CAPITOLO 2

#Sistemi_Operativi

Processi

Un processo è un programma in esecuzione. I processi rappresentano un'astrazione a livello utente che permetta l'esecuzione di programmi. Ad ogni processo è associato uno spazio di indirizzamento e un PID (Process ID) univoco.

Ogni processo presenta un layout comune identificato da:

- Stack : contiene i puntatori ai dati e viene utilizzato per la gestione delle chiamate di funzioni e delle variabili locali.
- Data : contiene le variabili del programma, incluse le variabili globali e i dati inizializzati.
- Text : contiene il codice eseguibile del programma.
- Gap : è uno spazio necessario per la gestione dinamica dell'allocazione e deallocazione di memoria nell'arco dell'esecuzione del processo.

Il ciclo di vita di un processo comprende diverse fasi, e le informazioni sui processi attivi vengono mantenute in una tabella dei processi attivi. Un processo può essere creato (allocato nella lista dei processi), terminato (deallocato dalla lista dei processi), messo in pausa e ripreso.

Inoltre, un processo può creare un altro processo, stabilendo una relazione padre-figlio .
Il possesso di un processo è assegnato a un utente identificato da un UID (User ID). Un processo figlio ha lo stesso UID del processo padre. Gli utenti possono appartenere a gruppi identificati da un GUID (Group ID).

In un sistema multiprogrammato, ciascuna CPU passa rapidamente da un processo all'altro, eseguendo ognuno per decine o centinaia di millisecondi. La CPU in realtà esegue un solo processo alla volta, ma nel corso di 1 sec. può elaborarne parecchi e dare l'illusione del parallelismo (Pseudoparallelismo). Mentre si parla di multiprocessore quando ci sono 2 o più processori a livello hardware.

Ogni processo ha il proprio flusso di controllo (istruzioni che deve eseguire) e la CPU tiene traccia della prossima istruzione da eseguire all'interno del registro PC Program Counter.

Esecuzione, creazione e terminazione dei processi

I processi possono essere eseguiti in tre modi:

- Esecuzione sequenziale : i processi vengono eseguiti uno dopo l'altro. Si tratta di un'esecuzione tipica dei sistemi a singolo processore. Facile gestire l'esecuzione dei processi, ma tempo considerevole per farlo.
- Esecuzione parallela : più core eseguono processi diversi, ogni core ha un processo in parallelo.
- Esecuzione concorrente : si simula l'esecuzione parallela tramite Multitasking . Quindi in questo modo più processi o thread possono essere attivi nello stesso momento, ma se si ha un singolo core non si possono eseguire davvero contemporaneamente.

Quando il SO decide di sospendere un processo (per dare la CPU a un altro processo) salva il PC del processo sospeso e altre informazioni di contesto nella tabella dei processi PCB, per poi riprenderle successivamente. I processi da eseguire vengono decisi secondo uno scheduler e degli algoritmi di scheduling.

I processi vengono creati quando:

- inizializzazione del sistema: vengono creati molti processi, sia attivi che in background
- esecuzione di una chiamata di sistema per la creazione di un processo da parte di un processo in esecuzione (fork()).
- richiesta dell'utente di creare un nuovo processo.
- avvio di un lavoro in modalità .

I processi terminano quando:

- uscita normale.
- uscita a causa di un errore (ad es. in C "return 0").
- errore fatale (ad es. in C "segmentation fault").
- ucciso da un altro processo.

Un processo può trovarsi in 3 stati possibili : running (in esecuzione), ready (eseguibile, temporaneamente fermo per consentire l'esecuzione di un altro processo), blocked (non può essere eseguito fino a quando non si verifica un evento esterno).

Fra i 3 stati disponibili 4 transizioni:

1. Il processo di blocca in attesa di input.
2. Lo scheduler sceglie un altro processo.
3. Lo scheduler sceglie questo processo.
4. Accada quando si verifica l'evento esterno di cui un processo era in attesa.

Interrupt, signals e thread

Un interrupt rappresenta un meccanismo adottato nel SO nel momento in cui avviene un evento asincrono, ossia un evento indipendente dall'esecuzione del software.

Si tratta di eventi legati all'hardware e per questo che devono essere gestiti e risolti immediatamente, e per farlo è necessaria una routine di servizio detta ISR (Interrupt service routine) .

Ogni periferica I/O e linea di interrupt presenta uno specifico interrupt vector che contiene, tra le altre cose, l'indirizzo dell'ISR necessaria per risolvere l'interrupt.
Quando avviene l'interrupt:

- viene salvato il contesto del processo interrotto.
- la CPU salta all'indirizzo specificato dall'interrupt vector, eseguendo la ISR identificata.
- terminata l'ISR viene chiamato lo scheduler per scegliere il prossimo processo da eseguire.
- Dopo ogni interrupt il processo torna esattamente allo stato in cui si trovava prima che avvenisse l'interrupt.

I signals sono eventi software generati da un processo o dal sistema operativo. Sono inviati asincronicamente ma possono essere gestiti in modo sincrono. Ogni tipo di segnale ha comportamento standard (es. Sigint : interruzione processo, Sigkill terminazione immediata programma, Sigterm terminazione programma).

Ogni processo dispone di uno spazio degli indirizzi e di un singolo thread di controllo. Tuttavia, ci sono frequenti situazioni in cui è utile avere più thread di controllo in esecuzione nello stesso spazio degli indirizzi, quasi in parallelo. La multithread execution implica che un processo può avere N thread in esecuzione.
I thread sono dei mini processi leggeri, poiché a differenza dei processi, non hanno un loro PID, un loro spazio di indirizzamento, ecc...
Inoltre, consentono un parallelismo efficiente in termini di spazio e di tempo, in quanto non devono essere gestiti dallo scheduler ma è il processo stesso che li deve gestire.

I thread si trovano tutti nella stessa area di memoria del processo che li ha generati, quindi sono in grado di scrivere e leggere dalla stessa memoria condivisa e questo comporta problemi di sincronizzazione tra thread.

Le uniche informazioni personali di ciascun thread sono : stack, registri, memoria, stato.

Sincronizzazione e comunicazione tra processi

I processi hanno bisogno di un modo per comunicare in modo da condividere i dati e sincronizzarsi con altri processi durante l'esecuzione. Vi sono 3 problemi : lo spazio condiviso in cui i processi possono accedere, il fatto che più processi non debbano intralciarsi il lavoro a vicenda e l'accettarsi che vi sia una corretta sequenzialità nell'esecuzione dei processi.

In particolare si introducono i concetti di:

- Race Conditions : si tratta di una situazione in cui due o più processi accedono nello stesso momento alla stessa risorsa condivisa. Se entrambi i processi devono scrivere dei dati, allora il risultato finale dipende esclusivamente dal tempo preciso di esecuzione dei processi. Ovvero, i processi "corrono" insieme per ottenere l'accesso ad una risorsa.
- Regione Critica : per evitare le race conditions serve una mutua esclusione, ossia un qualche sistema per essere certi che, se un processo sta usando una variabile o un file condiviso, agli altri processi venga impedito di fare la stessa cosa. Vi è la regione critica, ossia una sezione di codice in cui si accede a risorse condivise. Quindi la soluzione alle race conditions è di non fare accedere due processi/thread contemporaneamente nella regione critica. Per ottenere una buona soluzione servono 4 condizioni da rispettare:

1. Due processi non possono trovarsi contemporaneamente all'interno delle rispettive regioni critiche.
2. Non si possono fare ipotesi sulla velocità o sul numero di CPU.
3. Nessun processo in esecuzione al di fuori della proprio regione critica può bloccare altri processi.
4. Nessun processo deve aspettare all'infinito per entrare nella propria regione critica.

Mutua esclusione con busy waiting

Si tratta di una tecnica di sincronizzazione (evita le race conditions) per cui i processi o thread attendono in un ciclo attivo (ossia senza far nulla di produttivo) di accedere alla regione critica. Vediamo le strategie:

- Disabilitare gli interrupt : quando un processo entra nella sua regione critica, disabilita gli interrupt della CPU e non li riattiva finché non ha terminato. Ciò significa che in quel tempo la CPU non può rispondere ad eventi esterni che possano causare cambi di contesto, come un interrupt di clock (che attiverebbe un altro processo). In questo modo nessun altro processo può essere schedulato e quindi entrare nella regione critica. Non è opportuno dare la possibilità a processi utente di disabilitare gli interrupt ed inoltre questa tecnica funziona solo con sistemi UniCore.
- Bloccare le variabili : vengono utilizzate variabili di blocco per proteggere le regioni critiche, 0 se blocco libero e 1 se blocco occupato.
- Alternanza stretta : si utilizza una variabile turn per tenere traccia del turno in cui un processo può entrare nella regione.
- Peterson's Algorithm : Alice e Bob vogliono usare un'unica postazione computer in un ufficio. Ma ci sono delle regole:
    1. Solo una persona può usare il computer alla volta.

2. Se entrambi vogliono usarlo contemporaneamente, devono decidere chi va per primo.

Idea : Alice e Bob devono segnalare il loro interesse a usare il computer. Se l'altro non è interessato, la persona interessata può usarlo subito. Se entrambi mostrano interesse, registrano il loro nome su un foglio. Ma se scrivono quasi allo stesso tempo, l'ultimo nome sul foglio ha precedenza. Perché: garantisce che sempre uno avrà scritto e l'ultimo sarà quello che si prende la risorsa. La persona che non ha la precedenza aspetta finché l'altra ha finito. Una volta finito, la persona che ha usato il computer segnala che ha finito, e l'altra può iniziare.

Sleep e Wakeup

Si tratta di una tecnica di sincronizzazione che evita il busy waiting. Quest'ultimo infatti, poiché i processi rimanevano a ciclo attivo (quindi in loop controllando continuamente se la condizione di accesso alla regione critica risulta soddisfatta) consuma cicli della CPU senza lavoro utile.

Per evitare il busy waiting si utilizzano due funzioni primitive:

- Funzione di sleep : quando un processo, per continuare la propria esecuzione, necessita di accedere a una regione critica già occupata, invoca la funzione di sleep sospendendo la propria esecuzione e cedendo la CPU allo scheduler.
- Funzione di wakeup : viene chiamata dal processo o thread che detiene la risorsa condivisa, segnalando ai processi in sleep che la risorsa si è liberata e che quindi possono usufruirne riprendendo la propria esecuzione.

Semafori

Il semaforo è una variabile che può essere 0 (nessun wakeup) o un valore positivo (wakeup in attesa). Su questa variabile si possono eseguire le seguenti operazioni:

- Down : se il valore del semaforo è maggiore di 0, viene decrementato e il processo continua la sua esecuzione. Se è = 0, il processo che invoca down viene bloccato e messo in una coda di attesa associata al semaforo.
- Up : se il valore è 0, ci sono processi nella coda di attesa che vengono "svegliati". In ogni caso, il valore viene incrementato e il processo continua la sua esecuzione.

Le operazione di Up e Down su un semaforo sono operazioni atomiche, ossia vengono eseguite senza interruzioni.

Mutex

Un mutex è una versione esplicita e semplificata dei semafori, usata per gestire la mutua esclusione di risorse o codice condiviso, quando non è necessario contare gli accessi e altri

fenomeni. Può essere in due stati:

- locked (bloccato)
- unlocked (sbloccato)

Un solo bit può rappresentarlo, ma spesso viene utilizzato un intero (0-unlocked, altri-locked). Due procedure principali sono *mutex_lock* e mutex_unlock :

- Quando un thread vuole accedere a una regione critica, chiama mutex_lock.
- Se il mutex è unlocked, il thread può entrare; se è locked, il thread attende.
- Al termine dell'accesso, il thread chiama mutex_unlock per liberare la risorsa.
- IMPORTANTE : Non si utilizza il busy waiting. Se un thread non può acquisire un lock, chiama thread_yield per cedere la CPU ad un altro thread.

In definitiva :

- i semafori possono essere utilizzati sia per la gestione dell'accesso alle risorse condivise che per la sincronizzazione tra thread. Tuttavia, non avendo una semantica di proprietà (qualsiasi thread può incrementarne o decrementarne il valore indipendentemente da chi lo ha modificato l'ultima volta), è preferibile utilizzarli per la sincronizzazione tra thread.
- I mutex vengono principalmente utilizzati per la mutua esclusione poiché a differenza dei semafori, avendo la semantica di proprietà, garantiscono che due thread diversi non possono accedere alla stessa regione critica nello stesso istante.

Monitor

Un monitor è un costrutto di variabili, strutture dati e procedure utilizzato per la sincronizzazione di processi/thread. I processi possono chiamare le procedure ma non accedere direttamente alle strutture dati e alle variabili, e soltanto un processo alla volta può essere attivo nel monitor (mutua esclusione).
I monitor usano due variabili condizionali e due operazioni su di esse: wait e signal .
Wait mette in attesa il thread che l'ha invocato su una variabile condizionale e signal risveglia il thread in attesa su quella variabile.
Le variabili condizionali non accumulano segnali, per cui se signal è inviato prima che il thread venga messo in attesa allora questo viene perso.
I monitor sono quindi strumenti di sincronizzazione che garantiscono l'accesso mutualmente esclusivo a risorse condivise e forniscono un meccanismo per la sincronizzazione attraverso variabili condizionali.

Scambio di messaggi

E' un altro metodo per gestire la sincronizzazione tra processi, utilizza due primitive: send(dest, &msg) e receive(src, &msg).
E' utile in sistemi distribuiti (insiemi di computer indipendenti che appaiono agli utenti come singolo sistema) per la comunicazione diretta senza memoria condivisa.

Barriere

Sono un costrutto software utilizzato per sincronizzare processi in fasi diverse. Quando un processo raggiunge la barriera, attende fino a quando anche gli altri processi la raggiungono.

Si ha una situazione problematica quando si presenta l'inversione di priorità : un thread con priorità più bassa detiene una risorsa che un thread con più alta priorità dovrebbe utilizzare (si può risolvere disattivando l'interrupt oppure via Priority Ceiling).

Read-Copy-Update

L'obiettivo del Read-Copy-Update è di accedere in modo concorrente senza lock, cercando di evitare l'incosistenza dei dati. L'idea è di aggiornare strutture dati consentendo letture simulate senza incappare in versioni inconsistenti dei dati. I lettori vedono o la versione vecchia o quella nuova, mai un mix delle due. E' diffuso nel kernel dei sistemi operativi.

Introduzione allo scheduling

In un sistema multiprogrammato molteplici processi e thread competono per la CPU, quindi è necessario scegliere a quale processo o thread assegnare la CPU. La parte di sistema operativo che effettua lo scheduling è detto scheduler e l'algoritmo che utilizza si chiama algoritmo di scheduling.
Quando si parla di scheduling, ciò che è costoso in termini di tempo è: cambio di contesto (passaggio dall'esecuzione di un thread/processo a un altro), aggiornamento della MMU (Memory Management Unit), esecuzione dell'algoritmo, cambio di modalità utente a kernel, invalidazione della cache (per garantire che i dati siano consistenti rispetto a quelli in memoria), salvataggio dello stato del processo (Ready, Running, Blocked).

Gli obiettivi di scheduling sono tre:

- Equità : si vuole dividere equamente la CPU tra tutti i processi/thread.
- Bilanciamento : si vuole far si che tutte le componenti del sistema vengano mantenute attive, senza sovra o sottoutilizzazioni.
- Imposizione della policy
  : si vuole garantire che le politiche dichiarate per lo scheduling siano effettivamente rispettate.

Esistono due tipologie di processi :

- Processi CPU-bound : si tratta di processi con burst di CPU (ovvero tempi di utilizzazione) lunghi e intensi, ciò è dovuto al fatto che raramente necessitano e quindi aspettano operazioni di I/O.
- Processi I/O-bound : al contrario, si tratta di processi con burst di CPU brevi e frequenti attesa di operazioni I/O. Questa caratteristica non è dovuta alla durata del risolvere le richieste di I/O (infatti calcoli pochi e tempo per risolvere breve), quanto alla loro frequenza.

Lo scheduler opera principalmente all'avvenire di uno di questi 4 eventi:

- Interrupt di I/O
- Creazione di un nuovo processo (bisogna scegliere quale processo eseguire almeno tra padre e figlio appena creato, entrambi in stato di ready).
- Uscita di un processo (occorre quindi scegliere un altro tra i processi pronti).
- Blocco di un processo (ad es. a causa di un semaforo): occorre scegliere un nuovo processo).

Esistono infine due principali tipologie di algoritmi di scheduling, che variano a seconda della Prelazione (ossia della capacità di interrompere l'esecuzione di un processo per assegnarne un altro alla CPU):

- Non Preemptive (senza prelazione): lascia eseguire un processo scelto finché si blocca o lascia volontariamente la CPU.
- Preemptive (con prelazione): sceglie di eseguire un processo per un tempo fissato.

Scheduling nei sistemi batch

Un sistema Batch rappresenta un sistema in cui molti processi vengono eseguiti sequenzialmente e in modo automatico, senza interazione diretta con l'utente.
In sistemi di questo tipo è preferibile utilizzare algoritmi senza prelazione o al limite con prelazione ma dai tempi molto lunghi prima che il processo venga fermato.
Tra gli obiettivi in questo tipo di scheduling vi è quello di massimizzare il Throughput (numero di job eseguiti al secondo), minimizzare il tempo di TurnAround (tempo di esecuzione di un singolo job), mantenere la CPU sempre impegnata.

- FIFO (first-come, first served): algoritmo di scheduling senza prelazione in cui i processi vengono assegnati alla CPU nell'ordine di arrivo. Vi è in particolare una singola coda di processi in stato di pronto, quando un processo in esecuzione si blocca viene eseguito il successivo e quando quel processo torna in stato di Ready viene messo in fondo alla coda. Si tratta di un algoritmo facile da implementare ed equo in base all'ordine di arrivo dei processi, tuttavia ha lo svantaggio di presentare talvolta tempi di esecuzione molto lunghi.

- SJF (Shortest Job First) : algoritmo di scheduling senza prelazione. Data la precondizione che i tempi di esecuzione dei job si conoscano in anticipo, lo scheduler esegue il job più breve. Ha il vantaggio di minimizzare il tempo di TurnAround medio se i job arrivano contemporaneamente, ma non è ottimale se i job non sono disponibili in contemporanea..
- Shortest Remaining Time Next : variante di SJF con prelazione. L'algoritmo parte scegliendo il job più breve, quando poi arriva un nuovo job confronta il tempo necessario per eseguirlo col tempo rimanente per eseguire il job attuale. Se il tempo rimanente è superiore il processo attuale viene sospeso e il nuovo job eseguito. Altrimenti il nuovo job viene messo in una coda con priorità di job ready. I vantaggi di questo algoritmo risiedono nel fatto che minimizza il tempo di TurnAround medio ed è particolarmente efficiente in situazioni in cui i job hanno tempi di esecuzione molto diversi; lo svantaggio è che talvolta job con tempi molto lunghi vengono reinviati molteplici volte.

Scheduling nei sistemi interattivi

Nell'ambiente dei sistemi interattivi , progettati perché vi siano risposte rapide alle richieste dagli utenti in tempo reale, l'uso della prelazione è essenziale al fine di evitare che un processo monopolizzi la CPU e neghi il servizio agli altri. Tra gli obiettivi:

- Minimizzare il tempo di risposta alle richieste dell'utente.
- Soddisfare l'adeguatezza complessiva del sistema.

1. Round-Robin Scheduling : ogni processo riceve un intervallo di tempo, detto 'quanto', in cui viene eseguito. Quando il processo esaurisce il suo quanto, finisce in fondo alla coda di processi in attesa e la CPU prelaziona immediatamente un nuovo processo (indipendentemente dal fatto che quello precedente abbia terminato o meno).
2. Priority Scheduling : a ciascun processo è assegnata una certa priorità, e l'esecuzione è consentita al processo di priorità più alta. Per impedire che processi con alta priorità siano eseguiti indefinitivamente, talvolta lo scheduler può abbassarne le priorità (e alzarla se si arriva a priorità 0). Le priorità possono essere assegnate in modo:
    - Statico : rispetto alle caratteristiche di un processo.
    - Dinamico : basato sull'utilizzo della CPU e sul comportamento I/O-bound.
3. Shortest Process Next : sceglie di volta in volta i processi con stima di tempo di esecuzione più breve, ottimizzando così il tempo di TurnAround complessivo. Il calcolo della stima avviene in base al comportamento passato del processo in questione (Aging). Questa stima viene calcolata, ogni volta che viene eseguito il processo, come: T_{n+1} = a T_n + *(1-a)* T_{n+1} . Dove T_n è il tempo stimato , T_{n+1} è il tempo reale di esecuzione appena misurato e a è un parametro da 0 a 1. Più è grande, più si darà peso al tempo stimato precedente.
4. Guaranteed Scheduling : Il sistema tiene traccia di quanta CPU ha ottenuto ogni processo dal momento della sua creazione. Valuta il rapporto tra tempo CPU consumato e quello

dovuto (1/n), ed esegue il processo con rapporto più basso fintanto che non è più lui ad avere il rapporto più basso.

5. Lottery Scheduling : L'idea è dare ai processi dei biglietti per accedere alla CPU. Quando deve essere assegnata una risorsa, lo scheduler estrae un biglietto. Processi più importanti possono ottenere più biglietti ed inoltre questo algoritmo ha le proprietà di reattività e cooperazione tra processi (si possono scambiare biglietti tra loro).
6. Un altra proprietà è quella di Fair-Share Scheduling per cui si considerano anche gli utenti a cui appartengono i processi da schedulare e lo scheduler si assicura che ogni utente riceva la sua frazione, indipendentemente dal numero di processi posseduti (ad es. se un utente ha 9 processi e l'altro 1 allora il primo si prende 90% della CPU, non va bene).

Scheduling nei sistemi Real-Time

I sistemi Real-Time sono progettati per gestire applicazioni e processi che hanno requisiti temporali rigorosi e vincolanti. In sistemi di questo tipo la prelazione non è sempre necessaria perché i processi "sanno" di non poter essere eseguiti per lunghi periodi e quindi generalmente vengono eseguiti per poi bloccarsi rapidamente.

Tra gli obiettivi vi sono rispetto delle scadenze, prevedibilità, assicurarsi che il funzionamento sia costante.
Questi sistemi sono divisi in due categorie: Hard Real Time in caso di scadenze assolute da rispettare e Soft Real Time se c'è un grado di tollerabilità.

Gli eventi in un sistema Real Time sono di tipo periodico se avvengono a intervalli regolari e non periodico se avvengono in modo imprevedibile.

Dati m eventi periodici, l'evento i avviene in un periodo P_i e richiede C_i secondi di tempo della CPU per gestirlo, allora il carico può essere gestito solo se:

$$\sum_{i=1}^{m} \frac{C_i}{P_i} \leq 1$$

In tal caso il sistema si dirà schedulabile .

Scheduling di Thread

Lo scheduling differisce in base al tipo di thread, se sono a livello utente o kernel.

- Thread a livello utente : il kernel vede solo il processo ed ignora l'esistenza dei singoli thread che lo compongono (quindi il kernel gestisce solo i processi e non i thread individuali). Esiste uno scheduler interno ad ogni processo che decide quale thread eseguire, senza interruzioni del clock del sistema. Poiché il kernel assegna la CPU all'intero processo e non ai singoli thread, un thread a livello utente può consumare tutto il quanto

di tempo assegnato al processo. Lo scambio tra thread avviene inoltre molto rapidamente con poche istruzioni, poiché non si deve passare per ogni thread alla modalità kernel.

- Thread a livello kernel : il kernel seleziona il thread specifico per l'esecuzione. Se eccede il quanto, allora viene sospeso. In questo caso il passaggio tra thread richiede un cambio di contesto e quindi comporta un passaggio da modalità utente a kernel, ergo tempi di maggiori di esecuzione. Se un thread kernel richiede I/O non sospende stavolta tutto il processo, ma solo il thread in questione.