Politecnico di Milano

Introduzione ai thread

Professore
William Fornaciari
Politecnico di Milano
fornacia@elet.polimi.it
www.elet.polimi.it/~fornacia

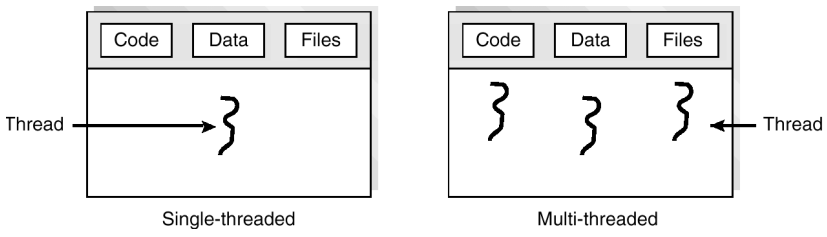
Sommario

- Cosa sono i thread
- Informazioni associate ai threads
- Potenziali benefici
- Modelli di cooperazione
- Meccanismi di sincronizzazione
- Esempi: WinNT, Solaris 2, Java

Cosa sono i thread

- Un processo viene diviso in due componenti:
  - Risorse allocate al processo
  - Contesto di esecuzione
- Un contesto di esecuzione è associato ad un thread
- Un processo deve avere almeno un thread
- Un processo può avere più di un thread
- I thread condividono tutte le risorse del processo a cui appartengono
- Analogamente ai processi sono previsti degli stati di esecuzione, come pronto, attivo, sospeso, terminato

Architetture a thread singolo e multiplo



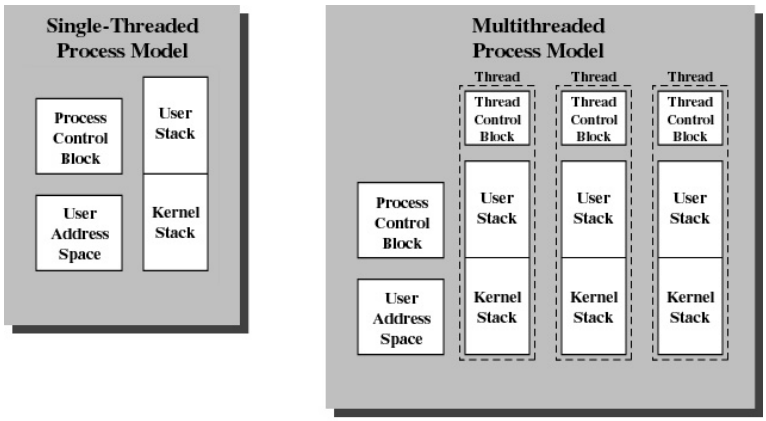
Esempio: file server su LAN

- Ad ogni nuova richiesta di file
  - Creazione di un thread per la gestione
- Il server gestisce molte richieste
  - Creazione e distruzione in breve tempo dei thread
- Se il server è multiprocessore
  - Molti thread dello stesso processo possono essere eseguiti parallelamente su differenti CPU
- Per sistema monoprocessore
  - Semplificazione della struttura dell'applicazione in funzioni logicamente distinte

Informazioni associate

- Le informazioni relative al processo sono:
  - Spazio degli indirizzi
  - Variabili globali
  - File aperti
  - Processi figli
  - Timer
  - Segnali
  - Semafori
  - Informazioni di accounting
- Ad ogni thread sono associati:
  - Program counter
  - Stack
  - Insieme dei registri
  - Thread figli
  - Stato

Modello dei processi con thread singolo e multiplo



Vantaggi offerti rispetto ai processi

- Maggior efficienza di creazione ed eliminazione
  - per sistemi unix-like efficienza 10 volte superiore
- Condivisione della memoria, per comunicare non sono obbligati a chiamare sempre il kernel
- Cambi di contesto più veloci
- Possibilità di implementare efficientemente alcune importanti architetture software
- Sfruttamento di un livello di parallelismo a grana più fine
- Sfruttamento più facile di architetture multiprocessore

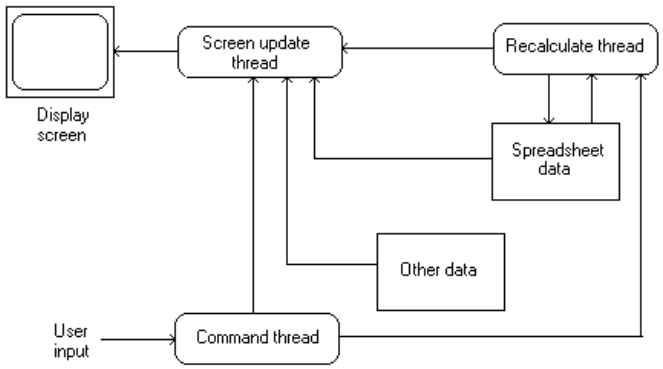
Tipi di thread e organizzazione

- Le architetture che implementano i principali modelli di cooperazione sono
  - Team model
  - Dispatcher
  - Pipeline
- Possono esistere
  - *Kernel* Thread
    - Gestione dei thread svolta da *user-level thread library*
    - Es. POSIX Pthreads, MAC C-Threads, Solaris Threads
  - *User* Thread
    - Supportati dal Kernel
    - Es.: Win 95/98/NT, Solaris, UNIX (by Digital)

Team model

- Utile per partizionare un'applicazione
- Si individuano alcune attività da separare
- Si crea un thread per ogni attività individuata
- Vantaggi
  - Maggior reattività agli input
  - Semplificazione del processo di implementazione, soprattutto per uno sviluppo effettuato da un gruppo di programmatori
- Esempio
  - gestione di un foglio elettronico

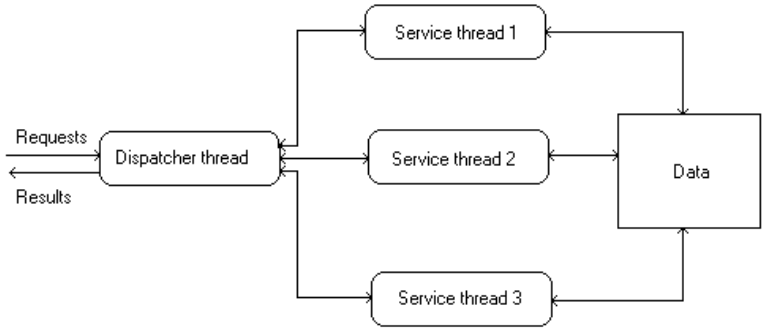
Team model: esempio



Architettura a dispatcher

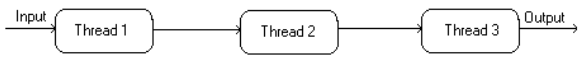
- Simile al team model
- Esiste un thread, detto dispatcher, che riceve richieste di servizio e le invia ai thread che le eseguono
- Si crea un thread per ogni richiesta di servizio, eventualmente prevedendo un numero massimo di thread
- Utile per replicare attività
- Esempio
  - file server

Architettura a dispatcher



Architettura a pipeline

- Partizionamento delle attività su base temporale
- Utile per gestire una catena di algoritmi ciascuno dei quali utilizzi come dato in ingresso l'uscita di un altro algoritmo
- Vantaggio
  - Se un thread deve mettersi in attesa di un evento (ad esempio per un accesso a disco), gli altri thread, se hanno i dati necessari, possono proseguire la computazione
- Esempio: script con la shell di UNIX



Thread package

- Un thread package è una libreria per l'utilizzo dei thread
- Offre le funzionalità per la creazione e lo scheduling dei thread di un processo
- Solitamente offre funzionalità di sincronizzazione fra thread (quali mutex e condition variable)
- Può essere realizzato
  - nello spazio di indirizzamento dell'utente (**user thread package**)
  - nel kernel (**kernel thread package**)

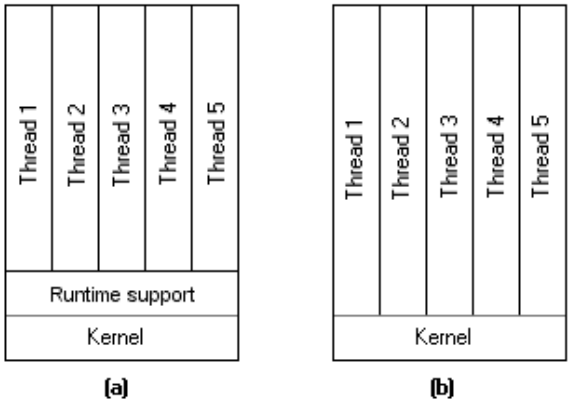
User thread package

- Viene implementato uno strato di supporto runtime sopra il kernel per gestire i thread
- Il supporto runtime si occupa dello scheduling dei thread attivi di un processo
- Vantaggi
  - Permette di aggiungere il supporto dei thread ad un sistema operativo che non li preveda (ad es. UNIX)
  - Permette di avere un algoritmo di scheduling personalizzato
- Svantaggi
  - Un thread rimane attivo fino a che non si sospende volontariamente
  - Difficile o impossibile sfruttamento multiprocessing
  - Il blocco di un thread può bloccare l'intero processo

Kernel thread package

- I thread vengono gestiti direttamente dal kernel
- Il kernel assegna a ciascun thread un quanto di tempo, trascorso il quale lo scheduler seleziona un altro thread per l'esecuzione
- Vantaggi
  - Se un thread esegue una chiamata di sistema che lo blocca, viene sospeso solo quel thread e non tutto il processo a cui appartiene
- Svantaggi
  - La gestione dei thread va effettuata con chiamate di sistema, introducendo un overhead rispetto alla soluzione degli user thread package

Architetture dei thread package



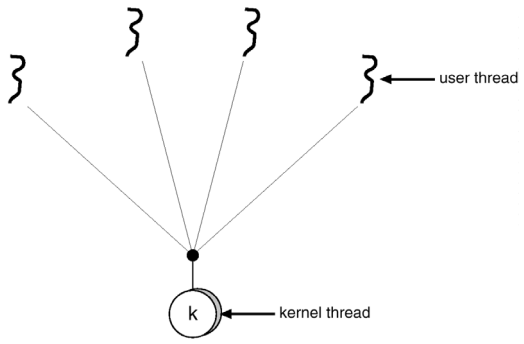
- User thread package (a): gestione thread nell'applic.
- Kernel thread package (b): nell'area applicazione si usa API

Modelli per multithreading

- Relazione fra thread e processi
  - Many-to-one (NT, Solaris, OS/2, ...)
  - One-to-one (molti UNIX)
  - Many-to-many (sperimentazioni in TRIX)
  - One-to-many (studi per sistemi distribuiti)

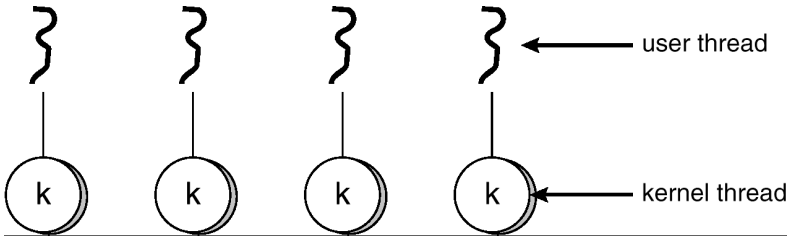
Many-to-one

- Molti User-Level Threads sono mappati su un solo Kernel Thread
- Usati in sistemi che non supportano i Kernel Threads

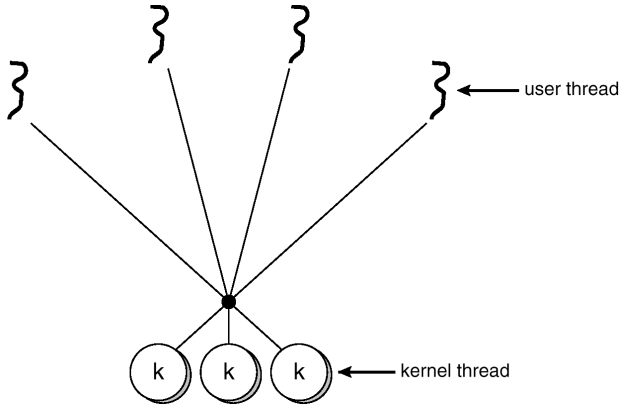


One-to-one

- Ogni User-Level Thread è mappato su Kernel Thread
- Exsempi
  - Windows 95/98/NT
  - OS/2



Many-to-many



Comunicazione e sincronizzazione

- Dato che i thread devono collaborare si rendono necessarie la comunicazione e la sincronizzazione
- Le principali soluzioni a questi problemi, comuni a processi e thread, sono
  - Semafori
  - Mutex
  - Condition variables
  - Monitor
  - Scambio di messaggi

Semafori

- Un semaforo è una variabile intera a cui sono associate tre operazioni:
  - Inizializzazione ad un valore non negativo
  - `wait()` decrementa il valore; se il valore del semaforo risulta negativo, il chiamante viene bloccato
  - `post()` (spesso chiamata `signal()`) incrementa il valore; se il valore del semaforo resta non positivo, uno dei processi bloccati su quel semaforo viene risvegliato
- Ad un semaforo è associata una lista di processi bloccati su di esso
- Necessaria una politica per la scelta del processo da risvegliare

Mutex

- Si tratta di un semaforo binario: ha solo due valori possibili, detti aperto e chiuso
- Le principali operazioni su un mutex sono
  - `init()` per inizializzarlo
  - `lock()` per chiuderlo; se il mutex è chiuso, il chiamante viene messo in attesa del rilascio del mutex
  - `trylock()` tenta di chiudere il mutex; se il mutex è chiuso, fallisce ma il chiamante non viene bloccato
  - `unlock()` rilascia il mutex

Condition variable

- Sono particolari variabili a cui è associata una coda di attesa
- Vengono utilizzate per attendere che sia verificata una condizione sullo stato della variabile
- L'accesso alle condition variable va effettuato in mutua esclusione
- La mutua esclusione viene garantita associandogli un mutex
- Il mutex associato va chiuso prima di utilizzarla e riaperto subito dopo

Condition variable

- Le operazioni su una condition variable sono:
  - `init()` per inizializzarla
  - `wait()` per mettersi in attesa che la condizione sia verificata; rilascia automaticamente il mutex associato per permettere ad altri processi di utilizzarla
  - `signal()` per risvegliare uno dei processi in attesa sulla variabile
  - `broadcast()` per risvegliare tutti i processi in attesa sulla variabile

Monitor e scambio messaggi

- Altre soluzioni per la sincronizzazione e la mutua esclusione sono i monitor e lo scambio di messaggi
- Monitor
  - E' un tipo di dato astratto. I dati interni possono essere letti e modificati solo tramite le funzioni offerte dal monitor
  - Un solo processo per volta può essere all'interno del monitor
- Scambio di messaggi
  - I processi si inviano a vicenda messaggi per sincronizzarsi o passarsi dati

Esempi di implementazione

- Verranno presentate le implementazioni offerte da
  - Windows NT
  - Solaris 2
  - Standard POSIX
  - Java Thread

Windows NT

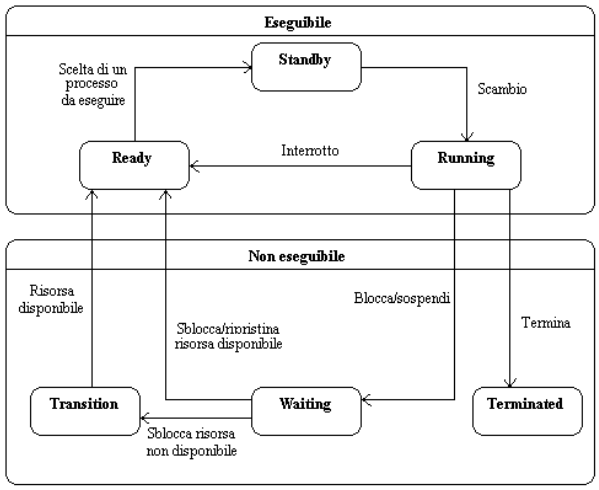
- Approccio generale per supportare diversi ambienti di sistema (Win 32, OS/2)
- I processi sono implementati come oggetti
- Un processo eseguibile può avere uno o più thread
- Gli oggetti di tipo processo e di tipo thread incorporano capacità di sincronizzazione
- La libreria dei thread è implementata nel kernel
- Due thread dello stesso processo possono essere eseguiti concorrentemente su processori separati, evitando l'overhead di processi multipli

Strutture dati e primitive in Windows NT

Processo
ID del processo
Descrittore della sicurezza
Priorità di base
Affinità di processore di default
Limiti di quota
Tempo di esecuzione
Contatori di I/O
Contatori delle operazioni di MV
Porte per le eccezioni
Stato di uscita
Creazione di processi
Apertura di processi
Richiesta di informazioni dei processi
Modifica di informazioni dei processi
Processo corrente
Termina il processo
Allocazione/rilascio di memoria virtuale
Lettura/scrittura di memoria virtuale
Protezione della memoria virtuale
Blocco/sblocco della memoria virtuale
Richiesta di memoria virtuale
Reclamo della memoria virtuale

Thread
ID del thread
Contesto del thread
Priorità dinamica
Priorità di base
Affinità di processore del thread
Tempo di esecuzione del thread
Stato di allerta
Contatore delle sospensioni
Token di impersonazione
Porta di terminazione
Stato di uscita del thread
Creazione di thread
Apertura di thread
Richiesta di informazioni dei thread
Modifica di informazioni dei thread
Thread corrente
Termina il thread
Leggi il contesto
Modifica il contesto
Sospendi
Ripristina
Allerta il thread
Test di allerta per il thread
Registra la porta di terminazione

Stati dei thread di Windows NT



Stati dei thread di Windows NT

- Ready
  - pronto per l'esecuzione; l'allocatore del microkernel shedula in ordine di priorità i thead in questo stato
- Standby
  - è stato selezionato per l'esecuzione su un processore, ma è in attesa che si liberi il processore. Se ha priorità suff. Alta può sostituire quello in esecuzione
- Running
  - è attualmente in esecuzione; se viene interrotto o scade il suo quanto di tempo passa allo stato ready

Stati dei thread di Windows NT

- Waiting
  - sta aspettando un evento, o è bloccato (volontariamente) per motivi di sincronizzazione o un sottosistema dell'ambiente lo ha forzato a sospendersi
  - quando è soddisfatta la condizione di attesa torna ready
- Transition
  - si passa dallo stato waiting a questo stato se è pronto per l'esecuzione ma qualche risorsa non è disponibile
  - passa allo stato ready quando la risorsa diventa disponibile
- Terminated
  - un thread può terminare per sua richiesta, per richiesta di un altro thread o quando termina il suo processo genitore

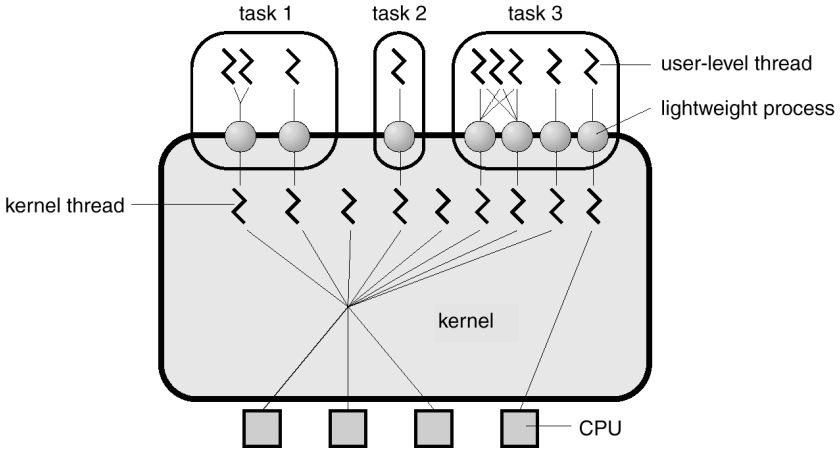
Solaris 2 threads: 4 concetti base

1. Processi
   - sono i normali processi UNIX
2. Thread a livello utente (ULT)
   - sono implementati con uno user thread package e forniscono l'interfaccia per il parallelismo rivolta alle applicazioni
   - sono invisibili al sistema operativo
   - ogni processo deve contenere almeno un ULT
3. Processi leggeri (LWP)
   - sono una mappatura degli ULT in thread del kernel
   - hanno una relazione uno a uno con i thread del kernel
   - Su multiprocessore possono venire eseguiti in parallelo
4. Thread del kernel
   - sono le entità che possono essere schedulate e allocate su uno dei processore del sistema

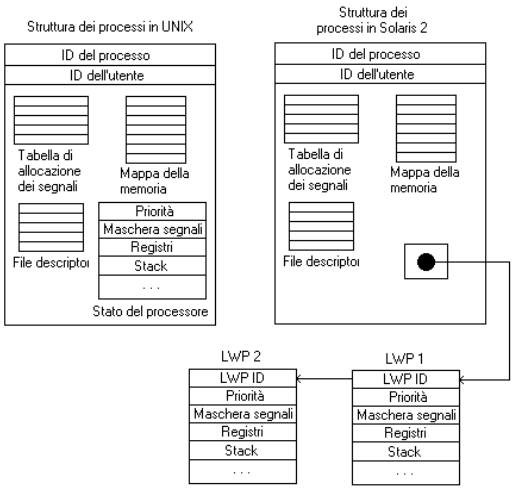
Solaris 2 threads

- Ad ogni LWP è associato un thread del kernel ma non è detto il contrario
- La mappatura fra ULT e LWP può avvenire secondo differenti modalità
  - **un LWP un solo ULT**: tutti i thread a livello utente possono essere eseguiti in parallelo
  - **un LWP per più ULT**: il thread a livello utente rimane attivo fino a che non ceda il posto ad un altro; se si sospende, tutti gli ULT mappati su quel LWP risultano sospesi; un solo ULT per volta può essere in esecuzione ad un dato istante
  - **n ULT su m LWP (con n>m):** al massimo m ULT possono essere eseguiti in parallelo;

Solaris 2 threads

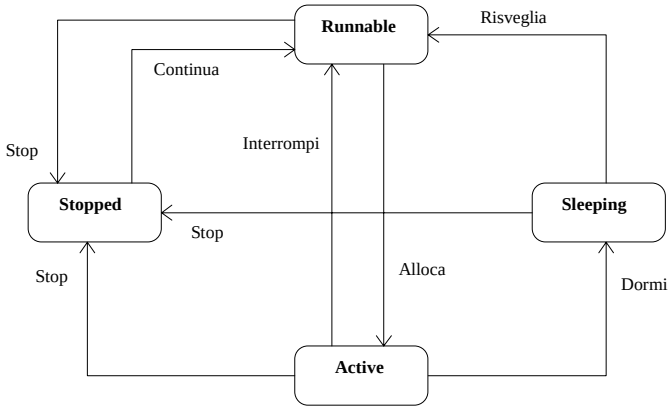


Strutture dati in Solaris 2



Stati di un ULT in Solaris 2

- Stati di un ULT quando un gruppo di ULT condividono lo stesso LWP

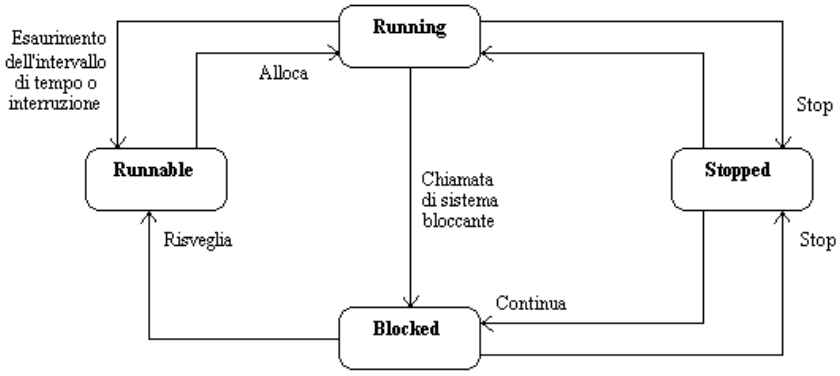


Stati di un ULT in Solaris 2

- T1 è ULT in stato active, eventi possibili:
  - *Sincronizzazione*: T1 passa in sleeping chiamando la primitiva di sincronizzazione. Ritorna runnable quando la condizione è soddisfatta
  - *Sospensione*: ogni thread può sospendere T1 facendolo passare a stopped, finchè non viene inviata richiesta di continuazione da altro thread che lo fa passare in runnable
  - *Prelazione*: se aumenta la priorità di altro thread T2 in stato runnable fino ad avere priorità maggiore di quella di T1, allora T1 passa in runnable e T2 viene associato al LWP resosi disponibile
  - *Concessione*: T1 esegue comando thr_yiels che spinge lo scheduler a cercare altro T2 con stessa priorità. Se esiste T1 è messo runnable e T2 assegnato a LWP altrimenti continua T1

Stati di un LWP in Solaris 2

- Diagramma degli stati di un LWP (dettaglio dello stato active di un ULT)



Stati di un LWP in Solaris 2

- Se un ULT è l'unico thread di un LWP
  - Stato runnable non ha senso, il thread è sempre eseguibile, passa subito ad active
  - Se un thread si sospende per sincronizzazione, anche il LWP lo farà, bloccandosi sulla variabile di sincronizzaizone a livello kernel
- Interrupt in Solaris 2
  - Gestiti come thread del kernel (con priorità superiore a quella degli altri del sistema), la sincronizzazione gestita con le primitive dei thread
  - Più flessibile ed efficiente della soluzione tradizionale di disabilitare interruzioni di priorità inferiore

POSIX: pthreads

- È il thread package specificato nello standard POSIX
- Specifica tipi e funzioni per gestire i thread, i mutex e le condition variables
- Principali tipi
  - `pthread_t` descrive un thread
  - `pthread_mutex_t` descrive un mutex
  - `pthread_cond_t` descrive una condition variable
  - `pthread_attr_t` descrive gli attributi di un thread

pthreads: principali attributi di un thread

- **detachstate** un thread può essere detached o joinable; nel primo caso quando termina viene cancellato, nel secondo caso viene cancellato solo dopo una `pthread_join` che lo abbia come argomento
- **inheritsched** determina se il thread eredita la politica di scheduling del thread che lo ha creato
- **schedpolicy** determina quale sia la politica di scheduling del thread (es. FIFO, RR, ...)
- **stacksize** determina la dimensione minima per lo stack del thread

Creazione thread: pthread_create()

- Sintassi
  - `int pthread_create(pthread_t *thread, const pthread_attr_t *attr, void *(*startroutine)(void *), void *arg);`
- Effetto
  - crea un thread
- Parametri
  - un puntatore al thread da creare
  - un puntatore all'attributo da utilizzare; se il puntatore è NULL, utilizza gli attributi di default
  - un puntatore alla funzione che il thread deve eseguire; la funzione accetta come parametro un puntatore a void e restituisce un puntatore a void
  - un puntatore al parametro della funzione

Terminazione di un thread

- Un thread termina in tre casi
  - al termine dell'esecuzione della funzione ad esso associata
  - se chiama la `pthread_exit(void *valueptr)`; il valore passato alla `exit` è quello che verrà restituito da una eventuale `pthread_join()` chiamata sul thread in questione
  - se viene terminato da un altro thread tramite la `pthread_cancel()`; tale funzione ha effetto solo se il thread è definito come cancellabile e l'effettiva cancellazione avviene quando il thread raggiunge una delle operazioni definite come cancellation point

pthread_join()

- Sintassi
  - `pthread_join(pthread_t thread, void **valueptr)`
- Effetto
  - il thread chiamante viene sospeso fino al momento in cui il thread specificato come parametro termina
- Parametri
  - un puntatore al thread da attendere
  - un puntatore puntatore a void restituito dal thread tramite la `pthread_exit()`

Esempio di codice n.1

- Il primo esempio di codice presentato è una semplice applicazione che scrive "hello world" utilizzando due thread:
  - Un thread scrive "hello", il secodo scrive "world" dopo avere atteso il primo, il main attende il termine del secondo
- Tratto dal sito www.cs.auckland.ac.nz/compsci711fc/lectures/pthread/pthread.html
- Scritto da *S. Manoharan (Department of Computer Science. University of Auckland)*

Esempio di codice

```
#include <iostream.h>
#include <pthread.h>
void * hello (void *)
{
    cout << "hello ";
    return 0;
}
void * world(void *arg)
{
  pthread_t *th=(pthread_t *)(arg);
  pthread_join(*th, 0);
  cout << "world";
  return 0;
}
```

```
int main()
{
   pthread_t th, tw;
   pthread_attr_t *tattr = 0;
   pthread_create(&th, tattr, hello, 0);
   pthread_create(&tw, tattr, world,&th);
   pthread_join(tw, 0);
   cout << endl;
   pthread_exit(0);
   return 0;
   // not reached
}
```

Creazione threads

Attende terminazione

Attende terminazione

Esempio di codice n.2

- Viene presentato un esempio dell'utilizzo della libreria pthread
- Il programma svolge un compito pressochè inutile
  - Inizializza tutti gli elementi un vettore a 1.5
  - Arrotonda tali elementi all'intero inferiore
- Ciascuna delle due fasi viene divisa in due parti
- La struct parametri permette di semplificare le dichiarazioni delle funzioni utilizzate come codice dei thead: consente di passare un solo puntatore
- I mutex e le condition variables vengono utilizzati per sincronizzare i vari thread

Esempio di codice n.2

- Sono creati 4 thread:
  1. Inizializza i primi cinquanta elementi del vettore
  2. Inizializza i rimanenti cinquanta elementi
  3. Arrotoonda i primi cinquanta elementi
  4. Arrotonda i rimanenti elementi
- Vengono usate condition variable per consentire ai thread che arrotondano di attendere che la corrispondente porzione del vettore sia inizializzata
- Tutti gli accessi a una condition variable avvengono fra un lock e unlock del mutex associato

Esempio di codice

```
#include <math.h>
#include <pthread.h>

//Inizializzo e definisco le
   strutture

double x[100];
pthread_mutex_t m1,m2;
pthread_cond_t cv1,cv2;
bool fineini1,fineini2;

typedef struct
{
    int primo;
    int ultimo;
    pthread_mutex_t *m;
    bool *fineini;
    pthread_cond_t *cv;
}parametri;
```

```
void * inizializza(void *arg)
{
    int temp,i;
    parametri *p=(parametri *)(arg);
    for (i=p->primo;i<p-
   >ultimo+1;i++)
    {
        x[i]=1.5;
    }
    temp=pthread_mutex_lock(p->m);
    *(p->fineini)=true;
    pthread_cond_signal(p->cv);
    temp=pthread_mutex_unlock(p->m);
}
```


Esempio di codice

```
void * calcola(void *arg)
{
    int temp,i;
    parametri *p=(parametri *)(arg);
    temp=pthread_mutex_lock(p->m);
    while (*(p->fineini)==false)
        pthread_cond_wait(p->cv,p->m);
    temp=pthread_mutex_unlock(p->m);
    for (i=p->primo;i<p->ultimo+1;i++)
    {
        x[i]=floor(x[i]);
    }
}
```

```
void main()
{
    int temp;
    parametri par;
    pthread_t *t1,*t2,*t3,*t4;

    fineini1=false;
    fineini2=false;

    //Creo i mutex
    temp=pthread_mutex_init(&m1,NULL);
    temp=pthread_mutex_init(&m2,NULL);
```


Esempio di codice

```
//Setto i parametri per la prima metà
par.primo=0;
par.ultimo=49;
par.m=&m1;
par.fineini=&fineini1;
par.cv=&cv1;
//Lancio i thread sulla prima metà
temp=pthread_create(t1,NULL,inizializza, &par);}
temp=pthread_create(t2,NULL,calcola, &par);

//Setto i parametri per la seconda metà
par.primo=50;
par.ultimo=99;
par.m=&m2;
par.fineini=&fineini2;
par.cv=&cv2;
//Lancio i thread sulla seconda metà
temp=pthread_create(t3,NULL,inizializza, &par);
temp=pthread_create(t4,NULL,calcola, &par);
```

```
//Attendo la terminazione dei
thread
temp=pthread_join(*t1,NULL);
temp=pthread_join(*t2,NULL);
temp=pthread_join(*t3,NULL);
temp=pthread_join(*t4,NULL);
```


Riferimenti bibliografici

- Standard IEEE 1003.1 1996 version
- Crowley, *Operating Systems: a constructive approach*, Irwin
- Stallings, *Sistemi operativi*, Jackson libri
- Tanenbaum, *I moderni sistemi operativi*, Prentice hall int., Jackson libri, 1994
- www.cs.auckland.ac.nz/compsci711fc/lectures/pthread/pthread.html

Java Threads

- Java Threads May be Created
  - Extending the Thread class
  - Implementing the Runnable interface

Extending the Thread Class

- Extending the Thread Class

```
class  Worker1 extends Thread
{   public void run() {
    System.out.println("I am a Worker Thread");
    }
}
```

- Creating the Thread

```
public class First
{   public static void main(String args[]) {
    Worker runner = new Worker1();
    runner.start();
    System.out.println("I am the main thread");
    }
}
```


The Runnable Interface

- The runnable Interface

```
public interface Runnable
{
    public abstract void run();
}
```

- Implementing the runnable interface

```
class  Worker2 implements Runnable
{
    public void run() {
    System.out.println("I am a Worker Thread");
    }
}
```


Creating the Thread

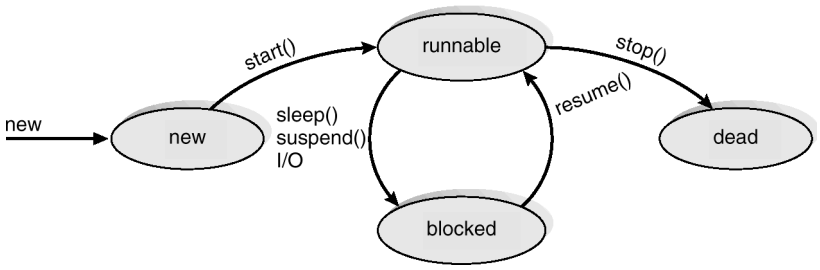
```
public class Second
{
  public static void main(String args[])
  {
     Runnable runner = new Worker2();
     Thread thrd = new Thread(runner);
     thrd.start();

     System.out.println("I am the main
  thread");
  }
}
```


Java Thread Management

- **suspend()**
  - suspends execution of the currently running thread
- **sleep()**
  - puts the currently running thread to sleep for a specified amount of time.
- **resume()**
  - resumes execution of a suspended thread.
- **stop()**
  - stops execution of a thread.

Java Thread States diagram



The Producer Consumer Problem

```
public class Server {
  public Server() {
       MessageQueue mailBox = new MessageQueue();

            Producer producerThread = new
  Producer(mailBox);
            Consumer consumerThread = new
  Consumer(mailBox);

            producerThread.start();
            consumerThread.start();
    }
    public static void main(String args[])   {
       Server server = new Server();
    }
}
```


Producer Thread

```
class Producer extends Thread {
  public Producer(MessageQueue m) {
       mbox = m;
  }

    public void run() {
   while (true) {
        // produce an item & enter it into the
  buffer
       Date message = new Date();
             mbox.send(message);
       }
    }
    private  MessageQueue mbox;
}
```


Consumer Thread

```
class Consumer extends Thread {
  public Consumer(MessageQueue m) {
       mbox = m;
  }

    public void run() {
   while (true) {
       Date message = (Date)mbox.receive();
       if (message != null)
             // consume the message
  }
    }
    private  MessageQueue mbox;
}
```