MODELLO DI ESECUZIONE NEI S.O.

MODELLO CONCORRENTE

Il **modello concorrente** è basato sul concetto di **processo.**

Processo:
- Attività controllata da un programma che si svolge su un processore (o più semplicemente un programma in esecuzione)
- Un processo non è un programma
    - Un programma specifica la sequenza di esecuzione di un insieme di istruzioni
    - Un processo rappresenta il modo in cui un programma viene eseguito nel tempo

Più processi possono eseguire lo stesso programma, ma ogni istanza viene considerata come un processo separato anche se possono condividere lo stesso codice mentre (in generale) i dati, l'immagine e lo stato rimangono separati.

In questo modello i processi vengono eseguiti in parallelo con parallelismo reale (multiprocessori) o apparente (multiprogrammazione su una singola CPU).

Il sistema operativo tiene traccia di tutti i processi del sistema tramite la **tabella dei processi**, le entry della tabella, **Process Control Block (PCB)** mantengono informazioni sui singoli processi come, a esempio, lo stato dei processi.

PROCESSI IN UNIX

Ogni processo, identificato dal **PID**, ha uno spazio di indirizzamento separato e quindi non vede le zone di memoria dedicate agli altri processi.
Un processo ha tre segmenti:
- **Stack**
- **Dati** (Heap e Dati statici)
- **Codice**.

L'indirizzamento è virtuale: codice, dati e heap sono memorizzati nella parte iniziale della memoria virtuale, lo stack nella parte finale.

CREAZIONE DI UN PROCESSO

La **fork** alloca spazio per il processo figlio
- nuove tabelle per la memoria virtuale
- nuova memoria segmenti dati e stack
- dati, stack e u-structure vengono copiati da quelli del padre preservando file aperti, UID, GID, ...
- il codice viene condiviso

La **execve** non crea nessun nuovo processo: semplicemente, i segmenti dati, codice e stack vengono rimpiazzati con quelli del nuovo programma.

CONTEXT SWITCH

Il kernel vieta context switch arbitrari per mantenere la consistenza delle sue strutture dati.
Il controllo può passare da un processo all'altro in quattro possibili scenari:
- Quando un processo si sospende
- Quando termina
- Quando torna a modo user da una chiamata di sistema ma non è più il processo a più alta priorità
- Quando torna a modo user dopo che il kernel ha terminato la gestione di un'interrupt a non è più il processo a più alta priorità.

In tutti questi casi il kernel lascia la decisione di quale processo è da eseguire allo scheduler.

MULTI-THREADING

I processi dei Sistemi Operativi hanno le seguenti caratteristiche:

- **Rappresentano unità di allocazione risorse:**
  Codice eseguibile, dati allocati staticamente (variabili globali) ed esplicitamente (heap), risorse mantenute dal kernel (file, I/O, workind dir), di controllo di accesso…

- **Rappresentano unità di esecuzione:**
  Un percorso di esecuzione attraverso uno o più programmi: stack di attivazione (variabili locali), stato (running, ready, waiting, …), priorità, parametri di scheduling…

Queste due componenti si possono considerare in maniera separata.

Un thread è un'unità di esecuzione:
- Program counter, insieme di registri
- Stack del processore
- Stato di esecuzione

Un thread condivide con gli altri suoi pari un'unità di allocazione risorse:
- Il codice eseguibile
- I dati
- Le risorse richieste al sistema operativo

Un t**ask** = un'unità di risorse + i thread che vi accedono.

CONDIVISIONE DI RISORSE TRA I THREAD

Vantaggi: maggiore efficienza
- Creare e cancellare thread è più veloce (100-1000 volte): meno informazione da duplicare/creare/cancellare e a volte non serve la system call.
- Lo scheduling tra thread dello stesso processo è molto più veloce che tra processi
- Cooperazione di più thread nello stesso task porta maggiore throughput e performance.

Svantaggi:
- Maggiore complessità di progettazione e programmazione
  - I processi devono essere "pensati" paralleli
  - Minore information hiding
  - Sincronizzazione tra i thread
  - Gestione dello scheduling tra i thread può essere demandato all'utente
- Inadatto per situazioni in cui i dati devono essere protetti

Ottimi per processi cooperanti che devono condividere strutture dati o comunicare (es: produttore-consumatore, server, …): la comunicazione non coinvolge il kernel.

User Level Thread (ULT)

Stack, program counter e operazioni su thread sono implementati in librerie a livello utente.

Vantaggi:

- efficiente: non c'è il costo della system call
- semplici da implementare su sistemi preesistenti
- portabile: possono soddisfare lo standard POSIX 1003.1c (pthread)
- lo scheduling può essere studiato specificatamente per l'applicazione

Svantaggi:

- Non c'è scheduling automatico tra i thread
  - Non c'è prelazione dei thread: se un thread non passa il controllo esplicitamente monopolizza la CPU (all'interno del processo)
  - System call bloccanti bloccano tutti i thread del processo: devono essere sostituite con delle routine di libreria, che blocchino solo il thread se i dati non sono pronti (**jacketing**)
- L'accesso al kernel è sequenziale
- Non sfrutta sistemi multiprocessore
- Poco utile per processi I/O Bound, come file server

Kernel Level Thread (KLT)

Il kernel gestisce direttamente i thread; le operazioni sono ottenute attraverso system call.

Vantaggi:

- lo scheduling del kernel è per thread, non per processo ⇒ un thread che si blocca non blocca l'intero processo
- Utile per i processi I/O bound e sistemi multiprocessor

Svantaggi:

- meno efficiente: costo della system call per ogni operazione sui thread
- necessita l'aggiunta e la riscrittura di system call dei kernel preesistenti
- meno portabile
- La politica di scheduling è fissata dal kernel e non può essere modificata

Implementazione ibride – ULT/KLT

Sistemi ibridi: permettono sia thread livello utente che kernel.

Vantaggi:

- tutti quelli dei ULT e KLT
- alta flessibilità: il programmatore può scegliere di volta in volta il tipo di thread che meglio si adatta

Svantaggio:

- Portabilità Es: Solaris 2 (thread/pthread e LWP), Linux (pthread e cloni), Mac OS X, Windows NT, . .

SINCRONIZZAZIONE

ISTRUZIONI ATOMICHE

Le istruzioni devono essere eseguite atomicamente, altrimenti possono portare ad inconsistenze.

RACE CONDITION

Più thread accedono concorrentemente agli stessi dati, e il risultato dipende dall'ordine di esecuzione di istruzioni selezionate dai diversi thread.

- Si possono verificare frequentemente nei sistemi operativi multitasking, sia per dati in user space sia per strutture in kernel.
- Sono estremamente pericolose: portano al malfunzionamento dei thread cooperanti, o anche (nel caso delle strutture in kernel space) dell'intero sistema
- Sono difficili da individuare e riprodurre: dipendono da decisioni dello scheduler, carico del sistema, utilizzo della memoria, numero di processori, ecc

PROBLEMA DELLA SEZIONE CRITICA (PSC)

Problema:
Assicurare che quando un thread esegue la sua sezione critica, nessun altro thread possa entrare nella propria sezione critica, mutua esclusione dalla sezione critica.

- **Deadlock:** Tutti i thread/processi sono bloccati.
- **Starvation:** Uno o più thread non ricevono mai l'accesso alla sezione critica.

CRITERI PER UNA SOLUZIONE CORRETTA DI PSC

Una buona soluzione deve soddisfare le seguenti proprietà, che definiscono uno scheduler ideale:

Mutua esclusione:

- Un solo thread alla volta deve acceder alla sezione critica.
- Se il thread Pi sta eseguendo la sua sezione critica, allora nessun altro thread può eseguire la propria sezione critica.

Progresso:

- il sistema deve progredire sempre, almeno un thread deve accedere alla sezione critica.
- Una situazione di **deadlock** (tutti i thread sono bloccati) rappresenta una possibile violazione della proprietà di progresso.

Attesa limitata:

- Vogliamo che un thread abbia un tempo di attesa (**upper bound**) per accedere alla sezione critica; quindi, vogliamo che tutti i thread abbiano le stesse possibilità.
- Definizioni alternative:
  - **Bounded Waiting:** Se un thread ha già richiesto l'ingresso nella sua sezione critica, esiste un limite superiore al numero di volte che si consente ad altri thread di entrare nelle rispettive sezioni critiche prima che si accordi la richiesta al primo thread.
  - La **starvation** (un thread non entra mai) rappresenta una violazione di bounded waiting.

ASSUNZIONI

- Dobbiamo dichiarare a priori quali istruzioni sono atomiche e quali no
- Ogni thread deve rimanere nella sezione critica solo un tempo finito
  - Altrimenti il problema perde di senso
  - Bisogna assumere che il thread **rimanga poco tempo nella sezione critica.**
- Ogni thread viene eseguito ad una velocità non nulla
  - Quindi un thread non si può fermare da solo)
  - Quindi entro un tempo finito venga eseguita un'istruzione.
- Non ci sono vincoli sulla velocità relativa dei thread (numero e tipo di CPU)

ALGORITMI CONCORRENTI PER PSC

BUSY WAIT

Notazione di attesa attiva di un evento da parte di un thread:

- Semplice da implementare
- Porta a consumi inaccettabili di CPU
- In genere da evitare, ma a volta è preferibile (in caso di attese molto brevi)

Un thread che attende attivamente su un variabile esegue uno **spin lock.**

ALGORITMO DI PETERSON PER 2 THREAD

Algoritmo per N thread basato su una combinazione di *richiesta* e *accesso*, soddisfa tutti i requisiti e risolve il problema della sezione critica, ma è ancora basato su spinlock.

Risorse condivise:

```
var turn : int;
var interested0 , interested1 : bool;
```

Processo P0:

```
while(TRUE) do
  interested0 := true;
  turn := 0;
  while(turn == 0 && interested1) do skip endwhile;
  #critical section
  interested0 := false;
  #non-critical section
endwhile
```

Processo P1:

```
while(TRUE) do
  interested1 := true;
  turn := 1;
  while(turn == 1 && interested0) do skip endwhile;
  #critical section
  interested1 := false;
  #non-critical section
endwhile
```

Tutto sta nell'assegnazione di *turn*, equivale quasi ad una random tra 0 e 1.
Algoritmo ottimista: andiamo avanti finché possiamo, in caso di errore facciamo roll-back.

ALGORITMO DI LAMPORT PER N THREAD

Basato su una coda generata da biglietti di ordine crescente distribuiti ai clienti, non centralizzato.
Risolve la sezione critica per n thread, generalizzando l'idea vista precedentemente.

- Prima di entrare nella sezione critica, ogni thread riceve un numero. Chi ha il numero più basso entra nella sezione critica.
- Se i thread Pi and Pj ricevono lo stesso numero: se i < j, allora Pi è servito per primo; altrimenti Pj è servito per primo.
- Lo schema di numerazione genera numeri in ordine crescente.
- Biglietto con valore 0, vuol dire che il thread non è interessato ad entrare nella sezione critica.

```
var choosing: array[1,. . . ,N] of boolean;
var number: array[1,. . . ,N] of integer;
#Process P(i):
while (TRUE) do
  choosing [i] = TRUE ;
  number [i] = max{number [1], . . . , number [N]} + 1;
  choosing [i] = FALSE ;
  for k : 1 to N do
    while choosing [k] do skip endwhile;
    #Mi basta una sola iterazione, perché scelgo biglietti sempre più grandi
    while (number [k] != 0 and ( ⟨number [k], k⟩ << ⟨number [i], i⟩ ))
      do skip
    endwhile;
  endfor;
  critical section
  number [i] = 0;
endwhile
```

Dove ⟨a, b⟩ << ⟨c, d⟩ se e solo se a<c oppure a=c e b<d

LOCK-FREE PROGRAMMING

Usata per ridurre l'overhead dovuto all'uso della sincronizzazione tramite lock, semafori e monitor.

Definizione: Programma Multi-thread che usa risorse condivise e che non presenta **interleaving** che bloccano i thread (deadlock, livelock).

TECNICHE USATE

Impone un ordine casuale forte usato per sincronizzare l'esecuzione.
Nel Lock-free programming si usano istruzioni a basso livello come le **memory fence** per garantire che ogni load/store effettuata fino alla chiamata di **mfence** sia visibile al sistema prima di quelle seguenti, così da garantire proprietà come la **write atomicity.**

SOLUZIONE HW ALLE RACE CONDITION

- Istruzioni per disabilitare interrupt
- Istruzioni speciali per rendere atomica l'esecuzione di un test e di un assegnamento

Disabilitare gli interrupt

Il processo può disabilitare TUTTI gli interrupt hw all'ingresso della sezione critica, e riabilitarli all'uscita

- **Soluzione semplice:** Garantisce la mutua esclusione poiché disabilita il context-switch.
- ma pericolosa: il processo può non riabilitare più gli interrupt, acquisendosi la macchina
- può allungare di molto i tempi di latenza
- non scala a macchine multiprocessore (a meno di non bloccare tutte le altre CPU)
- Inadatto come meccanismo di mutua esclusione tra processi utente.
- Adatto per brevi segmenti di codice affidabile (es: in kernel, accesso a strutture condivise).

Istruzioni Speciali: Test and Set (TS)

Testano e modificano atomicamente il contenuto di una variabile/cella di memoria.

$$TS(x,oldx) := atomico(\ oldx := x\ ;\ x := 1\ )$$

Ritorna in *oldx* il valore precedente di *x* ed assegna 1 ad *x*;

- Questi due passi devono essere atomici
- Cioè abbiamo bisogno di un'ipotetica istruzione: *TS RX, LOCK*
  - Che copi il contenuto della cella LOCK nel registro RX e poi imposti LOCK ad un valore ≠ 0.
- Il tutto atomicamente (bloccano il bus di memoria)

TS per mutua esclusione:

```
shared var lock = 0;

#Processo P
  bool old;
  while (true) do
    repeat
      TS(lock, old);
    until (old == 0);
  critical section
  lock := 0;
  #non-critical section
  endwhile
```

- Assicura mutua esclusione e progesso
- Tuttavia, è basato su spinlock e quindi busy wait
- C'è **Starvation !!**

Istruzioni speciali: Compare and Swap (CAS)

L'istruzione di Compare and Swap effettua uno swap del valore di due variabili (usando una variabile locale aux) in maniera atomica.

$$CAS(x,y) := atomico(\ aux{:=}x;\ x{:=}y;\ y{:=}aux;\ )$$

CAS per mutua esclusione

```
shared var lock=0;
process P:
  bool old;
  while (true) do
    repeat
      old=1;
      CAS(lock, old);
    until (old==0);
    critical section
    lock:=0;
    non-critical section
  endwhile
```

Evitare Busy Wait

Le soluzioni basate su spinlock portano a

- Busy wait: alto consumo di CPU
- Inversione di priorità: un processo a bassa priorità che blocca una risorsa viene ostacolato nella sua esecuzione da un processo ad alta priorità in busy wait sulla stessa risorsa

Idea migliore:

- Il processo viene messo in *wait* quando deve attendere un evento
- Il processo viene messo in *ready* quando l'evento avviene

Servono specifiche syscall o funzioni di kernel. Esempio:

- sleep(): il processo si autosospende (si mette in wait)
- wakeup(pid): il processo pid viene posto in ready, se era in wait

SEMAFORI

Principio base:
Due o più processi possono cooperare attraverso semplici segnali, in modo tale che un processo possa essere bloccato in specifici punti del suo programma finché non riceve un segnale da un altro processo.

Definizione: Tipo di dato astratto per il quale sono definite due operazioni
- V (verhogen, incrementare) chiamata anche up
  - viene invocata per inviare un segnale, quale il verificarsi di un evento o il rilascio di una risorsa
- P (proberen, testare) chiamata anche down
  - viene invocata per attendere il segnale

DESCRIZIONE INFORMALE

Un semaforo può essere visto come un contatore S condiviso che assume valori numerici interi e viene inizializzato ad un valore non negativo.

Operazione P(S) o down(S):
- Decrementa solo se il valore è > 0
- È bloccante quando il valore è > 0
- Non è bloccante quando il valore è < 0

Operazione V(S) o up(S):
- Incrementa il valore
- Non è bloccante

Le azioni P e V devono essere atomiche.

INVARIANTE

Siano:
- n_p il numero di azioni P/down completate
- n_v il numero di azioni V/up completate
- *init* il valore iniziale del semaforo (cioè il numero di risorse disponibili per i processi)

Allora vale la seguente proprietà invariante:

$$n_p \leq n_v + init$$

- Se *init = 0* il numero di attese dell'evento non deve essere superiore al numero di volte che l'evento si è verificato (np <= nv)
- Se *init* > 0 il numero di richieste soddisfatte (np) non deve essere superiore al numero inziale di risorse (*init*) + il numero di risorse restituite (nv)

ESEMPIO: SEZIONE CRITICA PER N PROCESSI

Variabili condivise:
- **var** *mutex* : *semaphore*
- inizialmente *mutex* = 1

C'è ancora starvation

Processo P_i

```
while (TRUE) {
    down(mutex);
        sezione critica
    up(mutex);
        sezione non critica
}
```

Mutex:
Semaforo binario [0,1], generalmente inzializzato a 1, poiché la prima operazione sarà una *down()* che decrementerà il valore a 0 per gli altri thread.

ESEMPIO: PRODUTTORE-CONSUMATORE

Vogliamo sincronizzare produttore e consumatore che comunicano attraverso un buffer con N slot. Vogliamo massimo parallelismo e sincronizzazione quando buffer pieno (Produttore attende) o vuoto (Consumatore attende).

- *Mutex:* protegge le strutture condivise
- *Empty e Full:* utilizzati per il meccanismo del contatore, lato produttore e consumatore
  - *empty* blocca solo quando finiscono gli elementi
- Questi semafori sono bloccanti solo con il valore 0

Variabili condivise

```
var mutex : semaphore
var empty : semaphore
var full : semaphore

inizializzate a

mutex = 1
empty=N (numero di slot)
full=0
```

Produttore

```
int item;
while(TRUE) do
  item=produce_item();
  down(empty);
  down(mutex);
  insert_item(item);
  up(mutex);
  up(full);
endwhile
```

Consumatore

```
int item;
while(TRUE) do
  down(full);
  down(mutex);
  item=remove_item();
  up(mutex);
  up(empty);
  consume_item(item); endwhile
```

- *down(mutex)*: Serve solo per proteggere l'operazione di inserimento.
- **Full e Empty sommati tra loro daranno sempre N**

IMPLEMENTAZIONE DEI SEMAFORI A LIVELLO DI KERNEL

L'implementazione della definizione classica dei semafori è basata su busy waiting:

```
procedure down(S):
  while (S≤0) skip endwhile;
  S:=S-1
```

```
procedure V(S):
  S:=S+1
```

Se i semafori sono implementati a livello Kernel è possibile limitare l'utilizzazione di busy waiting:

Per questo motivo:

- L'operazione down deve sospendere il processo che la invoca
- L'operazione up deve svegliare uno dei processi sospesi

Per ogni semaforo:

- Il S.O. deve mantenere una struttura dati contenente l'insieme dei processi sospesi
- Quando un processo deve essere svegliato, si seleziona uno dei processi sospesi

Semafori FIFO:

- politica first-in first-out: il processo che è stato sospeso più a lungo viene risvegliato per primo
- la struttura dati utilizzata dal S.O. è una coda

Struttura dati per semafori:
type semaphore = value integer; L list of process;

Assumiamo due operazioni fornite dal sistema operativo:

- *sleep()*: sospende il processo che la chiama (rilascia la CPU e va in stato waiting)
- *wakeup(Pid)*: pone in stato di *ready* il processo P.

RIASSUMENDO

- Utilizzando queste tecniche non abbiamo eliminato completamente la busy waiting
- Tuttavia, abbiamo limitato busy-waiting alle sezioni critiche delle operazioni P e V, queste sezioni critiche sono molto brevi
- Senza semafori: potenziali busy-waiting di lunga durata, meno frequenti
- Con semafori: busy-waiting di breve durata, più frequenti

SEMAFORI BINARI (MUTEX)

- I mutex sono semafori con due soli possibili valori: bloccato o non bloccato
- Utili per implementare mutua esclusione, sincronizzazione, . . .
- due primitive: **mutex lock** e **mutex unlock**.
- Semplici da implementare, anche in user space (p.e. per thread).

MONITOR

Tipo di dato astratto che fornisce funzionalità di mutua esclusione

- Collezione di dati privati e funzioni/procedure per accedervi
- I processi possono chiamare le procedure ma non accedere alle variabili locali.
- Un solo processo alla volta può eseguire codice di un monitor

Il programmatore raccoglie quindi i dati condivisi e tutte le sezioni critiche relative in un monitor
→ Questo risolve il problema della mutua esclusione.

Implementati dal compilatore con dei costrutti per mutua esclusione (p.e.: inserisce automaticamente lock mutex e unlock mutex all'inizio e fine di ogni procedura).

- Veri costrutti, non funzioni di libreria ⇒ bisogna modificare i compilatori.
- Implementati (in certe misure) in veri linguaggi, esempio: i metodi synchronized di Java
- Un problema che rimane (sia con i monitor che con i semafori):
  - È necessario avere memoria condivisa ⇒ questi costrutti non sono applicabili a sistemi distribuiti (reti di calcolatori) senza memoria fisica condivisa.

Il processo quando si "sveglia", quindi quando avrà accesso al monitor si troverà già nella posizione in cui dovrà eseguire l'operazione richiesta.

CONDITION VARIABLE

Per sospendere e riprendere i processi, ci sono le **variabili *condition***, simili agli eventi, con le operazioni

- wait(c): il processo che la esegue si blocca sulla condizione c.
- signal(c): uno dei processi in attesa su c viene risvegliato.

A questo punto, chi va in esecuzione nel monitor? Due varianti:

- Chi esegue la signal(c) si sospende automaticamente (monitor di Hoare)
- La signal(c) deve essere l'ultima istruzione di una procedura (così il processo lascia il monitor) (monitor di Brinch-Hansen)
- I processi risvegliati possono provare ad entrare nel monitor solo dopo che il processo attuale lo ha lasciato

Il successivo processo ad entrare viene scelto dallo scheduler di sistema.

i *signal* su una condizione senza processi in attesa vengono persi.

PRODUTTORE – CONSUMATORE

```
monitor ProducerConsumer
    condition full, empty;
    integer count;
    procedure insert(item: integer);
    begin
        if count = N then wait(full);
        insert_item(item);
        count := count + 1;
        if count = 1 then signal(empty)
    end;
    function remove: integer;
    begin
        if count = 0 then wait(empty);
        remove = remove_item;
        count := count – 1;
        if count = N – 1 then signal(full)
    end;
    count := 0;
end monitor;
```

Produttore:

```
while (true)
    item := ...
    ProducerConsumer.insert(item)
```

Consumatore:

```
while (true)
    item := ProducerConsumer.remove();
    ...
```

MEMORY MODEL

Modello formale che rappresenta la gestione della memoria, quindi come la memoria appare al programmatore.

OBBIETTIVI

Il memory model deve specificare:

- L'interazione tra thread e memoria
- Quale valore può restituire una read
- Quando una modifica diventa visibile agli altri thread
- Quali assunzioni si possono fare sul comportamento della memoria quando scriviamo un programma o applichiamo (il compilatore applica...) un'ottimizzazione

MULTI THREAD: STRICT CONSISTENCY

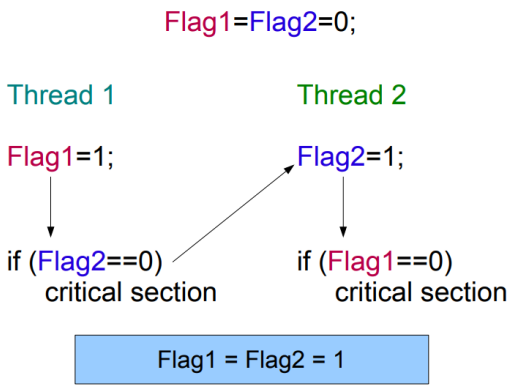
- Ogni operazione in memoria viene vista nell'ordine temporale in cui viene eseguita dai diversi thread
- Praticamente impossibile da ottenere in un sistema concorrente
- Solitamente si utilizzano nozioni più deboli che non rispettano l'ordine temporale di esecuzione ma l'ordine del programma e i possibili interleaving

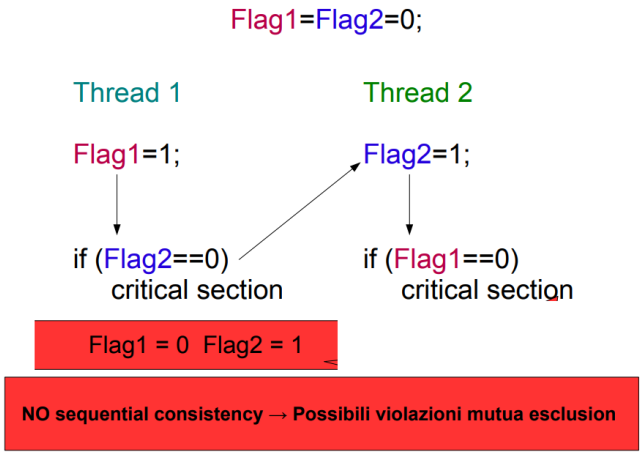
SEQUENTIAL CONSISTENCY (SC)

Un multiprocessore (sistema multithreaded) è sequentially consistent (SC) se il risultato di ogni sua esecuzione è lo stesso di un'esecuzione sequenziale delle istruzioni di tutti i processori (thread) e le istruzioni di ogni processore (thread) sono eseguite nell'ordine del corrispondente programma.
Richiede:

- **program order**: rispetta ordine nel programma
- **write atomicity**: tutte le write devono apparire a tutti i thread nello stesso ordine

Esempio: Algoritmo di Dekker





Sequential Consistency e Cache

- Per processori con cache locali occorre utilizzare un protocollo di consistenza tra memoria e linee di cache associate ad una certa locazione
- Ad esempio in caso di aggiornamento di una certa locazione di memoria:
  - Ciclo di invalidazione o di aggiornamento di tutte le linee associate ad L!

RELAXED MEMORY MODEL

Rilassiamo i requisiti per avere SC

- Program order (locazioni diverse): Si ammettono reorder es. read to write
- Write atomicity (stessa locazione)
  - Read other write early
  - Read own write early (leggi dal write buffer)

THREAD IN C

PTHREAD

La libreria pthread è definita in ambito POSIX, definisce un insieme di primitive per la programmazione di applicazioni multi-thread realizzate in C.

- I thread avranno tipo **pthread_t**, struttura che fa riferimento alla tabella dei thread del kernel.

Il thread è l'unità di scheduling, ed è univocamente individuato da un **identificatore** (intero):

- *pthread_t tid;*

Creazione

```
int pthread_create(pthread_t *thread, pthread_attr_t *attr,  void *(*start_routine)(void *), void * arg);
```

- thread: è il puntatore alla variabile che raccoglierà il thread_id
- start_routine: è il puntatore alla funzione che contiene il codice del nuovo thread
- arg: è il puntatore all'eventuale vettore contenente i parametri della funzione da eseguire
- attr: può essere usato per specificare eventuali attributi da associare al thread (di solito: NULL)

Attesa della terminazione

Un thread può sospendersi in attesa della terminazione di un altro thread con:

```
int pthread_join(pthread_t th, void *thread_return);
```

- th: è il pid del particolare thread da attendere
- thread_return: è il puntatore alla variabile dove verrà memorizzato il valore di ritorno del thread

Normalmente è necessario eseguire la pthread_join per ogni thread che termina la sua esecuzione, altrimenti rimangono allocate le aree di memoria ad esso assegnate.

In alternativa si può "staccare" il thread dagli altri con:

```
int pthread_detach(pthread_t th);
```

il thread rilascia automaticamente le risorse assegnategli quando termina.

Terminazione

Un thread può terminare chiamando:

```
void pthread_exit(void *retval);
```

- retval: è il puntatore alla variabile che contiene il valore di ritorno
  - Può essere raccolto da altri threads, vedi pthread_join.

MUTEX E MONITOR

Astrazione simile al concetto di semaforo binario, il valore può essere 0 oppure 1 (*occupato* o *libero*).
Sono utilizzati per risolvere problemi di mutua esclusione.
Il Mutex deve essere visibile a tutti i thread che si devono sincronizzare; quindi, deve stare nello spazio di memoria conidivisa.
Un mutex è definito dal tipo **pthread_mutex_t** che rappresenta:

- lo stato di mutex
- la coda dei processi sospesi in attesa che mutex sia libero.

Lock – Unlock

Sui mutex sono possibili solo due operazioni:

- `pthread_mutex_lock (pthread_mutex_t *M)`

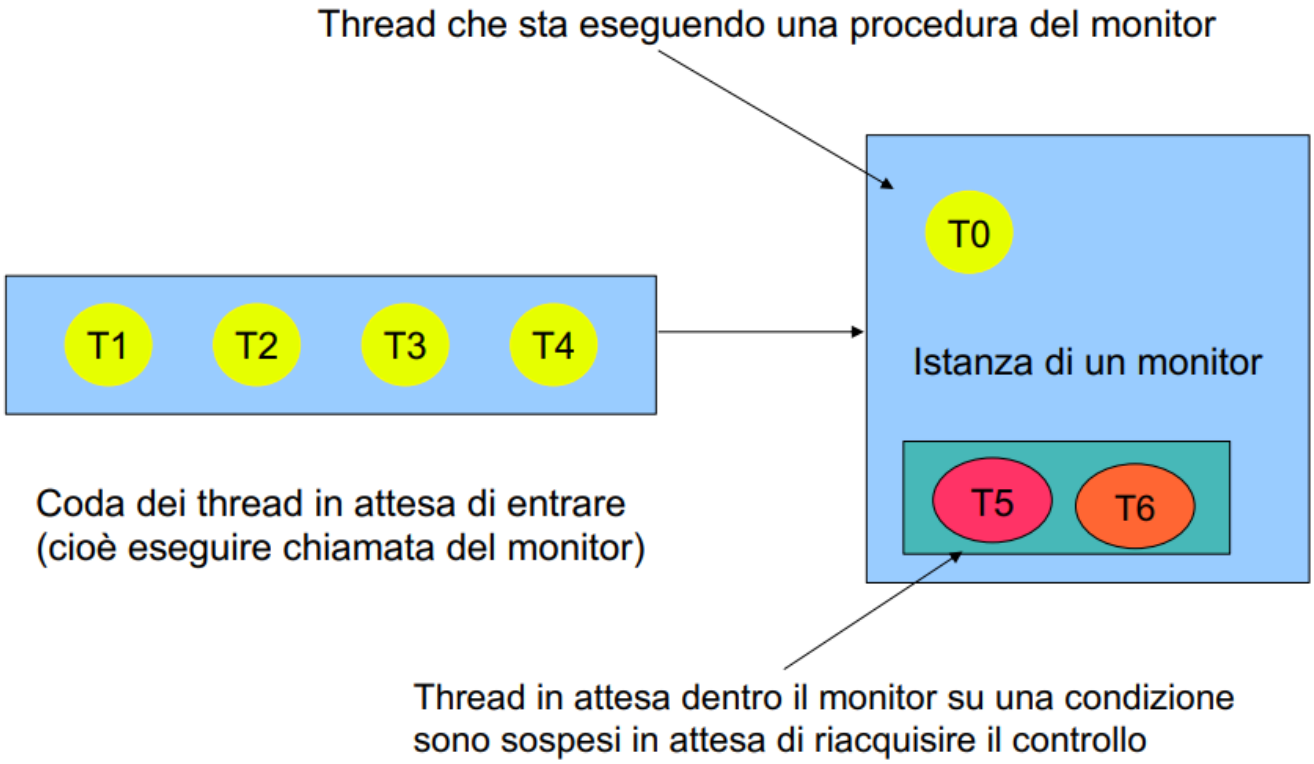
  se M è occupato (stato 0), il thread chiamante si sospende nella coda associata a M; altrimenti occupa M.

- `pthread_mutex_unlock (pthread_mutex_t *M)`

  se vi sono processi in attesa di M, ne risveglia uno, altrimenti libera M.

Condition Variable

- Strumento di sincronizzazione associato ai lock
- Consente la sospensione dei thread in attesa che sia soddisfatta
- Ad ogni condition variable viene associata una coda di attesa per i thread in attesa



Una variabile condizione C viene creata e inizializzata con attributi nel modo seguente:

```
p_thread_cond_t  C;
p_thread_cond_ init (&C,attr);
```

- attr è l'indirizzo della struttura che contiene eventuali attributi.

Le operazioni fanno sempre riferimento ad un mutex:

- **`pthread_cond_wait(&C,&m):`**
  - sospensione ed ingresso nella coda C nel monitor associato al mutex m

- **`pthread_cond_signal(&C,&m):`**
  - segnalazione ad un thread sospeso nella coda C del mutex m

- **`pthread_cond_broadcast(&C,&m):`**
  - notifica a tutti i thread sospesi nella coda C del mutex m

Le operazioni vanno eseguite dopo aver acquisto un lock.
Funzionamento:

- Il monitor è in memoria condivisa, protetto da lock
- Code FIFO associate al monitor

Il thread T1 esegue signal e il thread T2 è in attesa:

- non vogliamo due thread in esecuzione (deve valere mutua esclusione).

Il thread segnalato T2 viene trasferito dalla coda associata alla variabile condition alla entry_queue.

- T2 potrà rientrare nel monitor solo dopo l'uscita di T1.

Problema:
La signal sveglia un thread a caso, non ci garantisce un ordine; quindi, altri thread potrebbero entrare nel monitor prima di T2 e modificare le variabili condivise
La condizione che stiamo testando non è detto che sia ancora verificata nel momento in cui il thread che aveva eseguito la wait riprende il lock sul monitor.

Possibile "soluzione": Sostituire l'if con un ciclo while → Busy Wait

Esempio: Produttore – Consumatore

```
struct node {
  int info;
  struct node * next;
};

/* testa lista inizialmente vuota*/
struct node * head=NULL;

/* mutex e cond var*/
pthread_mutex_t mtx=PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t cond=PTHREAD_COND_INITIALIZER;
```

```
static void* consumer (void*arg) {
struct node * p;
 while(true) {
    pthread_mutex_lock(&mtx);
    while (head == NULL){
    pthread_cond_wait(&cond, &mtx);
    printf("Waken up!\n"); fflush(stdout);
 }
 p=estrai();
 pthread_mutex_unlock(&mtx);
 /* elaborazione p … … */
  }
}
```

```
static void* producer (void*arg) {
struct node * p;

for (i=0; i<N; i++) {
    p=produci();
    pthread_mutex_lock(&mtx);
    inserisci(p);
    pthread_cond_signal(&cond);
    pthread_mutex_unlock(&mtx);
 }

}
```

BARRIERE DI SINCRONIZZAZIONE

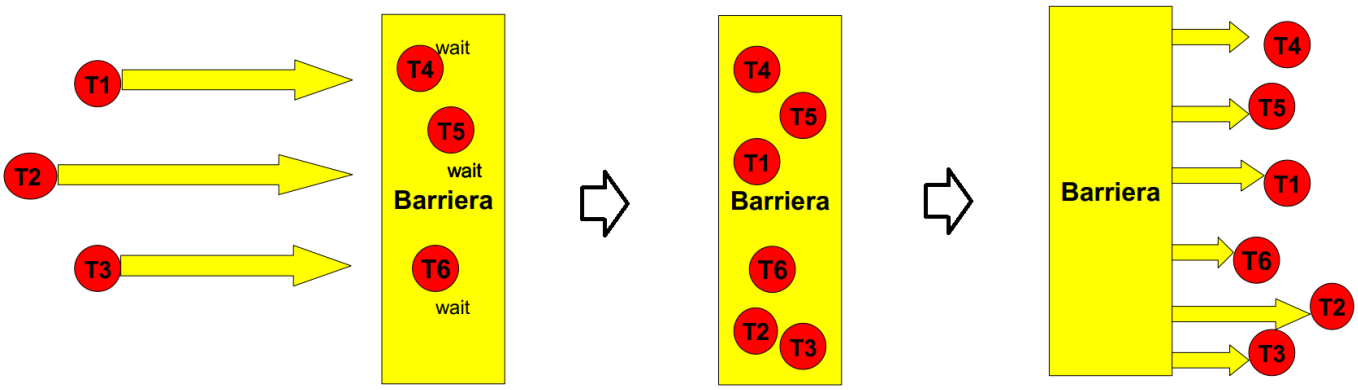
Le barriere sono un meccanismo di sincronizzazione utilizzato ad esempio nel calcolo parallelo.

Idea: una barriera rappresenta un punto di sincronizzazione per N thread

- I thread che arrivano alla barriera aspettano gli altri
- Solo quando tutti gli N thread arrivano alla barriera allora possono proseguire

È il singolo thread che decide dove creare la barriera; quindi, è un'operazione interna

- La barriera è un elemento condiviso in memoria tra tutti i thread



Creazione di una barriera

```
pthread_barrier_t barrier = PTHREAD_BARRIER_INITIALIZER(count);
```

- Count = numero di thread da sincronizzare

Attesa su una barriera all'interno del codice di un thread:

```
int pthread_barrier_wait(pthread_barrier_t *barrier);
```

- Quando N thread hanno eseguito wait sono tutti sbloccati e possono proseguire l'esecuzione!

Vantaggi rispetto alla Join:

- Non serve fare cicli di attesa
- Lo posso fare dove voglio
- Posso riusare lo stesso meccanismo più volte
- Non devo ricreare i thread, perché li posso riutilizzare e non ho bisogno del monitor esterno

BARRIERE DI MEMORIA

Una memory barrier è una classe di istruzioni fornite dalle architetture per imporre un certo ordine nelle operazioni di lettura e scrittura in memoria.

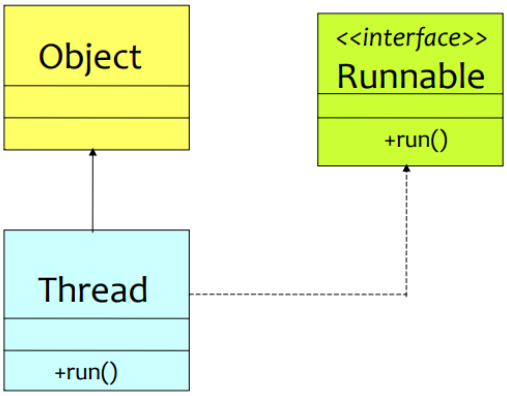
- Forza la write atomicity
- Elmina un possibile delay tra scrittura di una variabile e lettura successiva

In particolare le operazioni di scrittura prima di una certa barriera vengono consolidate prima di eseguire quelle successive alla barriera.

```
X=1;   // write(X,1) potrebbe essere nel buffer di scrittura
memory_fence();
if (Y==0) …          // X ha adesso valore 1 in memoria
```

MULTI-THREADING [JAVA]

CLASSE JAVA.LANG.THREAD



INTERFACCIA RUNNABLE

- Java non supporta l'ereditarietà multipla
- Multithreading per le classi già derivate:
  - Implementare l'interfaccia **Runnable**
  - Implementare il metodo **run**()
  - Creare un nuovo Thread passandogli un oggetto **Runnable**

- **Oggetto Runnable** = Task da eseguire
- **Oggetto Thread** = thread che esegue un task

```
// Lanciato su più thread
if (separateThread) {
  t=new MyRunnable("thread 1");
  (new Thread(t)).start();
  // lancio un nuovo thread
}
//Funziona su un solo thread
else {
  o=new MyRunnable("only main thread");
  o.run();
  // invoco metodo run dell'oggetto t1
};
```

METODI DELLA CLASSE THREAD

Oggetto di tipo Thread = oggetto che esegue un task in **modo asincrono**

Costruttori:

- `Thread()`
- `Thread(Runnable Obj)`
- `Thread(Runnable Obj, String Name)`

Run

- Il codice di esecuzione del thread
- Ridefinito nelle sottoclassi di Thread o nelle classi che implementano l'interfaccia Runnable
- Non solleva eccezioni

Start

- Lancia il thread e ritorna al chiamante
- Richiama **run**
- Richiamare start due volte genera un errore

COME USARE I THREAD

- Derivare una nuova classe da Thread
- Ridefinire il metodo **run()**
- Creare un oggetto di tale sottoclasse
- Richiamare **start()** su tale oggetto

```
public class MyThread extends Thread {
  //...
  public MyThread(String n){...}
  public void run() {...}

  public static void main(String[] args) {
      MyThread t1=new MyThread("thread 1");
      //t1 non e' ancora in esecuzione
      t1.start();
      //...
      // da qui in poi main e thr1 eseguono
      // in concorrenza
}
```

SLEEP

Pausa per x millesecondi, altri thread possono andare in esecuzione.

```
static sleep(milliseconds)
```

INTERRUZIONE

Nel programma bisogna specificare dei punti in cui il task relativo al thread può fare cose.

Interrupt():

- Viene messo a true il flag di interruzione
- Appena il thread entra in stato di blocked riceve una InterruptedException:
- Ci si affida alla correttezza del programma del thread: potrebbe ignorare tale eccezione!

Boolean isAlive()

- True se il thread non ha ancora terminato l'esecuzione.

```
// main
thr1.start();
thr1.interrupt();
status = thr1.isAlive() ? "is alive" : " is not alive";
```

```
// thread thr1
try {
  System.out.println("thread 1");
  Thread.sleep(3000L);
} catch (InterruptedException iex) { ... };
```

ATTESA TERMINAZIONE

t.join();

- Il chiamante attende che il thread t abbia terminato l'esecuzione

STATI DI UN THREAD

- Start state
  - Thread appena creato
  - Quando viene chiamato **start**() entra nel Ready state
- Runnable state:
  - Pronto per essere eseguito
- Running state
  - Il thread è effettivamente in esecuzione
  - Quando **run()** termina entra in stop state
- Stop state
  - Il thread può essere rimosso dal sistema
  - Quando **run**() termina oppure per eccezione non gestita
- Blocked state
  - Mentre è in running può entrare in questo stato
  - Per es. in attesa per una richiesta I/O
- Sleeping state
  - Quando viene chiamato il metodo **sleep**()
  - Rientra in stato runnable quando lo sleep time è passato
- Waiting state

SINCRONIZZAZIONE [JAVA]

SYNCHRONIZED METHODS

- I metodi di un oggetto si possono dichiarare **synchronized**
- Solo un thread alla volta può eseguire un tale metodo su uno stesso oggetto
- Se un thread sta eseguendo un metodo **synchronized** altri thread non possono eseguire altri metodi **synchronized** su uno stesso oggetto
- Lo stesso thread può eseguire altri metodi **synchronized** sullo stesso oggetto

Ogni oggetto ha al suo interno:
- un suo lock per eseguire la sincronizzazione
- e un **wait-set**, cioè una lista di thread in attesa su quell'oggetto (quindi è un monitor)

```
public synchronized int read() {
  return field;
}
```

```
public int read() {
  synchronized (this) {
      return field;
  }
}
```

MONITOR

Quando un thread deve eseguire un metodo synchronized su un oggetto si blocca finché non riesce ad ottenere il lock sull'oggetto.

- Quando lo ottiene può eseguire il metodo
- Gli altri thread rimarranno bloccati finché il lock non viene rilasciato
- Quando il thread esce dal metodo synchronized rilascia automaticamente il lock
- A quel punto gli altri thread proveranno ad acquisire il lock
- Solo uno ci riuscirà e gli altri torneranno in attesa

STATIC DATA

- I metodi e blocchi synchronized non assicurano l'accesso mutuamente esclusivo ai dati statici
- I dati statici sono condivisi da tutti gli oggetti della stessa classe
- Per accedere in modo sincronizzato ai dati statici si deve ottenere il lock su questo oggetto Class
    - si può dichiarare un metodo statico come synchronized
    - si può dichiarare un blocco come synchronized sull'oggetto

WAIT E NOTIFY

Un thread può chiamare **wait**() su un oggetto sul quale ha il lock:
- Il lock viene rilasciato
- Il thread va in stato di waiting

Altri thread possono ottenere tale lock, effettuano le opportune operazioni e invocano su un oggetto:
- **notify**() per risvegliare un singolo thread in attesa su quell'oggetto
- **notifyAll**() per risvegliare tutti i thread in attesa su quell'oggetto.

I thread risvegliati devono comunque riacquisire il lock.
I notify non sono cumulativi.

WAIT-SET

- Ad ogni oggetto Java è associato un wait-set: l'insieme dei thread che sono in attesa per l'oggetto
- Un thread viene inserito nel wait-set quando esegue la wait
- I thread sono rimossi dal wait-set attraverso le notifiche: notify / notifyAll

Produttore – Consumatore

```
class Produttore extends Thread {
  public void run() {
    for (int i = 1; i <= 10; ++i) {
      synchronized (cella) {
        //Funziona anche se si risveglia senza notify
        while (!cella.scrivibile) {
          try { cella.wait();
          } catch (InterruptedException e) { return;}
        }
        ++(cella.valore);
        cella.scrivibile = false; // cede il turno
        cella.notify(); // risveglia il consumatore
      }
    }
  }
}
```

```
class Consumatore extends Thread {
  public void run() {
    int valore letto;
    for (int i = 0; i < 10; ++i) {
      synchronized (cella) {
        while (cella.scrivibile) {
          try { cella.wait();
          }catch (InterruptedException e) { return; }
        }
        valore letto = cella.valore;
        cella.scrivibile = true; // cede il turno
        cella.notify(); // notifica il produttore
      }
    }
  }
}
```

Notify vs NotifyAll

Un notify effettuato su un oggetto su cui nessun thread è in wait viene perso (non è un semaforo)

Se ci possono essere più thread in attesa usare notifyAll:

- Risveglia tutti i thread in attesa
- Tutti si rimettono in coda per riacquisire il lock
- Ma solo uno alla volta riprenderà il lock

Prima che un thread in attesa riprenda l'esecuzione il thread che ha notificato deve rilasciare il monitor (uscire dal blocco synchronized).

HIGH LEVEL CONCURRENCY [JAVA]

Un oggetto runnable corrisponde alla definizione di un task (lavoro da eseguire), attraverso i metodi della classe Thread possiamo controllare inizio e fine dell'esecuzione di un task.

EXECUTOR

L'interfaccia Executor fornisce un pattern per separare:

- La definizione di un task (es. runnable object)
- Dal modo con cui si esegue (execution policy)

L'interfaccia ha un metodo execute che definisce la politica di esecuzione di un task.

```
new Thread(new(RunnableTask())).start();

Executor executor = anExecutor;
executor.execute(new RunnableTask1());
executor.execute(new RunnableTask2());
```

Executor = oggetto che definisce politica di esecuzione.

Esempio

Più task per thread

```
import java.util.concurrent.Executors;

class DirectExecutor implements Executor {
  public void execute(Runnable r) {
    r.run();
  }
}
```

Un task per thread

```
import java.util.concurrent.Executors;
ThreadPerTaskExecutor implements Executor {
  public void execute(Runnable r) {
    new Thread(r).start();
  }
}
```

EXECUTORS FACTORY METHODS

Per creare oggetti che implementano le politiche si possono usare factory methods:

- **Executors.newSingleThreadExecutor():** (single background thread)
- **Executors.newFixedThreadPool(int):** (fixed size thread pool)
- **Executors.newCachedThreadPool():** (unbounded thread pool, with automatic thread reclamation)

FIXED THREAD POOL

```
import java.util.concurrent.Executors;
import java.util.concurrent.ExecutorService;

public class RunnableTester {
  public static void main(String[] args) {
    Task task1 = new Task("thread1");
    Task task2 = new Task("thread2");
    System.out.println("Starting threads");

    ExecutorService threadExecutor=Executors.newFixedThreadPool(2);
    threadExecutor.execute(task1);
    threadExecutor.execute(task2);
    threadExecutor.shutdown();
    System.out.println("Threads started, main ends\n");
  }
}
```

I task vengono eseguiti usando un pool con N threads.

Le richieste di esecuzione vengono gestite con una coda.

Shutdown(): interrompe la mia thread pool

Gestione della coda

Dopo aver generato i core thread, quando viene sottomesso un nuovo task T

- se esiste un thread TH inattivo, T viene assegnato a TH
- se non esistono thread inattivi, T viene messo in coda
- **solo se la coda è piena**, si crea un nuovo thread
- se la coda è piena e sono attivi MaxPoolSize thread, il task viene respinto e viene sollevata un'eccezione

Tipi di coda

SynchronousQueue:

- Dimensione = zero.
- Un nuovo task T viene eseguito immediatamente se esiste un thread inattivo oppure se è possibile creare un nuovo thread (numero di thread ≤ maxPoolSize).
- Si può superare core ma non andare oltre max

ArrayBlockingQueue:

- Dimensione limitata, stabilita dal programmatore (si può andare oltre core)

LinkedBlockingQueue:

- Dimensione illimitata.
- È sempre possibile accodare un nuovo task, nel caso in cui tutti i thread siano attivi nell'esecuzione di altri task.
- La coda non può mai risultare piena e quindi la dimensione del pool **non supera mai core**

Test e Attesa di terminazione

boolean isShutdown( )

- Returns true if this executor has been shut down.

boolean isTerminated( )

- Returns true if all tasks have completed following shut down

boolean awaitTermination(long timeout, TimeUnit unit)

- Blocks until all tasks have completed execution after a shutdown request, or the timeout occurs, or the current thread is interrupted, whichever happens first.

Shutdown

void shutdown( ): Terminazione graduale

- non accetta ulteriori task
- i task sottomessi in precedenza vengono eseguiti, compresi quelli in coda
- tutti i thread del pool terminano la loro esecuzione

List shutdownNow( ): Terminazione immediata

- non accetta ulteriori task,
- elimina i task in coda e li restituisce in una lista
- tenta di terminare l'esecuzione dei thread che stanno eseguendo (con interrupt())
- Non può garantire la terminazione di tutti i thread.

```
void shutdownAndAwaitTermination(ExecutorService pool) {
  pool.shutdown();
  // non accettare piu thread
  try {
    // aspetta 1m e poi chiudi tutti i thread
    if (!pool.awaitTermination(60, TimeUnit.SECONDS)) {
      pool.shutdownNow();
      // aspetta ancora 1m e poi manda un msg di errore
      if (!pool.awaitTermination(60, TimeUnit.SECONDS))
        System.err.println("Pool did not terminate");
      }
  } catch (InterruptedException ie) {
 //…
  }
}
```

FUNZIONI ASINCRONE [JAVA]

THREAD COME PROCEDURE ASINCRONE

I thread possono essere visti come delle chiamate di procedura asincrona:

- il caller (chiamante) crea il thread
- il callee(chiamato) esegue il corpo della procedura e termina

L'asincronia è dovuta al fatto che la funzione chiamata non viene eseguita sequenzialmente rispetto al codice del caller.

Se abbiamo bisogno di ritornare un risultato ricompare il problema della sincronizzazione di quest'ultimo, per questo utilizzeremo i **Future**...

FUTURE

- I **Future** sono oggetti che rappresentano il risultato di un'esecuzione asincrona
- Il metodo **get**() può essere usato per recuperare il risultato
- L'invocazione di **get**() è bloccante se il risultato non è ancora disponibile

```
public interface Future <V>{
  V get( ) throws ...;
  V get (long timeout, TimeUnit) throws ...;
  void cancel (boolean mayInterrupt);
  boolean isCancelled( );
  boolean isDone( );
}
```

- get( ) si blocca fino alla terminazione del task e restituisce il valore calcolato
- È possibile definire un tempo massimo di attesa della terminazione del task, dopo cui viene sollevata una TimeoutException
- È possibile cancellare il task e verificare se la computazione è terminata oppure è stata cancellata

```
ExecutorService esecutore = Executors.factorymethod;

Future<Integer> f = esecutore.submit(new MyCallable());
... // faccio altre cose mentre il MyCallable viene eseguito

System.out.println("Risultato: " + f.get());
// attendo il risultato - puo' sollevare eccezioni

esecutore.shutdown();
// finisci i task e non accettarne altri
```

FUTURETASK

Se si gestiscono esplicitamente thread (invece di thread pool):

- si costruisce un oggetto della classe FutureTask (che implementa Future e Runnable) passando un oggetto di tipo Callable al costruttore
- si passa l'oggetto FutureTask al costruttore del thread

Un oggetto di tipo FutureTask ha a disposizione sia il risultato (Future) che il callable.

```
FutureTask task = new FutureTask (new MyCallable ());
Thread t =new Thread(task);
t.start();

try{
  int res = task.get(1000L, TimeUnit.MILLISECONDS);
}
catch(ExecutionException e) { ...}
catch(TimeoutException e){ System.out.println("tempo scaduto"); }
catch(InterruptedException e){ ... }
```

```
FutureTask task = new FutureTask (new MyCallable ());
//FutureTask = cancellable asynchronous computation.
//A FutureTask can be used to wrap a Callable or Runnable object.
//Because FutureTask implements Runnable, a FutureTask can be submitted
//to an Executor for execution.

ExecutorService es =Executors.newSingleThreadExecutor();
es.submit (task);

try {
  int result = task.get ();
  System.out.println ("Result from task.get () = " +result);
}
catch (Exception e) {System.err.println (e);}

es.shutdown ();
```

```
public class CallableTester {
  public static void main(String[] args) {
    Callable<Integer> callable = new CallableImpl(2);

    ExecutorService executor = new ScheduledThreadPoolExecutor(5);
    Future<Integer> future = executor.submit(callable);

    try {
      System.out.println("Future value: " + future.get());
    } catch (Exception e) {e.printStackTrace();}
  }
}
```

LIBRERIE CONCORRENTI [JAVA]

BLOCKING QUEUE

Struttura dati concorrente con operazioni thread-safe

Operazioni di inserimento:
- add (genera eccez. se non è eseguibile)
- offer (rest. valore speciale se non è eseguibile)
- put (bloccante)

Di rimozione
- remove (genera eccez. se non è eseguibile)
- poll (rest. valore speciale se non è eseguibile)
- take (bloccante)

Test
- peek (val. speciale)
- element (eccezione)

```
public class BlockingQueue {
  private List queue = new LinkedList();
  private int limit = 10;

  public BlockingQueue(int limit){
    this.limit = limit;
  }

  public synchronized void enqueue(Object item)
    throws InterruptedException {
      while(this.queue.size() == this.limit) {
        wait();
      }
      if(this.queue.size() == 0) {
        notifyAll();
      }
      this.queue.add(item);
    }
}
```

CONCURRENT HASHMAP

ConcurrentHashMap è una versione thread-safe delle HashMap:
- Non si usano lock sulla struttura dati in lettura.
- Si usano lock solo sulle parti modificate in scrittura

Esempi di metodi:
- get e put
- putIfAbsent
- contains

CYCLIC BARRIER

Cyclic Barrier implementano le barriere in Java.

Nel Main

```
CyclicBarrier barrier = new CyclicBarrier(2);
```

Nei Thread

```
barrier.await();
barrier.await(10, TimeUnit.SECONDS);
```

READWRITE LOCK

Read/Write Lock sono lock che implementano la politica multiple readers, single writer.

L'idea è:
- Garantire mutua esclusione tra writer e readers
- Vogliamo dare accesso concorrente ai Reader, quindi garantirne la massima concorrenza.

```
ReadWriteLock readWriteLock = new ReentrantReadWriteLock();

readWriteLock.readLock().lock();
//multiple readers, no writers
readWriteLock.readLock().unlock();

readWriteLock.writeLock().lock();
//one writer, no readers
readWriteLock.writeLock().unlock();
```

Quindi i reader possono coesistere nella memoria critica, mentre Reader e Writer non possono.

Esempio di implementazione

Variabili condivise

```
int Reader = 0;
M mutex
R mutex
```

Reader

```
//Entry
M.lock(); //Lock per reader, eventuale coda dei reader
Reader++;

//per dire che siamo il primo thread di read
if Reader == 1
  R.lock(); //di conseguenza questa sarà una coda per i writer

M.unlock();

//Exit
M.lock();
Reader--;

//Se entro vuol dire che sono l'ultimo ed ho il lock
if Reader == 0
  R.unlock();

M.unlock();
```

Writer

```
R.lock()
//----
Write(R)
//----
R.unlock()
```

ATOMIC INTEGER

La classe AtomicInteger fornisce interi con operazioni atomiche e schemi tipo CAS (compare and set).

```
AtomicInteger atomicInteger = new AtomicInteger(123);
int theValue = atomicInteger.get();
atomicInteger.set(234);

int expectedValue = 123;
int newValue = 234;
atomicInteger.compareAndSet(expectedValue,newValue);
```

ALTRE STRUTTURE DATI

- **AtomicBoolean**: booleani con operazioni atomiche
- **AtomicLong**: long con operazioni atomiche
- **AtomicReference**: reference ad oggetti con operazioni atomiche
- **Exchanger**: scambiare reference ad oggetti in maniera atomica

METODOLOGIE DI PROGRAMMAZIONE CONCORRENTE IN JAVA

VISIBILITÀ

La modifica di una variabile all'interno di un oggetto potrebbe non essere visibile a tutti i thread nello stesso momento.

Per garantire la **write-atomicity** dichiariamo le variabili come **Volatile**.

Write-Atomicity:
L'ultimo valore letto è l'ultimo valore scritto, quindi non c'è possibilità che due thread leggano un valore diverso.

VARIABILI LOCALI

Le variabili locali sono allocate sullo stack di un thread e quindi in generale sono thread-safe.

```
public void someMethod(){
  long threadSafeInt = 0;
  threadSafeInt++;
}
```

RIFERIMENTO A OGGETTI LOCALI

I riferimenti locali ad oggetti in un metodo sono thread-safe.
Gli oggetti stessi tuttavia sono creati nell'heap, quindi un'**oggetto creato localmente è thread-safe solo se non esce mai dallo scope del metodo in cui è stato creato.**

```
public void someMethod(){
  LocalObject localObject = new LocalObject();
  localObject.callMethod();
  method2(localObject);
}

public void method2(LocalObject localObject){
  localObject.setValue("value");
}
```

RIFERIMENTI A FIELD

I campi di un oggetto sono memorizzati nell'heap condiviso.
Se due thread chiamano un metodo sullo stesso oggetto che modifica lo stesso campo il metodo NON è thread-safe.

```
NotThreadSafe sharedInstance = new NotThreadSafe();
new Thread(new MyRunnable(sharedInstance)).start();
new Thread(new MyRunnable(sharedInstance)).start();

public class MyRunnable implements Runnable{
  NotThreadSafe instance = null;
  public MyRunnable(NotThreadSafe instance){
    this.instance = instance;
  }

  public void run(){
    this.instance.add("some text");
  }
}
```

Se i thread effettuano update su istanze diverse allora non ci sono race condition.

```
new Thread(new MyRunnable(new NotThreadSafe())).start();
new Thread(new MyRunnable(new NotThreadSafe())).start();
```

Thread Control Escape Rule

Se una risorsa (oggetto, field, file, ecc) viene creata, usata e rimossa all'interno del controllo dello stesso thread e non esce mai dal suo scope, allora tale risorsa è thread-safe.

Oggetti vs Operazioni Thread-safe

Anche se un oggetto è thread safe, se puntasse ad un oggetto condiviso allora l'applicazione potrebbe non essere thread safe.
Ad esempio, se t1 e t2 creano delle connessioni ad un database, le connessioni sono thread-safe ma l'uso del database potrebbe non esserlo.

Oggetto Immutabile

Possiamo garantire la thread-safeness per oggetti immutabili, una volta creati non vengono più modificati: **Non forniremo metodi setter.**

Per modificare un oggetto immutabile bisogna costruire una nuova istanza modificata:

```
public class ImmutableValue{
  private int value = 0;
  //...

  public ImmutableValue add(int valueToAdd){
    return new ImmutableValue(this.value + valueToAdd);
  }
}
```

Confinamento per Metodo

Per assicurare thread-safeness si possono usare tecniche di confinamento per metodo

```
class Plotter {
  // ...
  public void showNextPoint() {
    Point p = new Point();
    p.x = computeX(); p.y = computeY();
    display(p);
  }

  int computeX() { return 1; }
  int computeY() { return 1; }
  protected void display(Point p) {
    // somehow arrange to show p.
  }
}
```

Confinamento per Thread

Un oggetto è confinato in un thread quando solo il thread che lo crea può accedervi.
In Java si possono usare:

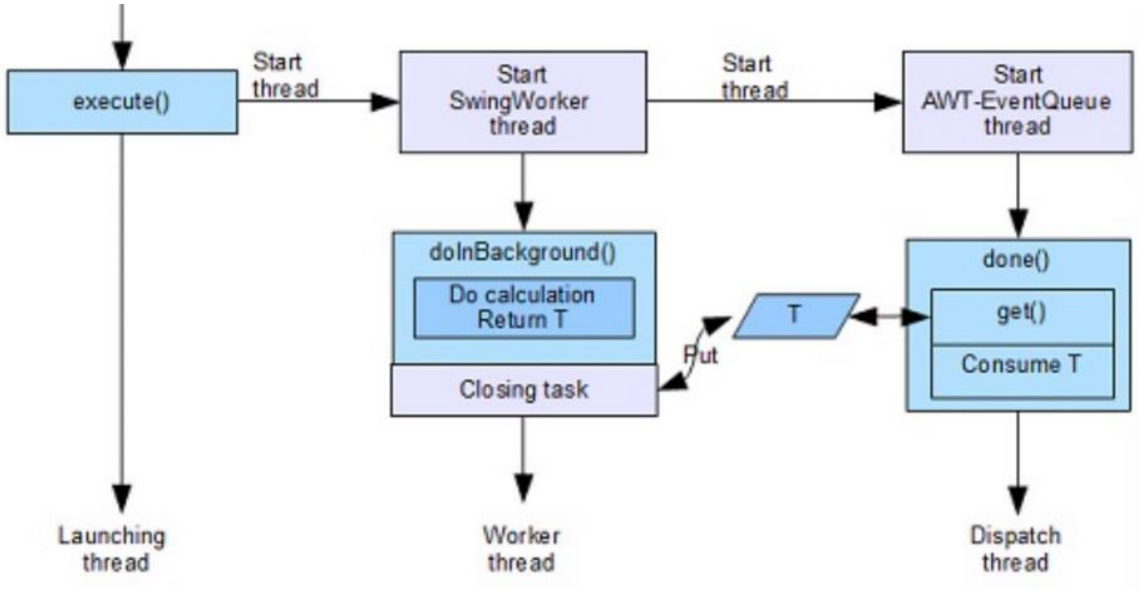
- Oggetti **java.lang.ThreadLocal** o **Campi privati di una classe** che deriva da java.lang.Thread

Troviamo altri esempi interessanti in framework in cui i thread sono usati per delegare long running task come Java Swing, FX e Android UI per gestire interfacce grafiche in applicazioni desktop e mobile.

Idea: UI è un'oggetto (complesso!) condiviso tra EDT (Event Dispatch Thread) che esegue i listener associati ad eventi di interazione con utente come click, ecc e tutti gli altri thread di un programma Java.

Single Thread Rule: un solo thread alla volta accede e modifica i campi della UI (es label, aspetto, posizione di un widget).
Per evitare di bloccare la UI con long running task vengono usati worker thread e metodi per delegare operazioni di update alla UI.

SWING WORKER NELLE UI



E ALTRI METODI DI SWINGWORKER

```
@Override
protected void done() {
}

@Override
protected void process(List<String> chunks) {
    for (String text : chunks) {
        ta.append(text);
        ta.append("\n");
    }
}

@Override
protected Void doInBackground() throws Exception {
    for (int index = 0; index < 100; index++) {
        publish("Line " + index);
        setProgress(index);
        Thread.sleep(125);
    }
    return null;
}
```

Eseguito nell'EDT ma definito nel worker!

Eseguito nell'EDT ma definito nel worker!

Eseguito nel worker

CONFINAMENTO PER OGGETTO

Nel confinamento per oggetto usiamo un oggetto host sincronizzato per proteggere l'accesso a risorse condivise.

- La sincronizzazione sul host protegge gli elementi interni.

Esempio

```
synchronizedCollection(Collection c) //collezione sincronizzata
synchronizedList(List l)
synchronizedMap(Map m)
unmodifiedCollection(Collection c)
unmodifiedList
```

SISTEMI DISTRIBUITI

Un sistema distribuito è una rete costituita da computer collegati tramite un middleware di distribuzione.
Aiutano a condividere diverse risorse e capacità per fornire agli utenti una rete coerente e integrata.

Caratteristiche principali:
- Diversi nodi autonomi, ognuno dei quali ha la sua memoria locale
- I nodi comunicano tra di loro tramite messaggi

PROPRIETÀ

- **Trasparenza della posizione**: i sistemi sono basati sulle risorse (resource-centric)
- **Computazione concorrente su differenti nodi, senza memoria condivisa**
- **Assenza di uno stato globale**
- **Tolleranza ai guasti**:
    - il sistema deve continuare a funzionare correttamente in caso di malfunzionamento di alcuni dei suoi componenti.
- **Eterogeneità nell'hardware e nel software**

SISTEMA DISTRIBUITO SINCRONO

Proprietà:
- Tempo di scadenza (Upper Bound) per l'invio di messaggi.
- Invio ordinato di messaggi
- Conoscenza dei Clock globali sincronizzati
- Esecuzione bassata su Lock Step

SISTEMA DISTRIBUITO ASINCRONO

- Il Clock può non essere accurato e non sincronizzato
- I messaggi possono essere ritardati per un periodo arbitrario di tempo.

PROBLEMI CLASSICI DEI SISTEMI DISTRIBUITI

- Selezione del leader
- Mutua esclusione
- Sincronizzazione del clock
- Global snapshot
- Consenso in presenza di failures
- Manutenzione di Routing/Route

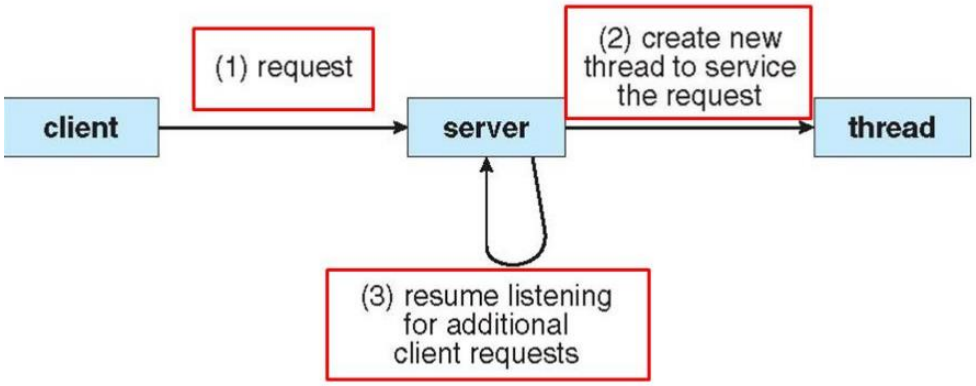
Assunzioni:
- Synchronous/Asynchronous
- Perdita o ritardo dei messaggi
- Node/Byzantine faults

ALGORITMI DISTRIBUITI

Gli algoritmi distribuiti sono spesso formulati sui grafi (percorso/ costo minimo, spanning tree, ...).

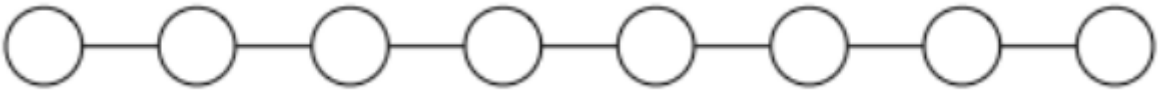
- Rete: Grafo
- Nodo: Processo
- Arco: Link di comunicazione tramite scambio di messaggi

ARCHITETTURA SERVER MULTITHREAD



COLOURING PATHS

Immaginiamo di avere *n* computer (nodi) che sono collegati l'uno all'altro tramite canali di comunicazione in modo tale che la topologia della rete sia un percorso:

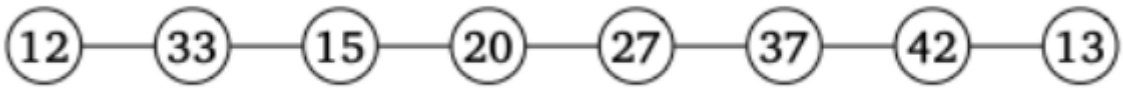


In questo esempio, il compito è trovare una colorazione appropriata del percorso con 3 colori. Ogni nodo deve mandare in output uno dei colori, 1, 2, o 3; qui c'è un esempio:

Sembra un problema facile da risolvere, ma i nodi non hanno conoscenza della rete globale. Nel nostro esempio, gli endpoints del percorso conoscono solo i loro vicini.

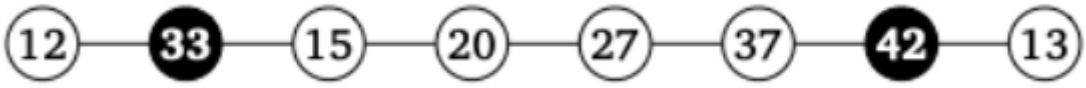
Tutti i nodi sul percorso sanno di avere due vicini:

Quando i nodi hanno id univoci

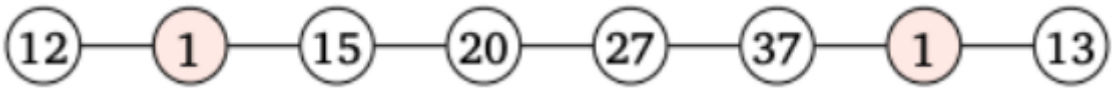


Con l'aiuto di identificatori unici, è facile progettare un algoritmo che colori il percorso.
Infatti, gli identificatori univoci formano già una colorazione con un grande numero di colori!
Tutto quello che dobbiamo fare è ridurre il numero di colori a 3.

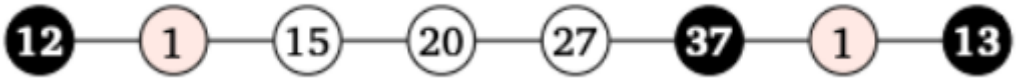
Possiamo usare la seguente strategia: in ogni step, un nodo è attivo se è "massimo locale", cioè se il suo colore corrente è maggiore del colore corrente dei suoi vicini:



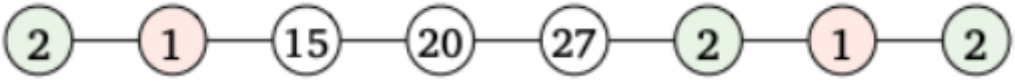
I nodi attivi prenderanno un nuovo colore dalla palette {1,2,3} così che non vada in conflitto con i colori correnti dei suoi vicini. Questo è sempre possibile, poiché ogni nodo in un percorso ha al massimo 2 vicini e abbiamo 3 colori nella nostra tavolozza di colori.



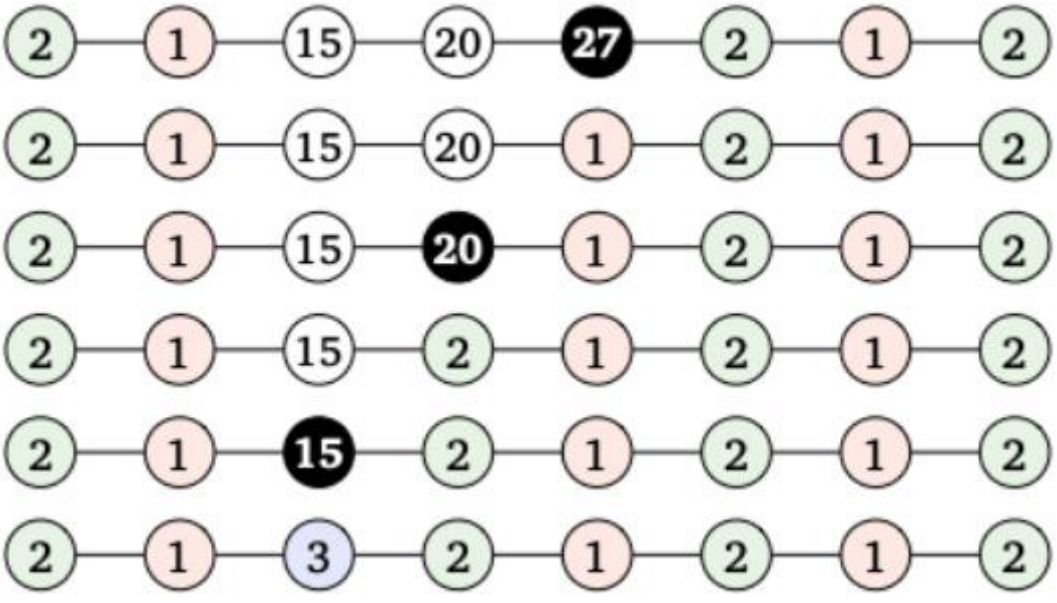
Ripetiamo la stessa procedura finché tutti i nodi hanno piccoli colori. Prima troviamo il massimo:



E poi ricoloriamo il massimo locale con colori da {1, 2, 3}



Continuando così avremo eventualmente un percorso che sarà propriamente colori con colori {1, 2, 3}:



Algoritmo

Ripeti per sempre:

- Invia il messaggio c a tutti i vicini
- Ricevi messaggi da tutti i vicini. M sarà l'insieme di messaggi ricevuti.
- *If c non appartiene a $\{1,2,3\}$ e $c > max\ M \rightarrow c = min(\{1,2,3\} \setminus M)$*

Proprietà

- I nodi sono macchine a stati che inviano ripetutamente messaggi ai loro vicini, ricevono messaggi dai loro vicini e aggiornano il loro stato. Tutti i nodi eseguono questi passi in modo sincrono e in parallelo
- Alcuni degli stati sono stati di arresto, e quando un nodo raggiunge uno di questi stati poi non modifica più il suo stato.
- In caso tutti i nodi avessero raggiunto uno stato di arresto, i loro stati dovrebbero rappresentare la soluzione al problema che volevamo risolvere.

Algoritmo randomizzato

Fin ora abbiamo usato gli identificatori univoci per rompere la simmetria.
Un'altra possibilità è quella di usare la randomicità per creare un algoritmo distribuito che trovi una corretta colorazione a 3 di un percorso: i nodi provano a prendere un colore dalla palette {1,2,3} randomicamente e si fermano quando riescono a prendere un colore diverso dai loro vicini.

Inizializzazione: state = unhappy, colour = 1

while state = unhappy:

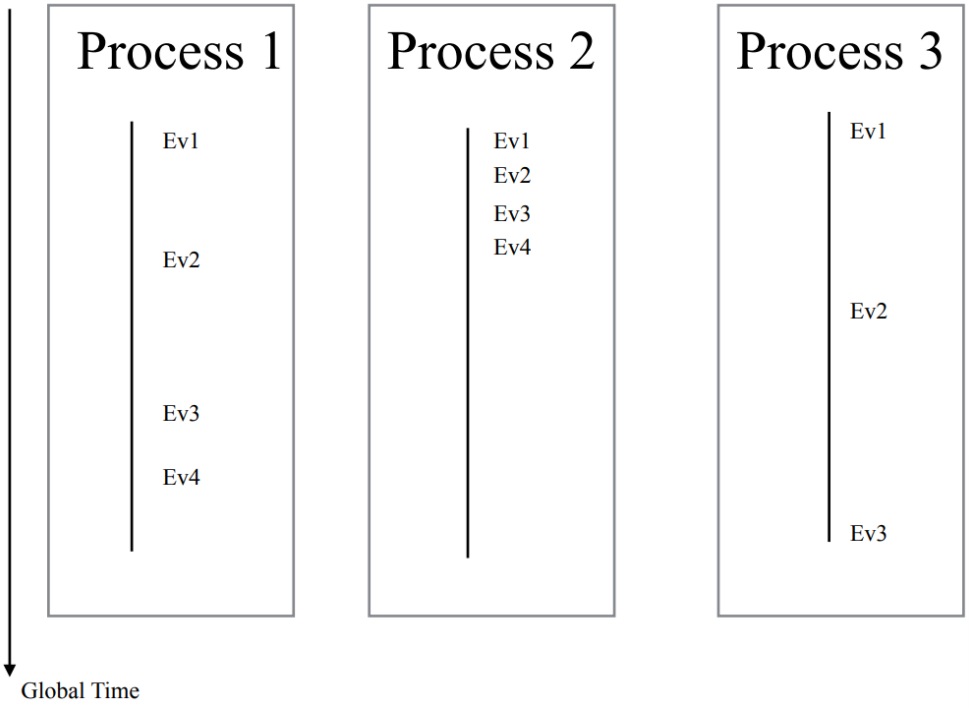
1. prendere un nuovo colore random da {1,2,3}
2. confronta i colori con i vicini
3. se diverso, imposta state = happy

LAMPORT'S LOGICAL CLOCKS

Clock logici

Algoritmi basati sui clock logici fanno uso di timestamps per ordinare le operazioni su diversi nodi di una rete.
Dobbiamo regolare il clock individuale per usare come timestamps globali.
Diverse versioni:

- clock scalari
- clock vettoriali



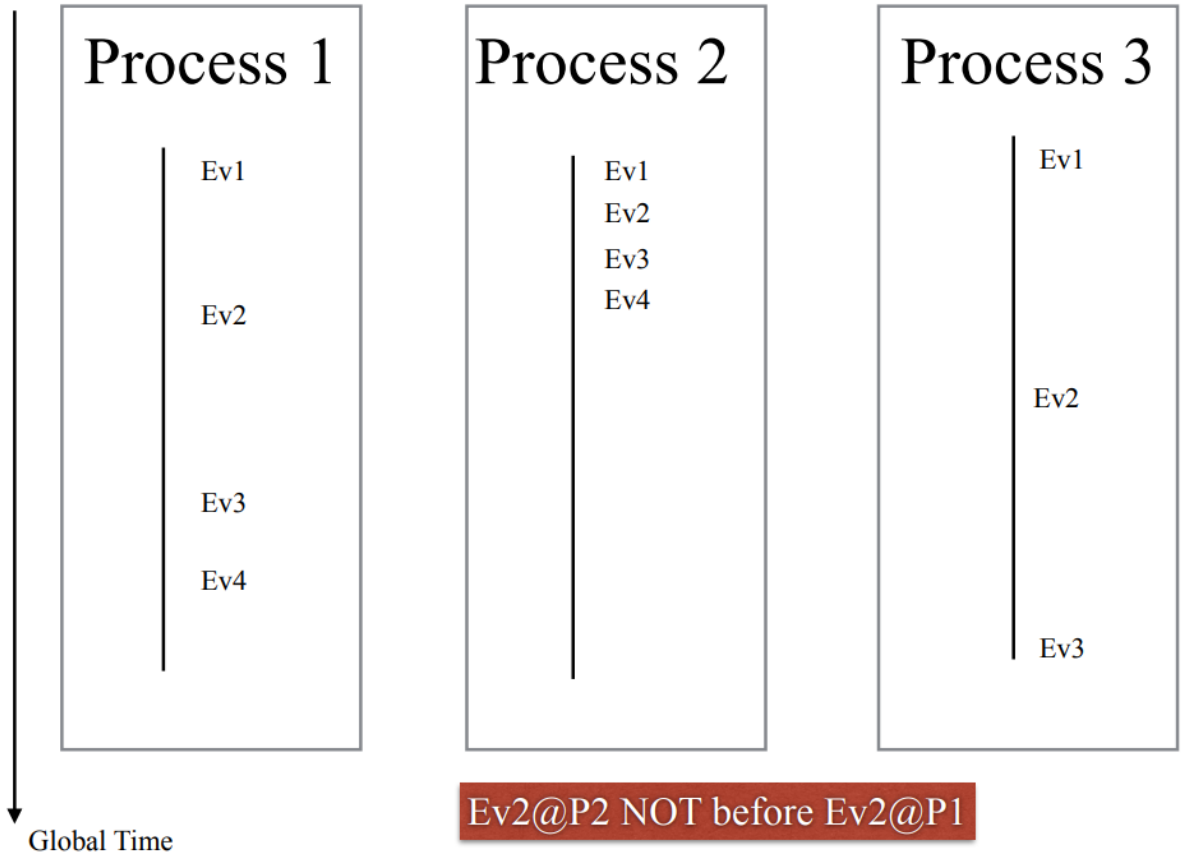
Proprietà

Per costruzione:

- a → b implica C(a) < C(b)

In generale:

- C(a) < C(b) non implica a → b
- In altre parole, i timestamp scalari non possono essere usati per identificare eventi concorrenti.



Clock Vettoriali (VC)

Ogni nodo mantiene una rappresentazione locale dei clock logici di tutti gli altri nodi (per N nodi un vettore di N elementi).

V[i] nel nodo j = quali nodi j conosce riguardo il clock logico del processo i.

Node with index I

local vector int V[N]={ 0,...,0 } ;

Internal event

V[i] ← V[i]+1

send

V[i] ← V[i]+1;
send(...,V,...);

receive

receive(...,T,...);
for every j : 1 ,…N :
V[j] ← max(V[j],T[j]);
V[i] ← V[i]+1;

- Il nodo i manda il suo vettore al nodo j.
- Visione parziale dello stato globale: Un nodo conosce solo quello che riceve.

Ordinamento dei Clock vettoriali

$$V \leq W \text{ iff } V[i] \leq W[i] \text{ for all } i : 1, \ldots, N$$
$$V < W \text{ iff } V \leq W \text{ there exists i s.t. } V[i] < W[i]$$
$$V = W \text{ iff } V \leq W \text{ and } W \leq V .$$

Sia C(e) il vettore di clock quando accade l'evento e, allora vale la seguente proprietà:

- C(a) < C(b) se e solo se a → b

SOMMARIO

- Sistemi distribuiti: Insieme di nodi cooperanti
- Algoritmi distribuiti: ogni nodo esegue un programma concorrente che scambia messaggi con altri nodi
- Un algoritmo distribuito eseguito a turni, in ogni turno i nodi mandano messaggi in parallelo e cambiano stato in accordo con i messaggi ricevuti
- Complessità: costo della comunicazione (numero di messaggi) e numero di turni.

JAVA NETWORKING AND IO

JAVA BLOCKING IO

Le API di Java Blocking IO sono incluse nel JDK sotto il pacchetto *java.net*.
Queste API si basano sui flussi di stream di byte e stream di caratteri che possono essere letti o scritti.

Non c'è un indice che si possa usare per spostarsi avanti e indietro, come in un array, è semplicemente un flusso continuo di dati.
È possibile creare pipeline di filtri attaccate agli stream di input e output (es: PrintStream, etc...).

SOCKET IN JAVA

```
//port scanning
for (int i = 1; i< 1024; i++){
  try{
   new Socket(host, i);
   System.out.println("Esiste un servizio sulla porta" + i);
  } catch (UnknownHostException ex){
     System.out.println("Host Sconosciuto");
  } catch (IOException ex) {
    System.out.println("Non esiste un servizio sulla porta"+i);
   }
}}}
```

FILTRI

```
import java.net.*;
import java.io.*;
import java.util.*;
...
InetAddress ia = InetAddress.getByName("localhost");
int port = 12345;
Socket s = null;
try{
  s = new Socket (ia, port);
  } catch (Exception e){...}

InputStream is = s.getInputStream( );
OutputStream os = s.getOutputStream( );
DataInputStream netIn = new DataInputStream(is);
DataOutputStream netOut = new DataOutputStream(os);
netOut.write(buffer);
....
```

SERVER SOCKET

```
import java.net.*;
import java.io.*;
int port= 12345;

ServerSocket s = new ServerSocket(port,2);
Socket sdati = s.accept();
InputStream is = sdati.getInputStream();
OutputStream out = sdati.getOutputStream();
DataInputStream netIn = new DataInputStream(is);
DataOutputStream netOut = new DataOutputStream(out);
...
```

SINGLE-THREAD SERVER

```
public class SingleThreadedServer implements Runnable{

    protected int          serverPort   = 8080;
    protected ServerSocket serverSocket = null;
    protected boolean      isStopped    = false;
    protected Thread       runningThread= null;

    public SingleThreadedServer(int port){
        this.serverPort = port;
    }

    public void run(){
        synchronized(this){
            this.runningThread = Thread.currentThread();
        }
        try {
            this.serverSocket = new ServerSocket(this.serverPort);
        } catch (IOException e) {
            throw new RuntimeException("Cannot open port 8080", e);
        }

        while(! isStopped()){
            Socket clientSocket = null;
            try {
                clientSocket = this.serverSocket.accept();
            } catch (IOException e) {
                if(isStopped()) {
                    System.out.println("Server Stopped.") ;
                    return;
                }
                throw new RuntimeException(
                    "Error accepting client connection", e);
            }
            try {
                processClientRequest(clientSocket);
            } catch (Exception e) {
                //log exception and go on to next request.
            }
        }

        System.out.println("Server Stopped.");
    }

    private void processClientRequest(Socket clientSocket)
    throws Exception {
        InputStream  input  = clientSocket.getInputStream();
        OutputStream output = clientSocket.getOutputStream();
        long time = System.currentTimeMillis();
        byte[] responseDocument = …
        byte[] responseHeader = …
        output.write(responseHeader);
        output.write(responseDocument);
        output.close();
        input.close();
        System.out.println("Request processed: " + time);
    }
```

```
    private synchronized boolean isStopped() {
        return this.isStopped;
    }

    public synchronized void stop(){
        this.isStopped = true;
        try {
            this.serverSocket.close();
        } catch (IOException e) {
            throw new RuntimeException("Error closing server", e);
        }
    }
}
```

MULTI-THREAD SERVER IN JAVA

Anziché elaborare le richieste in entrata nello stesso thread che accetta la connessione client, la connessione viene trasferita a un thread worker che elaborerà la richiesta.

Funzionamento del loop nella versione multi-thread:

```
while(! isStopped()){
    Socket clientSocket = null;
    try {
        clientSocket = this.serverSocket.accept();
    } catch (IOException e) {
        if(isStopped()) {
            System.out.println("Server Stopped.") ;
            return;
        }
        throw new RuntimeException(
            "Error accepting client connection", e);
    }

        new Thread(
        new WorkerRunnable(
        clientSocket, "Multithreaded Server")
        ).start();

}
```

Classe WorkerRunnable passata al costruttore del worker thread :

```
package servers;

import java.io.InputStream;
import java.io.OutputStream;
import java.io.IOException;
import java.net.Socket;

public class WorkerRunnable implements Runnable{
    protected Socket clientSocket = null;
    protected String serverText   = null;

    public WorkerRunnable(Socket clientSocket, String serverText) {
        this.clientSocket = clientSocket;
        this.serverText   = serverText;
    }

    public void run() {
        try {
            InputStream input  = clientSocket.getInputStream();
            OutputStream output = clientSocket.getOutputStream();
            output.write("…").getBytes());
            output.close();
            input.close();
            System.out.println("Request processed: " + time);
        } catch (IOException e) {
            //report exception somewhere.
            e.printStackTrace();
        }
    }
}
```

THREAD POOLED SERVER

Al posto di iniziare un nuovo thread per ogni connessione in arrivo, la connessione è raccolta in un Runnable e passata ad una thread pool con un numero fissato di thread.

I Runnable sono all'interno di una **coda** nella thread pool. Quando un thread nella thread pool è inattivo prenderà un Runnable dalla coda e lo eseguirà.

```
while(! isStopped()){
     Socket clientSocket = null;
     try {
         clientSocket = this.serverSocket.accept();
     } catch (IOException e) {
         if(isStopped()) {
             System.out.println("Server Stopped.") ;
             break;
         }
         throw new RuntimeException(
            "Error accepting client connection", e);
     }

        this.threadPool.execute(
        new WorkerRunnable(clientSocket, "Thread Pooled Server"));

 }
```

L'unico cambiamento nel loop per il server multi-thread è la parte in grassetto.
Al posto di iniziare un nuovo thread per ogni connessione in arrivo, il WorkerRunnable è passato alla thread pool per l'esecuzione quando un thread nel pool diventa inattivo.

```
…
public class WorkerRunnable implements Runnable{
    protected Socket clientSocket = null;
    protected String serverText   = null;

    public WorkerRunnable(Socket clientSocket, String serverText) {
        this.clientSocket = clientSocket;
        this.serverText   = serverText;
    }

    public void run() {
        try {
            InputStream input  = clientSocket.getInputStream();
            OutputStream output = clientSocket.getOutputStream();
            output.write(("…").getBytes());
            output.close();
            input.close();
            System.out.println("Request processed: " + time);
        } catch (IOException e) {
            //report exception somewhere.
            e.printStackTrace();
        }
    }
}
```

```
ThreadPooledServer server = new ThreadPooledServer(9000);
new Thread(server).start();

try {
    Thread.sleep(20 * 1000);
} catch (InterruptedException e) {
    e.printStackTrace();
}
System.out.println("Stopping Server");
server.stop();
```

Non saremo in grado di accettare più richieste del numero di thread disponibili sulla nostra macchina. Quindi se ci aspettiamo più connessioni dobbiamo trovare un'alternativa.

JAVA NIO (NEW IO)

API non bloccante per le connessioni socket, quindi non sono legati al numero di thread disponibili.
Elementi principali:

- **Channel:** i canali sono la combinazione di input e output stream, così da permettere di leggere e scrivere utilizzando i buffer.
- **Buffer:** Blocco di memoria usato per leggere da un Canale e scriverci sopra.
- **Selector:** Un selettore può registrare canali multipli e controllerà quali sono pronti per accettare nuove connessioni. Similmente al metodo *accept()* dei blocking IO, quando *select()* è invocata, bloccherà l'applicazione finché un canale è pronto per eseguire un'operazione. Perché un selettore può registrare molti canali
- **Selection Key**: Contiene proprietà per un particolare Canale, sono principalmente usate per conoscere l'interesse corrente del canale (isAcceptable(), isReadable(), isWritable()), prendere il canale e farci operazioni sopra.

Con questa libreria, un thread può gestire multiple connessioni contemporaneamente

```
1  var serverSocketChannel = ServerSocketChannel.open();
2  serverSocketChannel.configureBlocking(false);
3  serverSocketChannel.socket().bind(new InetSocketAddress(8080));
4
5  var selector = Selector.open();
6  serverSocketChannel.register(selector, SelectionKey.OP_ACCEPT);
7
8  while (true) {
9      selector.select();
10     var keys = selector.selectedKeys().iterator();
11
12     while (keys.hasNext()) {
13         var selectionKey = (SelectionKey) keys.next();
14
15         if (selectionKey.isAcceptable()) {
16             createChannel(serverSocketChannel, selectionKey);
17         } else if (selectionKey.isReadable()) {
18             doRead(selectionKey);
19         } else if (selectionKey.isWritable()) {
20             doWrite(selectionKey);
21         }
22         keys.remove();
23     }
24 }
25
```

1. Da riga 1 a 3, un ServerSocketChannel è creato e bisogna impostarlo esplicitamente come non bloccante. Il socket è configurato per ascoltare sulla porta 8080.
2. A riga 5 e 6, un Selector è creato e ServerSocketChannel è registrato sul selettore con una SelectionKey che punta a operazioni ACCEPT.
3. Per tenere l'applicazione in ascolto tutto il tempo, il metodo bloccante select() è dentro un loop infinito while e select() ritorna quando almeno un canale è selezionato, wakeup() è invocata o il thread è interrotto.
4. A riga 10, un insieme di chiavi è tornato dal Selector e itereranno su di loro in modo da eseguire i canali pronti. Ogni volta che una nuova connessione è creata, isAcceptable() sarà vera e un nuovo Channel sarà registrato nel Selector.

Esempio di creazione di canali

```
1  private static void createChannel(ServerSocketChannel serverSocketChannel, SelectionKey selectionKey) t
2      var socketChannel = serverSocketChannel.accept();
3      LOGGER.info("Accepted connection from " + socketChannel);
4      socketChannel.configureBlocking(false);
5      socketChannel.write(ByteBuffer.wrap(("Welcome: " + socketChannel.getRemoteAddress() +
6              "\nThe thread assigned to you is: " + Thread.currentThread().getId() + "\n").getBytes()));
7      dataMap.put(socketChannel, new LinkedList<>()); // store socket connection
8      LOGGER.info("Total clients connected: " + dataMap.size());
9      socketChannel.register(selectionKey.selector(), SelectionKey.OP_READ); // selector pointing to READ
10 }
```

1. Per tener traccia dei dati in ogni canale, sono inseriti in una Map con il socket come chiave ed una lista di ByteBuffers.

IO vs NIO

Java IO è orientato agli stream, questo significa che leggiamo uno o più byte alla volta, da un flusso. Quello che facciamo con i byte letti dipende da noi.
Non sono memorizzati nella cache da nessuna parte. Inoltre, non è possibile spostarsi avanti e indietro nei dati in uno stream. Se è necessario spostarsi avanti e indietro nei dati letti da un flusso, sarà necessario prima memorizzarli in un buffer.

I buffer di Java NIO hanno un approccio leggermente differente.
I dati vengono letti in un buffer dal quale vengono successivamente elaborati. Possiamo muoverci avanti e indietro nel buffer secondo se ne abbiamo bisogno.
Questo ci dà un po' più di flessibilità durante l'elaborazione. Tuttavia, dobbiamo anche controllare se il buffer contiene tutti i dati necessari per elaborarlo completamente. Inoltre, dobbiamo assicurarci che quando leggiamo più dati nel buffer, non sovrascriviamo i dati non ancora elaborati.

Blocking vs Non-Blocking IO

I vari flussi di Java IO sono bloccanti, ciò significa che quando un thread invoca read() o write(), quel thread viene bloccato finché non ci sono dati da leggere o i dati non vengono scritti completamente. Il thread non può fare nient'altro nel frattempo.

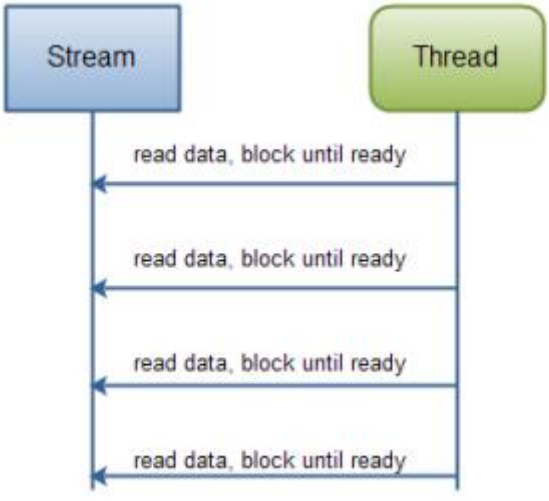
La modalità non bloccante di Java NIO consente a un thread di richiedere la lettura dei dati da un canale e di ottenere solo ciò che è attualmente disponibile, o nulla se non sono attualmente disponibili dati. Invece di rimanere bloccato fino a quando i dati non diventano disponibili per la lettura, il thread può continuare con qualcos'altro.

Lo stesso vale per le scritture non bloccanti. Un thread può richiedere che alcuni dati vengano scritti su un canale, ma non attendere che vangano scritti completamente. Il thread può quindi andare avanti e fare qualcos'altro nel frattempo.

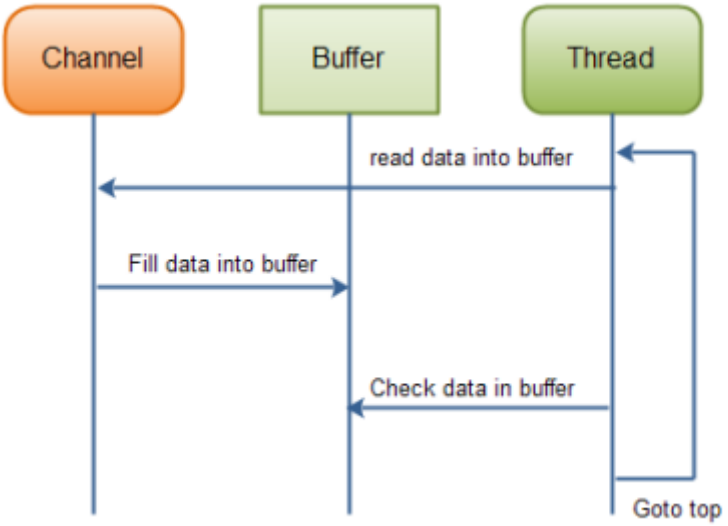
Quando i thread spendono il loro tempo in idle o quando non sono bloccati nelle chiamate IO, di solito eseguono IO su altri canali. Cioè un singolo thread può ora gestire più canali di input e output.

Selectors

I selettori di Java NIO permettono ad un singolo thread di monitorare multipli canali di input.
Possiamo registrare multipli canali con un selettore, poi usare un singolo thread per "selezionare" i canali che hanno input disponibili da processare o selezionare i canali che sono pronti per la scrittura.
Questo meccanismo di selezione rendere semplice per un singolo thread gestire diversi canali.



Java IO: Reading data from a blocking stream.

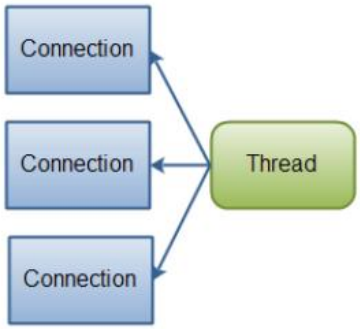


Java NIO: Reading data from a channel until all needed data is in buffer

SOMMARIO

NIO consente di gestire più canali (connessioni di rete o file) utilizzando solo uno (o pochi) thread, ma il costo è che l'analisi dei dati potrebbe essere un po' più complicata rispetto alla lettura dei dati da un flusso di blocco.

Se devi gestire migliaia di connessioni aperte contemporaneamente, ciascuna delle quali invia solo pochi dati, ad esempio un server di chat, implementare il server in NIO è probabilmente un vantaggio. Allo stesso modo, se è necessario mantenere molte connessioni aperte ad altri computer, ad es. in una rete P2P, l'utilizzo di un singolo thread per gestire tutte le connessioni in uscite potrebbe essere un vantaggio.
Questo design a un therad e connessioni multiple è illustrato in questo diagramma:



Java NIO: A single thread managing multiple connections.

Se ha meno connessioni con una larghezza di banda molto elevata, inviando molti dati alla volta, forse un'implementazione classica del server IO potrebbe essere la soluzione migliore. Questo diagramma illustra un classico design del server IO: