Sistemi Operativi

Appunti delle Lezioni di Sistemi Operativi

Anno Accademico: 2024/25

Giacomo Sturm

Dipartimento di Ingegneria Civile, Informatica e delle Tecnologie Aeronautiche
Università degli Studi "Roma Tre"

Sorgente del file LaTeX disponibile al seguente link:
https://github.com/00Darxk/Sistemi-Operativi

Indice

1 Introduzione a Docker

2 Introduzione alla Linea di Comando

3 Sistemi Operativi

3.1 Processi

Il kernel ha a disposizioni l'intero instruction set del processore.

I programmi eseguiti non hanno la possibilità di eseguire tutti le istruzioni fornite dalla CPU. Non possono eseguire istruzioni che accedono a risorse condivise, come la memoria, o il disco. Per poter accedere a queste istruzioni, i processi chiedono al kernel del sistema operativo, di effettuare queste operazioni per loro conto, poiché il kernel è in grado di effettuare queste operazioni. Il sistema operativo evita quindi conflitti e risolve questioni di sicurezza. Questo viene implementato tramite delle chiamate di sistema, essenzialmente un API con cui i processi possono interagire. Una chiamata di sistema blocca l'esecuzione del processo corrente, e attiva l'esecuzione del sistema operativo per eseguire effettivamente la chiamata di sistema. Si effettua un salto dal codice del programma e quello del sistema operativo. Inoltre si ha un'elevazione dei privilegi in modo da poter accedere a tutte le istruzioni del processore. Dopo aver effettuato la chiamata di sistema, si effettua un salto al processo in user space, e quindi diminuire il livello di privilegio nell'uso del processore. Per ritornare si effettua un salto all'istruzione immediatamente successiva alla chiamata di sistema.

Non si può effettuare un salto ad una qualsiasi istruzione, solamente alla prima istruzione della system call che si vuole invocare. Per questo viene inizializzata una tabella a tempo di avvio dove sono presenti tutti gli indirizzi di partenza per le system call disponibili ai programmi utenti. Rappresentano l'unica interfaccia tra i processi nello spazio utente al sistema operativo. Questa tabella viene chiamata "trap table". A livello di hardware viene impostato su un registro l'indirizzo del gestore delle chiamate di sistema. All'avvio di un programma viene creata un'entità processo da parte del sistema operativo, per allocare la memoria, ad una struttura dati del processo di tipo "proc t". Il codice del programma viene poi trasferito in memoria centrale, dal disco, e può essere svolto in maniera lazy, ovvero trasferendo solamente il codice strettamente necessario, ed successivamente si trasferisce solo il codice che viene utilizzato. Inoltre si realizza lo stack e l'heap dell'applicazione, e si realizza il doppio puntatore a caratteri argv, per gestire gli input per il processo. Si inserisce nello stack del kernel il registro ed il PC per tenere conto del processo in esecuzione, e tiene conto delle chiamate di sistema e dei loro argomenti per effettuare system call.

Alla fine di questo operazioni bisogna passare il controllo al programma appena avviato, utilizzando una chiamata "return from trap", e bisogna specificare la prima istruzione da eseguire. L'hardware conosce l'indirizzo di memoria base dello stack che contiene il valore dei registri da inizializzare per avviare il processo. Dopo aver popolato i registri ed il PC, cambia la modalità di funzionamento allo spazio utente e salta alla prima istruzione nel metodo main.

Ora il programma in esecuzione segue le istruzioni sequenzialmente, ed eventualmente deve eseguire delle chiamate di sistema, quindi invocando una funzione di trap per saltare al livello spazio kernel, ma bisogna salvare i valori del processo nello spazio riservato nel kernel. Questo salvataggio viene effettuato in modo completamente autonomo da parte dell'hardware prima di cedere il controllo al kernel. Dopo averli salvati si eleva il livello di privilegi e si salta alla prima istruzione del gestore della trap, eseguito dal punto di vista del kernel, si eseguono le istruzioni della chiamata di sistema, e si effettua una istruzione return from trap. Il processore rimuove i valori

salvati nello stack del kernel e si sposta allo spazio utente. Inoltre salta all'istruzione indicata dal PC dopo la chiamata trap.

Alla terminazione del processo non bisogna salvare i suoi valori poiché non avrà modo di effettuare altre chiamate di sistema, quindi l'hardware non svolge nessuna operazione. Il kernel quindi svuota svuota l'area di memoria assegnata al processo e lo rimuove dalla struttura dati utilizzata per contenere la lista dei processi.

Il sistema operativo ha come obiettivo la ricerca di prestazioni più alte possibili, bisogna quindi evitare di inserire livelli in più tra il processo al processore, per impedire rallentamenti nell'esecuzione del processo. Ma allo stesso tempo si vuole mantenere il controllo del processore da parte del sistema operativo. Data la modalità descritta precedentemente il processo restituisce il controllo al sistema operativo dopo una chiamata di sistema, quindi il processore mentre l'esecuzione del processo non sta eseguendo il sistema operativo. Questo rappresenta quindi un problema, poiché un processo malevolo potrebbe non effettuare mai chiamate di sistema, quindi non restituisce mai il controllo al sistema operativo e viene bloccata l'esecuzione del sistema operativo. Questo può essere realizzato facilmente tramite un ciclo infinito. Non necessariamente però dipende da una programmazione malevola, potrebbe essere causata ad un errore o bug nel programma.

Quindi è necessario cambiare l'approccio per riguadagnare il controllo del processore dal sistema operativo. Si definiscono quindi due approcci, uno cooperativo, l'altro non cooperativo. Il primo approccio consiste nella modalità precedente, dove il sistema operativo aspetta una chiamata di sistema per riprendere il controllo del processore, oppure se viene sollevata una qualche tipo di eccezione. Esiste una chiamata di sistema `yield()` che permette di cedere il controllo del processore al sistema operativo. Ma non è garantito che ogni processo contenga una chiamata di sistema, e che vengano chiamate periodicamente.

Si vorrebbe utilizzare un sistema dove il controllo viene ceduto al sistema operativo periodicamente, questo tuttavia non può essere realizzato interamente tramite software. Bisogna quindi ricorrere al supporto dell'hardware, generando un timer all'avvio di ogni processo. Quando questo timer arriva al termine viene invocato un interrupt che passa il controllo al sistema operativo, da parte di un gestore di interrupt. Questo rappresenta uno scheduling non cooperativo. I processi che vengono eseguiti non gestiscono il controllo del processore. In questo modo è possibile al sistema operativo riprendere il controllo nonostante la presenza di applicazioni malevoli. Questo timer lavora nell'ordine dei millisecondi, ed è indipendente dalla frequenza di aggiornamento del processore.

Un processo potrebbe non terminare tra due di questi timer interrupt, quindi il controllo passa al sistema operativo alla terminazione del programma, ed in alcuni casi viene disattivato il gestore degli interrupt e si scartano alcuni interrupt. Se dei processi hanno una priorità diversa, invece di terminare la loro esecuzione al primo interrupt, il controllo viene lasciato al processo invece del sistema operativo, fino ad un certo numero di interrupt.

Viene chiamata la funzione scheduling ad ogni interrupt per scegliere se si continua ad eseguire il processo che era in esecuzione oppure eseguire un nuovo processo Quando viene scelto di cambiare il processo da eseguire, si effettua un context switch, un cambio di contesto, poiché tutti i valori associati al processore e dei registri vengono salvati per usarne di nuovi.

Con questa modalità, all'avvio si inizializza la trap table, ed il gestore degli interrupt decide l'intervallo tra gli interrupt. Quando viene sollevato il timer interrupt all'esecuzione di un processo,

l'hardware effettua operazioni analoghe alla chiamata di sistema. I suoi valori e registri vengono salvati nel suo stack di processo, ed il processore cambia la modalità in kernel mode e passa il controllo al sistema operativo. Il sistema operativo sceglie dal suo scheduler a quale processo restituire il controllo del processore. Scelto il processo, l'hardware riassegna i valori ed i registri e salta allo spazio utente.

Se viene sollevato un interrupt mentre il processo sta scrivendo sul dati sul disco, oppure durante l'esecuzione di una chiamata di sistema, il sistema operativo può reagire in modi diversi. Potrebbe scartare gli interrupt sollevati durante una chiamata di sistema, oppure potrebbe usare sofisticati meccanismi di lock per proteggere l'accesso a strutture dati interne. Questi meccanismi saranno analizzati dettagliatamente in una successiva sezione del corso.

3.2 Scheduling

Dopo aver trattato il meccanismo di context-switch bisogna analizzare le politiche di scheduling. Verranno utilizzate una serie di assunzione irrealistiche. La prima di queste assunzione è che tutti i "job", lavori che il sistema deve eseguire, abbaino lo stesso tempo di esecuzione. Si assume che tutti i lavori arrivano allo stesso momento. Si considera che ciascuno di essi venga eseguito fino al suo completamento. Ognuno di questi utilizza esclusivamente il processore, non effettuano operazioni di input/output. Si suppone infine che si conosca a priori il tempo di esecuzione di tutti questi job.

Si vuole misurare in modo quantitativo le politiche di scheduling, si introducono quindi una serie di metriche. La prima chiamata turnaround time, è il tempo di lavoro di un job, si calcola come la differenza tra il tempo di arrivo ed il tempo di completamento, data l'assunzione che tutti questi job arrivano allo stesso momento, si impone sia nullo. Quindi con queste assunzioni il tempo di turnaround corrisponde al tempo ci completamento. La seconda metrica è la fairness, quanto le risorse vengono divise in modo equo rispetto a tutti i processi.

La prima politica di scheduling è la FIFO, o FCFS, "First Come, First Served", semplice e di facile implementazione. Si suppone arrivino sequenzialmente a distanza temporale minima, il primo job ad arrivare sarà il primo ad essere eseguito.

Nel momento in cui si considerano situazioni più realistiche, questa politica perde di efficienza, poiché il primo job potrebbe richiedere un tempo molto superiore ai successivi, quindi si verifica l'effetto convoglio. Il primo "vagone" è molto più grosso dei seguenti e quindi il turnaround time è molto elevato.

Nello stesso contesto si può utilizzare una gestione diversa, per migliorare il tempo di ritorno. Si considera allora una politica che esegue i lavori più corti prima, SJF "Shortest Job First", noto il tempo di esecuzione. In questo modo si minimizza il turnaround time, decisamente più basso della politica FIFO.

In uno scenario più realistico, non tutti i job arrivano nello stesso momento nel sistema, quindi è possibile che il primo lavoro che arriva sia l'unico possibile da eseguire, e sia anche il più lungo, quindi in queste condizioni si verifica nuovamente l'effetto convoglio.

Per cui si considera un sistema dove se arriva un job che richiede un tempo minore per essere completato, viene interrotta l'esecuzione del processo attuale e viene eseguito questo nuovo processo. Questa politica si chiama STCF "Shortest Time to Completion First", aggiunge un meccanismo di prevenzione a SJF, quindi viene chiamato anche PSJF, "Preemptive Shortest Job FIrst". Ogni

volta che un job entra nel sistema ne calcola il tempo di completamento e se è minore del processo attualmente in esecuzione allora si interrompe l'esecuzione del processo corrente e si esegue il nuovo processo. Quindi all'arrivo di un nuovo processo, il sistema operativo deve determinare il tempo di completamento per questi nuovi processi e per il processo attualmente in esecuzione.

Esistono altre metriche altrettanto significative come il tempo di risposta, definito come il tempo in cui il job è stato in coda, prima di essere eseguito per la prima volta. Un modo per minimizzare o controllare il tempo di risposta per un determinato job si considera la politica di scheduling "Round Robin" (RR). Questa politica effettua una suddivisione nel tempo cambiando periodicamente quale dei processi in coda viene eseguito, fino all'esaurimento dei job in coda. In questo modo viene garantito a tutti i job un quanto di tempo, essenzialmente l'intervallo di tempo tra due interrupt. Ma è possibile che questo quanto di tempo sia un multiplo del timer interrupt, in modo per sintonizzare gli interrupt e questi quanti di tempo. Secondo la definizione di fairness, questo è una politica fair poiché assegna ad ogni jon la stessa quantità di CPU. Ma ottiene delle prestazioni pessime rispetto al tempo di ritorno. Infatti per minimizzare il tempo di ritorno il processore deve concentrarsi su un unico job alla volta.

La lunghezza del quanto di tempo è critico per il tempo di risposta. Diminuendo il quanto di tempo, l'overhead rappresentato dal context switch rappresenta una parte considerevole del quanto di tempo, ma diminuisce il tempo di risposta. Se aumenta il quanto di tempo, viene ammortizzato l'effetto del context switch, il tempo di risposta aumenta. Questa politica è tra le peggiori per il tempo di ritorno, ma è la più equa nella sua distribuzione delle risorse.

Si considerano ora job in grado di eseguire operazioni input/output o accessi a risorse esterne al processore, quindi possono passare dallo stato running allo stato blocked.