PROCESSI E TRHEADS

Fork() && Wait()

Un OS deve poter gestire correttamente i processi e più dettagliatamente la loro esecuzione contemporanea, le politiche d'allocazione di risorse, creazione e comunicazione inter-processo.

Un processo è composto dal codice del programma, un insieme di dati, e una serie di attributi che ne specificano l'evoluzione durante l'esecuzione: l'ID, lo stato, la priorità, il program counter, i puntatori di memoria, ecc.

Essi sono contenuti all'interno del Process Control Block (PCB), creato e gestito dall'OS.

Un processo che avvia la syscall fork() crea un processo "figlio", eseguito in concorrenza, che risulta copia del processo che lo crea.

E' possibile, in sequenza, che sia il padre che il figlio effettuino altre fork, il risultato è un albero di processi.

Un processo può decidere se aspettare la terminazione di un figlio utilizzando la syscall wait(), rimanendo così bloccato e infine raccogliere lo stato di terminazione del figlio.

All'avvio di un PC, o al riavvio, parte il bootstrap program il quale ha il compito di inizializzare i registri della CPU, fa un checkup della memoria e dei dispositivi, carica l'OS nella memoria e lo avvia. Una volta avviato l'OS, esso avvia il primo processo, chiamato "init", e aspetta di gestire qualche evento causato da interrupt o trap.

La fork() ritorna -1 se c'è stato un errore, 0 nel figlio, l'ID del figlio nel padre.

Il processo figlio eredita dal padre una COPIA identica di memoria, i registri della CPU e tutti i descrittori dei file aperti dal padre. Da notare che il PCB del figlio è una copia del PCB del padre al momento della syscall fork().

In uscita un figlio dovrebbe chiamare la syscall exit(), chiude i file descriptors, dealloca la memoria, il figlio muore se il processo padre è morto oppure se il padre chiama la wait la exit ritorna il valore, se il padre è vivo e non chiama la wait il figlio diventa zombie.

La wait() ritorna il process ID del figlio appena terminato e può salvare il suo valore di ritorno nell'area di memoria puntata dal puntatore che prende in ingresso, o -1 se il figlio non esiste o è già terminato. Oppure con waitpid posso attendere uno specifico figlio.

Risorse ed esecuzione dei threads

Un processo ha due caratteristiche fondamentali: è proprietario di uno spazio virtuale di memoria dove montare la propria immagine ed è trattato indipendentemente dall'OS, ossia potrebbe eseguire il proprio task in maniera concorrente con altri processi.

Il proprietario delle risorse è il process o task, chi invia messaggi o computa dati è il thread.

Il multithreading è l'abilità di un OS di gestire più percorsi concorrenti di esecuzione all'interno di un singolo processo.

Un processo in sistemi multithread ha il compito di proteggere la propria immagine da altri processi, dalla CPU, files, ecc. Mentre i thread, che attenzione condividono tutti l'area di memoria del processo, svolgono un proprio task indipendente dagli altri thread. Essi hanno un proprio stato (contenuto nel TCB) , una stack di esecuzione, e anche una piccola area di memoria dove conservare le proprie variabili locali.

I benefici dei thread sono: il minor tempo di creazione rispetto a un processo, anche lo switching richiede meno tempo, condividendo la stessa area di memoria non richiedono l'intervento del kernel per comunicare.

Siccome l'OS gestisce i thread a livello dei processi, se il processo che contiene thread in questione venisse sospeso o chiuso, quest'azione si proietterebbe anche su tutti i thread.

I thread avendo anch'essi uno stato (spawn, block, unblock, finish) possono essere sincronizzati.

Abbiamo due tipi di thread User Level Thread (ULT) oppure i Kerner Level Thread(KLT), anche chiamati processi leggeri. I primi vengono totalmente gestiti dall'applicazione, il kernel gestisce solo il processo che li crea. Dello scheduling dei secondi invece se ne occupa direttamente il kernel, windows è un esempio di questo approccio.

I vantaggi dei primi sono che lo switch tra i ULTs avviene senza il passaggio alla kernel mode e funzionano su qualsiasi OS poiché implementati attraverso una libreria a livello utente. Lo svantaggio è che una chiamata da parte di un thread bloccherebbe tutti gli altri contemporaneamente. Perderemmo vantaggi delle architetture multicores.

I vantaggi dei secondi sono che vari KLT possono essere schedulati su core differenti, se un solo thread effettua una chiamata bloccante non ci sono effetti sugli altri thread, gli stessi task del kernel possono essere multithread. Lo svantaggio è che lo switch tra thread richiede l'intervento del kernel.

Potremmo anche avere un approccio combinato M:N.

Multiprocessing simmetrico

Tradizionalmente il pc è visto come una macchina che esegue una istruzione per volta, in modo sequenziale.

-Single Instruction Single Data (SISD): un singolo processore esegue un solo flusso di istruzioni che operano su dati memorizzati in una singola memoria.

-Single Instruction Multiple Data (SIMD): ogni istruzione è eseguita su un insieme diverso di dati da parte di diversi processori.

-Multiple Instruction Single Data (MISD): Un insieme di dati viene trasmessa a più processori che eseguono una sequenza differente di istruzioni.

-Multiple Instruction Multiple Data (MIMD): Una serie di processori eseguono simultaneamente diverse sequenze di istruzioni su differenti insiemi di dati.

Gli ultimi due sono esempi di Parallel Processor. Dei MIMD differenziamo quelli a memoria condivisa, di cui fanno parte i Symmetric Multiprocessors (SMP), e quelli a memoria distribuita, di cui fanno parte i Clusters.

I principali problemi di progettazioni degli OS multiprocessor sono: la concorrenza simultanea dei processi e dei thread, lo scheduling, la sincronizzazione, la gestione di memoria, gestione dei Fault.

PThread

Pthreads sono definiti in <pthread.h>. I thread devono svolgere task completamente indipendenti oppure devono essere sincronizzati tra loro in riferimento agli accessi alla memoria condivisa.

Per creare un thread si usa la syscall pthread_create(), una volta creati i thread sono pari e il loro massimo numero dipende dall'OS. Per terminare un thread posso o inserire un return statement, o chiamare un pthread_exit(), o una exit() (ma chiuderebbe tutto il programma), oppure il main termina senza eseguire pthread_create(). Attenzione pthread_exit() non chiude i file aperti dai thread.