Processi e Thread

Il concetto fondamentale di goni Sistema Operativo è il **processo**: l'astrazione di un programma in esecuzione.

Introduzione ai processi

Come detto un processo è semplicemente un programma in esecuzione.
I processi consento al Sistema Operativo di:

- **Allocare risorse**
- **Limitare risorse**
- **Contabilizzare le risorse**

Quando un processo è terminato viene chiamata una funzione neceessaria a sostituire i processi in memoria, il **context switch**, la quale è un operazione molto complessa quindi è meglio evitarla.

Un compito del Sistema Operativo è quello di mantenre informazioni sulle risorse e sullo stato di ogni processo, per farlo ad ogni processo è associata una tabella in memoria con tutte le informazioni

Modelli di processo

Tutto il software disponibile del computer è organizzato in un certo numero di **processi sequenziali**, ognuno dei quali ha la sua CPU virtuale.
In realtà ogni CPU passa avanti e indietro da processo a processo in maniera così veloce che all'occhio umano sembra in parallelo, questo avanti e indietro è appunto chiamato **multiprogrammazione.

> ⚠ **Viene sempre eseguito solo un processo alla volta, anche nella macchine multicore.**

Quindi sarebbe più coretto dire che i processi vengono eseguiti in maniera **pseudoparallela**, poiché viene eseguito un solo processo alla volta, ma il quanto di tempo e il tempo che ci mette il context-switch è cosi minuscolo che è impercettibile all'uomo.

Non tutti i processi sono uguali infatti esistono processi:

- **Con priorità**

- **Senza priorità**

Inoltre i processi sono indipendenti e separati, ognuno ha il suo spazio di memoria non accedibile da altri processi, ma in qualche modo li dobbiamo far comunicare tra loro.

> **Context Switch**
>
> È il meccanismo attraverso il quale il sistema operativo salva lo stato di un processo e carica lo stato di un altro. Anche se è un'operazione complessa, è fondamentale per la multiprogrammazione.

Creazione del processo

I processi sono solitamente creati da:

- inizializzazione del sistema(processo init);
- Esecuzione di chiamata di processo;
- Richiesta da parte di utenti;
- Avvio di un lavoro batch;

I processi terminano quando:

- Uscita normale(volontaria);
- A causa di un errore(volontaria);
- Errore fatale(involontari -- ex: segmentatio fault(acesso ad area di memoria non data));
- Ucciso da un altro processo(involontaria);

> **Volontario e Involtorario**
>
> Quando diciamo che è una terminazione volontaria intendiamo che è un qualcosa di gestibile o deciso dal processo.

Possiamo gestire i processi tramite:

- **fork**: Ci permette di creare un processo clone, quindi di creare una gerarchia padre-figlio, nella quale il figlio è una copia del padre, il figlio può condividere alcune risorse con il padre, ma hanno delle differenze come il PID. La clonazione ha valore di sicurezza perché i due processi non condividono variabili e quindi non si possono incasinare a vicenda. Cosa che invece non è disponibile nei thread.
- **exec**: Esegue un nuovo processo, utilizza fork dentro.

- **exit**: Causa la terminazione volontaria del processo.
- **kill**: Invia un segnale al processo.

> **Nota sui processi figli e padri:**
>
> Tutti i processi figli e i genitori costituiscono un gruppo, quando mandiamo un segnale a tutti i membri del gruppo ognuno dovrà individualmente intercettare il segnale, ignorarlo o eseguire la funzione di default.

Gli stati di un processo

Figura 2.2

Un processo può essere in stato di esecuzione, bloccato o pronto. Sono illustrate le transizioni fra questi stati.

Ogni processo è un' entità indipendente, ognuna delle quali èuò avere uno stato doverso, sono tre i possibili stati:

- **Esecuzione**: Il processo è dentro al processore è viene eseguito;
- **Pronto**: puo essere eseguito, ma temporaneamente sospeso per consentire aun altro processo di essere eseguito;
- **Bloccato**: Incapace di essere eseguito perché manca una qualche risorsa esterna.

Il programma che ha il compito di scegliere come devono essere eseguiti i processi si chiama **Scheduler**.

Ogni processo come già detto deve mantenere delle informazioni come:

- PID,UID,GID
- Spazio di indirizzi di memoria
- Registri hardware
- File aperti
- Segnali

- Interrupt

> **Dove si trovano queste informazioni?**
>
> Sono informazioni all'interno di una struttura all'interno dela memoria ed ogni processo alla sua.

Per comunicare, due processi devono usare:

- **Interrupt**: Origini da componenti hardware, gestito tramite routine di servizio di interrupt (ISR), serve per far comunicare hardware e software, si verifiicano in manirera asincrona. Hanno proprità su tutti i processi, infatti quando un interrupt viene chiamato si blocca il processo in esecuzione per far spazio all'interrupt. Sono gestiti dal SO che chiama lo scheduler (unica entità che permette di cambiare stato ai processi e di fare il context switch).
- **Segnali**: Eventi di natura software generati da processo, gestione personalizzata o comportamento predefinito, sono quindi utili per gestire casi eccezionali ==(involontari) e ==generalmente sono asincroni (possono arrivare in qualsiasi momento).

Possiamo dire che il dizionario con cui i processi comunicano è quello dato dai segnali.

> **Note di poca importanza:**
>
> In alcuni casi l'interrupt possono essere ignorati
> il muose è un interrupt continuo.

La gestione di un processo può essere:

- **Asincrona**: Può essere fatto nel tempo.
- **Sincrono**: eseguito nel momento in cui arriva.

> **Interrupt vector:**
>
> E' una tabella che memorizza i vari interrupt a tutti i dspositivi di I/O.

Implementazione dei processi

L'implementazione di processi che danno l'illusione del parallelismo si basa su otto passaggi.

1. L'ahrdware mette nello stack le informazioni relative al processo.

2. L'hardware carica il nuovo contatore di programma dal vettore di interrupt.
3. La procedure in linguaggio assembly salva i registri.
4. La procedura in linguaggio assembly imposta il nuovo stack.
5. Eseguita la procedura di servizio in C.
6. Lo scheduler decide quale processo eseguire come successivo.
7. La procedura in C ritorna il codice in linguaggio assembly.
8. La procedura in linguaggio assembly avvia il nuovo processo corrente.

Il controllo deve essere sempre e solo del SO, quindi un processo non può cedere la CPU a un altro processo (contexct switch) senza passare per lo scheduler, che ottiene il controllo ad ogni interruzione.

Gestione dei segnali

Esistono diversi tipi di segnali:

- **Indotti dall'hardware**
- **Indotti dal software**

Azioni possibili:

- Term
- Ign
- Core
- Stop
- Cont

il **cathing dei segnali** avviene nel seguente modo:

- Il processo registra il gestore del segnale.
- Il Sistema Operativo invia il segnale e consente al processo di eseguire l'handler.
- Il contesto d'esecuzione corrente deve essere selavato/ripristinato.

Mentre invece la gestione:

- Il kernel invia un segnale;
- Interrompe il codice in esecuzione;
- Salva il contesto;
- Eseguie il codice di gestione del segnale;
- Ripristina il contesto originale;

Thread

Fino ad adesso abbiamo parlato di processi che dispongono di uno spazio degli indirizzi e di un singolo thread di controllo.
Ma esistono dei modelli in cui un processo può eseguire diversi thread.

> **Perché consentire piu thread per processo?**
>
> - **Lightweight processes:** Consente un parallelismo più efficiente
> - Comunicazione e sincronizzazione più semplice siccome si trovano nello stesso spazio.

Ogni processo ha n thread e ciò consente un parallelismo efficienti per spazio e tempo.
Inoltre mi permettono di svolgere più operazioni senza dovermi preouccupare del cambio di contesto.

Thread e processi

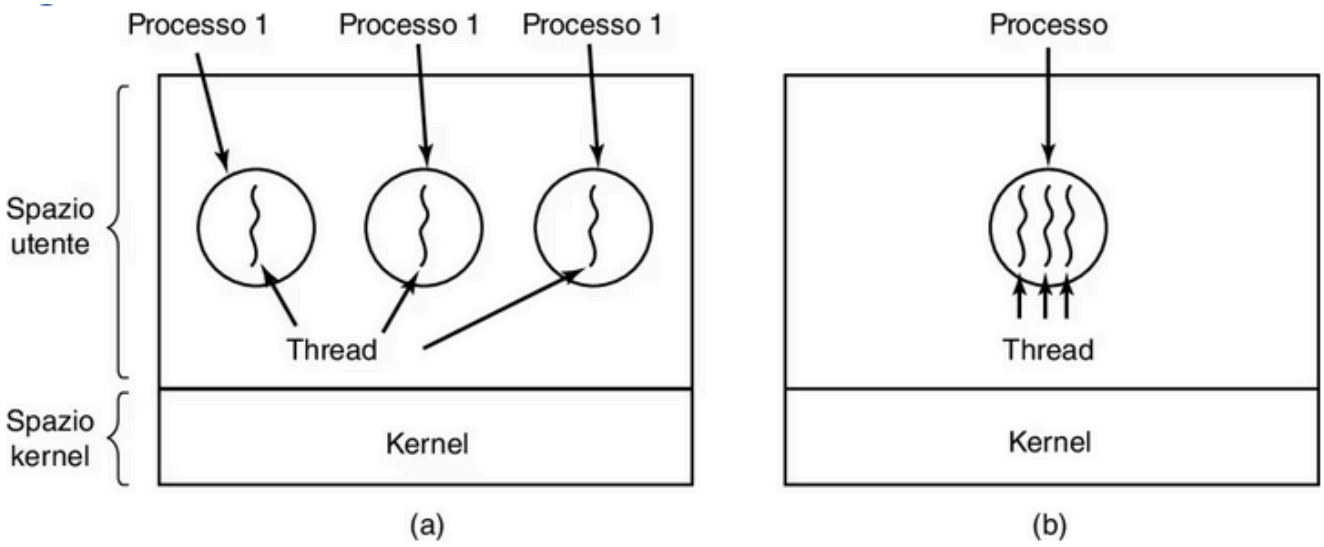
Un processo può essere visto come un modo comodo per raggruppare risorse correlate con le quali possiamo eseguire un programma.

Ogni thread ha:

- **Uno Stack**
- **Dei Registri Hardware**
- **Srato indipendente dal processo**

Semplicemente un thread è una parte del processo che stiamo eseguendo, un solo processo può contenere diversi thread che lavorano in parallelo tramite la tecnica della **multiprogrammazione**, inoltre condividendo lo stesso spazio di memoria rendono più efficiente l'esecuzione dei processi, poiché non si deve effettuare nessun cambio di contesto.

Invece di avere tanti processi con un solo thread ciascuno abbiamo un solo processo con tanti thread.



(a) Tre processi con un thread ciascuno. (b) Un processo con tre thread.

Caratteristiche comuni dei modelli di thread classici:

- **Condivisione delle risorse**: I thread all'interno dello stesso processo condividono lo stesso spazio di indirizzo e possono accedere facilmente alle variabili globali e ad altre risorse.
- **Sincronizzazione**: Meccanismi come mutex e semafori sono utilizzati per sincronizzare l'accesso alle risorse condivise, evitando condizioni di gara.
- **Creazione e distruzione**: I thread possono essere creati e distrutti dinamicamente, permettendo una gestione flessibile delle risorse.

Esistono inoltre diversi tipi di implementazione di questo modello:

- **Modello dell'utente**
- **Modello del kernel**
- **Modello ibrido**

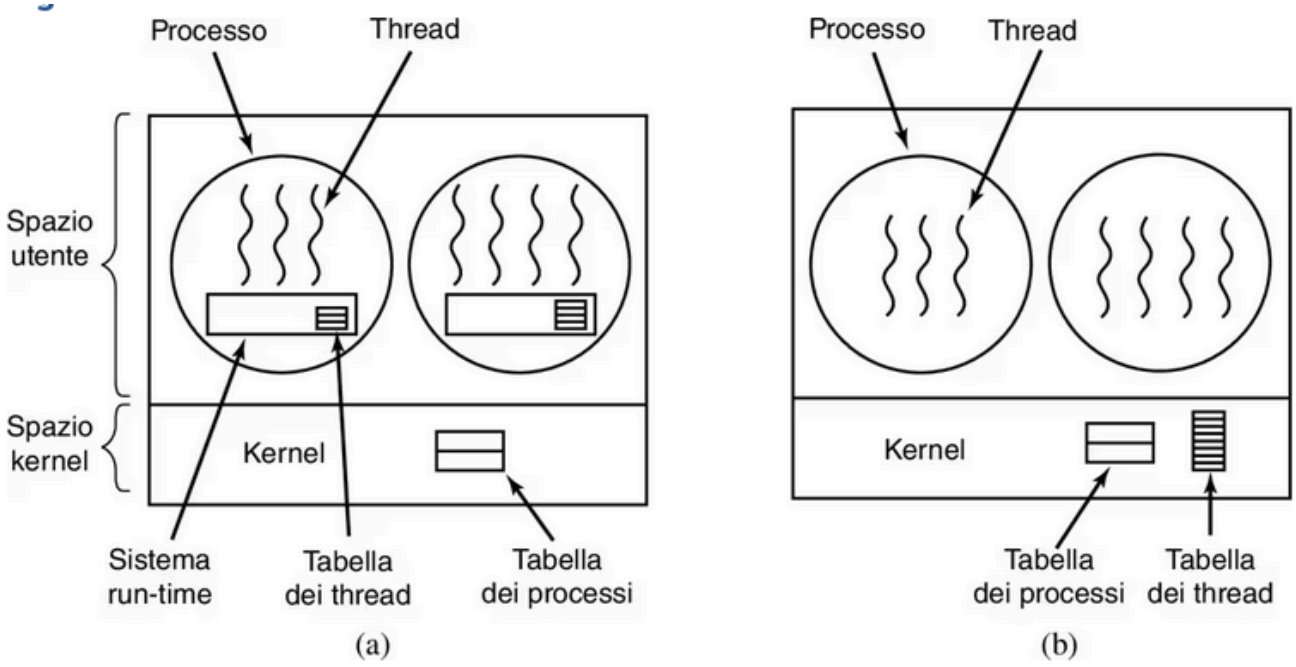
Alcune chiamate per la libreria Pthread

Chiamata	Descrizione
pthread_create	Crea un nuovo thread
pthread_exit	Termina il thread chiamante
pthread_join	Attende che un thread specifico esca
pthread_yield	Rilascia la CPU affinché venga eseguito un altro thread
pthread_attr_init	Crea e inizializza la struttura degli attributi di un thread
pthread_attr_destroy	Rimuove la struttura degli attributi di un thread

Alcune chiamate di funzione di Pthread.

Implementare i thread nello spazio utente

In questa implementazione i **thread nello spazio utente** sono gestiti dal kernel come processi a thread singolo.
Vengono gestiti tramite una libreria e possono essere esguiti su SO che non supportano direttamente i thread.
Ogni processo che usa i thread a livello utente deve avere una **tabella** in cui si trovano stati e proprietà deoìi thread del processo.
L'interruzione e il cambio tra thread, siccome si trovano nello stesso indirizzo di memoria, non richiede nessun cambio di contesto completo.
Inoltre offrono la possibilità di **personalizzare l'algoritmo di scheduling** per ogni processo.



(a) Un pacchetto di thread a livello utente. (b) Un pacchetto di thread gestito dal kernel.

I problemi principali con questo tipo di implementazione sono:

- Se un thread fa una **chiamante bloccante**, tutti gli altri pthread del processo vengono bloccati.
- I thread del processo non hanno interrupt del clock impedendo quindi un scheduling round-robin.
- Sebbene sono molto flessibili sono poco adatti per applicazioni in cui i thread si bloccano spesso.

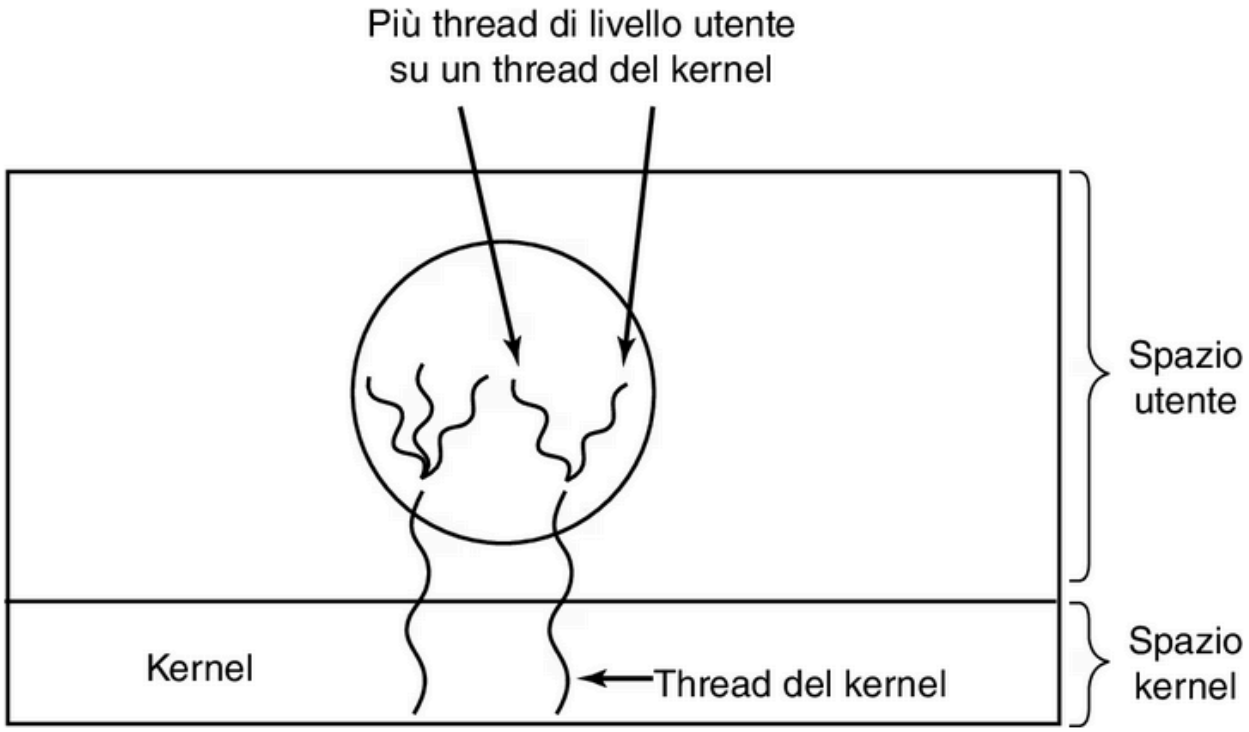
Implementare i thread nello spazio kernel

Questa implementazione riesce a risolvere alcuni prblemi di quella precedente, infatti le chiamate che potrebbero bloccare un thread vengono eseguite come chiamate di funzione che hanno però dei costi maggiori da un punto di vista di utilizzo di risorse.
Per questo motivo alcuni sistemi riciclano thread per ridurre costi invece di terminarli.
Se un thread viene bloccato a causa di un errore invece di bloccare il sistema, il kernel esegue altri thread.
Un problema però è come implementiamo i thread poiché , **stando a livello kernel, dobbiamo fare più cautela.**

Implementazione ibrida

Alcuni sistemi effettuano il **multiplexing dei thread utente sui thread del kernel** combinando i vantaggi dei due approcci.
Quindi i programmatori decidono quanti thread del kernel usare e quanti thread utente multiplexare.

Implementazione ibrida: Serve a mettere assieme i vantaggi dei due approcci alcuni thread visti dal kernel e alcuni dall'utente.



Esecuzione del multiplexing di thread di livello utente su thread del kernel.

Alcuni problemi con i thread

- Molte procedure di librerie possono causare conflitti se un thread sovrascrive dati cruciali, l'implementazione di **wrappers** può evitare conflitti, ma limita il parallelismo.
- La gestione dei segnali è complicata, perché il segnale arriva al processo, ma non tutti i thread devono tenerne conto, l'implementazione della gestione dei segnali può risultare complicata.