02a - Processi e Thread

Processi e operazioni sui processi

Un processo è un programma in esecuzione, e vi sono definite proprietà e metodi. Può essere scritto formalmente con la notazione $P = (C, S)$

dove P indica il processo, C il codice eseguibile ed S lo stato dell'esecuzione del processo.

Stato del processo

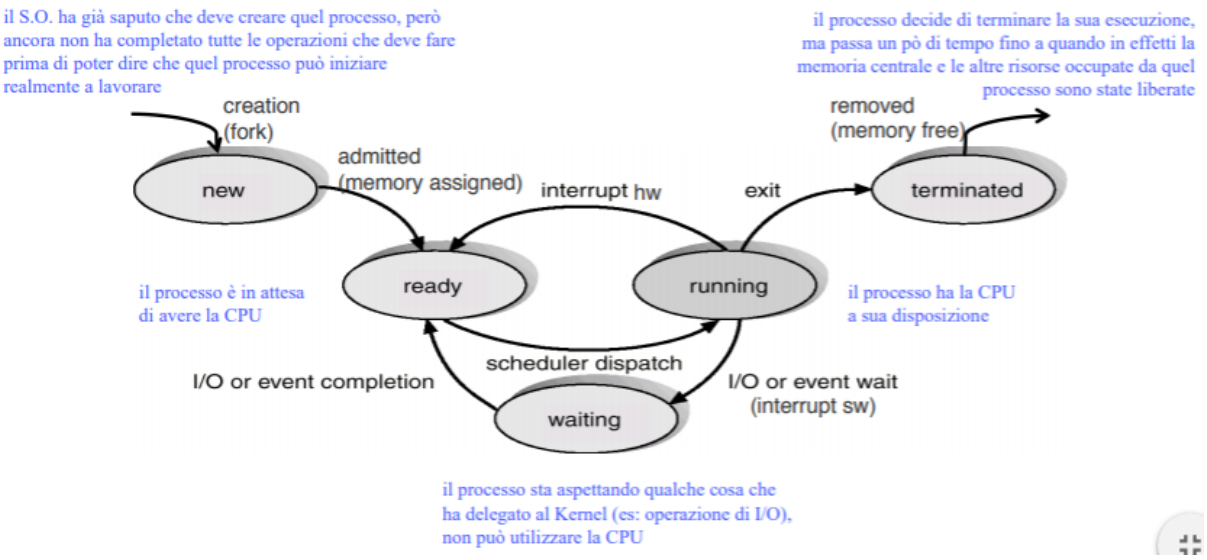
Un processo in esecuzione è soggetto a cambiamenti di stato. Gli stati del processo sono:

- *Nuovo* (**new**): si crea il processo.
  Il SO ha già saputo che deve creare quel processo, però ancora non ha

completato tutte le operazioni che deve fare prima di poter dire che quel processo può iniziare realmente a lavorare
.

- *Pronto* (**ready**): il processo attende di essere assegnato ad un'unità di elaborazione.
  Il processo è in attesa di avere la CPU.
- *Esecuzione* (**running**): esegue le operazioni del programma.
  Il processo ha la CPU a sua disposizione.
- *Attesa* (**waiting**): attende che si verifichi qualche evento.
  Il processo sta aspettando qualche cosa che ha delegato al Kernel (es: operazione di I/O), non può utilizzare la CPU.
- *Terminato* (**terminated**): il processo ha terminato l'esecuzione.
  Il processo decide di terminare la sua esecuzione, ma passa un pò di tempo fino a quando in effetti la memoria centrale e le altre risorse occupate da quel processo sono state liberate.

Scheduler



L'inizio è rappresentato dalla creazione del processo : l'interfaccia di input, interpretata e compresa la richiesta, chiede al SO di creare un processo associato a quel programma.

Un processo applicativo, ovvero che lavora in modalità utente (l'interfaccia del SO è un processo che lavora in modalità utente) *come fa a chiedere al Kernel di fare qualche cosa?*
Fa una chiamata di sistema, ovvero esegue un'interruzione software, il cui oggetto è la creazione di un processo.

Storicamente la chiamata di sistema per creare un processo viene indicata con il nome di "**fork**", perchè nei sistemi UNIX da lì in poi c'è una biforcazione (se fino a quel momento c'era un solo processo, da lì in poi ci sono due processi : il processo che ha chiesto la creazione dell'altro e l'altro processo). Il processo che chiede la creazione di un altro processo viene chiamato processo genitore, il processo che è stato creato come conseguenza di quella richiesta viene chiamato processo figlio.

La prima azione che fa quella routine del kernel che si deve occupare di creare un processo, è creare una struttura dati per contenere lo stato del processo stesso, che va sotto il nome di "**Descrittore di processo**" (*Process Control Block*), dove sono presenti lo stato, il PC, i vari registri...

Process Control Block (PCB) :

Nel sistema operativo il processo è rappresentato da un *Process Control Block* (**PCB**, descrittore di processo), che contiene le informazioni di un processo specifico:

- Stato di un processo
- PC
(contiene l'indirizzo dell'istruzione successiva)

- Registri CPU (variano a seconda dell'architettura del calcolatore)

- Informazioni sullo scheduling della CPU
- Informazioni sulla gestione della memoria
(si possono esprimere attraverso i valori dei registri di base, tabelle delle pagine e dei segmenti)
-
Informazioni contabili, tempo di uso della CPU ecc
- I
nformazioni sullo stato dell'I/O, lista dei dispositivi di I/O assegnati ad un processo

La *fork* come seconda cosa va a mettere dentro la variabile stato del PCB il valore *new*.

Il processo ha iniziato ad esistere (perchè esiste il PCB) però non è ancora in grado di funzionare.

La terza cosa che fa la *fork* è cercare dello spazio di memoria dove allocare il nostro processo; una volta trovato va a cambiare le informazioni sul PCB relative alla gestione della memoria. Se la memoria è piena il processo rimane in stato di *new* in attesa che si liberi della memoria.

Si iniziano a caricare nella memoria che abbiamo allocato le istruzioni del programma di cui abbiamo richiesto l'esecuzione.
A questo punto possiamo mettere nel PC del PCB l'indirizzo della prima istruzione del nostro programma che deve essere eseguita.
Arrivati a questo punto il nostro programma è pronto per iniziare a lavorare.
Adesso lo stato del nostro programma può essere messo a *ready.*

La *fork* termina la sua esecuzione e manda in esecuzione lo **scheduler** (un'altra parte del Kernel). Lo scheduler della CPU vede l'elenco di tutti i processi che sono in stato di *ready* (gestisce una coda dei processi *ready*, la coda è una coda dei PCB), ne sceglie uno e passa il puntatore di quel PCB ad un altro pezzo del Kernel chiamato "**dispatcher**".

Il dispatcher prende il PCB che gli è stato passato e lo utilizza per mandare in esecuzione quel processo (in generale, per far riprendere l'esecuzione di quel processo), ovvero va a prendere il contenuto dei registri della CPU, dei registri base e limite (mi permettono di stabilire qual è la memoria assegnata al processo), mi va ad assegnare questi valori ai rispettivi registri (legge dal PCB i valori che devono essere caricati sui registri), cambia lo stato e lo mette a *running*, per ultimo prende il valore del PCB e lo assegna al registro PC (per ultimo perchè per come funziona la CPU la successiva istruzione da eseguire è quella contenuta nella locazione di memoria il cui indirizzo è il PC).

Adesso possono succedere tre cose :
1) il processo potrebbe aver terminato la sua esecuzione, facendo una system call (interruzione software) per richiamare una funzione particolare del Kernel, la "
exit". La *exit* mette lo stato a *terminated*, successivamente inizia a deallocare tutte le risorse allocate e infine rimuove anche il PCB.

2) il processo chiede al Kernel di svolgere un qualche compito, viene eseguita un'interruzione software e aspetta che l'operazione richiesta venga eseguita; si mette nello stato di *waiting*.

3) un qualche dispositivo, a seguito di operazioni richieste da altri processi, invia un segnale di interruzione (hardware) al processo, il quale viene riportato nello stato di *ready*.

Context Switch

In presenza di un'interruzione il sistema deve salvare il contesto del processo corrente, per poterlo poi ripristinare. Si esegue quindi un salvataggio dello stato corrente della CPU e poi un ripristino dello stato per riprendere l'elaborazione. Il passaggio della CPU ad un nuovo processo implica la registrazione dello stato del processo vecchio e il caricamento dello stato precedentemente registrato dal nuovo processo.

Il meccanismo che intercorre nel passaggio da un processo utente ad un altro processo utente prende il nome di **Context Switch**.

Esempio: Il processo P_0 è in esecuzione, ma ad un certo punto c'è un'interruzione, a seguito della quale viene salvato lo stato di P_0 in PCB_0; lo stato di P_0 viene messo a *ready* o *waiting*. Va in esecuzione la routine di gestione di quell'interruzione, al termine della quale va in esecuzione lo scheduler per selezionare il nuovo processo (P_1) a cui assegnare la CPU; successivamente viene passato il PCB_1 al dispatcher, il quale legge i valori contenuti in PCB_1 per ripristinare la situazione da cui deve (ri)partire P_1 (assegna i valori ai vari registri, e da PCB_1 a PC); adesso va in esecuzione P_1.

L'intervallo di tempo che passa dal momento in cui P_0 si è (viene) interrotto al momento in cui inizia (riparte) l'esecuzione di P_1 viene chiamato "**Context Switch**"; è il tempo in cui la CPU è utilizzata dal Kernel, considerato non utile dal punto di vista dei processi utente, e per questo sarebbe meglio se fossero i più brevi possibile.

Operazioni su processi

Nei SO i processi si devono poter creare e cancellare dinamicamente.

Creazione

Durante l'esecuzione un processo può creare altri processi. Il processo creante è chiamato processo genitore, il nuovo processo è il processo figlio. Ciascun processo può creare altri processi formando una gerarchia chiamata *albero dei processi*, in cui genitori e figli possono condividere risorse (condividono tutte le risorse || i figli condividono un sottoinsieme delle risorse del genitore || non condividono risorse).
In fase di esecuzione, genitore e figli vengono eseguiti concorrentemente **OPPURE** il genitore attende che tutti o alcuni figli terminino.
Per quanto riguarda lo spazio degli indirizzi, il figlio è un duplicato del genitore **OPPURE** nel figlio viene caricato un programma (eventualmente diverso da quello del genitore).

In ambiente UNIX ogni processo ha discendenza da altri processi: il Root è servitore di tre processi, tra cui INIT, che genera a sua volta altri processi che sono gli utenti del SO.
Un nuovo processo si crea per mezzo della chiamata di sistema `fork()`. Generalmente dopo una chiamata `fork()` uno dei processi impiega una chiamata di sistema `exec()` per sostituire lo spazio di memoria del processo con un nuovo programma.

Terminazione

Un processo termina quando finisce l'esecuzione della sua ultima istruzione e inoltra la richiesta al SO di essere cancellato usando il comando `exit()`.
Il processo figlio può riportare alcuni dati al processo genitore, che li riceve attraverso la chiamata di sistema `wait()`.

La terminazione di un processo si può verificare anche quando un processo genitore pone termine all'esecuzione di uno dei suoi processi figli (`abort()`) per diverse ragioni:
- Il figlio ha ecceduto nell'uso di alcune risorse che gli sono state assegnate.
- Il compito assegnato al processo figlio non è più richiesto.
- Il genitore termina e il SO non consente al figlio di continuare l'esecuzione e avviene una terminazione a cascata.

Processi e Java

Introduzione

Nel linguaggio di programmazione Java, la programmazione concorrente riguarda principalmente i thread. Tuttavia, anche i processi possono essere gestiti in Java.

Esistono delle classi predefinite che possono essere utilizzate per gestire i processi.

Classe `ProcessBuilder`

- Questa classe viene utilizzata per creare processi del SO.
- Ogni istanza di *ProcessBuilder* gestisce una raccolta di attributi di processo.
- Il metodo `start()` crea una nuova istanza di processo con tali attributi. Il metodo `start()` può essere richiamato ripetutamente dalla stessa istanza per creare nuovi sottoprocessi con attributi identici o correlati.

Classe `Process`

- Il metodo `ProcessBuilder.start()` crea un processo nativo e restituisce un'istanza di una sottoclasse di *Process* che può essere utilizzata per controllare il processo e ottenere informazioni su di esso.
- La classe *Process* fornisce metodi per eseguire input dal processo, eseguire l'output nel processo, attendere il completamento del processo, controllare lo stato di uscita del processo e distruggere (uccidere) il processo.

Il processo genitore, se vuole creare un nuovo processo, invoca il metodo `start()` della classe *ProcessBuilder* ("equivalente" della *fork*) e viene creato il processo figlio. Il risultato del metodo `start()` restituito al processo genitore è un riferimento al processo figlio, una istanza della classe *Process*.

Esiste una classe *Process* le cui istanze sono idealmente i PCB dei vari processi figli.

I metodi della classe *Process* possono essere utilizzati dal genitore per comunicare con il figlio.

Esempio #1 (busy waiting)

```
public class SimpleProcess {
    public static void main(String[] args) throws Interrupt
edException, IOException {
```

```
        ProcessBuilder pb = new ProcessBuilder("ls", "-l
a");
        System.out.println("Run command");
        Process process = pb.start();

        while (process.isAlive()) {           // busy waiti
ng
            System.out.println("The process is still runnni
ng"); //
        }

        System.out.println("The process stopped. Any error
s? " + (process.exitValue() == 0 ? "No" : "Yes"));
        System.out.println("Echo Output:\n" + output(proces
s.getInputStream()));
}
```

Questo programma chiede di creare un nuovo processo, però tale operazione passa attraverso una serie di passaggi intermedi.

Innanzitutto bisogna creare una istanza di *ProcessBuilder* (`pb`), perchè devo dire che tipo di processo voglio creare; indico come parametro qual è il comando che deve essere eseguito per creare quel processo (nell'esempio chiedo la creazione di un processo figlio che corrisponde al programma "*ls*", l'operazione "*-la*" mi serve per dire che voglio l'elenco completo). Poi effettivamente chiedo la creazione di tale processo, invocando il metodo `start()` dell'istanza *pb*.

Da qui in poi, per come è fatto il metodo `start()`, il genitore prosegue la sua esecuzione e il figlio inizia la sua esecuzione; il genitore non viene sospeso automaticamente se non è lui a chiederlo esplicitamente.

Il metodo `start()` restituisce una istanza della classe *Process* (nell'esempio la variabile `process` contiene l'istanza del processo appena creato).

Supponiamo di voler interagire, in particolare supponiamo che il processo genitore voglia essere informato quando il processo figlio termina la sua esecuzione.
Uno dei tanti metodi di
Process che possono essere invocati e che ci aiutano a capire in che stato si trova il processo figlio è `isAlive()`, il quale serve solo per sapere se il processo

figlio è ancora in esecuzione oppure ha terminato la sua esecuzione; è un metodo che restituisce un valore booleano : se è `true` il processo è ancora in vita.
Ad un certo punto il processo terminerà la sua esecuzione e quindi `process.isAlive()` mi restituirà come valore `false`.

Questo qui è un esempio di **attesa attiva** (*busy waiting*), perchè il processo genitore per sapere quando il processo figlio termina la sua esecuzione, va a leggere continuamente uno degli attributi dell'istanza process per andare a vedere quando cambia valore; finché `isAlive()` è `true` continua a leggere il valore, e smette solo quando è diventato `false`.

Dal punto di vista del SO abbiamo due processi che sono in esecuzione (il processo genitore e il processo figlio) e che hanno bisogno entrambi della CPU; ma il processo genitore per eseguire `process.isAlive()` occupa la CPU (se fosse un sistema monoprocessore occuperebbe la CPU a discapito del processo figlio).

Dunque si definisce *attesa attiva* perchè il processo genitore usa la CPU non per fare qualcosa di utile, bensì per sapere quand'è che deve finire di aspettare.

Situazione che di solito non è molto efficiente, occupando inutilmente la CPU (nella gestione di I/O, quando un processo sta aspettando un'operazione di I/O viene sospeso e viene risvegliato solo quando tale operazione è stata completata).

Esco da questo ciclo ed eseguo le istruzioni successive; una di queste istruzioni (la prima) contiene il metodo `exitValue()` di *Process*, il quale permette di andare a vedere qual è il valore restituito dal processo figlio al processo genitore.

Il processo genitore ha mandato in esecuzione il processo "*ls -la*", che produce un elenco dei file contenuti in una cartella; se il processo genitore vuole acquisire quello che è stato mandato nello stream di output da parte del processo figlio, lui lo può acquisire come stream di input, quindi può avere accesso all'output generato dal processo figlio (attraverso il metodo `getInputStream()`).

Esempio #2 (blocking)

```
public class SimpleProcess {
    public static void main(String[] args) throws Interrupt
```

```
edException, IOException {
        ProcessBuilder pb = new ProcessBuilder("ls", "-l
a");
        System.out.println("Run command");
        Process process = pb.start();

        int errCode = process.waitFor(); //blocking

        System.out.println("Echo command executed, any erro
rs? " + (errCode == 0 ? "No" : "Yes"));
        System.out.println("Echo Output:\n" + output(proces
s.getInputStream()));
}
```

Questa volta, invece di attendere attivamente la terminazione del processo figlio, il processo genitore si sospende fino a quando il processo figlio non termina, invocando il metodo `waitFor()`.

Il suo effetto è quello di eseguire un'interruzione software : il processo genitore si sospende, il suo stato viene messo a *waiting*, il Kernel prende nota che deve essere risvegliato fino a quando il suo processo figlio terminerà la sua esecuzione, e a quel punto il Kernel cambia il suo stato a *ready*.

Esempio #3 (blocking)

```
public class SimpleProcess {
    public static void main(String[] args) throws Interrupt
edException, IOException {
        ProcessBuilder pb = new ProcessBuilder("ls", "-l
a");
        System.out.println("Run command");
        Process process = pb.start();

        Thread.sleep(1); // Sleep for 1 ms   //blocking

        if (process.isAlive()) {
        process.destroy();
        System.out.println("The process has been killed. An
y errors?" + (process.exitValue() == 0 ? "No" : "Yes"));
```

```
        }
        else
        {
        System.out.println("The process stopped. Any error
s? " + (process.exitValue() == 0 ? "No" : "Yes"));
        }
        System.out.println("Echo Output:\n" + output(proces
s.getInputStream()));
}
```

Un processo genitore potrebbe sì sospendersi, ma non in attesa della terminazione di un processo figlio; potrebbe chiedere di sospendersi per una certa quantità di tempo.

Il metodo `sleep()` sospende il processo per una certa quantità di tempo; l'argomento che viene passato a questo comando è per quanto tempo si deve sospendere.

L'esecuzione di questo comando comporta una chiamata di sistema (interruzione software) da parte del nostro processo, che viene messo in stato di *waiting*; il Kernel prende nota che allo scadere della quantità di tempo indicata lo stato del processo deve essere portato da *waiting* a *ready*.
Un modo con cui un processo genitore può chiedere la terminazione di un processo figlio è invocando il metodo `destroy()`.

Thread e operazioni sui thread

Thread

Un processo è un programma che si esegue seguendo un unico percorso di esecuzione, detto **thread**. L'esecuzione di un processo avviene seguendo una singola sequenza di istruzioni, quindi il processo può svolgere un solo compito alla volta. Un thread può essere quindi definito come una suddivisione di un processo in due o più filoni o sottoprocessi, che vengono eseguiti concorrentemente da un sistema di elaborazione.

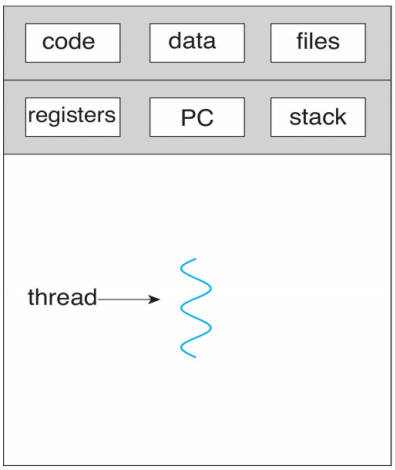
Anche i thread si suddividono in base al modo di utilizzo:

- Thread a livello utente: gestiti da una libreria a livello utente. Implementati dall'utente in fase di sviluppo.
-
Thread a livello kernel: sono gestiti direttamente dal kernel. Visti al livello di SO.

Un processo può essere scomposto in più thread (multithreading) in modo da permettere che un processo possa svolgere più di un compito alla volta.

Problemi

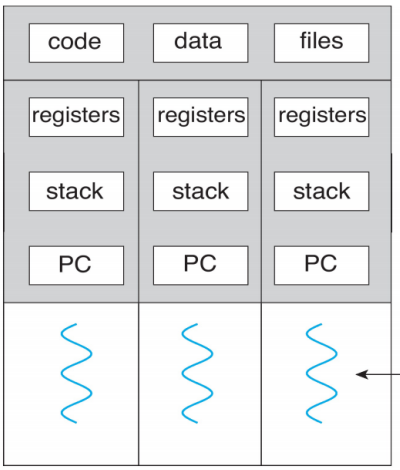
1. Il processo figlio deve lavorare sul messaggio che è stato ricevuto dal processo genitore; quindi processo genitore e processo figlio devono condividere lo spazio di memoria dove è contenuta la richiesta ricevuta dal genitore, in modo che il genitore mette in quello spazio di memoria tale richiesta e il processo figlio sa che può andare a leggere lì per trovare la richiesta che deve risolvere. Non sempre è conveniente avere processi che condividono lo stesso spazio di indirizzi (anche se parzialmente)
2. Il processo genitore e il processo figlio si possono trovare nella necessità di utilizzare lo stesso codice; se ogni processo ha un proprio spazio degli indirizzi diverso dagli altri, sullo spazio di memoria di ogni processo deve essere caricato il codice; in tal modo occupiamo inutilmente spazio di memoria. L'idea potrebbe essere che alcune parti della memoria sono condivise e altre parti no
3. Ogni volta che viene chiesta la creazione di un processo figlio, questo richiede una lunga sequenza di operazioni da parte del kernel; se noi abbiamo bisogno in continuazione di creare nuovi processi, potrebbe non essere così conveniente perchè il tempo di creazione di un processo diventa paragonabile al tempo di esecuzione del processo stesso; alla luce del punto precedente, sarebbe però conveniente far condividere alcune aree di memoria. Da qui è nata l'idea della programmazione *multithread*.



single-threaded process

Processo a thread singolo

viene eseguita un'istruzione alla volta in sequenza



multithreaded process

Processo multithread

tutti i thread hanno in comune il codice (lo carico una volta sola per tutti), i dati e i file

un thread si distingue dagli altri per le informazioni strettamente connesse con la sua esecuzione

per l'esecuzione di un thread mi serve conoscere il PC, i valori dei registri (della CPU) e lo stack

abbiamo un problema molto complesso da risolvere; per aumentare la velocità di esecuzione lo scomponiamo in sottoproblemi e ognuno lo facciamo affrontare da un processo diverso in parallelo con gli altri processi

Un processo multithread ha il vantaggio (rispetto al caso multiprocesso) che una volta che viene creato il primo thread e questo va in esecuzione, ad un certo punto ha bisogno del parallelismo : invece di creare un nuovo processo crea un nuovo thread. Questo vuol dire che non c'è bisogno di cercare spazio di memoria per il nuovo thread perchè già c'è (lo stesso del genitore), l'unica cosa che c'è da creare è un PCB associato a quel nuovo thread.

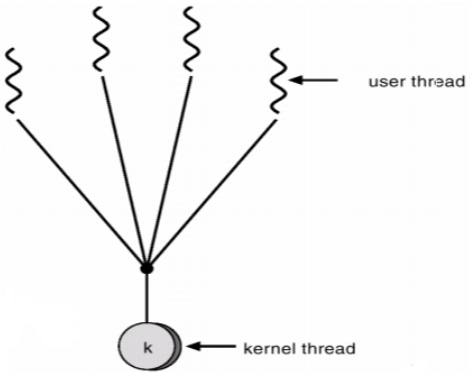
Lo svantaggio dei thread (svantaggio di una qualsiasi soluzione che prevede la condivisione della memoria) è che tutti i thread che appartengono allo stesso job (processo inteso come insieme di thread) condividono la stessa area di memoria, quindi se non facciamo attenzione un thread potrebbe andare a modificare i dati che servono ad un altro thread.

Modelli di programmazione multithread

Deve esistere una relazione tra i thread utente e i thread del kernel. Ne analizziamo la natura in tre casi comuni.

Molti-a-uno

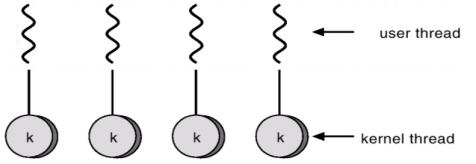
Fa corrispondere molti thread a livello utente ad un solo thread a livello kernel. La gestione dei thread risulta efficiente, ma l'intero processo risulta bloccato se un thread invoca una chiamata di sistema di tipo bloccante. Poiché un solo thread alla volta può accedere al kernel è impossibile eseguire thread multipli in parallelo in sistemi multiprocessore.



Sarà compito dell'ambiente di esecuzione stabilire quale tra i vari thread a livello utente che sono in esecuzione in quel momento debba occupare la CPU che il kernel mette a disposizione per il nostro processo, perchè per il kernel è in esecuzione un solo processo.

Uno-a-uno

Mette in corrispondenza ciascun thread a livello utente con un thread a livello kernel. Anche se un thread invoca una chiamata di sistema bloccante è possibile eseguire un altro thread; il modello permette anche l'esecuzione di thread in parallelo in sistemi multiprocessore. Uno svantaggio è che la creazione di ogni thread a livello utente comporta la creazione del corrispondente thread a livello kernel.

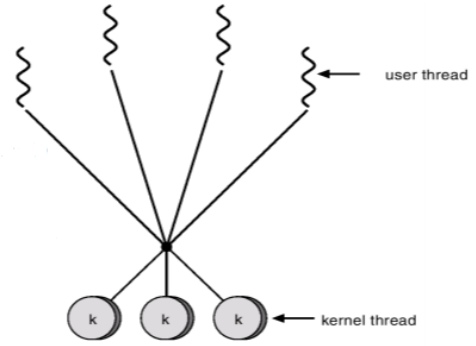


Ideale visto che ogni volta che un thread chiede la creazione di un altro thread, è direttamente il kernel a prendersi carico di schedularlo (assegnargli la CPU...)

Molti-a-molti

Mette in corrispondenza più thread a livello utente con un numero minore o uguale di thread a livello kernel. I programmatori possono creare liberamente i thread che ritengono necessari e i corrispondenti thread a livello kernel si possono eseguire in parallelo nelle architetture multiprocessore. Se un thread impiega una chiamata di sistema bloccante il kernel può fare in modo che si esegua un altro thread.

Rispetto al caso precedente c'è un ambiente di esecuzione che fa da interfaccia.



Thread e Java

Il programma ha una variabile contatore che viene incrementata da diversi thread

Esempio

```
public class ConcurrentCount {
  public static void main(String[] args) {
    CCounter c = new CCounter();

    // threads t and u, sharing counter
    Thread t = new Thread(c);
    Thread u = new Thread(c);

    t.start(); // increment once
    u.start(); // increment twice
          Abbiamo messo a ready i due thread
    try { // wait for t and u to terminate
      t.join(); u.join(); }
    catch (InterruptedException e)
    { System.out.println("Interrupted!"); }

    // print final value of counter
    System.out.println(c.getCounter());
  } }
```

vengono eseguiti in sequenza

join = wait

blocking

```
public class CCounter
  implements Runnable
{
  private int counter = 0;

  // increment counter by one
  public void run() {
     setCounter(getCounter() + 1);
  }

  // getter
  public int getCounter() {
     return counter;
  }
  //setter
  public void setCounter(int v) {
     counter = v;
  }

}
```

questa classe, implementando un'interfaccia, può essere passata come argomento del costruttore di una istanza di thread

Sistemi Operativi A.A. 2020/2021

27

In generale :

1) il risultato, se non stiamo attenti, dipende dall'ordine in cui le istruzioni dei vari thread in parallelo vengono eseguite;
2) noi non possiamo prevedere quale sarà l'ordine in cui vengono eseguite le istruzioni dei singoli thread.

Se vogliamo che il risultato non dipenda da questo (dall'ordine di esecuzione) dobbiamo mettere dei controlli, dobbiamo SINCRONIZZARE fra di loro i thread, in modo che il funzionamento dia sempre un risultato coerente che non dipenda dalla velocità di esecuzione dei singoli thread.

Task di Linux

Linux utilizza la medesima rappresentazione interna sia per i processi sia per i thread che in entrambi casi chiamiamo **task**. La differenza sta nel fatto che un thread è un task che condivide lo stesso spazio degli indirizzi con il genitore, mentre un processo ha uno spazio diverso; si nota questa differenza nel momento della creazione del task:

- `Fork` : crea un nuovo task con un suo nuovo contesto, cioè un processo.
- `Clone` : crea invece un nuovo task con una sua identità propria, ma con i dati condivisi con il genitore, cioè crea un thread.

Le proprietà di un task Linux possono essere classificate in 3 categorie: *Identità del task*, *Ambiente del task* e *Contesto del task*.

Identità di un task

- **Process ID** (*PID*). Identifica in modo univoco un task. Per esempio, se una applicazione deve effettuare una system call per inviare un messaggio, modificare o attendere un task, il parametro da passare al SO è il PID del task.
- **Credential**. Ad ogni task viene associato uno user ID ed uno o più group ID. Serve per determinare i diritti di accesso del task a risorse e a file.
- **Personality**. Questa caratteristica non si trova nei tradizionali sistemi Unix. Sotto Linux, ad ogni task viene associato un personality ID. Modifica la semantica di certe system call. Indica l'insieme di system call a cui fa riferimento quel task. Principalmente viene utilizzato da librerie di emulazione per rendere le system call compatibili con determinate caratteristiche di Unix.

- **Namespace**: vista specifica sulla gerarchia del file system. La maggior parte dei processi condivide lo spazio dei nomi comune e opera su una gerarchia di file system condivisa, ma ognuno può avere una gerarchia di file system univoca con la propria directory principale e un insieme di file system montati.

Ambiente di un task

Ogni task eredita dal genitore due array:

- Il **vettore degli argomenti** contiene tutti gli argomenti presenti nella linea di comando per avviare il programma. Convenzionalmente il primo argomento è il nome del programma stesso.
  Questo è uno strumento per parametrizzare l'esecuzione dei programmi.
- Il **vettore dell'ambiente** è una lista di coppie del tipo "NAME=VALUE" che associa ad ogni variabile di ambiente un valore. In questo modo ogni task può avere il SO configurato ad hoc, invece di avere una configurazione unica per tutti i task.

Inoltre le variabili possono essere utilizzate per passare informazioni da un task ad un altro.

Far ereditare ad un task l'ambiente del task genitore è un modo per far passare informazioni dal genitore al figlio.

Contesto di un task

Lo stato dell'esecuzione di un task (le informazioni contenute in un descrittore di task).

- Lo **scheduling context** è la parte più importante. Sono le informazioni necessarie allo scheduler per sospendere e riattivare un task.
- **Informazioni contabili** su tutte le risorse usate attualmente ed utilizzate fino a quel momento dal task.
- La **tabella dei file aperti** è un array di puntatori ai descrittori di file (fd)
  Quando si effettuano chiamate di sistema I/O di file, i processi fanno riferimento ai file in base al loro indice in questa tabella, mentre la tabella dei file elenca i file aperti esistenti, il file system context si applica alle richieste di apertura di nuovi file.
  Le attuali directory di root e di default da utilizzare per le nuove ricerche di file vengono memorizzate qui.

- La **signal-handler table** indica quali routine (appartenenti allo spazio degli indirizzi del task) devono essere chiamate quando al task arriva uno specifico segnale (signal).
- La **virtual-memory context** indica l'intero contenuto dello spazio degli indirizzi del task.