Riassunto Sistemi Operativi1 ©Andrea Ierardi

Sistema operativo e caratteristiche

Che cos'è il sistema operativo

Il sistema operativo **nasconde** al programmatore la verità sull'hardware mostrandolo in modo semplice e chiaro. Presenta inoltre un'interfaccia a file e nasconde soprattutto meccanismi come i timer, la gestione della memoria e altre caratteristiche a basso livello. La funzione del sistema operativo è quella di mostrare all'utente una macchina virtuale che sia più facile da programmare. Il sistema operativo fornisce servizi di cui i programmi posso usufruire attraverso funzioni speciali chiamate System Call.

Hardware per calcolatori

Processori

Il cervello del calcolatore è la CPU che recupera le **istruzioni dalla memoria e le esegue**. Siccome l'accesso in memoria per il prelievo delle istruzioni richiede più tempo, spesso le variabili e i risultati temporanei vengono salvati su registri.

Vi sono anche parecchi **registri speciali** come il Program Counter (PC) che contiene l'indirizzo di memoria della prossima istruzione, stack pointer (SP) che punta alla cima dello stack corrente di memoria e il Program Status Word (PSW) che contiene i bit di codice di condizione che vengono inizializzai da istruzioni di confronto priorità della CPU modalità kernel o utente. Il registro PSW svolge un ruolo fondamentale per le chiamate di sistema.

La maggior parte delle CPU hanno due modalità, modalità kernel e utente e generalmente un bit nel registro PSW controlla la modalità. Quando è in esecuzione in modalità kernel, la CPU po' eseguire qualsiasi istruzione ed usare ogni caratteristica dell'hardware. Il sistema operativo viene eseguito in modalità kernel in modo che abbia accesso completo. I programmi utente, vengono eseguiti in modalità utente ed hanno accesso solo ad un sottoinsieme di funzioni. Di solito sono bandite tutte le istruzioni che coinvolgono I/O e la protezione della memoria e di conseguenza non può neanche cambiare il registro PSW.

Per ottenere servizi dal sistema operativo un programma deve eseguire una chiamata di sistema che esegue una trap al kernel e invoca il sistema operativo. L'istruzione trap cambia da modalità utente a modalità kernel e avvia il sistema operativo. Quando il lavoro è concluso il controllo ritorna al programma utente.

Memoria

La memoria dovrebbe essere estremamente **veloce**, **abbondante** e a **bassissimo costo**. Nessuna tecnologia attuale rispetta questi obiettivi, così si è adottata una gerarchia di strati.

Lo strato più alto si compone dei registri interni alla CPU, essi sono costruiti con la stessa tecnologia della CPU e sono quindi veloci quanto questa però la capacità di memoria è parecchio limitata. Subito dopo viene la memoria cache che viene controllata dall'hardware ed è divisa in linee di cache. Le linee di cache usate più di frequente sono mantenute in una

cache ad alta velocità situata all'interno o vicino alla CPU. Quando il programma ha bisogno di leggere una linea dalla cache l'hardware della cache controllo se la linea è presente e così si ha cache hit ma se non trovata si ha una cache miss. Possono esserci più livelli di cache.

Di seguito vi è la RAM, infatti tutte le richieste della CPU che non possono essere soddisfatte dalla cache vanno alla memoria principale.

Il prossimo della gerarchia è il disco fisso ed è due ordini di grandezza più economica della RAM e anche di grandezza. L'unico problema è che il tempo di accesso ai dati è parecchio più lungo ed è dovuto in particolare che il disco è un dispositivo meccanico.

L'ultimo membro della gerarchia è il nastro magnetico ed è utilizzato come backup.

È possibile tenere due o più programmi contemporaneamente nella memoria principale ma occorre protegge i programmi l'uno dall'latro e il kernel dai programmi, ma anche gestire la rilocazione. Per fare ciò si utilizzano due registri chiamati base e limite. Quando si esegue un programma il registro base è impostato al punto di inizio del suo codice mentre il registro limite è impostato all'ultimo elemento del codice del programma. Il dispositivo **Memory Management Unit (MMU)** che effettua il controllo che gli indirizzi a cui il programma accede siano i suoi e non vadano oltre i registri base e limite.

I/O

L'ingresso e l'uscita possono essere effettuati in tre modi diversi:

- Un **programma utente emette una chiamata di sistema** che il kernel traduce in una chiamata di procedura al driver che successivamente darà il via all'operazione I/O ed entrerà in un ciclo continuo interrogando il dispositivo per verificare la fine dell'operazione. Questo però genera busy waiting e tiene occupata la CPU.
- Il secondo metodo prevede che **il driver faccia partire il dispositivo e mandi un'interruzione al driver** quando ha finito dopodiché il driver termina l'esecuzione. Quando il controllo rivela la fine di un trasferimento genera un interrupt

Interrupt

Le **interruzioni** sono importanti nei sistemi operativi. Quando avviene un'interruzione e la CPU la accetta il PC e il PSW vengono messi sulla pila corrente e la CPU passa in **modalità kernel**.

Il numero di dispositivo può essere usato come indice per cercare l'indirizzo del gestore di interruzioni del dispositivo. Questa parte di memoria è chiamata vettore delle interruzioni. Questo dispositivo fa parte del **driver del dispositivo** che ha mandato l'interruzione. Quando il gestore ha terminato il controllo ritorna alla prima istruzione non ancora eseguita nel programma utente precedentemente in esecuzione.

Concetti base sul sistema operativo

I processi

Un processo è essenzialmente **un'astrazione del sistema operativo**. Ad ogni processo viene associato il suo **spazio di indirizzamento** e una lista di **locazioni in memoria** che il processo può leggere e scrivere. Inoltre, ad ogni processo sono associati un insieme di registri tra cui il PC, SP. Quando un processo viene sospeso, tutti i puntatori vanno salvati in modo che il processo possa ripartire dal punto in cui è stato sospeso. In molti sistemi operativi, infatti le informazioni di ciascun processo vengono salvate in una tabella di sistema chiamata **tabella dei processi** (**process table**) che è una lista per ogni processo esistente. Quindi un processo è costituito dal suo spazio di indirizzamenti che viene anche chiamato immagine in memoria. Le principali chiamate di sistema per la gestione dei processi sino quelle che si occupano della creazione e della terminazione. Un processo può creare uno o più processi detti processi figli. È possibile far comunicare i processi tra di loro attraverso dei meccanismi come la **condivisione di memoria** o **pipe**.

Gestione della memoria

Il sistema operativo inoltre si occupa della **gestione dei processi in memoria**. In alcuni sistemi è possibile tenere più processi in memoria contemporaneamente ma c'è bisogno di meccanismi di protezione che sono in hardware ma devono essere controllati dal sistema operativo. Inoltre il sistema operativo fornisce meccanismi per poter gestione indirizzi più grandi di 32 o 64 bit attraverso l'uso di una **memoria virtuale** che fa sì che il sistema gestisca lo spazio di indirizzamento mentendo una parte in memoria principale e una su disco trasportando avanti e indietro in caso di necessità.

L'ingresso/uscita

Ogni sistema operativo ha un sottosistema di ingresso e uscita per la **gestione dei dispositivi I/O**. Un calcolatore sarebbe inutile se non potesse comunicare con un utente attraverso dispositivi.

I file

Un altro concetto importante in un sistema operativo è la **gestione del file system**. Il sistema operativo quindi maschera le particolarità dei dischi e di altri dispositivi mantenendo un modello pulito e astratto di file indipendentemente dal dispositivo. Per fornire un posto dove tenere i file si usa un concetto di directory come scatola in cui raggruppare i file.

Inoltre possono essere utilizzati file speciale come le pipe che possono permettere la comunicazione tra processi diversi.

Sicurezza

In UNIX per garantire la sicurezza si utilizzano **3 bit per i permessi** di lettura, scrittura ed esecuzione di certi programmi e inoltre vi è una divisione tra utente, gruppo e altri.

Chiamate di sistema

Ogni calcolatore a singola CPU può eseguire un'istruzione per volta e se un processo che sta eseguendo un programma utente in modalità utente ha bisogno di un **servizio di sistema**, come leggere dati da un file, deve eseguire una trap o un'istruzione che esegue una chiamata per trasferire il controllo al sistema operativo. Il sistema operativo riesce a capire cosa vuole il processo chiamante esaminando i parametri e dopodiché restituisce il controllo all'istruzione seguente alla **chiamata di sistema**. In sintesi eseguire una chiamata di sistema è come eseguire una chiamata ad una procedura speciale solo che le chiamate di sistema entrano a far parte del kernel, mentre quelli di procedura no. La procedura di libreria (generalmente in Assembler) mette il numero di chiamata di sistema in un posto noto al sistema, per esempio un registro ed esegue un'istruzione di TRAP per passare dalla modalità utente alla modalità kernel e inizia l'esecuzione ad un indirizzo fisso del kernel. Questo esamina il numero della chiamata di sistema e la smista al corretto gestore della chiamata di sistema. Viene eseguito il gestore delle chiamate di sistema, completa il suo lavoro e infine restituisce il controllo alla procedura di libreria nello spazio utente all'istruzione che segue la trap tornando al programma utente.

Processi e threads

Processi → Raggruppamento delle risorse ed esecuzione.

- **Esecuzione** -> processo vuole essere eseguito
- **Pronto** -> processo vuole essere eseguito ma non c'è CPU disponibile in quel momento
- **Bloccato** -> processo in attesa di un evento esterno

Implementazione dei processi avviene grazie ad una tabella dei processi con un elemento entry per ogni processo **Process Control Block**(**PCB**) che contiene informazioni sullo stato del processo (PC, SP, file aperti, allocazione in memoria).

Ad ogni dispositivo I/O viene associata una locazione chiamata **vettore delle interruzioni**, che contiene l'indirizzo di procedura di gestione delle interruzioni.

Le interruzioni iniziano tutte col salvataggio dei registri.

Threads → Raggruppamento delle risorse correlate. Processi raggruppano risorse, threads entità schedulate per l'esecuzione nella CPU.

Ha una stack che contiene la storia dell'esecuzione, PC e registri che mantengono le variabili di lavoro.

I threads condividono i file aperti, lo spazio di indirizzamento e altre risorse, mentre i processi condividono la memoria fisica, i dischi, le stampanti e altro.

Il termine **multithreading** è una situazione in cui ad un solo processò sono assegnati più threads. Avere un thread in tre processi è diverso da avere tre threads in un processo. Nel primo caso i threads non condividono la memoria e le variabili, mentre nel secondo caso condividono lo stesso spazio d'indirizzamento.

Su un sistema a CPU singola i threads multipli danno l'illusione di essere eseguiti in parallelo mentre in realtà la CPU passa avanti e indietro tra i threads.

Un thread può leggere, scrivere o cancellare la stack di un altro thread. Non c'è protezione tra threads poiché non si può realizzare e perché non dovrebbe essere necessaria.

Le transizioni di stato dei processi è lo stesso dei threads (Esecuzione, Bloccato, Pronto, Terminato).

Vantaggi threads rispetto ai processi:

- **Non vi sono cambi di contesto**, interruzioni pesanti e la possibilità di poter condividere spazio di indirizzamento tra di loro
- È possibile la **creazione e la distruzione dei threads è più facile** rispetto a quella dei processi poiché non hanno alcuna risorsa associata.
- **Le prestazioni sono migliori**. Con parecchie operazioni di I/O vi è la sovrapposizione di queste attività, velocizzando le operazioni

Implementazioni possibili dei threads: **livello utente, livello kernel o ibrida**

Threads livello utente

I threads vengono interamente inseriti a livello utente e il kernel non sa nulla di loro, quindi gestirà i processi ordinari ad un solo thread. Essendo gestiti a livello utente ogni threads avrà la propria **tabella dei threads** per tenere traccia dei propri thread. Se thread bloccato in esecuzione (attesa di un altro thread) memorizza i registri nella tabella e cerca un nuovo thread pronto e carica i registri macchina del nuovo thread.

Vantaggi:

Implementazione su sistemi operativi che non supportano i threads.

Non sono necessari **cambi di contesto, né trap del kernel** e la memoria **cache non** deve essere **svuotata** (schedulazione più veloce).

Possibile implementazione di algoritmi di Scheduling.

Non è richiesto troppo spazio in memoria per le tabelle e lo stack e ciò è problematico se il thread è molto grande.

Svantaggi:

Non è possibile utilizzare **primitive bloccanti** poiché questa bloccherebbe tutti i threads.

Una soluzione potrebbe essere una modifica alle chiamate di sistema ma uno dei vantaggi che si voleva ottenere è la portabilità tra diversi sistemi operativi, quindi si potrebbero utilizzare del codice **jacket** per esegui controlli sulle chiamate di sistema.

Possibili **Fault di Pagina** se il programma chiama o salta a istruzioni che non sono in memoria e il sistema operativo dovrà andare a recuperare le istruzioni mancanti da disco. Se un thread causa un Fault di Pagina e il kernel non sapendo dell'esistenza di altri threads bloccherà l'intero processo.

Se un thread inizia l'esecuzione nessun altro verrà eseguito a meno che non venga rilasciata la CPU volontariamente dal primo.

I programmatori necessitano di più threads per calcoli intensivi quindi l'utilizzo di quelli a livello kernel non è conveniente.

Threads livello kernel.

Non serve un sistema a tempo di esecuzione e non c'è una tabella dei threads per ogni processo ma è il kernel ad avere una tabella dei threads del sistema.

Per la creazione o la distruzione di un thread si esegue una **chiamata di sistema** che effettua l'operazione e aggiorna la tabella dei threads del kernel.

Tutte le **chiamate bloccanti** sono implementate come chiamate di sistema che sono però più costose rispetto a procedure a tempo di esecuzione.

Quando un **thread si blocca**, il kernel sceglie se eseguire un altro thread dello stesso processo (se ce n'è uno pronto) o di un altro processo.

Si utilizza un metodo di **riciclaggio dei thread** marcando quelli distrutti come 'non eseguibile' ma le strutture non vengono modificate in modo tale che ad una successiva chiamata non si abbia troppo overhead.

Se si hanno maggiori operazioni sui thread il costo sarà alto (creazione, terminazione etc..).

Threads ibridi

Sono stati realizzati per combinare i vantaggi dei threads a livello utente e a livello kernel.

Si potrebbe quindi utilizzare threads a livello kernel per mappare(multiplexing) molti thread a livello utente su alcuni o tutti i threads nel kernel.

Per combinare i vantaggi delle due realizzazioni (in particolare, risparmiare spazio del kernel e diminuire il costo del context switch rispetto alla gestione interamente a carico del kernel) è stata introdotta una terza possibilità, per un processo P può esistere:

- un numero **N** di thread a **livello utente**
- un numero **M** ($\leq$ N) di thread a **livello kernel** Il sistema operativo si occupa di schedulare i thread del kernel Il sistema di gestione dei thread si occupa della corrispondenza "insieme thread utente – thread del kernel" e decide quale fra i k thread utente associati ad un dato thread del kernel far eseguire

Sincronizzazione

Spesso i processi necessitano di comunicare tra di loro, preferibilmente in modo strutturato e senza utilizzare interruzioni.

Tre questioni da considerare:

- Come può un processo **passare informazioni** ad un altro?
- Fare in modo che **due processi non entrino in conflitto** tra di loro generando corse critiche
- Mettere **in sequenza i processi** in modo adeguato quando esistono delle dipendenze.

  Ex. Se processo A produce e B stampa risultati, allora B deve aspettare che A abbia prodotto qualche dato prima di consumarlo.

Queste tre considerazioni si applicano anche ai threads tranne la prima, poiché per questi è più facile il passaggio d'informazioni dato che condividono uno spazio d'indirizzamento comune.

Corse Critiche e Sezioni Critiche

In alcuni sistemi operativi i processi possono **condividere una parte di memoria comune**. Questa può essere in memoria principale (magari struttura dati nel kernel) oppure può essere semplicemente un file condiviso.

Una condizione di corsa critica (race condition) è una situazione nella quale due o più processi stanno leggendo o scrivendo un qualunque dato condiviso ed il risultato finale dipende dall'ordine in cui vengono eseguiti i processi. L'ordine può cambiare ad ogni esecuzione.

Per evitare le corse critiche vi è bisogno di una **mutua esclusione** che è realizzabile attraverso la scelta di primitive bloccanti appropriate ed è infatti uno dei problemi maggiori in fase di progettazione di ogni sistema operativo.

La parte del programma in cui un processo accede alla memoria condivisa viene chiamata **sezione critica**.

Quindi per evitare corse critiche bisogna evitare che due o più processi entrino nella stessa sezione critica contemporaneamente.

Per avere una buona soluzione bisogna soddisfare quattro condizioni:

- Due processi non devono mai trovarsi all'interno delle loro **sezioni critiche contemporaneamente**
- Non si deve fare alcuna **ipotesi sulla velocità** e sul numero delle CPU poiché non è possibile determinare quale processo sarà eseguito per primo
- Nessun processo in esecuzione fuori dalla sua sezione critica può **bloccare altri processi**
- Nessun processo deve **aspettare all'infinito** per poter entrare in una sua sezione critica

Mutua esclusione con attesa attiva (busy waiting)

Disabilitazione degli interrupt

La soluzione più semplice è permettere ad ogni processo di disabilitare le proprie interruzioni non appena entra nella sua regione critica, e riabilitarle non appena ne esce.

Non è saggio dare ad un processo utente la possibilità di poter disabilitare gli interrupt, poiché se non le riabilitasse più sarebbe la fine per il sistema. In più in un sistema multiprocessore si disabiliterebbero solo su una delle CPU e le altre continuerebbero a girare comunque. Non è una tecnica appropriata per la mutua esclusione.

Variabili di lock

Si Utilizza una variabile condivisa lock inizializzata a 0. Se un processo vuole entrare nella regione critica vede che lock vale 0 e la imposta a 1 ed entra. Un secondo processo vedendo lock a 1 dovrà aspettare fino a che non viene. Sfortunatamente è possibile che un processo veda il valore di lock a 0 e prima che possa metterlo a 1 allo stesso tempo un altro processo venga schedulato e imposti lock a 1. Non appena il primo processo verrà di nuovo schedulato si avranno quindi due processi nelle loro sezioni critiche.

Alternanza stretta

Si utilizza la **variabile intera turno inizializzata a 0** che tiene traccia del processo quale tocca entrare in sezione critica. Un ‘processo 0’ legge turno ed entra, anche il ‘processo 1’ trova turno a 0 e entra in un piccolo ciclo che continua a testare turno per vedere se sia stata posta a 1 (busy waiting). Quando ‘processo 0’ lascia la sezione critica e pone turno a a1, a questo punto entrambi i processi sono in esecuzione nelle loro sezioni non critiche. Improvvisamente ‘processo 0’ termina la sezione non critica e ritorna all’inizio del ciclo ma ora non può entrare perché turno è a 1 e ‘processo 1’ è ancora occupato nel suo ciclo while. Non è una buona idea stabilire dei turni se un processo è più lento dell’altro.

Non soddisfa il requisito di “progresso”: se è il turno di un processo, anche se lui non vuole entrare, l’altro non può entrare

La soluzione di Peterson

Peterson scoprì un algoritmo molto più semplice per garantire la **mutua esclusione**. Ogni processo prima di usare variabili condivise richiama la **‘entra_nella_regione’** con il proprio numero di processo 0 o 1 come parametro per far sì che il processo debba in caso aspettare o entrare nella regione se sicuro.

Dopo aver lavorato su variabili condivise, viene chiamata la **‘lascia_la_regione’** per indicare che ha finito e per permettere ad un altro processo di entrare.

Se entrambi i processi chiamano ‘entra_nella_regione’ ed entrambi memorizzano il numero processo nella variabile turno. Il secondo valore memorizzato è quello che conta e il primo

verrebbe sovrascritto. Se il processo 1 scrive per ultimo nella variabile turno così che valga 1. Quando entrambi i processi arrivano all'istruzione while, il processo 0 non la esegue neanche una volta, ed entra dirittamente nella sezione critica, mentre il processo 1 cicla e non entra nella sezione critica finché il processo 0 non esce.

L'istruzione TSL

Per ovviare a ciò si può richiedere un piccolo aiuto in hardware con l'utilizzo di istruzioni a livello ISA. Queste istruzioni sono dette di **Test and Set Lock** (**TSL** – verifica e imposta il blocco). Viene **salvato** il valore della **variabile lock in un registro**. Le operazioni di lettura e scrittura sono invisibili →nessun altro processore può accedere finché l'istruzione non è finita. Quando CPU esegue un'istruzione TSL, il bus di memoria viene bloccato per impedire ad altre CPU di accedere. In poche parole è uguale alla soluzione proposta con lock ma non è possibile che due processi entrino in sezione critica allo stesso momento. Questo però genera busy waiting e quindi uso continuo di CPU e va bene nel caso di problemi non troppo complessi. Un'altra complicanza è la possibile inversione di priorità, quindi in caso un processo con priorità maggiore sia in attesa attiva ci rimanga indefinitamente poiché un processo secondario con priorità minore non verrà mai schedulato.

Sospensione e risveglio

Una soluzione al problema del busy-waiting potrebbe essere l'utilizzo di primitive di comunicazione che bloccano i processi anziché tenerli in attesa-attiva. Quindi un processo passerebbe allo stato di attesa invece di rimanere Running. Si potrebbero utilizzare quindi due primitive chiamate **sleep ()** e **wakeup(processo)**.

Il problema del produttore consumatore

Problematica → Potrebbe accadere che il buffer vuoto ed il consumatore ha appena letto cont per vedere se vale 0, se in quel momento lo Scheduler decidesse di sospendere temporaneamente il consumatore e manda in esecuzione il produttore. Il produttore chiamerebbe **una wakeup () che andrebbe persa** siccome il consumatore non era ancora entrato in attesa. Non appena lo scheduler riprende il consumatore entrerà in attesa per sempre poiché la wakeup è andata persa.

I Semafori

L'introduzione di un nuovo tipo di variabile, chiamato **semaforo** contribuì alla risoluzione del problema della mutua esclusione. Un semaforo può avere valore 0 se nessun wakeup è stata chiamata o un qualche valore positivo se una o più wakeup sono pervenute.

Vi sono due operazioni principali sui semafori → **down** e **up**.

La down su un semaforo controlla che il valore del semaforo sia maggiore di 0; se così fosse decrementa il valore e il processo continua l'esecuzione, mentre se il valore del semaforo fosse 0, il processo verrebbe quindi sospeso senza completare la down.

Queste operazioni sono tutte atomiche quindi indivisibili. Non possono avvenire altre operazioni finché non si ha terminato.

L'up, invece, incrementa il semaforo su cui viene invocata. Se uno o più processi erano sospesi sul semaforo, quindi impossibilitati a completare un'operazione di down, uno di questi viene scelto dal sistema(Scheduler) per permettergli di continuare l'operazione.

Semaforo Binario

Un semaforo binario può assumere solo i **valori 0 o 1** (o si tratta di un semaforo generale, usato solo in modo che assuma tali valori) Ha le stesse operazioni già viste; ma nel caso si tratti di un vero e proprio semaforo binario (non un semaforo generale usato come binario) se si esegue una up() quando s.val è già 1, questa non ha alcun effetto (oppure, come è comodo dire negli standard: l'effetto è indefinito, cioè: meglio scrivere i programmi in modo che non succeda, perché le conseguenze sono a scelta dell'implementazione) Possono essere utilizzati per garantire la mutua esclusione, inizializzandoli a 1 Se usati solo a tale scopo possono essere chiamati mutex, eventualmente con inizializzazione implicita Ma è un semaforo binario anche quello usato per "B in Pk solo dopo A in Pi" Va inizializzato a 0.

Semaforo Contatore

Un semaforo contatore può assumere qualsiasi **valore ≥ 0**. Può essere usato ad esempio per il problema di sincronizzazione con un numero N di risorse da assegnare: si inizializza a N, numero di "risorse" (in senso lato) disponibili preleva = down(&s) rilascia = up(&s) N processi/threads possono superare down senza nessuna up, l'N+1 esimo viene sospeso In generale, in ogni momento s.val è il numero di "risorse" disponibili, ≥ 0, e vale: N – numero down completate + numero up completate cioè risorse totali – risorse prelevate + risorse rilasciate

Semafori Privati

I **semafori privati** sono tali solo per come vengono usati: il meccanismo messo a disposizione dalle funzioni è lo stesso, senza alcun controllo su quale processo/thread usa i semafori e come:

- **Un semaforo privato** s_priv_P «di» un processo P (o di una classe di processi) è **inizializzato a 0**
- **solo il processo P** (o un processo della classe associata al semaforo) **esegue down**(&s_priv_P); la esegue quando deve attendere che diventi vera una condizione (booleana) di sincronizzazione
- **qualsiasi processo** (P incluso) **può eseguire up**(&s_priv_P) se serve svegliare P, o serve non farlo sospendere se fa down, perché è vera la condizione di sincronizzazione

Spinlock

Sono implementati tramite **attesa attiva**: questa scelta è efficace solo se:

- i threads che li utilizzano sono in esecuzione su due diverse CPU (il parallelismo quindi è reale, non solo emulato con il time-sharing) → mentre il thread in "attesa attiva" consuma cicli della CPU su un processore, l'altro può continuare finché non ha finito;
- **la sezione critica è breve**: sotto queste condizioni l'attesa attiva può essere vantaggiosa, perché evita di effettuare un cambio di contesto (context switch) sulla CPU su cui gira il thread che deve attendere.

Futex

i **futex** (fast user space mutex): con una istruzione tipo **TSL** (singola, non un ciclo) si decrementa una variabile di lock leggendo il valore precedente; se era 1 (libero) si va avanti, senza effettuare chiamate di sistema che comportano il passaggio a stato kernel; se no, si fa una chiamata al kernel per sospendere il thread In uscita si fa una operazione atomica di "incremento e test" e solo se necessario (ci sono thread sospesi: la var era < 0) si chiama il kernel per svegliarne uno. Non si chiama il kernel se non c'è "contesa" per l'accesso alla sezione critica.

I Mutex

I **mutex** sono utili per la **gestione della mutua esclusione** e sono utili nei threads nello spazio utente perché facili da implementare. Un mutex può avere solo due stati: **bloccato** e **non bloccato**, quindi è necessario solo un intero con il valore a 0 quando non bloccato e gli altri valori che indicano bloccato. Se il mutex è bloccato un thread viene bloccato finché un altro thread non esce dalla zona critica e lo sblocca. Essendo semplici si possono implementare nello spazio utente con un'istruzione TSL.

Nei thread a livello kernel è possibile salvare strutture dati e stati dei semafori nel kernel in modo tale da renderle condivisibili.

I Monitor

Sono **primitive bloccanti** di livello più alto rispetto ai semafori. I semafori devono essere posti nella posizione giusta nel codice altrimenti si rischierebbe di non far funzionare il programma. (Esempio un down prima del controllo se il buffer è pieno da parte del produttore→ Si bloccherà sul mutex, idem il consumatore che richiamerà successivamente una down).

Un monitor è una **collezione di procedure, strutture e variabili** che vengono raggruppate in un tipo speciale di modulo (o package). Solo un solo processo può essere attivo in un monitor ma non può accedere direttamente alle strutture dati ma solo richiamarle.

Se un processo richiama una procedura di un monitor in cui è già attivo un altro processo, questo verrà sospeso finché il monitor non sarà libero.

Bisogna però trovare un modo per far bloccare un processo su un monitor. La soluzione è l'utilizzo di **variabili di tipo condizione** (condition variables) con due operazioni associate **wait e signal**. Quando una procedura di un monitor scopre di non poter continuare chiama una wait su una variabile di tipo condizione. Questo provoca il blocco del processo chiamante e permette l'ingresso ad un altro processo a cui era stato precedentemente proibito entrare nel monitor. Questo consumatore può risvegliare il partner sospeso chiamando una signal

sulla variabile di condizione. Se viene eseguita una signal su una variabile di tipo condizione su cui nessuno è sospeso viene persa, quindi la wait deve avvenire prima di una signal.

Per l'uso dei semafori di solito si usano linguaggi di livello più alto rispetto al C (Java) poiché ci vuole un linguaggio che li sappia gestire in modo adeguato. Un altro problema con i monitor è che su un sistema a CPU multiple con memoria privata, collegate grazie ad una rete locale, sono inapplicabili.

Lo scambio di messaggi

Un metodo di comunicazione tra processi è lo scambio di messaggi. Questo metodo usa due primitive: send e receive che, come i semafori, sono chiamate di sistema e non costrutti di un linguaggio (monitor).

Send (destinazione, &messaggio); → Spedisce un messaggio ad una determinata destinazione

Receive (sorgente, &messaggio); →Riceve un messaggio da una determinata sorgente

Se non è disponibile nessun messaggio, il ricevente potrebbe bloccarsi finché non ne arriva uno o in caso terminare con un codice d'errore.

Si può utilizzare una struttura dati chiamata mailbox per poter bufferizzare un certo numero di messaggi. I parametri della send e della receive in questo caso diventeranno indirizzi del mailbox.

Il problema dei filosofi a cena

Un esempio di **problema di sincronizzazione** tra processi è il problema dei filosofi a cena (dining philosopher). Chiunque inventi una primitiva si sente obbligato a dimostrare come risolvere questo problema.

Vi sono **5 filosofi** ad un tavolo tondo con una forchetta ciascuno ma per poter mangiare gli spaghetti necessitano di due forchette poiché gli spaghetti sono troppo scivolosi. Si alternano periodi in cui i filosofi mangiano e pensano.

- Si potrebbe pensare per esempio che ogni filosofo debba aspettare una forchetta specifica prima di poter mangiare. Il problema è che se tutti i filosofi prendono la forchetta a sinistra ci sarà un deadlock poiché tutti aspetteranno di ricevere la forchetta a destra che non verrà mai ceduta.
- Si potrebbe fare un programma che, non appena un filosofo prende la forchetta sinistra, venga eseguito **un controllo sulla forchetta destra**. Se questa non è disponibile il filosofo rilascia la forchetta sinistra. Il problema potrebbe verificarsi se tutti i filosofi prelevano la forchetta a sinistra e la riposino accorgendosi che non vi sono forchette a destra e continuare così per un periodo di tempo indefinito. Questo fenomeno è chiamata starvation.
- Si potrebbe fare in modo che i filosofi potessero **aspettare un tempo casuale** tra un tentativo e l'altro. Ridurrebbe le probabilità di blocco del programma in modo drastico

ma se si volesse una soluzione che funzioni sempre non fa al caso nostro (esempio centrale nucleare)

- Un'altra soluzione più prestante potrebbe essere l'utilizzo di un **semaforo binario**. Prima di prendere possesso delle forchette un filosofo dovrebbe fare una down su un mutex e dopo averle deposte dovrebbe fare l'up. Il problema è che in questo modo non si avrebbe un parallelismo poiché un filoso alla volta può mangiare. (Mentre con cinque filosofi due dovrebbero mangiare allo stesso tempo)
- La soluzione quindi è l'utilizzo di un **vettore di semafori**, uno per filosofo, in modo che i filosofi affamati possano bloccarsi se le forchette di cui hanno bisogno risultano occupate. Se nessuno dei vicini sta mangiando allora un filoso può iniziare a mangiare.

In definitiva il problema dei filosofi a cena è utile per modellare processi che competono per l'acquisto esclusivo ad un numero di risorse limitato come nei dispositivi I/O.

Il problema dei lettori e scrittori

Un altro problema frequente è quello dei lettori e scrittori che modellano l'accesso ad un Database.

- **La prima soluzione** consiste in una down sul primo lettore che ottiene l'accesso al DB. I lettori incrementano un **contatore 'rc'** e via via che i lettori escono riducono rc e l'ultimo esegue una up sul semaforo per permettere ad uno scrittore bloccato (se c'è) di accedere alla base di dati. Il problema è che un continuo flusso di lettori che accedono al DB genereranno una situazione di **starvation** allo scrittore che rimarrà sospeso fino a che tutti i lettori avranno finito.
- Per evitare questa situazione quando un lettore arriva e uno scrittore sta aspettando il lettore viene sospeso dietro lo scrittore invece di accedere direttamente nel DB. In questo modo uno scrittore deve aspettare che i lettori attivi terminino e non viene sovrastato da nuovi lettori che cercano di accedere al DB. Il problema è che c'è **meno concorrenza** e quindi **peggiori prestazioni**.

Si possono quindi utilizzare due semafori privati uno per i scrittori e uno per i lettori entrambi inizializzati a 0 che servono a i lettori o agli scrittori per sospendersi quando non posso accedere al DB.

Scheduling

Introduzione

Quando il sistema è multi programmato ha spesso molti processi che competono per l'uso della CPU contemporaneamente e ciò si verifica se molto processi sono nello stato di pronto. Se è disponibile una sola CPU, bisogna scegliere il prossimo da eseguire. La parte del sistema operativo che si occupa di questa decisione di chiama **Scheduler**.

Lo scheduler oltre a doversi occupare del processo giusto da eseguire deve preoccuparsi di fare un uso efficiente della CPU perché il cambio di processo è costoso. Deve cambiare la modalità utente alla modalità kernel, deve quindi salvare lo stato del processo corrente, memorizzando i suoi registri nella tabella dei processi in modo che poi possano essere ricaricati. Quindi un algoritmo di schedulazione sceglie quale processo rendere running e ricarica la MMU con la mappa di memoria del novo processo e infine far partire il processo.

Alcuni processi spendono la maggior parte del tempo in calcoli e vengono chiamati CPU bound, mentre altri spendo più tempo in attesa di I/O e vengono chiamati I/O bound.

È importante che però in caso di processi orientati all'ingresso/uscita debbano essere eseguiti entro tempi previ in modo che possa impartire una richiesta al disco per tenerlo occupato.

Quando schedulare

Esistono varie situazioni in cui prendere le decisioni di schedulazione di un processo ready:

- Quando viene **creato un nuovo processo**, si deve decidere se eseguire il processo genitore o quello figlio
- Quando un processo **termina il processo** non può essere eseguito e bisogna scegliere un altro processo dall'insieme dei processi in stato di pronto. Se non ce ne sono presenti viene eseguito il processo inattivo fornito dal sistema
- Quando un **processo si blocca** su un'operazione di I/O o su un semaforo.
- Quando **deve essere presa una decisione di schedulazione** quando si verifica una interruzione da I/O→ quando viene terminata l'operazione per il dispositivo di I/O viene reso ready il processo che è stato sospeso dall'operazione dispositivo. È lo Scheduler a dover decidere se far riprendere il processo sospeso o far continuare il processo che era attivo prima dell'interruzione oppure un altro ancora.
- Quando avviene una o 'k' (quindi più) **interruzioni di clock** hardware.

Preemptive e non preemptive

Gli algoritmi di schedulazione possono dividersi in **preemptive** (con prerilascio) o in **nonpreemptive** (senza prerilascio). Un processo gestito con un algoritmo nonpreemptive sceglie un processo da eseguire e lo lascia in esecuzione finché questo non si blocca in un'operazione I/O, in attesa di un altro processo o non rilascia volontariamente la CPU.

Un processo del genere potrebbe andare avanti per ore senza essere interrotto e in caso di interruzioni di clock lo scheduler manderebbe avanti sempre lo stesso processo.

Invece per un processo preemptive un processo può andare avanti fino ad un massimo della quantità prefissata di tempo. Se questo processo rimane attivo allo scadere del tempo viene sospeso e lo scheduler sceglie un altro processo da eseguire se disponibile.

Vi sono vari tipi di algoritmi di schedulazione e sono divisi in base all'ambiente: Sistemi batch, interattivi e real time.

- Per i sistemi batch si possono usare sia algoritmi senza prerilascio sia algoritmi con prerilascio con un periodo lungo per ogni processo.
- Per i sistemi interattivi è importante che vi sia il prerilascio altrimenti un processo potrebbe impossessarsi della CPU per un tempo indeterminato.
- Per i sistemi real time non è sempre necessario poiché i processi i processi fanno il loro lavoro e si bloccano in fretta.

Schedulazione nei Sistemi Batch

Per gestire la schedulazione nei sistemi batch di solito si considerano due metriche per verificare se i sistemi lavorano bene, infatti si analizzano:

- Il **throughput** → ovvero il numero dei job per ora che il sistema completa. Più è alto meglio è!
- Il **turnaround** → Il tempo medio dalla richiesta di un job al completamento

Un sistema che massimizza il throughput non necessariamente minimizza il tempo di turnaround. Se i job sono brevi si avrà un throughput alto ma un turnaround alto.

First-Come First-Served (FCFS)

Primo arrivato, prima servito. È un algoritmo senza prerilascio e funziona che il primo job che arriva è il primo a cui viene assegnata la CPU. Come arrivano i job vengono messi in una coda singola ed è facile da programmare. Il problema è in caso vi siano molti processi I/O bound che usano poco tempo di CPU senza prerilascio impiegherà più tempo.

Shortest Job First (SJF)

Il job più corto è il primo. È un algoritmo senza prerilascio che presuppone la conoscenza anticipata dei tempi di esecuzione.

Shortest Remaining Time Next (SRTN)

Versione con prerilascio della **SJB** è la **Shortest Remaining Time Next**, quindi viene eseguito per primo il processo con il tempo rimasto più breve. Lo schedulatore sceglie sempre il processo il cui tempo di esecuzione residuo è più breve.

Schedulazione nei Sistemi Interattivi

Si cerca in particolare di minimizzare il tempo di risposta che è il tempo tra l'invio del comando e l'arrivo del risultato. Gli utenti non accettano che per richieste percepite come semplici il tempo impiegato sia troppo elevato. Lo schedulatore non può far nulla sul tempo di risposta ma in alcuni casi il ritardo può essere causato da scelte scadenti di ordine di processi.

Non è possibile la schedulazione a tre livelli ma a due (Schedulatore di memoria e schedulatore di CPU).

Schedulazione Round Robin

In **Round Robin** ad ogni processo viene assegnato un determinato **intervallo di tempo** chiamato quanto, nel quale può rimanere in esecuzione. Se allo scadere del **quanto** il processo è ancora in esecuzione, la CPU viene rilasciata e assegnata ad un altro processo. Se il processo ha terminato prima dello scadere del quanto, l'assegnazione della CPU ad un altro processo viene eseguita prima. Allo scadere del quanto il processo viene inserito in fondo alla coda.

Assegnare un quanto troppo breve provoca troppi campi di contesto e peggiora le prestazioni, ma assegnarlo troppo lungo può provocare tempi di risposta lunghi per richieste interattive brevi. La durata di un quanto dovrebbe essere di circa 20-50 millisecondi per evitare questi due problemi.

Schedulazione con priorità

Con **Round Robin** si presuppone che ogni processo abbia la stessa importanza ma in realtà ci potrebbero essere processi più 'urgenti' rispetto ad altri (Per esempio se richiesta I/O necessita CPU gliela si dovrebbe lasciare in modo tale da consentirle di andare avanti nel proprio compito senza rimanere troppo in attesa ed evitare che occupi per troppo tempo la memoria). L'idea è che ad **ogni processo viene assegnata una priorità** e viene concessa l'esecuzione al processo con priorità più alto.

Code multiple

Spesso è conveniente usare la schedulazione a **priorità** fra le classi con schedulatore **Round Robin** all'interno di ciascuna classe. Fino a che esistono processi eseguibili nella classe di priorità 4 manda in esecuzione ciascuno di essi per un quanto con scheduling round Robin e senza preoccuparsi delle classi di priorità più bassa. Se la classe di priorità 4 è vuota, esegui

quelli della classe 3 in round Robin ecc... Se la priorità non vengono aggiornate di tanto in tanto le classi di priorità più bassa potrebbero rimanere in starvation all'infinito. Per risolvere il problema della starvation si utilizza l'aging in modo tale che se un processo rimane troppo tempo in una coda con priorità bassa allo viene spostata in quella più alta.

Altri metodi di scheduling interattivi:

- **Scheduling a quote** → Se vi sono n processi ad ogni processo viene fornito 1/n potenza di CPU. Viene eseguito il rapporto tra tempo di CPU utilizzato e quello di cui ha diritto. Viene mandato in esecuzione il processo che ha il rapporto più basso.

- **Scheduling a lotteria** → Vengono assegnati dei biglietti della lotteria ad ogni processo in base alla priorità. Viene estratto un biglietto a caso e se appartiene ad un processo allora viene eseguito. (Chi possiede più biglietti ha più probabilità di essere eseguito)

Schedulazione nei sistemi Real Time

Nei sistemi real time è importante la **prevedibilità**. Non rispettare una scadenza ogni tanto non è grave! Il tempo gioca un ruolo essenziale, infatti di solito dispositivi esterni generano impulsi ai quali il calcolatore deve reagire entro una certa quantità di tempo.

Hard real time e soft real time

I real time sono classificati in: **hard real time**, quindi che hanno bisogno di rispettare le scadenze, e **soft real time** dove occasionalmente si può non rispettare la scadenza senza troppi danni.

Di solito in questo tipo di schedulazione i processi non durano più di un secondo e il comportamento è prevedibile.

Gli eventi a cui un sistema real time può dover reagire sono detti periodici, quindi che si verificano ad intervalli di tempo regolari e aperiodici, che si verificano in modo imprevedibile. Se ci sono m eventi periodici e l'evento 'i' arriva con un periodo Pi e richiedere Ci secondi di tempo di CPU per essere gestito, il carico può essere gestito solo se:

$$\sum_{i=1}^{m} \frac{Ci}{Pi} \leq 1$$

Solo un sistema real time che rispetta questo criterio è schedulabile.

Deadlock

Risorse

Si possono verificare situazioni di stallo sia su risorse hardware, sia su risorse software, quindi con dispositivi I/O e senza.

Un deadlock può verificarsi quando a d un processo viene garantito l'accesso esclusivo a risorse ovvero dispositivi I/O, file, ecc. Vi sono due tipi di risorse:

- **Prerilasciabili** → è una risorsa che si può togliere al processo che la sta usando senza provocare effetti dannosi.
- **Non prerilasciabili** → Una risorsa che non può essere ceduta ad un altro processo senza provocare il fallimento dell'esecuzione.

Di solito è più probabile avere deadlock con risorse non prerilasciabili. Qualche volta i processi necessitano di due o più risorse quindi le acquisiscono in maniera sequenziale e si ha bisogno di più risorse vengo acquisite una dopo l'altra. Finché è coinvolto un solo processo tutto funziona poiché non c'è competizione per acquisire la risorsa.

Un insieme di processi si trova in una situazione di stallo se ogni processo dell'insieme aspetta un evento che solo un altro processo può provocare. Ogni membro in stallo aspetta una risorsa che può essere rilasciata solo da un altro processo che si trova in stallo. Nessun processo viene eseguito e nessun sbloccato.

Condizioni per il deadlock

Devono valere quattro condizioni perché possa verificarsi una situazione di deadlock:

- Ogni risorsa è assegnata ad un solo processo oppure è disponibile → condizione di **mutua esclusione**
- **Condizione Hold and Wait** → i processi che hanno già richiesto e ottenuto risorse ne possono chiedere altre
- Condizione di **mancanza di prerilascio** → le risorse che sono già state assegnate ad un processo non gli possono essere tolte in modo forzate ma devono essere rilasciate volontariamente
- Condizione di **attesa circolare**→ deve esistere una catena circolare di processi, ognuno dei quali deve aspettare il rilascio di una risorsa da parte del processo che lo segue.

Strategie per trattare i deadlock

- **Ignorare il problema** → Per essere sicuri di risolvere il problema bisognerebbe costringere gli utenti ad aprire un unico file, usare un unico processo. Nessun utente è disposto a tanto, meglio qualche caso di Deadlock occasionale
- **Rilevare e risolvere** → Se si rileva una situazione di deadlock si cerca di risolverlo mediante il prerilascio o il ritorno ad uno stato precedente o l'eliminazione del processo che l'ha causato.
- Cerca di **evitare** situazioni di **stallo** con una politica di allocazione delle risorse
- **Prevenire** le situazioni di **stallo** cercando di negare una delle quattro condizioni

Gestione della memoria

Esistono vari sistemi di gestione della memoria e possono essere divisi in due classi:

Quelli che spostano i processi avanti e indietro dai dischi alla memoria durante l'esecuzione (Swapping o paginazione) e quelli che non lo fanno e hanno partizioni fisse. Lo swapping e la paginazione sono dei sistemi che servono soprattutto per la mancanza della memoria principale che non è in grado di contenere tutti i programmi.

Monoprogrammazione senza swapping o paginazione

Lo schema di gestione della memoria più semplice è quello di eseguire solo un programma alla volta. Non è più utilizzato nei tempi moderni. Era utilizzato nei vecchi personal computer.

Multiprogrammazione con partizioni fisse

La monoprogrammazione non viene più utilizzata, anzi la maggior parte dei sistemi operativi attuali consente a diversi processi di girare contemporaneamente.

Il modo più semplice per realizzare la **multiprogrammazione** è quello di dividere la memoria in n **partizioni**. Il problema delle partizioni fisse è che tutto lo spazio di una partizione non usato da un job va sprecato. Organizzare i job in code separate in caso vi siano job in coda in partizione piccola quando nel frattempo vi è una partizione di memoria grande non utilizzata (poiché i job si mettono in coda della più piccola partizione che può contenerli).

Si potrebbe invece organizzare i job in una singola coda, dove ogni volta che una partizione si libera viene caricato il job più grande che può entrarci. I job piccoli vengono discriminati in questo modo.

Probabilità di n processi di rimanere in attesa e utilizzo di CPU

Usando la multiprogrammazione si può incrementare l'utilizzo della CPU. Se un processo esegue computazioni per il 20% del tempo in cui si trova in memoria, con cinque processi in contemporanea in memoria la CPU potrebbe essere occupata per tutto il tempo (Modello troppo ottimistico, si presuppone che nessuno dei processi non sia in attesa di operazioni di I/O).

La probabilità che un processo spenda una frazione p del tempo in attesa di completamento di operazioni I/O allora con n processi in memoria la probabilità che tutti gli n processi stiano aspettando il completamento di un I/O (quindi CPU inattiva) è p^n.

L'utilizzo della CPU sarebbe quindi → utilizzo CPU = 1 - p^n.

Rilocazione e Protezione

La multiprogrammazione introduce due problemi essenziali:

- **La protezione** → I programmi. agendo su indirizzi assoluti e non relativi, possono accedere a qualsiasi indirizzo di memoria
- **La rilocazione** → Caricare indirizzi di partizioni diverse e riuscire ad accedere agli indirizzi necessari

Per risolvere questi due problemi sono stati introdotti due registri hardware chiamati registro base e limite. Quando un processo viene schedulato, nel registro base viene caricato l'indirizzo di inizio della sua partizione e nel registro limite la lunghezza della partizione. Ad ogni indirizzo di memoria generato viene automaticamente sommato il contenuto del registro base. Gli indirizzi vengono confrontati con il registro limite per assicurarsi che non tentino indirizzi al di fuori della partizione corrente. Il problema è la necessita di dover eseguire una somma ed un confronto ad ogni riferimento in memoria.

Swapping e Memoria Virtuale

Per i sistemi batch è facile organizzare la memoria in partizioni fisse. Ogni volta che un job raggiunge la testa della coda, viene caricato in partizione e rimane in memoria fino a che non termina.

Per i sistemi timesharing (a condivisione di tempo) o PC orientati alla grafica può capitare che non vi sia abbastanza memoria principale per mantenere tutti i processi attivi, infatti questi vengono mantenuti su disco e in caso introdotti in memoria.

Possono essere usati due approcci principali per la gestione di memoria a seconda dell'hardware disponibile:

Swapping → La soluzione più semplice e consiste nel caricare interamente in memoria ogni processo ed eseguirlo per un tot di tempo e poi spostarlo su disco.

- **Memoria virtuale** → Consente ai programmi di girare anche quando sono caricati solo parzialmente in memoria.

Swapping

La differenza principale fra le partizioni di memoria fisse e quelle variabili è che nel secondo caso il numero, la dimensione e le posizioni delle partizioni variano dinamicamente, mentre nel primo caso sono fisse. Quando lo swapping crea molto buchi in memoria è possibile ricompattarli spostando tutti i processi più indietro possibile. Questa operazione è chiamata compattazione della memoria ma non viene eseguita normalmente poiché richiede parecchio tempo di CPU. Il problema delle partizioni che possono crescere è che per esempio se si alloca un Heap di memoria ci si aspetta che cresca e se la partizione è adiacente ad un altro processo bisognerebbe spostarlo in una partizione più grande o scaricare uno o più processi per creare un buco abbastanza grande. Quindi se ci si aspetta che cresca allora è una buona idea allocare un po' di memoria extra quando un processo viene caricato dal disco per ridurre l'overhead dovuto allo scaricamento.

Un modo per poter gestire la memoria è attraverso una lista ordinata per indirizzo che tenga traccia dei segmenti di memoria liberi e occupati. In questo modo si possono usare algoritmi per allocare la memoria ad un processo:

- **First fit** (primo posto sufficiente) → Viene fatta scorrere la lista fino a che il gestore della memoria non trova un buco abbastanza grande.
- **Next fit** (prossimo posto sufficiente) → Lavoro come il First fit ma si ricorda di dove ha trovato il buco adatto e procede la ricerca da quel punto. È poco meno efficiente del first fit
- **Best fit** (miglior posto sufficiente) →Scorre tutta la lista e prende il più piccolo dei buchi che possono essere usati. È meno efficiente del first fit e produce uno spreco di memoria maggiore poiché si creano in memoria buchi piccolo e difficili da riempire.
- **Worst fit** (peggior posto sufficiente) → Per evitare di avere parti di scarto troppo piccole si può adottare il worst fit che va a prendere sempre il buco più grande così da generare scarti grandi in modo da poterli utilizzare in seguito.

Memoria virtuale

Negli anni passati non si potevano caricare tutti i programmi in memoria centrale poiché erano troppo grandi, di conseguenza si adottava l'overlay, ovvero la divisione del programma in parti.

Si trovò una soluzione migliore: la memoria virtuale. Il sistema operativo mantiene in memoria solo le parti che sono in uso in un certo momento, mentre le altre vengono mantenute su disco. La memoria virtuale può anche funzionare con sistemi multiprogrammati con pezzi di molti programmi in memoria allo stesso tempo.

La Paginazione

MMU e pagefault

La maggior parte dei sistemi con memoria virtuale usa una tecnica detta paginazione.

Gli indirizzi possono essere generati usando un sistema d'indicizzazione, i registri base e altri metodi. Questi indirizzi generati dal programma sono detti indirizzi virtuali e formano lo spazio di indirizzamento virtuale. Su calcolatori privi di memoria virtuale l'indirizzo viene spedito direttamente sul bus della memoria. Invece, quando si usa la memoria virtuale, gli indirizzi virtuali non finiscono direttamente sul bus della memoria ma vanno a finire nella **MMU (Memory Management Unit)** che traduce gli indirizzi virtuali in indirizzi fisici.

Lo spazio d'indirizzamento virtuale viene diviso in unità chiamate **pagine**, mentre nella memoria fisica vengono chiamate **pagine fisiche** (page frame). Entrambe queste pagine hanno sempre la stessa dimensione.

Se si cerca di accedere ad una pagina virtuale non presente la MMU se ne accorge e fa in modo che CPU esegua una trap; questo passaggio è detto pagefault. Il sistema ricerca una pagina fisica poco usata e ne scrive il contenuto su disco e successivamente preleva la pagina

appena riferita, la mette nella pagina fisica resa libera, cambia la funzione di traduzione e fa ripartire l'istruzione che era interrotta.

Il numero di pagina viene usato come indice nella tabella delle pagine, che contiene il numero della pagina fisica corrispondente alla pagina virtuale.

Traduzione da indirizzo logico a fisico

NP = IL / DimPagina O = IL % DimPagina

IF = NF + O = DimPag * NF + O

Tabella delle pagine

Nella traduzione l'indirizzo virtuale viene diviso in un numero di pagina virtuale e in un offset. Il numero di pagina virtuale viene usato come indice nella tabella delle pagine per trovare l'elemento relativo a quella pagina virtuale e da questo si ricava il numero di pagina fisica (se esiste) e gli viene attaccato alla parte dell'offset, in modo tale da rimpiazzare il numero di pagina virtuale per formare un indirizzo fisico spendibile alla memoria.

Due problemi principali della tabella delle pagine:

- La tabella delle pagine può essere molto grande
- La traduzione deve essere veloce

Per risolvere il problema della presenza costante in memoria di tabelle delle pagine molto grosse, molti calcolatori usano una tabella delle pagine a più livelli.

Di solito gli elementi nella tabella delle pagine sono costituiti da 32 bit:

- Il numero di pagina fisica → lo scopo della tabella è proprio questo numero
- Il bit presente/assente → se a 0 causa page fault se riferita
- Il bit di protezione → se 0 indica il diritto di lettura/scrittura, se 1 indica il diritto di sola lettura (può essere anche a 3 bit)
- Il bit di modifica → mantengono traccia dell'uso di una pagina. Se si reclama una pagina fisica e la pagina è stata modificata viene messo a 1 ogni volta che si fa un riferimento alla pagina. Se la pagina è stata modificata (sporca) deve essere scritta su disco, altrimenti (pulita) può essere semplicemente abbandonata visto che la copia su disco è ancora valida.
- Il bit usata → viene messo a 1 ogni volta che si fa un riferimento alla pagina, se a 0 la pagina potrebbe essere una candidata ideale per un algoritmo di rimpiazzamento.
- Il bit cache → utile per le pagine che devono essere mappate nei registri invece che in memoria.

Translation Lookaside Buffer (TLB)

A causa della grandezza delle tabelle delle pagine in memoria è utile servirsi di un dispositivo hardware chiamato Translation Lookaside Buffer. Questo funge da cache della tabella delle pagine. Di solito si trova nel MMU e ha una grandezza di massimo 64 elementi. È utile perché

permette di registrare gli indirizzi virtuali più utilizzati senza dover accedere alla tabella delle pagine.

Se viene ricercato un elemento nel TLB e si è trovato (page hit) altrimenti esegue una ricerca normale nella tabella delle pagine (page miss). In seguito scarica uno degli elementi del TLB e lo rimpiazza con l'elemento della tabella delle pagine appena trovato.

Algoritmi di rimpiazzamento delle pagine

Quando si verifica un fault di pagina, il sistema operativo deve scegliere una pagina da rimuovere dalla memoria per fare spazio alla pagina che deve essere caricata. Se la pagina che deve essere rimossa è stata modificata allora prima di rimuoverla deve essere riscritta su disco per aggiornare la sua copia. Se la pagina non è stata modificata allora non è necessaria la riscrittura su disco.

Gli algoritmi di rimpiazzamento sono utili poiché è probabili che si riesca a scegliere una pagina non usata frequentemente. Se si dovesse scegliere una pagina a caso da rimpiazzare si rischierebbe di prelevare una pagina usata di frequente e ridurre le prestazioni.

Algoritmo ottimale di rimpiazzamento delle pagine

È teoricamente l'algoritmo di rimpiazzamento migliore ma non è possibili realizzarlo in pratica poiché è basato sui riferimenti futuri. La pagina scelta come vittima è quella che verrà riferita più lontano nel futuro. Il problema è che il sistema operativo non ha queste informazioni.

Algoritmo Not Recently Used (NRL) di rimpiazzamento delle pagine

Questo algoritmo permette di raccogliere statistiche utili sulle pagine usate e non usate grazie a due bit associati a ciascuna pagina. Il bit R viene messo a 1 ogni volta che la pagina viene riferita(r/w) e M viene messo a 1 quando la pagina viene scritta (cioè modificata). Questi bit devono essere aggiornati ad ogni riferimento in memoria, quindi è essenziale che vengano assegnati dall'hardware e ogni volta che un bit è stato messo a 1, rimane fino a che il sistema operativo lo rimette a 0 via software.

Questo metodo può essere utilizzato per generare classi di pagine in base ai bit R e M. Appena un processo viene lanciato i bit vengono impostati a zero e periodicamente (magari ad ogni interruzione di clock) il bit R viene rimesso a 0 per distinguere le pagine utilizzate di recente.

Classe 0 → non usata, non modificata

Classe 1 → non usata, modificata

Classe 2 → usata, non modificata

Classe 3 → usata, modificata

Viene rimossa una pagina a caso della classe non vuota di numero più basso, è implicito che è meglio rimuovere una pagina modificata cui non si è fatto riferimento per almeno un

periodo del clock invece che una pagina pulita che viene usata frequentemente. Facile da implementare e prestazioni soddisfacenti.

Algoritmo First in First Out (FIFO) di rimpiazzamento delle pagine

Un altro algoritmo è **FIFO**, ovvero primo arrivato, primo ad uscire. Il sistema operativo mantiene una lista di tutte le pagine contemporaneamente in memoria, dove la pagina in testa è la più vecchia e quelli in coda è quella più recente. Al momento della page fault la pagina in testa viene rimossa e la nuova pagina aggiunta in coda. Questo algoritmo non si usa spesso poiché potrebbe rimpiazzare anche pagine importanti o che verranno quasi sicuramente riferite in futuro.

Algoritmo 'della seconda scelta' di rimpiazzamento delle pagine

Una semplice modifica dell'algoritmo FIFO che evita il problema di dover scaricare una pagina molto usata è quello della seconda possibilità dove si controlla il bit R della pagina più vecchia. Se è a 0 la pagina è sia vecchia sia non usata così può essere rimpiazzata. Se il bit R vale 1 viene posto a 0 e la pagina viene spostata alla fine della coda delle pagine, ed il suo tempo di caricamento viene aggiornato come se la pagina fosse appena arrivata in memoria. Il problema è che se tutte le pagine sono state usate allora l'algoritmo degenera e diventa uguale alla FIFO.

Algoritmo 'dell'orologio' di rimpiazzamento delle pagine

Un approccio migliore è quello di mantenere le pagine in una lista circolare a forma di orologio, dove la lancetta punta alla pagina più vecchia. Se il suo bit R è a 0, la pagina viene scaricata, la pagina nuova viene inserita al suo posto e la lancetta scorre in avanti di una posizione, se invece il bit R è a 1 allora viene messo a zero e la lancetta viene spostata in avanti di una posizione.

Algoritmo Least Recently Used (LRU) di rimpiazzamento delle pagine

L'algoritmo LRU, sceglie la pagina usata meno di recente quando si verifica una page fault. L'algoritmo si basa sull'osservazione delle pagine che sono state frequentemente usate nelle ultime istruzioni lo saranno probabilmente anche nelle prossime, quindi buttare la pagina che non è stata usata da più tempo. Il problema è che non è economica da realizzare.

Il modello Working Set

L'insieme delle pagine usate correntemente da un processo viene chiamato **Working Set o insieme di lavoro**. Se tutto l'insieme di lavoro fosse in memoria il processo non causerebbe molti fault di pagina girando ma se la memoria è troppo piccola per poter contenere tutto il Working Set il processo causerà molti fault di pagina e girerà lentamente dato che vi saranno continui rimpiazzamenti. Un programma che genera un fault di pagine dopo l'esecuzione di poche istruzioni è detto in una situazione di trashing.

In un sistema multiprogrammato i processi vengono frequentemente spostati su disco per far sì che altri processi godano del loro turno di accesso alla CPU. Quando un processo deve essere caricano esso provocherà dei fault di pagina fino a che il Working Set non sarà caricato in memoria. Il problema è che avere troppo page fault per caricare un processo non conviene quindi ci si assicura che l'insieme di lavoro sia stato caricato in memoria prima di far riprendere l'esecuzione di un processo e ciò viene chiamato pre-paging.

È noto che la maggior parte dei riferimenti dei programmi tendono a concentrarsi in un piccolo numero di pagine. Ad ogni istante t esiste un insieme costruito da tutte le pagine utilizzate dai k riferimenti in memoria più recenti, questo insieme w (k, t) è l'insieme di lavoro.

Invece di tenere conto dei precedenti riferimenti, si tiene conto dell'insieme delle pagine utilizzate durante i 't' secondi precedenti di tempo di esecuzione. Ogni processo conta solo il suo tempo di esecuzione, quindi se un processo inizia al tempo T e ha avuto 40ms di CPU al tempo reale T+100ms, il suo tempo è 40 ms.

L'ammontare di tempo di CPU che un processo ha effettivamente utilizzato da quando è iniziato è chiamato tempo virtuale corrente. L'insieme di lavoro di un processo è l'insieme delle pagine che ha riferito durante i precedenti τ secondi di tempo virtuale. Un algoritmo di rimpiazzamento delle pagine basate sul working set di solito se trova una pagina che non è nel working set la scarica. Ogni elemento contiene almeno due dati per ogni pagina:

Il tempo approssimato a cui la pagina è stata usate per l'ultima volta e il bit R(riferita).

L'hardware si occupa di impostare i bit R e M. Quando viene esaminato un elemento, viene esaminato anche il bit R: se vale 1, il tempo virtuale corrente viene scritto nel campo tempo di ultimo utilizzo nella tabella delle pagine, il quale indica che la pagina era in uso nel momento in cui è avvenuto il fault di pagina e quindi non è una candidata ad essere rimossa;

mentre se R vale 0 la pagina non è stata riferita durante il tick del clock corrente e può essere una candidata alla rimozione. Per verificare se rimuoverla o meno si confronta il tempo di ultimo utilizzo e τ. Se il tempo è maggiore di τ allora la pagina non è più nell'insieme di lavoro, viene elimina e caricata una nuova pagina. Se tutte le pagine hanno bit R = 1 allora viene scelta una pagina a caso, possibilmente pulita (non modificata).

Algoritmo WSClock di rimpiazzamento delle pagine basato sul Working Set

L'algoritmo base dell'insieme di lavoro non è troppo efficiente poiché ad ogni page fault deve essere analizzata tutta la tabella delle pagine fino a che non viene trovata la pagina candidata.

L'algoritmo **WSClock** è come l'algoritmo dell'orologio ma che utilizza anche le informazioni dell'insieme di lavoro ed è largamente utilizzato per le prestazioni e la semplicità. Ogni elemento dell'orologio contiene il campo relativo al tempo di ultimo utilizzo derivante dall'algoritmo base del Working Set e contiene il bit M e R. Se pagina ha bit R a 1 allora la pagina è stata usata di recente, viene reimpostato a 0 e si scorre alla pagina successiva poiché non è una buona candidata. Se il bit R è a 0 e l'età è maggiore di τ e la pagina non è stata modificata allora significa che non è nel **Working Set**, viene scaricata su disco. Per le operazioni di I/O tutte le pagine potrebbero essere schedulate su disco in un giro di orologio. Per ridurre il traffico sul disco si potrebbe imporre un limite sulla scrittura delle pagine.

Anomalia di Belady

Intuitivamente è facile pensare che più pagine fisiche ha una memoria, meno page fault avrà un programma. Questo non è sempre vero, infatti **Belady** riuscì a dimostrare che l'algoritmo **FIFO** provocava più fault di pagina con 4 pagine fisiche che con 3. Questa strana situazione viene infatti chiamata anomalia di Belady. Gli algoritmi come LRU non possiedono l'anomalia di Belady infatti questi vengono chiamati algoritmi a stack, mentre l'algoritmo FIFO la possiede.

Alcuni algoritmi (come **LRU**) rispettano la seguente proprietà:

M (m, r) ⊆ M (m+1, r)

Problematiche dei sistemi con Paginazione

Overhead totale dovuto alla tabella delle pagine

Lo spazio occupato dalla tabella delle pagine cresce al decrescere della dimensione delle pagine. Questo si può discutere a livello matematico. Sia s byte la dimensione media di un processo e p byte la dimensione delle pagine. Si suppone che ogni elemento della tabella delle pagine richieda e byte. Il numero di pagine richiesto per ogni processo è di circa s/p ed esse occupano s/p * e byte nella tabella delle pagine. La memoria sprecata nell'ultima pagina per via della frammentazione interna è p/2 byte.

$$\boldsymbol{Overhead} = \frac{\boldsymbol{s*e}}{\boldsymbol{p}} + \frac{\boldsymbol{p}}{\boldsymbol{2}}$$

Il primo termine (dim tabella) è grande quando la dimensione delle pagine è piccola, il secondo termine (frammentazione interna) è grande quando la dimensione delle pagine è grande, quindi l'ottimo deve stare in qualche punto intermedio.

La dimensione ottima di una pagina dovrebbe quindi essere: $p = \sqrt{2se}$.

Pagine condivise

Un altro problema dell'implementazione è quello della condivisione. Nei sistemi multiprogrammati è abbastanza comune la situazione in cui più utenti fanno girare lo stesso programma nello stesso momento. Per evitare di avere più copie della stessa pagina allo stesso instante risulta più efficiente condividere la stessa pagina. Il problema è che non tutte le pagine possono essere condivise, per esempio le pagine a sola lettura possono essere condivise mentre quelle di dati non possono essere condivise. In un sistema paginato quello che si fa per condividere dati e testo si fornisce a ciascun processo la propria tabella delle pagine ed entrambe che puntano allo stesso insieme di pagine. Le pagini di date sono in entrambi i processi READ ONLY. Non appena un processo tenta di scrivere sulla pagina viene generata un trap al sistema operativo e viene fatta una copia della pagina in modo tale che ognuno abbia la propria copia privata ed impostata a READ-WRITE. In questo modo solo in caso di necessita si creano copie delle pagine.

Trattamento dei Page Fault

Trattamento di un page fault nel dettaglio:

- L'hardware provoca una trap al kernel, salvando il PC sullo stack.
- Viene fatta partire una procedura per salvare i registri e le informazioni volati, per evitare che vengano alterate dal sistema operativo e questo codice chiama il sistema operativo come se fosse una procedura.
- Il sistema operativo scopre che si è verificato un fault di pagina e cerca di capire qual è la pagina virtuale richiesta.
- Una volta noto l'indirizzo virtuale che ha causato il fault, il sistema operativo controlla che sia un indirizzo valido e che rispetti la protezione sull'accesso (base e limite). Se non è così al processo viene spedito un segnale o viene ucciso, mentre se l'indirizzo è valido il sistema tenta di ottenere una pagina fisica dalla lista delle pagine libere e se non ci sono si applica un algoritmo di rimpiazzamento.
- Se la pagina selezionata è stata modificata, la si schedula per essere trasferita su disco e si ha un cambio di contesto che sospende il processo che aveva generato il page fault e ne manda in esecuzione un altro fino a che il trasferimento su disco non è completo.
- Se la pagina non è stata modificata il sistema operativo cerca l'indirizzo sul disco dove si trova la pagina richiesta e schedula una richiesta al disco per caricarla. Mentre si carica la pagina, il processo che ha generato il fault è sospeso e viene mandano in esecuzione un altro processo.
- Quando l'interruzione dal disco indica che la pagina è arrivata, le tabelle delle pagine vengo aggiornate per riportare la sua posizione e la pagina fisica viene marcata come stato normale.
- Il processo che aveva causato il fault viene schedulato ed il sistema operativo restituisce il controllo alla procedura Assembler che lo aveva chiamato.
- Questa procedura recupera i registri e le altre informazioni e restituisce il controllo allo spazio utente perché l'esecuzione continui come se non fosse accaduto alcun fault di pagina.

Segmentazione

Con la paginazione si ha una dimensione fissa degli spazi d'indirizzamento. In caso in cui si debbano allocare dati e quindi si avrebbe uno spazio di indirizzi in espansione si potrebbero incontrare problemi con la paginazione. Ciò che serve è un modo per liberare il programmatore dalla gestione delle tabelle ed eliminare il problema di dover organizzare il programma in overlay. La soluzione più semplice è quella di dotare la macchina di spazi di indirizzamento completamente indipendenti, chiamati **segmenti**. Questi corrispondono ad una sequenza lineare di indirizzi da 0 a un qualche massimo. Segmenti diversi possono avere lunghezze diverse e possono anche cambiare durante l'esecuzione. Se si modifica l'indirizzo di una procedura non è necessario cambiarla ad altre procedure. La segmentazione, però può generare una frammentazione esterna, ovvero spazi di memoria troppo piccoli per essere utilizzati e può risolvere attraverso una compattazione.

Segmentazione Paginata

Se i segmenti sono grandi può risultare poco conveniente caricarli in memoria nella loro interezza, questo portò all'idea di paginare i segmenti in maniera tale che debbano essere presenti solo le pagine effettivamente necessarie.

UNIX

Intro

in Unix le chiamate di sistema sono disponibili:

1) Come istruzioni aggiuntive alle istruzioni macchina

2) Come funzioni C, la cui interfaccia è definita nello standard POSIX in modo che i programmi sviluppati su uno Unix girino anche su un altro.

Esistono altre librerie di funzioni (insiemi di funzioni predefinite a disposizione dei programmatori) che non sono chiamate di sistema.

- Le chiamate di un sistema in standard POSIX sono offerte da tutti gli Unix, ma non necessariamente da altri S.O., mentre ad es. la printf è offerta da ogni ambiente di programmazione in C (compilatore+collegatore). Inoltre alcune librerie (fra cui proprio quelle per l'I/O) sono realizzate usando le chiamate di sistema.
- Durante l'esecuzione di una c.s. la CPU passa, con un piccolo ma a volte non trascurabile costo, a modalità kernel (nucleo del sistema operativo) nella quale si possono fare cose – tipo accedere ai dati del S.O. - che normalmente i programmi non possono fare.

Chiamate di sistema – file

- creat("paperino", ...) crea il file (anche open lo fa con opportune opzioni)
- read(fd,puntatore,n) legge n bytes dal file aperto identificato da fd, scritti in memoria a partire da puntatore
- write(fd,puntatore,n) analoga
- close(fd) simmetrica di open – la riga fd della tabella diventa "libera"
- lseek(fd,...) sposta l'offset
- dup(fd) e dup2(fd,n) duplicano la riga fd della tabella dei file aperti, nella prima libera o in quella n

Processi e Threads

Processi

Istruzioni con processi:

- fork(); che duplica il processo in esecuzione ("padre" – parent process), creandone una copia ("figlio" – child process) quasi identica:

ha una copia dell'immagine, ha un diverso identificatore, la funzione fork restituisce al processo padre il PID del figlio (un valore $\neq 0$) , il nuovo processo "nasce" uscendo anch'esso dalla chiamata di funzione fork() ottenendo però come risultato 0.

- wait()

Un processo p1 può attendere la terminazione di un processo figlio con: y = wait(&x);

- exec

In effetti, spesso non si fa così, nel ramo del p. figlio si può usare una chiamata di sistema della "famiglia" exec

Threads

- pthread_create che permette di creare, all'interno del processo corrente, un nuovo thread
- pthread_exit che permette ad un thread di terminare
- pthread_join che permette ad un thread di attendere la terminazione di un altro thread

Segnali

Il S.O. "consegna" (delivers) il segnale al processo. Per ogni tipo di segnale c'è un comportamento per default che può essere:

- ignorare il segnale
- terminare (variante: terminare salvando un file "core" che può servire per il debugging)
- fermarsi – poi si può farlo ripartire

System Call dei segnali:

- kill(pid_destinatario, tipo_segnale) → si invia un segnale
- signal
- struct sigaction act,old; → nuova/vecchia azione
- act.sa_handler=f; → oppure SIG_IGN o SIG_DFL
- sigemptyset(&act.sa_mask);
- act.sa_flags = 0;
- sigaction(s,&act,&old)

Pipes

Le pipes (pipe = tubo, tubazione) sono un meccanismo di comunicazione tra processi che si appoggia alle chiamate di sistema per accedere ai files.

Una pipe si apre con la chiamata di sistema pipe:

- int pipefd[2];
- pipe(pipefd);

Una read da una pipe può sospendere il processo chiamante (Q), se tutti i dati scritti nella pipe sono già stati letti (pipe "vuota") Una write su una pipe può sospendere il processo chiamante (P), se i dati scritti e non ancora letti occupano tutto lo spazio allocato (pipe "piena") Possono esserci più "produttori" e "consumatori".

Memoria Condivisa

Il meccanismo di gestione della memoria, qualunque esso sia, garantisce di allocare in RAM l'immagine di più processi in indirizzi non sovrapposti per evitare che uno danneggi l'altro per errore o per "malizia".

- id = shmget(key, size, flags) → un processo può tuttavia chiedere che venga allocato un "segmento" di memoria condivisa, o meglio condivisibile,

- p = shmat(id, 0, 0) → il segmento può poi essere "collegato" allo spazio di indirizzamento di P

Semafori/Condizioni

Semafori

- Un semaforo si dichiara con→ sem_t s;
- Si inizializza con→ sem_init(&s,0,val); - dove "val" è il valore a cui viene inizializzato (il secondo parametro 0 serve ad indicare che il semaforo può essere usato solo dai threads del processo che l'ha creato)
- L'equivalente della funzione "up" → sem_post(&s);
- L'equivalente della "down" → sem_wait(&s);

Su un singolo semaforo dell'array si fanno con semop operazioni identificate da un intero (non vediamo i dettagli):

- -1 : corrisponde a down, se il decremento dovesse rendere il semaforo
- 1 : corrisponde a up

- -2, -3, -4 ... : generalizzazione di down, se il decremento di 2, 3, 4, ... dovesse rendere il semaforo
- 2, 3, 4,... : generalizzazione di up

NB un'operazione con -2 è diversa da due operazioni con -1 (su un semaforo che vale 1).

Mutex

In UNIX i Phtreads si possono sincronizzare con un meccanismo ispirato a quello dei monitor:

- Usando esplicitamente dei mutex per la mutua esclusione (non c'è gestione automatica come per le procedure dei monitor o i metodi synchronized)
- Variabili condizione con operazioni wait, signal e broadcast (segnala una condizione a tutti i thread in attesa su quella)

Per i mutex si hanno a disposizione le funzioni:

- pthread_mutex_lock(&m); → per prendere la mutua esclusione
- pthread_mutex_unlock(&m); → per lasciare la mutua esclusione

Un thread, tipicamente dopo avere conquistato l'accesso esclusivo a variabili condivise con pthread_mutex_lock(&m); può sospendersi su una variabile condizione cond (tipo pthread_cond_t) con:

- pthread_cond_wait(&cond,&m); → Thread può sospendersi su una variabile condizione cond dopo che conquista l'accesso esclusivo a variabili condivise
- pthread_cond_signal(&cond); →"risveglia" uno dei threads bloccati su cond (se non ce ne sono, non ha effetto)
- pthread_cond_broadcast(&cond);→"risveglia" tutti i threads bloccati sulla condizione