Sistemi operativi 1 – 2/3/17

**Spiegare che cosa si intende per *timesharing*, per che scopo lo si è introdotto, illustrare quale meccanismo permette di realizzarlo e come, in generale, in una politica di *scheduling* basata su una forma di priorità si tiene anche conto dell’obiettivo del *timesharing.***

**Timesharing** significa suddividere il tempo di CPU fra tutti i processi, per evitare che un processo monopolizzi la CPU,e con alternanza sufficientemente frequente per garantire tempi di risposta brevi ai processi interattivi. Il meccanismo che permette di realizzarlo è l’interruzione da timer, la cui routine di gestione (almeno ogni k volte) chiama lo scheduler che nella forma più semplice per ottenere il timesharing utilizza una politica Round Robin. Se la politica di scheduling utilizza una forma di priorità, tale politica può tendere a favorire i processi interattivi; in ogni caso, per evitare di dare troppo poco tempo di CPU *(starvation)* ai processi non favoriti dal criterio di priorità, lo si modifica aumentando la priorità diprocessi che non vengono scelti per periodi lunghi.

**Descrivere quali tipi di eventi possono far passare un processo o thread nello stato di “pronto” (*ready*), partendo da quali altri stati; e quali eventi possono farlo uscire da tale stato, indicando qual è lo stato successivo.**

Un processo o thread può passare allo stato di pronto in seguito a principalmente due eventi: L’arrivo di un interrupt handler(timer) che allo scadere del suo tempo rimette nella lista di processi pronti il processo per dare spazio a qualche altro processo in modo tale non monopolizzare la CPU. Il secondo evento è invece il passaggio da stato di waiting a quello “pronto”: quando un processo o thread ha terminato l’I/O oppure è avvenuto un evento (come l’arrivo di un dato in ingresso) che un processo attendeva. Se nessun altro processo è in esecuzione in quell’istante, il processo verrà rimesso in esecuzione altrimenti occorrerà che il processo aspetti finchè la CPU non torna disponibile e non arriva il suo turno.  
Come detto precedentemente, è possibile uscire dallo stato di ready solo quando lo schedulatore preleva il processo/thread dalla “coda” di processi pronti.

**Illustrare le differenze tra la creazione di un processo e la creazione di un *thread,* mettendo in luce a che cosa possono accedere processi diversi, o *thread* diversi, e quali informazioni devono essere mantenute dal sistema operativo per ogni processo e per ogni *thread*.**

Per la creazione di un processo, viene effettuata una fork che duplica il processo in esecuzione creandone una copia quasi identica, con una copia dell’immagine (codice, dati, dati p/so) e ha un diverso ID, la funzione fork restituisce al processo padre il pid del figlio sicuramente != 0 e il nuovo processo nasce ottenendo come risultato della fork 0.  
Il processo mantiene in memoria il codice che può essere condiviso, una copia dei dati come variabili globali, pila e heap e una copia dei dati P-SO ad es tabella file aperti. Il SO mantiene la PCB che serve per avere info su stato processo, PC, SP, allocazione di memoria, stato dei suoi file aperti, info per addebito costi,schedulazione etc, per ogni processo del sistema. La tabella dei processi contiene l’insieme dei PCB di tutti i processi presenti nel sistema. Quando viene effettuato un cambio di processo, lo stato dei registri viene salvato nel PCB e viene attuato il Context Switch. I processi possono inoltre cooperare condividendo risorse software (programma,dati,file aperti) oltre a quelle hardware.

La creazione dei thread può avvenire a livello kernel o livello utente, nel caso del livello utente avremo come vantaggio il tempo di context switch molto basso, può essere implementato al di sopra di qualsiasi SO e si possono attuare politiche di scheduling ad-hoc per applicazione. Lo svantaggio principale è che è molto complesso fare in modo che quando un thread ha bisogno di eseguire una SC bloccante non vengano bloccati anche gli altri thread ( PCB imposta stato==bloccato) se ne deve quindi occupare la libreria di sistema. Inoltre, non potrannò esserci più di un thread running per processo anche se il SO ha più CPU. I thread non sono automaticamente time sharing ma vanno utilizzate chiamate esplicite con cui un thread permette al sistema di gestione di assegnare la CPU ad un altro thread.  
Il vantaggio dei thread a livello kernel: Si possono avere più thread running per processo con sistemi multicpu. La gestione di SC bloccanti non pone alcun problema. Lo svantaggio è il tempo di context switch fra i vari thread di uno stesso processo che richiede passaggio a mod kernel. IL SO deve predisporre strutture dati per gestire tutti i thread. Ogni thread ha un suo stato e i suoi registri CPU, lo spazio di indirizzamento è condiviso e non c’è protezione fra essi.

**Spiegare che cosa si intende per *deadlock* (stallo), quali sono le condizioni necessarie affinché avvenga, e, almeno per una di esse, come si può basarsi su di essa (garantire che …) per prevenire il verificarsi del *deadlock*.**

Il deadlock è una condizione che si verifica quando un insieme di processi è in attesa di un evento che può essere causato solo da un altro processo dell’insieme, quindi nessuno può girare, nessuno può causare eventi e rimangono tutti sospesi. Avviene in particolare nel caso di allocazione di risorse: si sta attendendo il rilascio di alcune risorse, devono valere necessariamente 4 condizioni:

* LE RISORSE SONO IN MUTUA ESCLUSIONE
* LE RISORSE NON SONO PREEMPTIVE: NON HA SENSO PORTARLE VIA AL PROCESSO CHE LE STA USANDO
* UN PROCESSO A CUI SONO ALLOCATE RISORSE NE PUO’ RICHIEDERE ALTRE (HOLD&WAIT)
* SI HA ATTESA CIRCOLARE NEL SENSO GIA’ INDICATO: OGNI PROCESSO ATTENDE UNA RISORSA DETENUTA DA UN ALTRO PROCESSO

Per evitare l’hold and wait si può imporre di chiedere tutte le risorse in anticipo prima di utilizzarne una oppure, se si detengono risorse e ne servono altre, prima si rilasciano quelle detenute. Per evitare l’attesa circolare si può imporre di richiedere le risorse in un ordine prefissato.

**Spiegare che cosa si intende per “attesa attiva” (*busy waiting*) nella sincronizzazione di processi e *threads*, ed indicare se le soluzioni basate sui semafori comportano attesa attiva.**

L’attesa attiva nella sincronizzazione di processi e thread riguarda quello ‘stato’ del processo/thread che rimane in attesa che si verifichi una determinata condizione, quindi testando ripetutamente se quella condizione viene soddisfatta. I semafori non comportano attesa attiva per ilperiodo in cui un processo/thread è sospeso sul semaforo, tempo che può essere più o meno lungo a seconda dei programmi che usano i semafori; possono comportarla (a seconda della realizzazione delle operazioni su di essi, e in particolare su sistemi multiprocessore) per il breve tempo per cui un processo sta eseguendo una operazione su un semaforo (ma non è ancora stato sospeso, in caso di *down)*: in tale periodo, un altro processo che tenta di eseguire una operazione sul semaforo farà attesaattiva prima di poter iniziare l’operazione.

**Spiegare perché, nella sincronizzazione basata su variabili condizione, fra le soluzioni specifiche che forniscono tali variabili e le operazioni su di esse (una delle quali è qui indicata genericamente come wait(cond) senza riferirsi ad una soluzione specifica), alcune fanno sì che tipicamente nel codice l’attesa di una condizione vada scritta:**

**if (test) wait(cond)**

**mentre altre prevedono che tipicamente vada scritta:**

**while (test) wait(cond)**

1. posto che (!test) è la condizione booleana che il processo/thread richiede valga per procedere, il frammento di programma con if è adeguato se:

* 1. la signal viene effettuata solo se test è falso (non, più in generale, se *potrebbe* essere falso); la scrittura del codice che usa signal deve quindi tener conto di come sono scritte le parti che usano wait
  2. dopo la segnalazione, il processo/thread che la riceve è il primo a girare in mutua esclusione, (o più in generale, quando gira, il test sulle variabili condivise è ancora falso; questo tipicamente richiede che la segnalazione non sia una broadcast, altrimenti è verosimile che tra più processi risvegliati, chi non è il primo a girare possa non avere più test falso)

Il punto (1) e la seconda parte del (2) riguardano come vanno scritti i programmi, la prima parte del

1. dipende invece da che cosa effettivamente garantisce una realizzazione delle operazioni sulle variabili condizione. Le realizzazioni nei *monitor* lo garantivano sospendendo il processo che effettua la segnalazione, o prevedendo che la segnalazione comportasse l’uscita dall’operazione sulle variabili condivise e quindi il rilascio della mutua esclusione.

Le operazioni fornite per i Pthreads e in Java non prevedono alcuna garanzia sull’ordine della mutua esclusione, è quindi opportuno il while , di conseguenza la scrittura dei programmi non richiede

(1).

**Descrivere la traduzione degli indirizzi nel caso della paginazione, indicando anche il supporto fornito dall’hardware per tale traduzione.**

La traduzione degli indirizzi logici in indirizzi fisici nel caso della paginazione avviene nel seguente modo:

La cpu invia l’indirizzo virtuale alla MMU. L’hardware controlla dapprima se il relativo numero di pagina virtuale è presente nel TLB confrontandolo simultaneamente con tutti gli elementi. Se si trova un elemento con lo stesso numero di pagina virtuale ( page hit) e non si ha violazione dei bit di protezione, il numero della pagina fisica viene direttamente preso dal TLB senza bisogno di dover accedere alla tabella delle pagine. Se il numero di pagina virtuale risulta presente nel TLB ma l’istruzione sta tentando di scrivere in una pagina a sola lettura, viene generato un errore di protezione, in maniere del tutto analoga a quanto verrebbe fatto in seguito a un accesso alla tabella della pagine. Il caso interessante si ha quando il numero di pagina virtuale non è presente nel TLB. La MMU si accorge della mancanza per via di un page miss ed esegue una normale ricerca sulla tabella delle pagine. In seguito scarica uno degli elementi del TLB e lo rimpiazza con l’elemento della tabella delle pagine appena trovato. Così se a quella pagina si farà riferimento nell’immediato futuro si avrà un page hit. Quando si cancella un elemento dal TLB il bit di modifica viene copiato nella tabella delle pagine in memoria mentre gli altri valori si trovano già nell’elemento che sta in memoria. Quando si carica il TLB dalla tabella delle pagine, tutti i campi vengono presi dalla memoria.