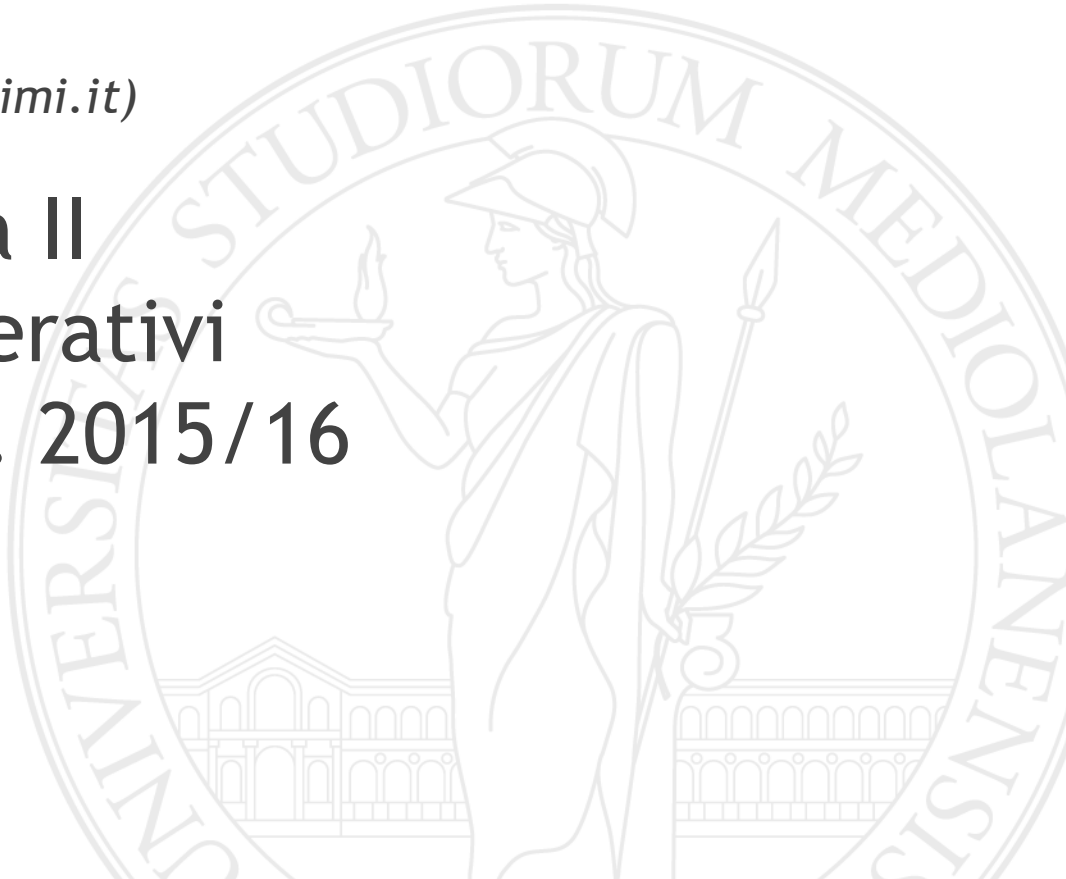
UNIVERSITÀ DEGLI STUDI DI MILANO
DIPARTIMENTO DI INFORMATICA

Alberto Ceselli
(alberto.ceselli@unimi.it)

Informatica II
Sistemi Operativi
DIGIP - a.a. 2015/16



Sistemi Operativi

(modulo di Informatica II)

I thread

Patrizia Scandurra

Università degli Studi di Bergamo

Sommario

- Generalità: il concetto di thread ed il multi-threading
- Modelli multithread
- Problematiche relative ai thread
- Esempi di librerie di thread
- Thread safety & condizioni di Bernstein

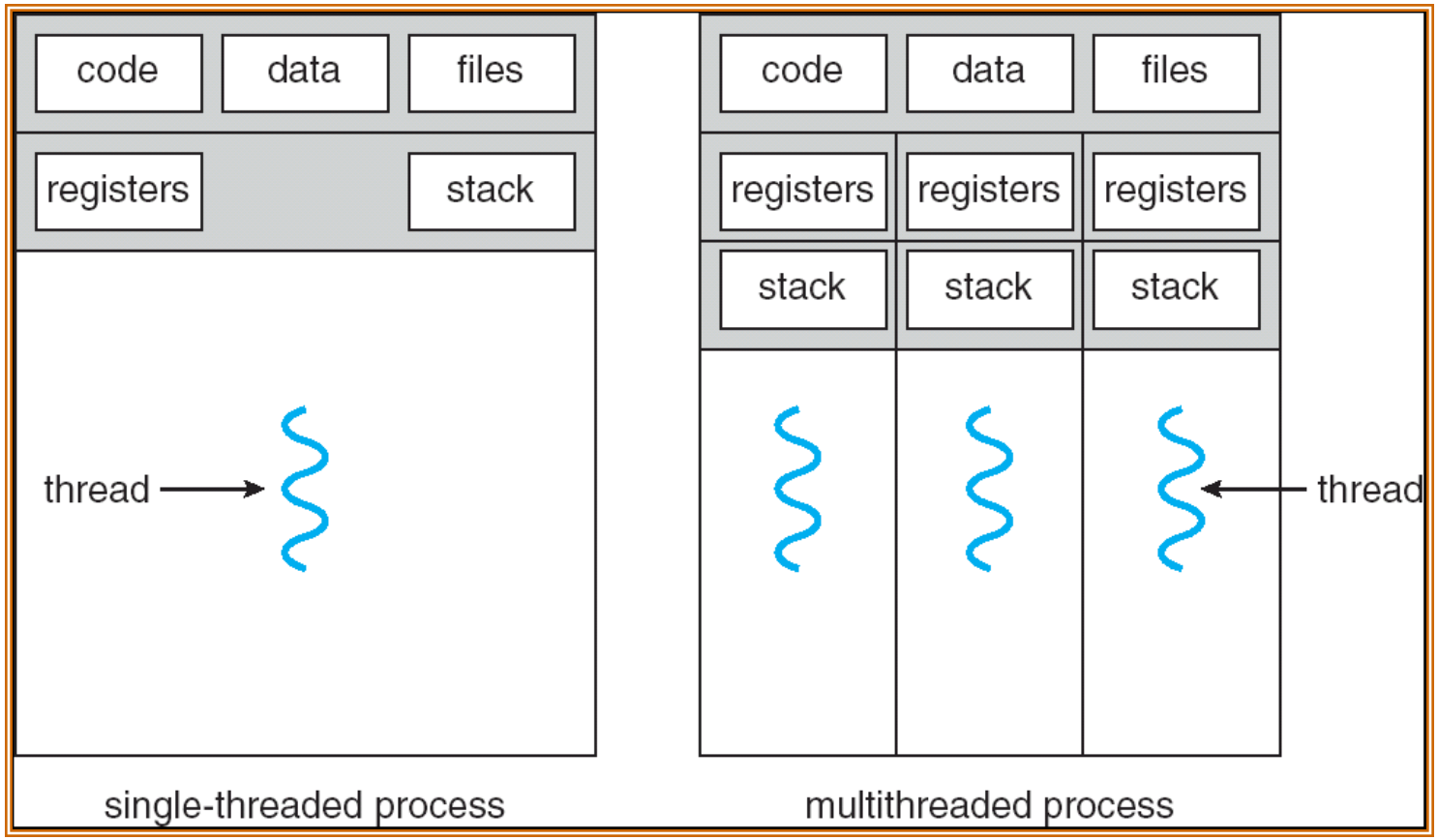
Sommario

- Generalità: il concetto di thread ed il multi-threading
- Modelli multithread
- Problematiche relative ai thread
- Esempi di librerie di thread
- Thread safety & condizioni di Bernstein

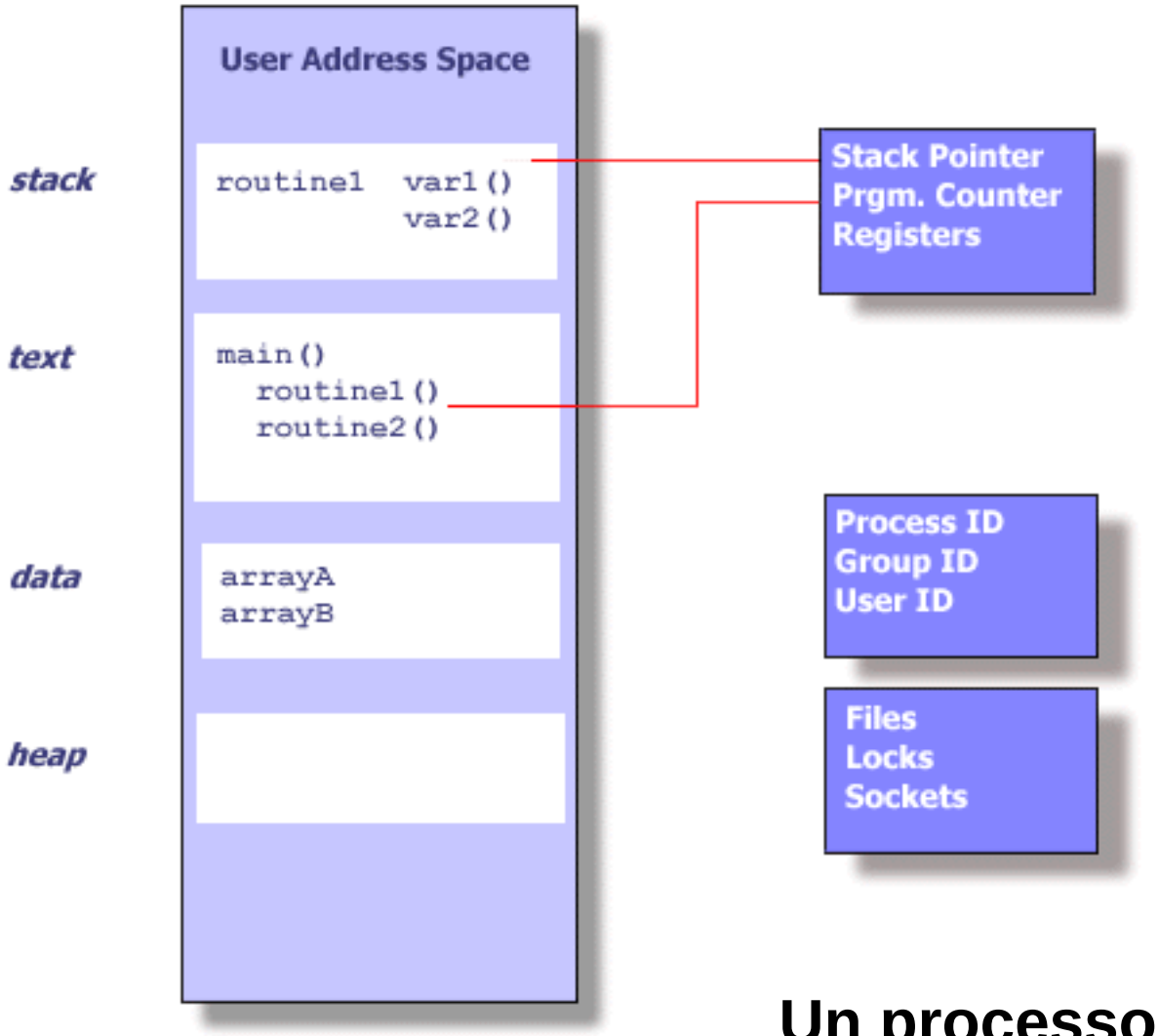
Concetto di Thread

- Anche chiamati ***lightweight process*** perché possiedono un contesto più snello rispetto ai processi
- **Flusso di esecuzione indipendente**
  - **interno ad un processo**
  - condivide lo spazio di indirizzamento con gli altri thread del processo
  - Rappresentato da un ***thread control block*** **(TCB**) che **punta al PCB del processo contenitore**
- Esempio di applicazione multi-thread: programma di elaborazione dei testi:
  - Thread per l'input da tastiera
  - Thread per la rappresentazione del testo
  - Thread per la correzione ortografica

Processi a singolo thread e multithread

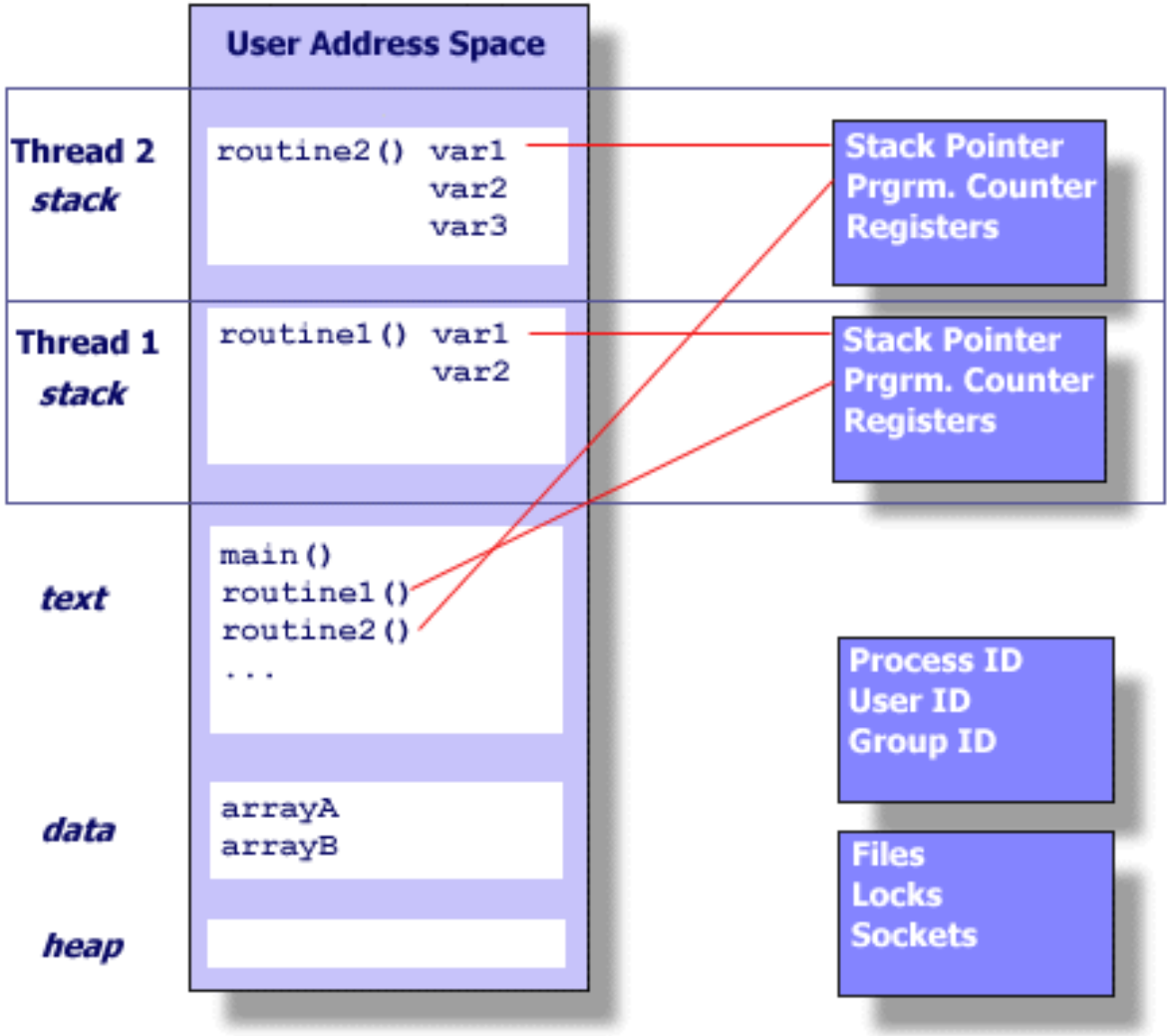


In Unix: Thread e Processi (1)



Un processo

In Unix: Thread e Processi (2)



Un thread all'interno di un processo

Contesto: Thread vs Processi

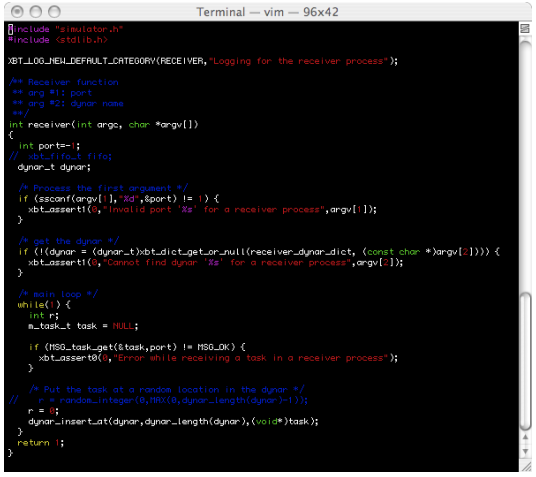
- **Contesto di un Thread**
  - stato della computazione (registri, stack, PC…)
  - attributi (schedulazione, priorità)
  - descrittore di thread (tid, priorità, segnali pendenti, …)
  - memoria privata (TSD)
- **Contesto di un processo**; tutto quello che è nel contesto di un thread, ed inoltre:
  - spazio di memoria
  - risorse private
    (con le corrispondenti tabelle dei descrittori)

Concorrenza (multithreading)

- ***Def.:*** ***Esecuzione di "task" multipli nello "stesso" tempo***
- In un **programma non-concorrente**, o **sequenziale**
  - In ogni momento è possibile interrompere il programma e dire esattamente quale task si stava eseguendo, quale era la sequenza di chiamate, ecc.
  - **Esecuzione deterministica**
- In un **programma concorrente**, si individuano un certo numero di task da eseguire come "flussi indipendenti"
  - Ogni task ha un compito specificico da portare avanti
  - I task possono comunicare tra loro: "flussi cooperanti"
  - **Regioni differenti del codice** eseguite allo stesso tempo
  - Lo stato di un programma ha più di una dimensione
  - **Esecuzione non deterministica**

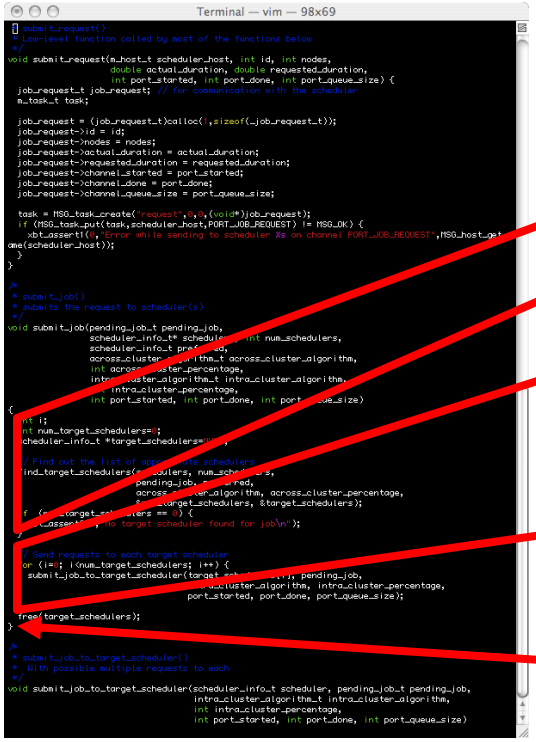
Programmi

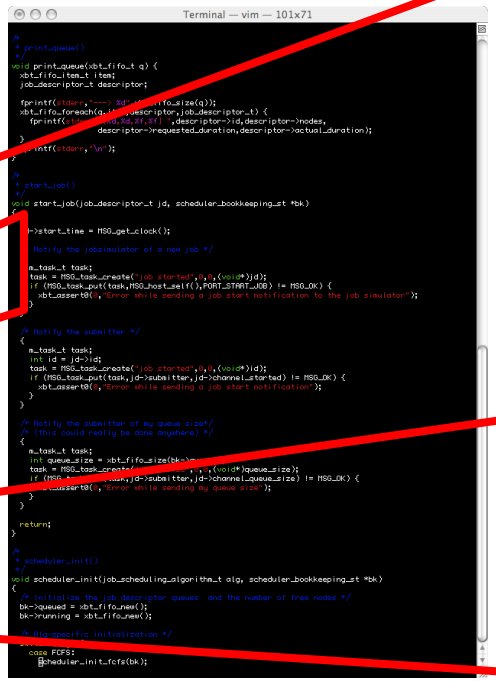
- Un programma è costituito da file/moduli/classi/funzioni

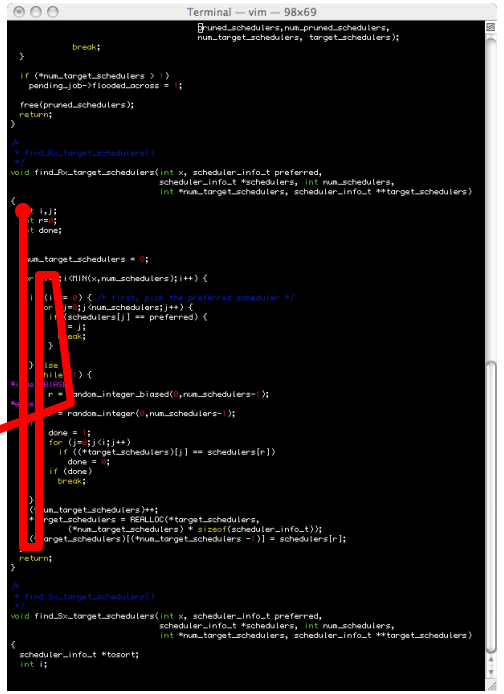


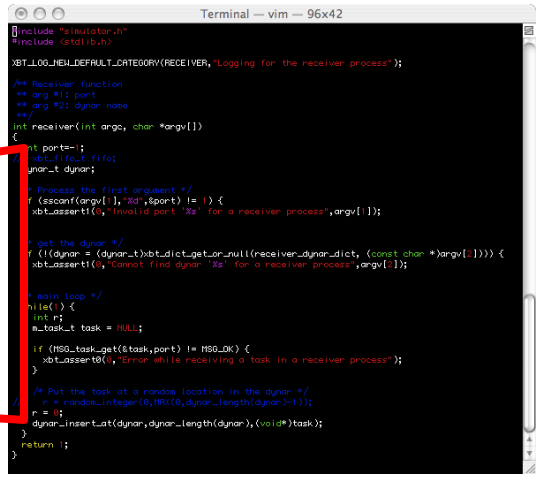
Programmi sequenziali

- Un programma **sequenziale**:
  un unico flusso rosso di esecuzione



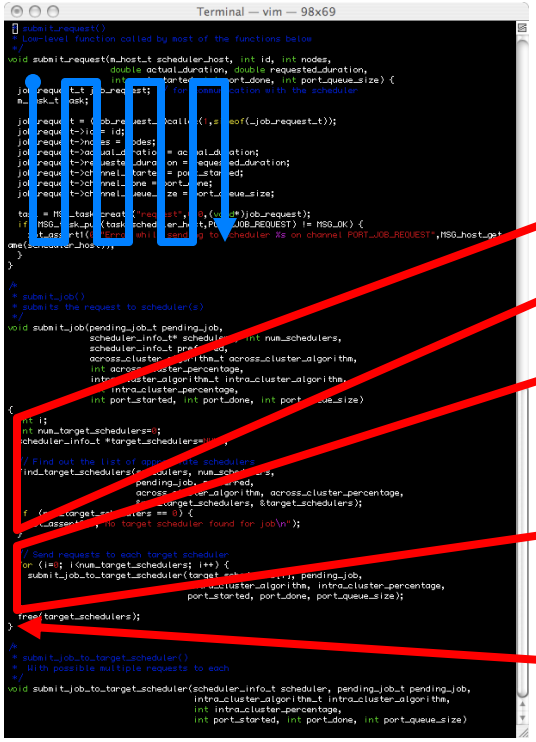


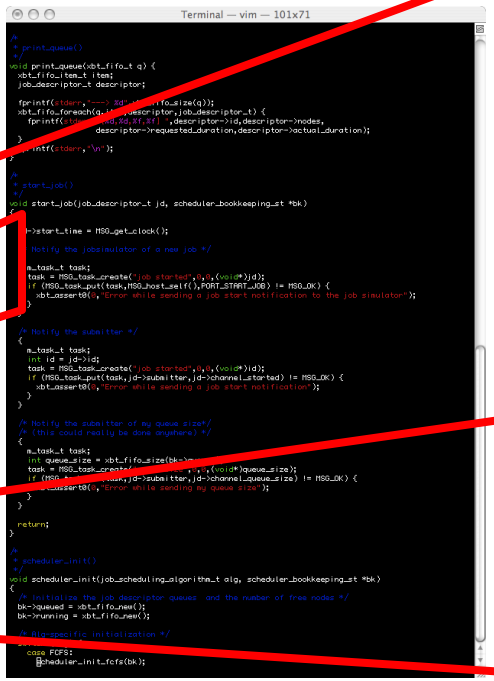


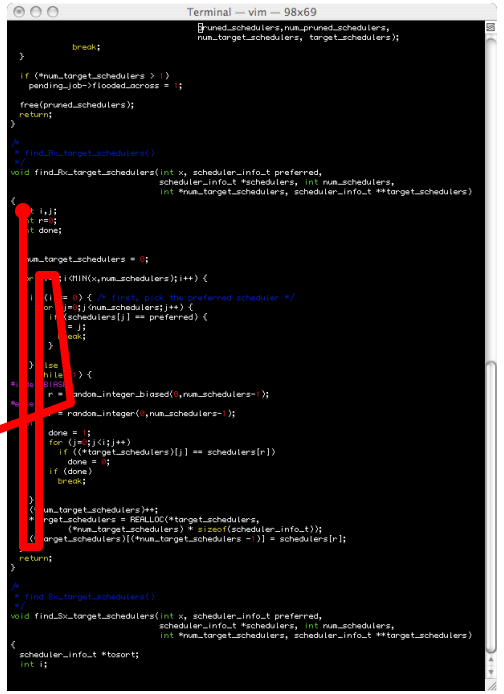


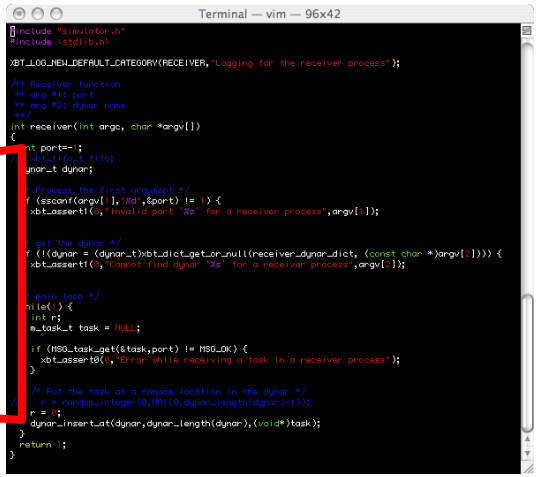
Programmi concorrenti

- Un programma **concorrente**
  - un task blu
  - un task rosso





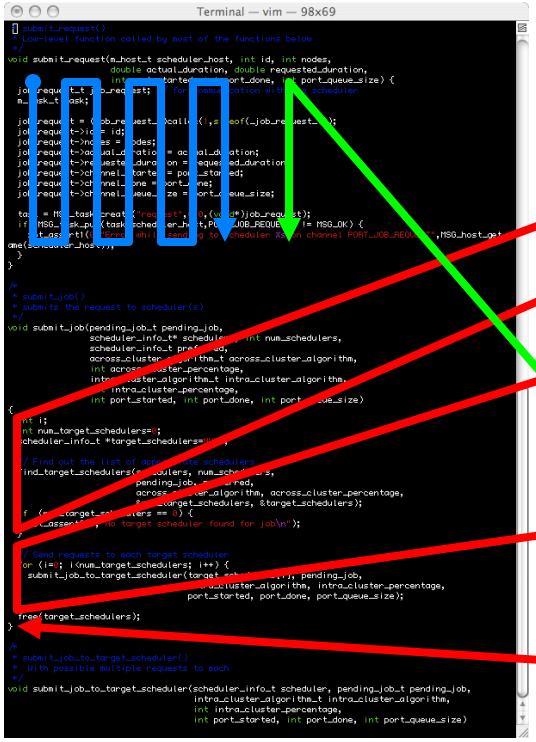


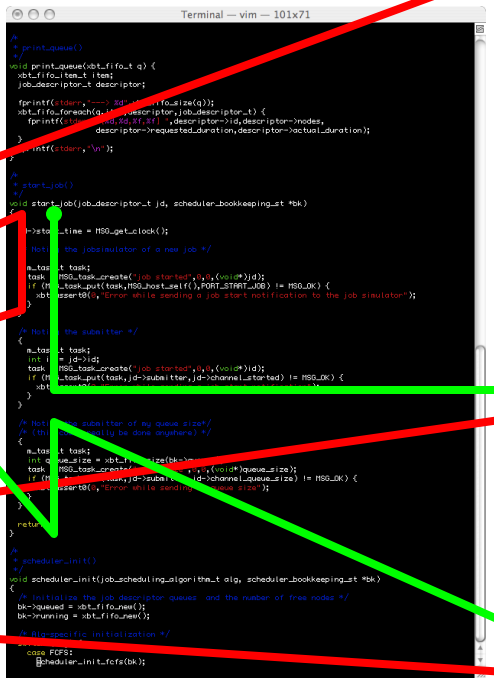


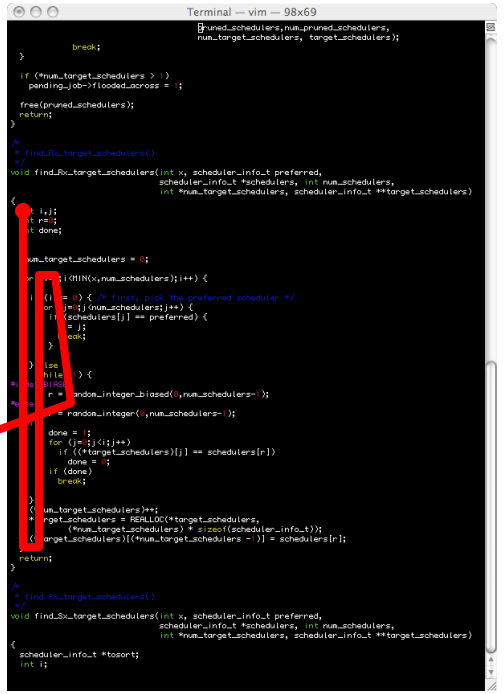
Programmi concorrenti

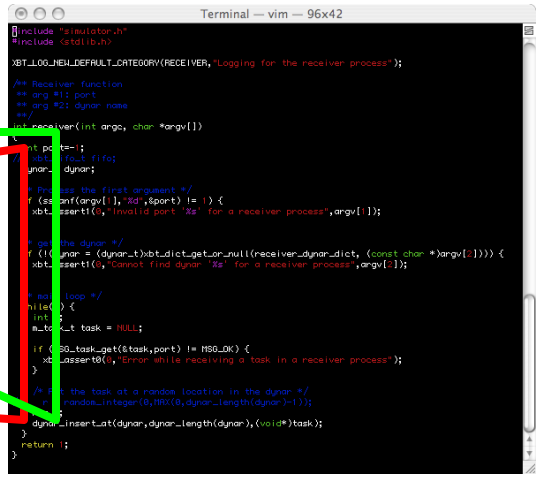
- oppure
  - un task blu
  - un task rosso
  - e un task verde

e così via …









Si ha vera concorrenza?

- Può essere verificato lanciando l'applicazione con due thread che ciclamente producono un simbolo diverso a video

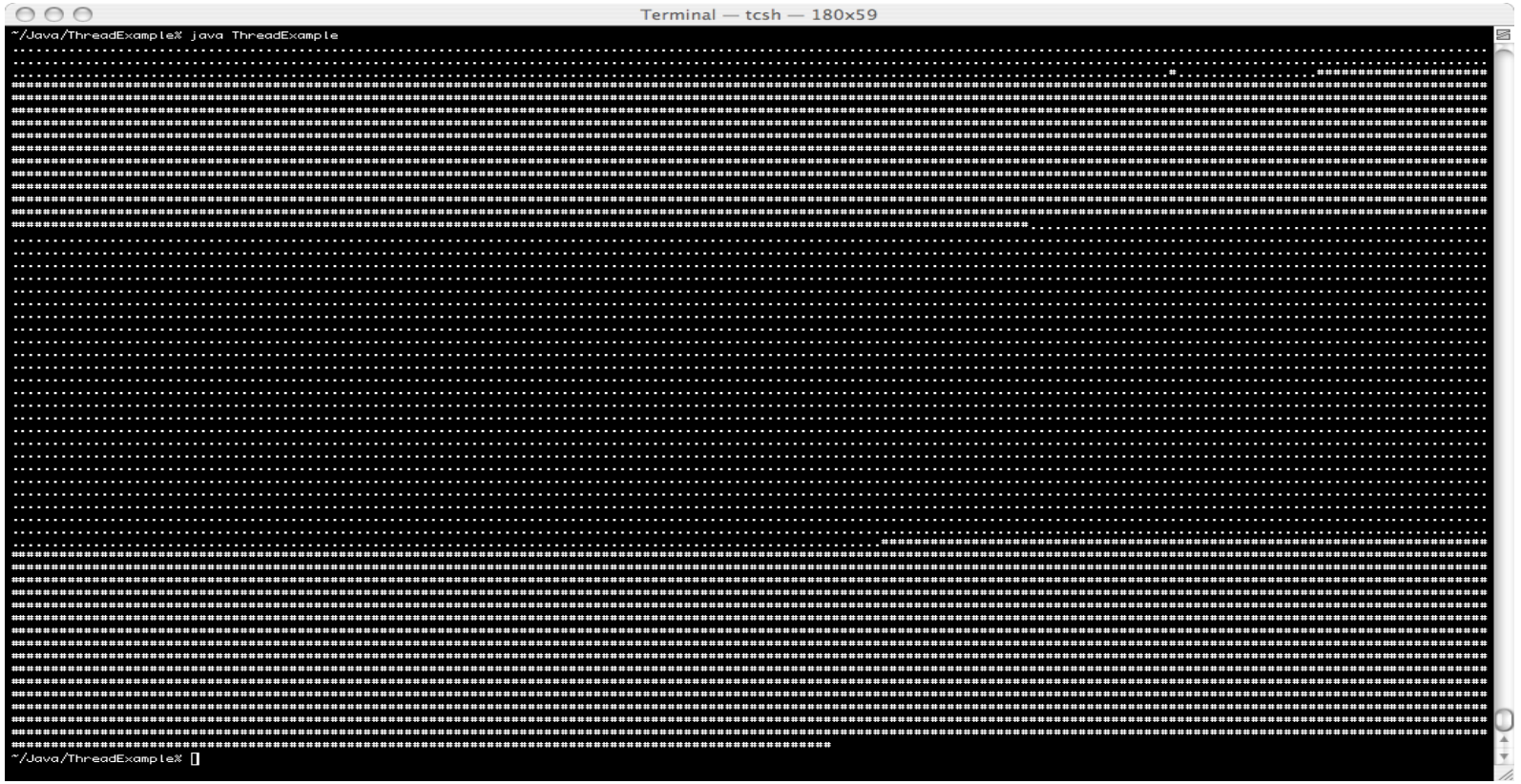
Output del nuovo processo "."
Output del main thread "#"

- e valutando il grado di "*interleaving*"

Esempio Esecuzione #1

- Su un Single Core

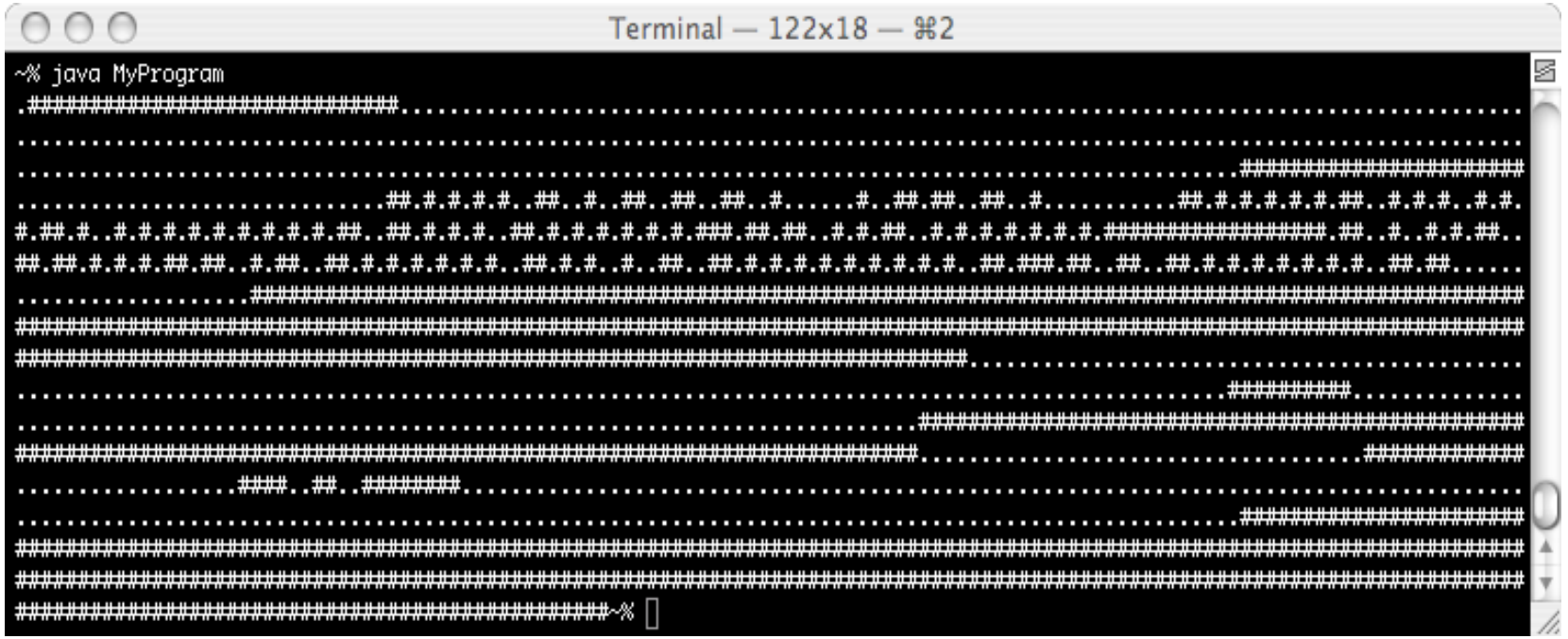
Concorrenza: **l'esecuzione dei thread è avvicendata nel tempo** (***interleaved***) perché la CPU può eseguire un solo thread alla volta!



Esempio Esecuzione #2

- Su un Dual Core Mac OS X

C'è vera concorrenza, ovvero **parallelismo**: i thread possono eseguire in parallelo, perché il sistema può assegnare thread diversi a ciascun core.



Esempio Esecuzione #3

- Su un Dual Core Mac OS X

L'esecuzione è non deterministica!

Perchè la concorrenza?

- Alcuni problemi (e quindi i programmi che li risolvono) sono per *natura* concorrenti
  - es., un video game in cui i caratteri vengono autonomamente generati dal computer
    - Ogni carattere è un task
  - es., un programma che suona musica e visualizza frattali
    - Il player è un task, la visualizzazione è un altro task
  - ecc..

Thread versus Processi (1) - esempio

- Un processo con un solo thread (heavyweight process) può eseguire un'attività alla volta
- **Problema**: supponiamo di implementare un web-server come un processo a singolo thread
  - Potremmo soddisfare uno solo dei numerosi client alla volta
  - Lasciando tutti gli altri bloccati
- **Soluzione A**: implementare il server come un processo che attende le richieste e avvia nuovi processi per processarle
  - Tuttavia creare un nuovo processo è oneroso (tempi e costi in termini di allocazione di nuove risorse)
  - In *Solaris* si stima che:
    - Creare un processo costa 30 volte di più che creare un nuovo thread
    - Un context switch di un processo costa 5 volte di più che quello di un thread

Thread versus Processi (2) - esempio

- Se un processo deve eseguire incarichi simili a quelli del primo è più conveniente creare un nuovo thread
- **Soluzione B**: implementare il server come un processo con un thread che attende le richieste e crea nuovi thread che le processano
  - Un implementazione simile è utilizzata per il server *RPC (Remote Procedure Call)*

Thread VS Processi

- Vantaggi e Svantaggi?

Vantaggi dei Thread (1)

- **Semplicità di comunicazione inter-thread**
  - Condivisione di risorse più naturale, programmazione più semplice
    - I thread condividono per default la memoria e le risorse del processo che li genera (cioè, molti thread nello stesso spazio di indirizzi)
    - I thread quindi comunicano tramite condivisione di informazioni nella memoria del processo contenitore
    - I processi comunicano invece tramite meccanismi di *scambio di messaggi* messi in atto esplicitamente (dal programmatore)
    - *Esempio*: in un editor di testo almeno 3 thread condividono lo stesso file: il paginatore, il thread che legge i caratteri battuti sulla tastiera e il correttore ortografico. Con 3 processi, l'effetto non sarebbe lo stesso: i processi richiederebbero l'uso esclusivo del file e "cambi di contesto"

Vantaggi dei Thread (2)

- **Efficienza e scalabilità**
  - Creazione e distruzione di thread in tempi rapidi
  - Context switch rapido e quindi scheduling più veloce dello scheduling dei processi
  - La comunicazione inter-thread è il più veloce dei meccanismi di comunicazione
  - Un programma multi-thread può continuare la computazione anche se uno dei suoi thread è bloccato (per esempio in attesa di I/O)
    - Rendere multi-thread un'applicazione reattiva ne migliora il tempo di risposta!
  - Su architetture multiprocessore i thread possono essere eseguiti in parallelo su distinti core di elaborazione
    - Maggior scalabilità

Svantaggi dei Thread

- **Difficoltà di ottenere risorse private**
  - per ottenere memoria privata all'interno di un thread esistono appositi meccanismi
- **Pericolo di interferenza maggiore**
  - la condivisione delle risorse accentua il pericolo di interferenza
  - gli accessi concorrenti alle risorse del processo contenitore devono essere sincronizzati per evitare interferenze (thread safety)

Complementarietà Thread / Processi

- Processo
  - "Unità di allocazione delle risorse"
- Thread
  - flusso di esecuzione indipendente ma esistente nel contesto di un processo
- Quali utilizzare?

  dipende, una scelta da valutare di caso in caso

Modelli di supporto dei Thread: User vs Kernel

I thread si possono implementare:

- **User-Level:** thread all'interno del processo utente gestiti da una libreria specifica (da un supporto run-time)
  - il kernel non è a conoscenza dell'esistenza dei thread
  - lo switch non richiede chiamate al kernel
- **Kernel-Level**: gestione dei thread affidata al sistema operativo tramite chiamate di sistema
  - gestione integrata processi e thread dal kernel
  - lo switch è provocato da chiamate al kernel

In entrambi i casi, deve esistere una relazione tra i thread a livello user e a livello kernel per permettere l'accesso alle risorse!

Sommario

- Generalità: il concetto di thread ed il multi-threading
- Modelli multithread
- Problematiche relative ai thread
- Esempi di librerie di thread
- Thread safety & condizioni di Bernstein

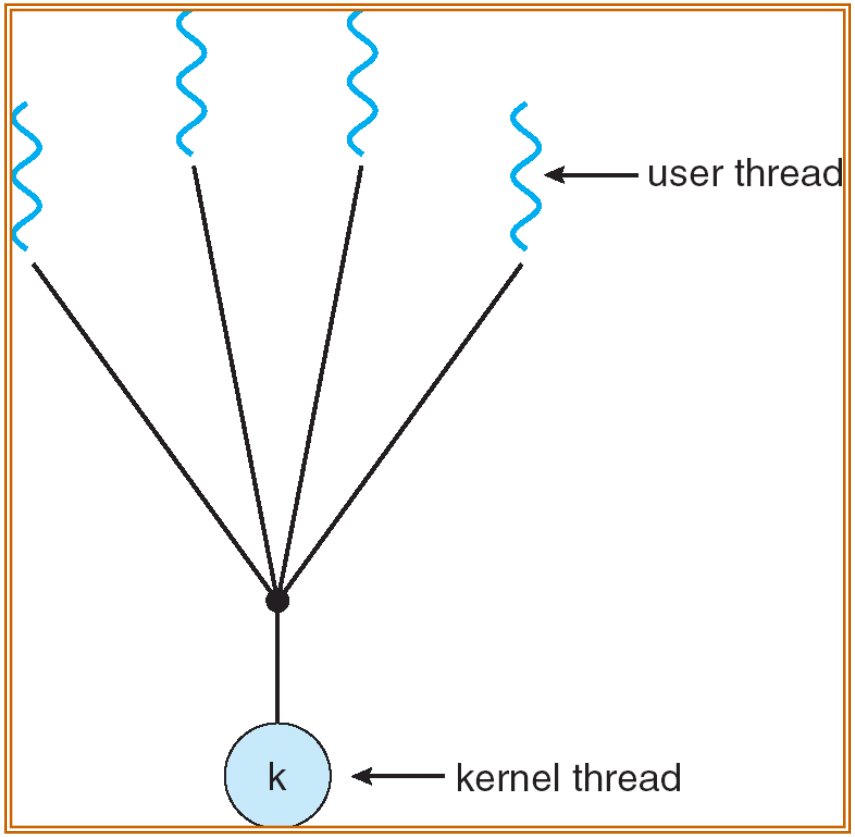
Modelli multi-thread (1)

- Esistono diversi tipi di relazione tra thread a livello utente e thread a livello kernel:
  - Molti-a-uno
  - Uno-a-uno
  - Molti-a-molti
  - A due livelli

Il modello molti-a-uno

- Il modello molti-a-uno riunisce molti thread di livello utente in un unico kernel thread

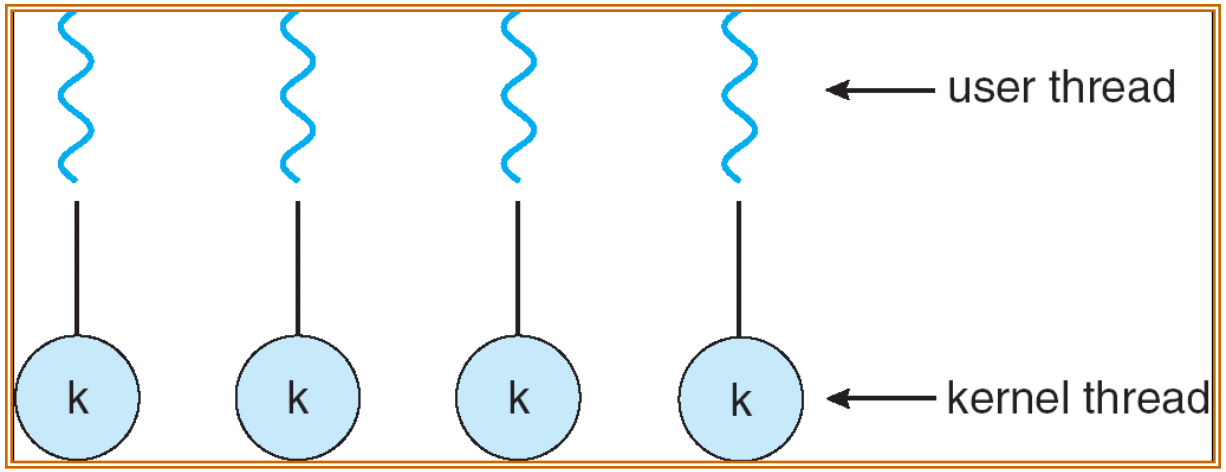
  - Gestione dei thread efficiente, perché gestita da una libreria a livello utente
  - Un solo thread alla volta può accedere al Kernel
  - Intero processo bloccato se un thread invoca una chiamata di sistema bloccante
  - Impossibile eseguire thread in parallelo su architetture multi-core
  - Esempi di **librerie di thread**:
    - Solaris Green Threads (adottata nelle prime versioni di Java su SOLARIS)
    - GNU Portable Threads



Il modello uno-a-uno

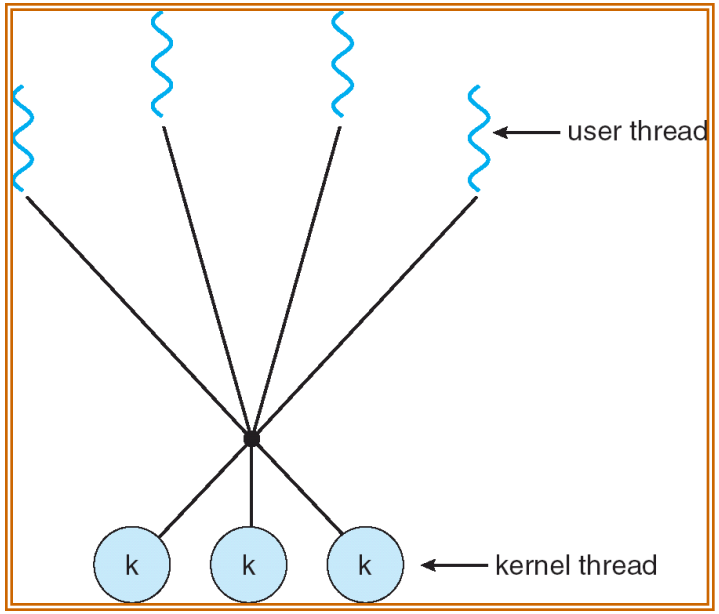
- Il modello uno-a-uno mappa ciascun thread utente con un thread kernel separato
- Maggiore concorrenza (possibilità di eseguire i thread in parallelo su sistemi multi-core e le chiamate bloccanti non bloccano tutti i thread)
- La creazione di molti thread a livello kernel compromette però le prestazioni dell'applicazione
- Tipicamente le realizzazioni di questo modello limitano il numero di thread a livello kernel

- Esempi:
  - Windows
  - Linux
  - Solaris 9



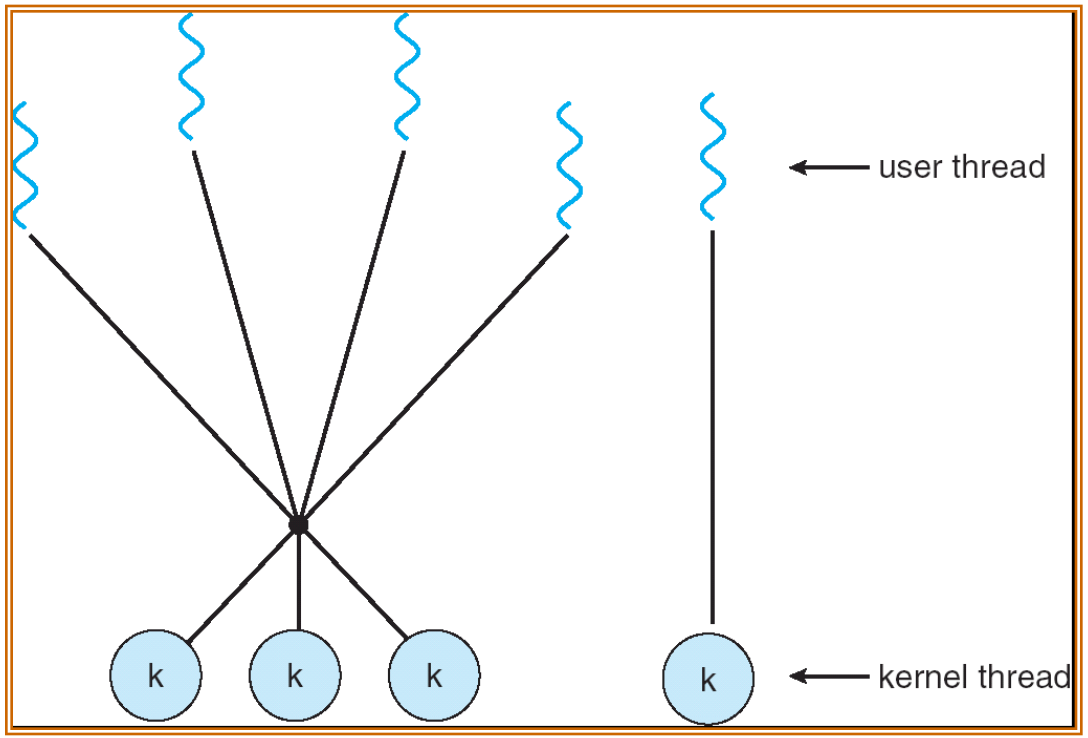
Il modello molti-a-molti

- Permette di aggregare molti thread a livello utente verso un numero più piccolo o equivalente di kernel thread
- Risolve le limitazioni dei modelli precedenti:
  - Permette al SO di creare un numero sufficiente di kernel thread
  - Il numero max di processi a livello kernel può essere personalizzato in base all'architettura o applicazione specifica
    - n_max maggiore per architettura multicore
- Esempi:
  - Solaris, versioni precedenti alla 9
  - True64Unix
  - Windows NT/2000 con il pacchetto ThreadFiber



Il modello a due livelli

- Simile al modello molti-a-molti, ma permette anche di associare un thread di livello utente ad un kernel thread
- Esempi:
  - IRIX
  - HP-UX
  - Tru64 UNIX
  - Solaris vesioni precedenti alla 9



Vantaggi/Svantaggi: User Level

- Vantaggi:
  - Lo switch non coinvolge il kernel, e quindi **non ci sono cambiamenti della modalità di esecuzione**
  - Maggiore **libertà nella scelta dell'algoritmo di scheduling** che può anche essere personalizzato
  - Poichè le chiamate possono essere raccolte in una libreria, c'è **maggiore portabilità tra SO**
- Svantaggi:
  - **Una chiamata al kernel può bloccare tutti i thread di un processo**, indipendentemente dal fatto che in realtà solo uno dei suoi thread ha causato la chiamata bloccante
  - **In sistemi** a multiprocessore simmetrico (**SMP**) **due processori non risulteranno mai associati a due thread del medesimo processo**

Vantaggi/Svantaggi: Kernel Level

- Vantaggi:
  - il kernel può **eseguire più thread** dello stesso processo anche su più processori
  - **il kernel stesso può essere scritto multithread**

- Svantaggi:
  - **lo switch coinvolge chiamate al kernel** e questo comporta un costo maggiore
  - **l'algoritmo di scheduling è meno facilmente personalizzabile**
  - **meno portabile**

Sommario

- Generalità: il concetto di thread ed il multi-threading
- Modelli multithread
- Problematiche relative ai thread
- Esempi di librerie di thread
- Thread safety & condizioni di Bernstein

Librerie dei Thread

- Fornisce una API al programmatore per creare e gestire i thread
- Può essere implementata come:
  - **Libreria collocata a livello utente**
    - Codice e strutture dati per la libreria risiedono nello spazio utente
    - Invocare una funzione della libreria non significa invocare una chiamata di sistema
  - **Libreria collocata a livello kernel**
    - Codice e strutture dati per la libreria risiedono nello spazio del kernel
    - Invocare una funzione della libreria significa invocare una chiamata di sistema
- Tre principali (più popolari) librerie di thread:
  - *POSIX Pthread* (sia a livello utente che a livello kernel)
  - *Win32 thread* (a livello kernel per sistemi Windows)
  - *Java thread* (dipende dal sistema operativo ospitante, può essere implementata in Pthreads su Unix e Linux, in Win32 su Windows, ecc..)

Pthread

- Uno **standard POSIX (IEEE 1003.1c) API** per la creazione e la sincronizzazione dei thread
- **Fornisce una specifica** per il comportamento della libreria dei thread, **non un'implementazione**!
  - i progettisti di sistemi operativi possono implementare la specifica nel modo che desiderano
- Frequente nei sistemi operativi UNIX-like (Solaris, Linux, Mac OS X), implementazioni da terze parti anche per Windows.

Esempio pthread – Calcolo sommatoria

```
1.  #include <pthread.h>
2.  #include <stdio.h>

3.  int sum; /* this data is shared by the thread(s) */

4.  void *runner(void *param); /* the thread function */

5.  int main(int argc, char *argv[])
6.  {
7.  pthread_t tid; /* the thread identifier */
8.  pthread_attr_t attr; /* set of attributes for the thread */

9.  if (argc != 2) {
10.         fprintf(stderr,"usage: a.out <integer value>\n");
11.         /*exit(1);*/
12.         return -1;
13. }

14. if (atoi(argv[1]) < 0) {
15.         fprintf(stderr,"Argument %d must be non-negative\n",atoi(argv[1]));
16.         /*exit(1);*/
17.         return -1;
18. }

19. /* get the default attributes */
20. pthread_attr_init(&attr);
```

`pthread_t` specifica l'identificatore di un thread

`pthread_attr` specifica gli attributi di un thread (e.g. proprietà per lo scheduling

pthread_attr_init inizializza gli attributi di un thread ai valori di default

Esempio pthread – Calcolo sommatoria

```
1.  /* create the thread */
2.  pthread_create(&tid,&attr,runner,argv[1]);

3.  /* now wait for the thread to exit */
4.  pthread_join(tid,NULL);

5.  printf("sum = %d\n",sum);
6.  }

7.  /**
8.   * The thread will begin control in this function
9.   */
10. void *runner(void *param)
11. {
12. int i, upper = atoi(param);
13. sum = 0;

14.         if (upper > 0) {
15.                     for (i = 1; i <= upper; i++)
16.                                 sum += i;
17.         }

18.         pthread_exit(0);
19. }
```

`pthread_create` crea un user thread al qualle associa gli attributi inizializzati e la funzione runner. Alla funzione viene passato l'argomento argv[1]

`pthread_join` metta in pausa il thread principale fino alla terminazione del thread con identificatore `tid`. Se il secondo parametro non è nullo viene usato per restituire il valore di ritorno del thread

`pthread_exit` termina il thread

FINE LEZIONE

Terminologia

- **Job**: insieme di processi

- **Processo** (o task): contenitore di risorse
  - che il kernel tratta atomicamente per allocare risorse
  - ha almeno un thread di esecuzione

- **Thread**: unità di esecuzione schedulate dal kernel
  - vivono all'interno di un processo
  - possono accedere a tutte le risorse del processo
  - possiedono uno stato (contesto) di computazione
  - e un'area privata per le variabili locali

- **Fiber**: thread leggero gestito solo nello spazio utente

Problematiche relative ai thread

- Creazione, avvio e cancellazione dei thread
- La gestione dei segnali
- Gruppi di thread
- Dati specifici dei thread
- Attivazione dello schedulatore

Creazione di thread: le chiamate di sistema fork() ed exec()

- Se un thread in un processo effettua la chiamata di sistema **fork(), il nuovo processo contiene un duplicato del solo thread che ha invocato la fork() oppure di tutti i thread?**
  - Alcuni sistemi Unix hanno due versioni di fork() per entrambe le semantiche a seconda dell'applicazione

- La chiamata **exec()** da parte di un thread funziona allo stesso modo
  - il programma specificato come parametro **rimpiazzerà l'intero processo**, **inclusi tutti i thread**

Cancellazione dei thread

- È l'atto di terminare un thread prima che abbia completato l'esecuzione
- La cancellazione del thread può avvenire in due differenti scenari:
  - **Cancellazione asincrona**: un thread termina immediatamente il thread target
  - **Cancellazione differita**: il thread target può periodicamente controllare se deve terminare

La gestione dei segnali (1)

- Si usano nei sistemi UNIX-like per notificare un evento ad un processo
- I segnali vengono elaborati secondo questo **schema**:
  1. il verificarsi di un particolare evento genera un segnale
  2. il segnale generato viene consegnato ad un processo
  3. il segnale viene gestito
- I segnali possono essere gestiti attraverso:
  - il gestore predefinito dello specifico segnale (esiste per ogni segnale)
  - una funzione di gestione definita dall'utente (override del gestore predefinito)

La gestione dei segnali (2)

I segnali possono essere:

- *Sincroni*: sono inviati allo stesso processo che ne ha causato la generazione
  - Esempi: divisione per 0, accesso illegale alla memoria
- *Asincroni*: sono inviati ad un processo differente da quello che ne ha causato la generazione
  - Esempi: intercettazione di una combinazione di tasti (ctrl + c), scadenza di un timer

La gestione dei segnali (3)

- Per i processi a singolo thread la gestione dei segnali è semplice
- Per i processi multithread ci sono diverse **opzioni di consegna**:
  - Consegnare il segnale al thread cui il segnale viene applicato
  - Consegnare il segnale ad ogni thread del processo
  - Consegnare il segnale ad alcuni thread del processo
  - Designare un thread specifico che riceva tutti i segnali per il processo
- Per i segnali asincroni si usa sempre la prima opzione
- Per la seconda e terza opzione invece è possibile istruire i processi su quali segnali accettare e quali ignorare
- In sistemi Unix-like: le chiamate per inviare segnali specificano l'id del processo o thread a cui far recapitare il segnale
  - kill(pid_t pid, int signal)
  - Pthread_kill(pthread_t tid, int signal)
- In Windows: emulazione dei segnali con chiamate di procedura asincrone (APC) rivolte solo a singoli thread

Gruppi di thread

- L'esempio del web-server che crea un thread per ogni richiesta presenta diversi **problemi**:
  - Esso non pone un limite massimo al numero di thread creabili
  - Creare un nuovo thread ogni volta è meno costoso che creare un processo ma ha comunque un costo
- **Soluzione**: creare un gruppo di thread (pool) all'avvio del processo e assegnarli un lavoro quando richiesto
  - Al completamento del lavoro il thread torna nel gruppo d'attesa
- **Vantaggi**:
  - Servire la richiesta all'interno di un thread esistente è tipicamente più veloce che attendere la creazione di un thread
  - Un pool di thread limita il numero di thread esistenti in contemporanea
  - Il numero massimo può essere adattato a runtime in architetture raffinate

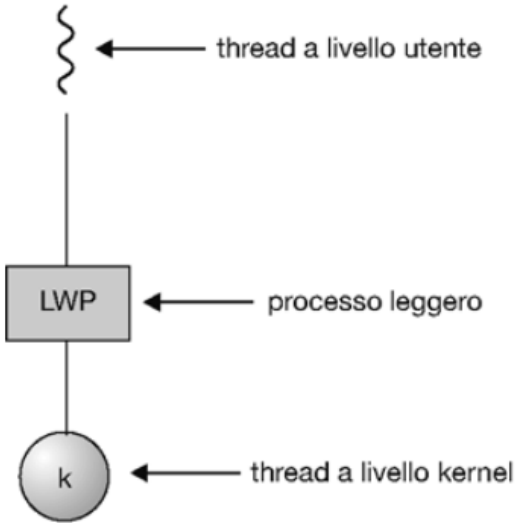
Dati specifici dei thread

- Permette ad ogni thread di avere la **propria copia di dati**
- Utile quando non si ha il controllo sul processo di creazione dei thread
  - ad esempio quando si utilizza un gruppo di thread per gestire transazioni di un server di servizi
  - l'id univoco di una transazione và memorizzato nella zona di memoria privata del thread che la esegue

Attivazione dello schedulatore (1)

- Le applicazioni multi-thread con modello molti-a-molti o a due livelli richiedono una **comunicazione con il kernel**
  - per variare dinamicamente il numero di kernel thread allocati
  - per far sì che il kernel informi l'applicazione di certi eventi (segnali), come il blocco di un thread dell'applicazione (ad es. per un evento di I/O)

- In molti di questi sistemi esiste una struttura dati posta tra i thread kernel e i thread utente: **lightweight process (LWP)**
  - Le LWP sono come dei ***processori virtuali*** su cui schedulare i vari thread utente
  - Il blocco di un thread kernel (per ese. per I/O) causa il blocco del LWP
    - Di conseguenza si bloccano i thread utente schedulati sul LWP
  - È quindi necessario un LWP per ogni chiamata di sistema concorrente bloccante
    - ad es. se un'applicazione ha bisogno di fare 5 letture da file servono 5 LWP



Attivazione dello schedulatore (2)

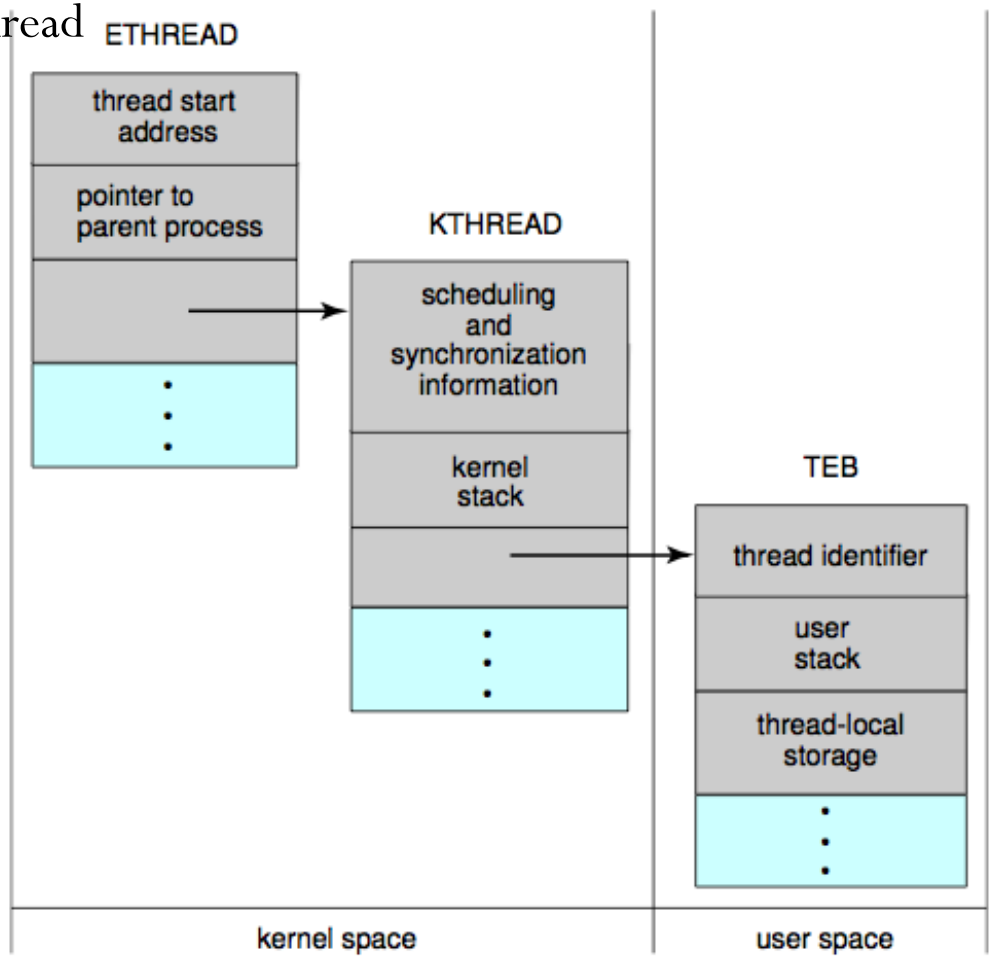
- Uno dei modelli di comunicazione tra libreria a livello utente e il kernel
- I thread kernel possono comunicare eventi alla libreria dei thread tramite le **upcall** (**chiamate al thread**)
- Le upcall sono gestite a livello utente con **un gestore di upcall** in esecuzione "just in time" (cioè quando avviene una comunicazione dal kernel all'applicazione) **su di un LWP**
  - Effettua lo scheduling dei thread utente sul processore virtuale LWP

I thread di Windows (da Win98 fino a Win7) (1)

- Implementa la **mappatura uno-a-uno**
- Ogni thread contiene:
  - Un idetificatore del thread
  - Un set di registri
  - Due stack: User e kernel stack separati
  - Un'area di memoria privata
- Il set di registri, gli stack e l'area di deposito sono noti come il **contesto del thread**

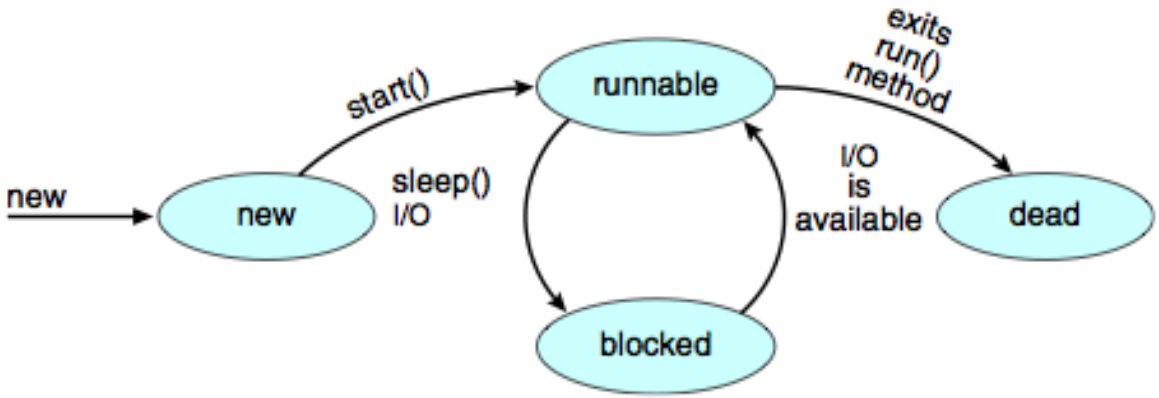
I thread di WindowsXP (2)

- Strutture dati primarie di un thread:
  - ETHREAD – blocco di esecuzione del thread
  - KTHREAD – blocco di kernel del thread
  - TEB – blocco di ambiente del thread



I thread di Java

- I thread di Java sono **gestiti dalla JVM**
- I thread di Java possono essere creati tramite:
  - L'estensione della classe Thread
  - L'interfaccia Runnable
- Una volta creati:



Li tratteremo nei dettagli durante le ore di esercitazione in laboratorio!

I thread di Linux

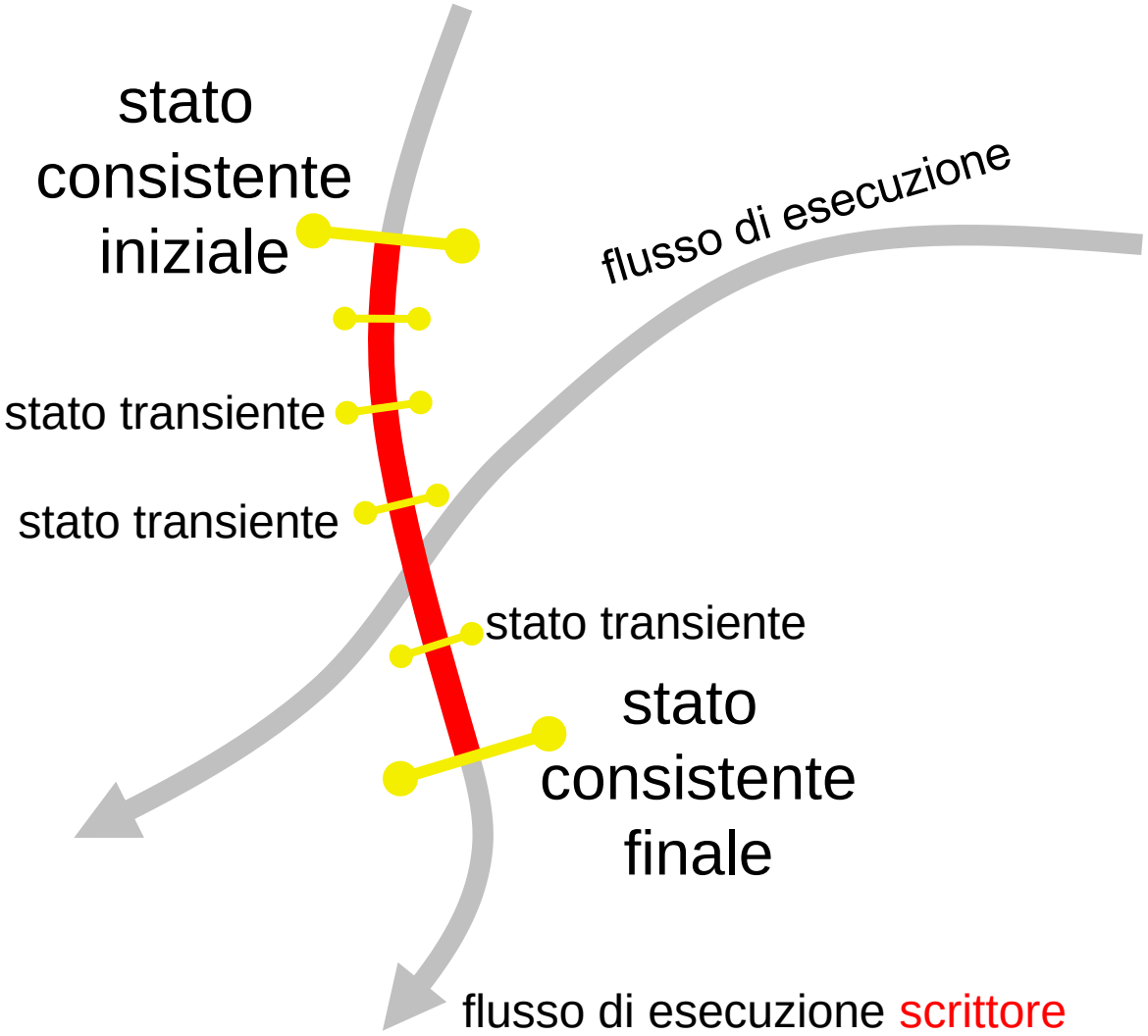
- Linux li definisce come *task piuttosto che thread*
- La creazione di un thread avviene attraverso la chiamata di sistema **clone()**
  - che **permette di stabilire il "grado di condivisione"** tramite flag parametri
  - **se nessun flag è impostato, non c'è alcuna condivisione e l'effetto di clone() è simile a quello della fork()**

flag	meaning
CLONE_FS	File-system information is shared.
CLONE_VM	The same memory space is shared.
CLONE_SIGHAND	Signal handlers are shared.
CLONE_FILES	The set of open files is shared.

Stati transienti ed interferenze

- Le strutture dati accedute da un programma multithread sono oggetto di aggiornamenti da parte di più thread
- Gli **aggiornamenti non avvengono atomicamente**, ma sono decomponibili in varie operazioni di modifica intermedie e di una certa durata
- Durante il transitorio la struttura dati "perde significato" (inconsistente), e passa per una serie di ***stati transienti***
- Un tale stato **non dovrebbe essere visibile** a thread diversi dal thread che esegue l'aggiornamento, altrimenti si generano **interferenze**

Origine dei fenomeni di interferenza



Interferenza

- Si ha **interferenza** in presenza di
  - due o più flussi di esecuzione
  - almeno un flusso di esecuzione esegue scritture (aggiorna la struttura dati!)
- **Perché**
  - un flusso esegue un cambio di stato dell'area di memoria in maniera non atomica
  - gli stati *transienti* che intercorrono tra quello iniziale a quello finale sono visibili a flussi di esecuzione diversi da quello che li sta producendo

Esempio di Interferenza (1)

- La disponibilità di un volo di una compagnia aerea è memorizzata in **POSTI=1**. Due signori nel medesimo istante ma da due postazioni distinte, chiedono rispettivamente di prenotare l'ultimo posto e di disdire la prenotazione già effettuata.
- Le due richieste vengono tradotte in queste sequenze di istruzioni elementari indivisibili:

procedure Prenota
begin
  $R_a \leftarrow$ POSTI - 1;
  POSTI $\leftarrow R_a$;
end

procedure Disdici
begin
  $R_b \leftarrow$ POSTI + 1;
  POSTI $\leftarrow R_b$;
end

Esempio di Interferenza (2)

- Inizialmente POSTI=1
- L'esecuzione concorrente da luogo ad una qualsiasi delle possibili sequenze di interleaving
- Consideriamo un campione di tre sequenze:

$R_a \leftarrow$ POSTI - 1;	$R_a \leftarrow$ POSTI - 1;	$R_b \leftarrow$ POSTI + 1;
$R_b \leftarrow$ POSTI + 1;	POSTI $\leftarrow R_a$;	$R_a \leftarrow$ POSTI - 1;
POSTI $\leftarrow R_b$;	$R_b \leftarrow$ POSTI + 1;	POSTI $\leftarrow R_a$;
POSTI $\leftarrow R_a$;	POSTI $\leftarrow R_b$;	POSTI $\leftarrow R_b$;
(POSTI=0)	(POSTI=1)	(POSTI=2)
ERRORE	OK	ERRORE

Thread Safeness

- *Def.: Programma thread-safe*: Un programma si dice ***thread safe*** se garantisce che **nessun thread possa accedere a dati in uno stato inconsistente**
- Un **programma thread safe protegge l'accesso alle strutture** in stato inconsistente da parte di altri thread **per evitare interferenze**
  - costringendoli in attesa (passiva) del suo ritorno in uno stato consistente
- Il termine *thread safeness* si applica anche a librerie ed a strutture dati ad indicare la loro predisposizione ad essere inseriti in programmi multithread

Dominio e Rango

- Indichiamo con A, B, … X, Y, … un'area di memoria
- **Una istruzione *i***
  - dipende da una o più aree di memoria che denotiamo domain(*i*), ovvero dominio di *i*
  - altera il contenuto di una o più aree di memoria che denotiamo range(*i*) di *i*, ovvero rango di *i*
- Ad es. per la procedura **P**

procedure P
begin

$X \leftarrow A + X;$

$Y \leftarrow A * B;$

end

domain(**P**) = {A, B, X}

range(**P**) = {X, Y}

Condizioni di Bernstein

Quando è lecito eseguire
concorrentemente due istruzioni i_a e i_b ?

- se valgono le seguenti condizioni, dette Condizioni di Bernstein:

1. $\text{range}(i_a) \cap \text{range}(i_b) = \emptyset$
2. $\text{range}(i_a) \cap \text{domain}(i_b) = \emptyset$
3. $\text{domain}(i_a) \cap \text{range}(i_b) = \emptyset$

Condizioni di Bernstein (2)

- Si osservi che non si impone alcuna condizione su $\text{domain}(i_a) \cap \text{domain}(i_b)$
- Sono banalmente estendibili al caso di tre o più istruzioni

- Esempi di violazione per le due istruzioni:
  - *X* ← *Y* + *1;* *X* ← *Y* – *1;* *(violano la 1.)*
  - *X* ← *Y* + *1;* *Y* ← *X* - *1* ; *(violano la 2. e la 3.)*
  - *scrivi X;* *X* ← *X* + *Y;* *(violano la 3.)*

Effetti delle violazioni

- Quando un insieme di istruzioni soddisfa le condizioni di Bernstein, il loro esito complessivo sarà sempre lo stesso
  - **indipendentemente dall'ordine e dalle velocità relative** con cui vengono eseguite
  - ovvero, sarà sempre equivalente ad una loro esecuzione seriale

- Al contrario, **in caso di violazione, gli errori dipendono dall'ordine e dalle velocità relative** generando il fenomeno dell'*interferenze*

Il Programmatore e gli errori dipendenti dal tempo

- Un programma che (implicitamente od esplicitamente) basa la propria correttezza su ipotesi circa
  - **la velocità relativa dei vari processori virtuali** o
  - sulla **sequenza di interleaving eseguita**,

  è **scorretto**
- Esiste una sola assunzione che possono fare i programmatori sulla velocità dei processori virtuali…

Assunzione di Progresso Finito

Tutti i processori virtuali hanno una velocità finita non nulla

- Questa assunzione è **l'unica** che si può fare sui processori virtuali e sulle loro velocità relative

Thread Safeness & Condizioni di Bernstein

- **Dato un programma multithread, quali strutture dati bisogna proteggere per garantire la thread safeness?**

  Tutte le strutture dati oggetto di accessi concorrenti che violano le condizioni di Bernstein

  in altre parole,

  le strutture dati oggetto di scritture concorrenti da parte di due o più thread