Dario Olianas A.A. 2014/2015

Appunti di Programmazione Concorrente e Algoritmi Distribuiti (PCAD)

1. Introduzione

Il paradigma sequenziale è profondamente diverso da quello concorrente: nel paradigma sequenziale le istruzioni sono totalmente ordinate, abbiamo un'unica memoria virtuale sia per i dati dichiarati staticamente che per quelli dichiarati dinamicamente e l'esecuzione avviene una singola istruzione per volta.
Il paradigma concorrente invece comporta parallelismo, che può essere reale (in caso di macchine multiprocessore o sistemi distribuiti) o virtuale (simulato attraverso la multiprogrammazione con scheduling). La programmazione concorrente sfrutta tecniche e features offerte dai linguaggi per sfruttare al massimo il parallelismo.
L'idea di base della programmazione concorrente è che le istruzioni sono ordinate solo parzialmente. Questo permette ad esempio, se stiamo calcolando il valore di un'espressione, di calcolare contemporaneamente le sottoespressioni che la compongono per arrivare prima al risultato finale, o per velocizzare il rendering di un'immagine.
La programmazione concorrente permette di sfruttare le architetture multiprocessore, migliorare la reattività delle interfacce grafiche, risolvere con maggiore naturalezza determinati problemi.
Questi sono gli aspetti più applicativi, che adesso interessano maggiormente, ma in origine la concorrenza è nata per migliorare design, comprensione e prestazioni dei sistemi operativi: è nata per permettere il multitasking.

1.1 Concorrenza nei sistemi operativi

Nei sistemi operativi, in particolare UNIX-like, la concorrenza è stata una scelta di design: il sistema operativo è infatti composto da un gran numero di attività eseguite più o meno contemporaneamente dal processore. Senza un modello adeguato a coordinare queste attività, la loro coesistenza sarebbe molto difficile: per questo è stato ideato il **modello concorrente**, fondato sul concetto di **processo**. Un processo è un programma in esecuzione, composto dal codice del programma e da tutte le informazioni necessarie all'esecuzione. A livello di sistema operativo, l'unità con cui si gestisce l'esecuzione è il processo. I processi che compongono il sistema operativo vengono eseguiti in parallelo, con parallelismo reale (multiprocessore) o apparente (multiprogrammazione su una singola CPU).
Più processi possono eseguire lo stesso programma: il codice viene condiviso ma i dati, l'immagine e lo stato rimangono separati, ogni istanza viene considerata come un processo separato. Le informazioni che identificano un processo sono:

- pid
- program counter (il punto a cui si è arrivati nell'esecuzione)
- stato (ready, wait, running, zombie)
- stack di chiamate
- tutto ciò che ha a che fare con le risorse: diritti di accesso, file descriptor aperti ecc.
- heap

Il sistema operativo (Unix) tiene traccia di tutti i processi in esecuzione tramite la **tabella dei processi**, che solitamente mantiene più informazioni per ogni processo (puntatore al codice, puntatore allo heap e tutto ciò che può servire). Ma queste informazioni non bastano: il sistema operativo deve decidere anche come alternare i processi in esecuzione. Questo compito è demandato allo scheduler, che può adottare diverse strategie: una possibile soluzione è una lista linkata collegata alla tabella dei processi, che dice con che ordine vanno messe in esecuzione.

Lo scheduling è una forma di euristica molto complessa: il SO non può perdere troppo tempo a decidere quale processo mandare in esecuzione, ma non può neanche farlo a casaccio. Non è l'unica scelta critica che deve essere fatta dallo scheduler: oltre all'ordine di esecuzione dei processi deve infatti decidere anche ogni quanto cambiare processo in esecuzione (**context switch**). Una scelta sensata è fare context switch subito dopo un'operazione di I/O: è infatti probabile che ci sia un processo che deve elaborare i dati ricevuti. Quando fa context switch, lo scheduler deve salvare lo stato del processo in esecuzione e ripristinare lo stato del processo da mandare in esecuzione. È un'operazione talmente critica che solitamente l'hardware fornisce istruzioni per fare context switch: potrebbe infatti essere troppo costoso farlo solo a livello software. Il modello a processi è stato quindi integrato nell'hardware (questa cosa succede spesso: un'invenzione a livello applicativo scende tutti i livelli di astrazione fino ad arrivare all'hardware, che diventa più potente).

1.1.1 Processi Unix

Un processo UNIX ha tre segmenti: stack, dati (suddivisi in dati statici ed heap) e codice. L'indirizzamento è virtuale: lo stack è memorizzato nella parte iniziale della memoria virtuale, lo heap in quella finale. L'unico modo di creare un nuovo processo è la funzione fork(), che crea un processo figlio clonato dal codice del processo padre. È un operazione di sistema, pensata per chi sviluppa a livello di sistema, non a livello applicativo. La clone() è una versione di fork() ancora più difficile da usare perché il figlio invece di avere una propria copia dei dati condivide i dati del padre. Per eseguire un programma diverso nel figlio si usa execve.
//mi sa che manca qualcosa

Per risparmiare context switch, in UNIX i processi possono operare in modalità user e in modalità kernel: in questo modo se ad esempio deve essere gestito un interrupt può essere fatto senza fare context switch, ma semplicemente passando da modalità user a modalità kernel. Il processo è però sempre lo stesso, cambia solo il livello di priorità assegnato (un processo in modalità utente infatti non può accedere allo spazio di indirizzi del kernel). E' possibile avere più livelli di contesto: livello utente e vari livelli kernel a seconda del motivo dell'intervento del kernel: system call o interrupt (possono esserci vari livelli a seconda del tipo di interrupt).

L'algoritmo per la gestione degli interrupt deve:

- salvare il contesto del processo corrente
- determinare la fonte dell'interrupt (trap, interrupt I/O ecc.)
- recuperare l'indirizzo di partenza dell'interrupt handler appropriato per quel tipo di interrupt
- invocare l'interrupt handler
- una volta terminata la gestione dell'interrupt, ripristinare il livello di contesto precedente

Per motivi di efficienza, parte della gestione di interrupt/trap viene svolta direttamente dal processore.

Il context switch è un operazione molto critica, perché manipola le strutture dati del kernel: può avvenire solo in determinati momenti della vita di un processo, il kernel vieta context switch arbitrari. Un context switch può avvenire

- quando un processo si sospende
- quando termina
- quando torna alla modalità utente dopo una system call, ma non è più il processo a più alta priorità
- quando torna alla modalità utente dopo che il kernel ha terminato la gestione di un interrupt, e il processo non ha più la massima priorità

Il kernel decide solo quando può avvenire il context switch: la decisione su quale processo eseguire è lasciata allo scheduler.

1.1.2 Thread

I processi, per come li abbiamo visti finora, sono unità di allocazione risorse (dati, sia statici che dinamici, permessi di accesso, risorse gestite dal kernel ecc.) e unità di esecuzione (un percorso di esecuzione attraverso uno o più programmi, costituito da stack di chiamate, stato, priorità). Queste due caratteristiche possono essere separate: i thread infatti sono unità di esecuzione ma non di allocazione: le risorse allocate appartengono al processo e sono condivise dai thread che convivono in esso: sono propri di ogni thread un program counter, un insieme di registri, uno stack di chiamate e uno stato di esecuzione; sono condivisi dai thread di uno stesso processo il codice eseguibile, i dati e le risorse richieste al sistema operativo.
L'utilizzo di thread porta una maggiore efficienza, soprattutto in fase di creazione: creare un thread è tra le 100 e le 1000 volte più veloce che creare un processo, perché ci sono meno informazioni da duplicare/creare/cancellare, a volte non serve neanche la system call. È più veloce anche la schedulazione, perché il context switch tra thread è più leggero. La comunicazione è più veloce perché non supera i confini del processo: non devo utilizzare gli strumenti di passaggio di messaggi offerti dal kernel (basati su buffer allocati nel segmento kernel), posso usare le strutture dati interne al processo.
Lo svantaggio dell'utilizzo di thread è che il programma va pensato parallelo, e questo aumenta notevolmente la complessità di programmazione: dovremo anche tenere conto della sincronizzazione, evitando problemi come deadlock e race conditions. Diminuisce anche l'information hiding, perché i thread dovranno comunicare tra loro. Inoltre, in alcuni casi lo scheduling potrebbe essere demandato all'utente.
Quando serve il multithreading? Ad esempio, quando il lavoro va suddiviso in foreground e background: un thread che gestisce l'I/O con l'utente, altri thread che operano in background sui dati. Anche in caso di operazioni asincrone, come il salvataggio automatico, devono fare uso di thread. Ci sono poi operazioni intrinsecamente parallele: web server e DBMS devono operare in parallelo per non far attendere i client.

User Level Thread

In un sistema operativo che utilizza User Level Thread stack, program counter e operazioni sui thread vengono implementate a livello utente. Questa scelta migliora l'efficienza perché non richiede costose system call per gestire i thread, li rende semplici da implementare su sistemi preesistenti e permette di personalizzare lo scheduling. Quest'ultimo è sia un vantaggio che uno svantaggio: non avendo lo scheduling automatico se un thread prende la CPU e non la molla più gli altri thread del processo non saranno eseguiti mai. Inoltre, se un thread fa una system call

bloccante vengono bloccati tutti i thread del processo, non solo quello che la chiamata: non è un comportamento desiderabile. Inoltre, l'accesso al kernel è sequenziale, non vengono sfruttati i sistemi multi processore ed è poco utile per i processi I/O bound come i file server.
Questa implementazione dei thread era usata da Mac OS (fino al 9), CMU e alcune implementazioni POSIX: le prime implementazioni dei thread erano a livello utente, con accorgimenti per evitare il problema delle system call bloccanti.

Kernel Level Thread
Quando i thread sono gestiti dal kernel la granularità dello scheduling di sistema è a livello di thread, non di processi: un thread che si blocca non bloccherà l'intero processo. È utile per processi I/O bound e permette di sfruttare i sistemi multiprocessore. Gli svantaggi sono il calo dell'efficienza, perché dovremo usare system call; la necessità di modifiche al sistema operativo per aggiungere le system call di gestione dei thread; infine il fatto di non poter gestire manualmente lo scheduling. È questa l'implementazione adottata da Unix.

Implementazioni ibride ULT/KLT
Un'implementazione ibrida è quella che utilizza thread a livello utente mappati su thread a livello kernel (non necessariamente in mapping 1:1). I vantaggi sono tutti quelli degli approcci precedenti. È la scelta usata da Windows NT, Mac OS X, Linux (pthread).

1.2 Pseudocodice per programmi concorrenti
Notazione in pseudocodice per programmi concorrenti:

```
cobegin
..statement 1...
||
...statement 2...
||
...
||
...statement K...
coend
```

Ogni statement 1..k è eseguito in concorrenza, e le istruzioni successive al coend vengono eseguite solo quando tutti gli statement concorrenti sono terminati. È un ordinamento parziale delle istruzioni: le istruzioni fuori dal cobegin/coend sono ordinate, quelle dentro no.
Supponiamo di voler definire un algoritmo parallelo di Mergesort, usando le funzioni sort(v,i,j) che ordina gli elementi dell'array v dall'indice i all'indice j e la funzione merge(v,i,j,k) che fa merge dei due segmenti (già ordinati) di v che vanno da i a j e da j a k.

La funzione mergesort sarà:

```
mergesort(v,l) {
   m = l/2;
   cobegin
        sort(v,1,m);
    ||
        sort(v,m+1,l);
    ||
         merge(v,1,m,l);
```

```
     coend;
}
```

Questa soluzione non è corretta perché non so cosa viene eseguito per primo: potrebbe anche venir eseguita prima la merge su array non ancora ordinati.

La soluzione corretta è:

```
mergesort(v,l) {
   m = l/2;
   cobegin
        sort(v,1,m);
    ||
      sort(v,m+1,l);
    coend;
        merge(v,1,m,l);
}
```

In questo modo, non so quale delle due metà ordino per prima e non mi importa, ma sono sicuro che al momento del merge entrambe le metà saranno ordinate.

Un'altra possibile sintassi è:

```
Risorse condivise
Dichiarazioni di variabili

Processo P1 {...istruzioni...}
....
Processo PK {...istruzioni...}
```

P_1,...,P_k sono programmi sequenziali eseguiti in parallelo tra loro.

Esempio:

```
Risorse
var x=0

Processo P1 { x := 500; }
Processo P2 { x:=0; }
Processo P3 { write(x); }
```

Quale sarà l'output del programma? Intuitivamente o 0 o 500, a seconda di come vengono schedulati i tre processi. In realtà, dipende dal livello di atomicità delle operazioni: se l'assegnazione non è eseguita atomicamente potrebbe produrre valori non ben definiti (se avviene un context switch durante l'assegnazione).

Quelli che vediamo ora sono esempi di **lock free programming**: programmi eseguiti in concorrenza ma senza meccanismi di sincronizzazione (lock, monitor, semafori ecc.)
In contesti reali ovviamente non basta, come si vede da questo esempio

```
Risorse
var x = 10

Processo P1 {
    x:= x+1;
}

Processo P2 {
    x:= x-1;
}
```

Questo codice provoca una **race condition**: il risultato cambia a seconda di chi viene eseguito per primo. Se l'assegnazione non è atomica[1], infatti, un processo potrebbe venire interrotto quando non ha ancora finito di salvare il risultato in x: in questo caso l'altro processo valuterà la sua espressione con un risultato errato di x.
Posso rappresentare la non atomicità dell'assegnazione scrivendone il codice in questo modo:

```
Risorse
var x = 10

Processo P1 {
    var aux1;
    aux1:= x+1;
    x:=aux1;
}

Processo P2 {
    var aux2;
    aux2:= x-1;
    x:=aux2;
}
```

Andare a vedere come viene eseguito un programma concorrente vuol dire assegnare un ordinamento totale alle sue istruzioni (che hanno ordinamento parziale: sono ordinate all'interno di uno stesso processo, ma non è definito a priori un ordine tra le istruzioni di processi diversi).

[1] **istruzione atomica**: istruzione di linguaggio ad alto livello che viene tradotta in una singola istruzione assembly

2. Sincronizzazione

La sincronizzazione può essere fatta sia lock free (a livello più basso) sia utilizzando i lock (che sono un meccanismo di alto livello fornito da librerie).
Problema del produttore-consumatore: abbiamo due processi: un processo produttore che produce degli elementi che saranno ricevuti da un processo consumatore che li utilizza. Come vengono passati gli elementi? In una soluzione a memoria condivisa, si alloca un buffer (di dimensione prefissata) per la comunicazione tra i due processi.

Possibile soluzione a memoria condivisa:

Inizializzazione variabili globali

```
type item = ...;
var buffer = array [0...n-1] of item;
in, out: 0...n-1;
counter: 0..n;
in, out, counter = 0;
```

Nel processo produttore, siccome il buffer ha dimensione limitata, bisognerà stare attenti a non sovrascrivere elementi non ancora consumati dal processo consumatore: la variabile counter serve a questo.

Processo produttore:

```
while true do
produci un item di nome nextp
while counter = n do skip endwhile
buffer[in] := nextp;
in = in +1 mod n;
counter++;
endwhile
```

Il while counter = n serve ad uscire dal ciclo se counter = n (buffer pieno). Se il buffer non è pieno, mette nextp nella prima posizione libera e aggiorna l'indice con una somma circolare: se supera le dimensioni del buffer ricomincia dall'inizio. Infine, incrementa il contatore. Il processo consumatore lavorerà in maniera speculare

Processo consumatore:

```
while true do
while counter = 0 do skip endwhile
nextc := buffer[out];
out := out +1 mod n;
counter--;
...
consuma l'item in nextc
...
end
```

Questo approccio presenta alcuni problemi: se il produttore termina, il consumatore rimane bloccato: è un caso di livelock, in cui un processo rimane bloccato in attesa per sempre. Inoltre, se le istruzioni counter++ e counter-- non vengono eseguite atomicamente possono verificarsi race conditions.
Quindi questa implementazione non è corretta, visto anche che il modello produttore-consumatore solitamente è presente anche nel kernel, e se si verificassero race conditions sarebbe compromessa la stabilità dell'intero sistema.
Le **race conditions** sono errori che si verificano quando più processi accedono concorrentemente agli stessi dati, e il risultato dipende dall'ordine di esecuzione dei processi. Sono errori frequenti nei sistemi operativi multitasking ed estremamente pericolose perché portano al malfunzionamento di più processi: se si verificano in strutture kernel space, addirittura dell'intero sistema.
Sono anche estremamente difficili da individuare e riprodurre, perché dipendono da situazioni fuori dal controllo dello sviluppatore (ordine di scheduling, determinato da molti parametri come carico del sistema, utilizzo della memoria, numero di processori disponibili).

2.1 Problema della sezione critica

È un modo che è stato pensato per formalizzare la gestione delle race conditions nei sistemi operativi. È stato studiato molti anni fa, ma è ancora valido.
Abbiamo n processi che competono per usare dati condivisi: la parte di processo in cui vengono utilizzati i dati condivisi è detta sezione critica. Perché non si verifichino race conditions, bisogna assicurarsi che possa essere eseguita solo una sezione critica per volta: quando un processo sta eseguendo la sua sezione critica, nessun altro processo deve poter eseguire la propria. È necessario uno strumento di controllo per entrare nella sezione critica, qualcosa che garantisca la **mutua esclusione**.
Per come è stato descritto fin qui, il problema ammette anche soluzioni ovvie (ma inutili), come ad esempio che nessuno esegua la sua sezione critica. La mutua esclusione è garantita, ma non è la soluzione che volevo.
Per questo le soluzioni al problema della sezione critica deve avere tre proprietà:

1. **Mutua esclusione**: se il processo Pi sta eseguendo la sua sezione critica, allora nessun altro deve poter eseguire la proprio sezione critica
2. **Progresso**: se nessun processo sta eseguendo la sua sezione critica ed esiste un processo che desidera entrare nella propria sezione critica, deve poterlo fare; l'esecuzione di tale processo non può essere posposta indefinitamente: elimina la soluzione del non uso della risorsa, quella in cui nessuno esegue la sua sezione critica. Il deadlock rappresenta una violazione del progresso.
3. **Attesa limitata (bounded waiting)**: se un processo P ha richiesto di entrare nella propria sezione critica, allora il numero di volte che si concede agli altri processi di accedere alla propria sezione critica prima del processo P deve essere limitato. Elimina la soluzione dell'uso esclusivo, quella in cui un solo processo usa la risorsa per sempre. La starvation (un processo che non riesce mai ad accedere alla risorsa) rappresenta una violazione del bounded waiting.

Sono necessarie alcune assunzioni: bisogna sapere a priori quali istruzioni sono atomiche e quali no; bisogna assumere anche che ogni processo rimanga nella sezione critica per un tempo finito e che venga eseguito ad una velocità non nulla (un processo non può fermarsi da solo senza ragione). Non poniamo vincoli sull'architettura o sullo scheduling (può esserci un qualunque

numero di CPU ed una qualunque politica di scheduling che non possiamo controllare e che non conosciamo).

Soluzione: flag condiviso
Mantengo una variabile booleana condivisa che vale 1 quando qualcuno sta usando la sezione critica, vale 0 quando nessuno la sta usando. Ogni processo prima di entrare in sezione critica setterà a 1 il flag e lo setterà a 0 quando esce.

```
var occupato: (0, 1);
occupato = 0
```

Processo Px:
```
while true do
     while (occupato != 0) do skip endwhile;
           occupato = 1;
                 // sezione critica
          occupato = 0
             // sezione non critica
       endwhile
```

Che problema ha questa soluzione? Che abbiamo semplicemente spostato la race condition dalla sezione critica al flag di controllo: lo scheduler passa il controllo subito dopo che un processo ha eseguito il ciclo di controllo ma prima di aver settato il flag a 1, c'era un altro processo nello stesso stato, quindi tutti e due riescono ad ottenere l'accesso alla sezione critica contemporaneamente (violazione della mutua esclusione).

Soluzione: alternanza stretta
Come variabile di controllo uso una variabile turn (0..n): quando turn vale i il processo Pi può eseguire la sezione critica. Il problema qui è se un processo ha una sezione non critica molto pesante che non termina mai: in questo caso quando arriverà il suo turno non potrà entrare nella sezione critica perché bloccato nell'eseguire la computazione della sua sezione non critica, ma neanche gli altri processi potranno entrarci perché il loro turno non arriverà mai (dovrebbe infatti essere il processo bloccato a cambiare il valore di turn una volta eseguita la sua sezione critica).

Nel caso in cui abbiamo 2 processi, il problema si risolve con l'**algoritmo di Peterson per 2 processi**, che mette insieme l'alternanza stretta con la condivisione di un flag booleano. È un algoritmo lock-free di mutua esclusione: sincronizza le azioni dei processi senza usare il lock in maniera esplicita (sfrutta spin lock e busy wait: cicli che simulano una condizione di attesa).
Il problema dell'algoritmo del flag booleano è che è troppo simmetrico: tutti i processi fanno la stessa cosa, senza badare a quello che fanno gli altri. Il problema dell'algoritmo dell'alternanza stretta è che l'autorizzazione al passaggio viene data dal processo stesso all'uscita della sezione critica: se si blocca, si blocca tutto.
In Peterson abbiamo come risorse condivise una variabile per il turno e due variabili booleane, che rappresentano l'interesse dei due processi ad entrare nella sezione critica. Nel processo P0, all'interno di un ciclo infinito setta il suo booleano a true e la variabile turn a 0 (assumendo che 0 sia per il processo 0 e 1 sia per il processo 1). Subito dopo, controllo se turn è sempre 0 e se l'altro processo è interessato: se è così salto ad endwhile. Se supero questo spinlock sono nella sezione critica: alla fine setterò di nuovo interested0 = false.

Vediamo che c'è una race condition su turn, e che turn non indica il processo che deve essere eseguito: indica chi è arrivato per ultimo. Se lo spinlock trova turn=0, infatti, vuol dire che P0 è stato l'ultimo processo a scriverla, quindi è arrivato per ultimo, quindi verrà eseguito P1 (assumiamo atomica l'assegnazione).
Questo algoritmo garantisce tutte e 3 le proprietà, però funziona solo per due processi. Come lo generalizzo per N processi? È possibile adottare una soluzione in cui Peterson è solo un modulo: i processi si sfidano due a due finché non ne rimane solo uno che guadagna il diritto di entrare in sezione critica.
Esercizio: provare una soluzione per N processi che usa Peterson come sottoprocedura.

Algoritmo di Lamport per N processi
L'algoritmo di Lamport è basato sull'idea di fornire una forma di turnazione dinamica: voglio costruire una coda di processi di attesa per entrare in sezione critica. Devo però trovare un modo di gestire una coda in maniera condivisa.
L'algoritmo deve costruire una priorità tra i processi, e lo fa in modo simile alla coda dal panettiere: ognuno ha un numero e si viene serviti dal più basso al più alto. Prima di entrare in sezione critica, ogni processo riceve un numero, e chi ha il numero più basso entra. Se due processi Pi e Pj ricevono lo stesso numero, viene servito quello con l'indice (i o j) minore.

I biglietti sono identificati da un array di numeri, di dimensione N uguale al numero di processi. Il biglietto 0 è speciale: indica che il processo non è interessato alla sezione critica.

2.2 Sincronizzazione ed ottimizzazione

Nel mondo ideale, l'algoritmo di Peterson ha lo stesso identico effetto di un lock. In realtà, le ottimizzazioni del processore nelle architetture moderne interferiscono con l'esecuzione dell'algoritmo, come possiamo vedere dai risultati del codice di test. In breve: ci sono due thread eseguiti concorrentemente e sincronizzati con l'algoritmo di Peterson, e la loro sezione critica consiste semplicemente nell'incremento del valore di una variabile condivisa (che all'inizio vale 0). Il programma prende in input il numero di volte che vogliamo eseguire i thread: quindi al termine di tutte le esecuzioni ci aspetteremmo che la variabile condivisa valga due volte il numero di esecuzioni (ogni esecuzione di ogni thread la incrementa di 1). In realtà, se diamo in input numeri molto grandi vediamo che il contatore vale leggermente meno del numero di esecuzioni (su Windows 7 64 bit, CPU Intel i7 2630QM si inizia a notare per valori tra 10000 e 100000). A cosa è dovuto? Non c'è modo di scoprirlo via software, in questi casi, anche l'uso di un debugger influisce sul risultato dell'esecuzione. Il problema infatti è causato dalle ottimizzazioni applicate dal processore, in particolare dalla bufferizzazione delle scritture.

Nel codice su AulaWeb, alcune variabili sono segnate come volatile: questo impedisce al compilatore di riordinare le istruzioni che lavorano su quelle variabili (il compilatore solitamente riordina le istruzioni per ottimizzare l'esecuzione). In programmi concorrenti però questa ottimizzazione può essere pericolosa. È diverso dal volatile di Java, lo vedremo quando faremo la concorrenza in Java.

Al giorno d'oggi, il parallelismo è a tutti i livelli. Sicuramente a livello applicativo, ogni applicazione minimamente complessa oggi è multithreaded, ma si spinge fino al livello dell'hardware.

Differenza tra parallelismo e concorrenza: scopo del parallelismo è rendere più efficiente l'esecuzione di codice eseguendone in parallelo alcune parti, ma volendo conservare un comportamento deterministico. La concorrenza invece introduce modularità nel codice, sfruttando quando possibile il parallelismo attraverso il multithreading. Il risultato dell'esecuzione spesso è non deterministico. Il parallelismo guarda più all'efficienza, la concorrenza invece è una questione di design del programma.
A livello software il parallelismo consiste nel **multithreading**, solitamente eseguito su hardware multicore: sperabilmente, i diversi thread saranno mappati su diversi core ed eseguiti in parallelo. L'**hyperthreading** (trademark Intel) è l'implementazione a livello hardware del multithreading: un processore single core in grado di eseguire istruzioni da più thread contemporaneamente (due core logici in un solo core fisico). Attraverso il SIMD invece una singola istruzione viene eseguita in parallelo da più ALU.
I diversi core installati su uno stesso chip hanno una loro cache privata, ma condividono anche un livello di cache comune.

L'anomalia che abbiamo notato nell'esecuzione del codice di Peterson può essere dovuta al caching delle scritture: ci sono dei momenti in cui l'incremento della variabile condivisa è ancora cached. La programmazione lock free infatti richiede istruzioni di basso livello, per agire sulle ottimizzazioni hardware: nel codice che abbiamo usato non ce ne sono quindi abbiamo risultati imprevisti.

2.2.1 Memory model

Per avere consapevolezza completa di quello che succede durante l'esecuzione, il programmatore deve conoscere il modello di memoria su cui sta lavorando. Il memory model è la specifica formale di come la gestione della memoria appare al programmatore, che serve ad eliminare il gap tra il comportamento atteso del programma ed il suo comportamento reale a runtime. Il memory model deve specificare anche l'interazione tra thread e memoria: quale valore viene restituito da una lettura, quando una scrittura diventa visibile agli altri thread, quali ottimizzazioni verranno applicate e quali effetti hanno.
In un modello **single thread** ad esempio, sappiamo che gli accessi in memoria vengono eseguiti uno per volta: quindi una read dovrà restituire come risultato il valore che l'ultima write ha scritto in quella posizione. Le ottimizzazioni possono cambiare l'ordine delle istruzioni purché la semantica rimanga inalterata
Nel modello **multi threaded strict consistency** ogni operazione in memoria viene vista nell'ordine temporale in cui viene eseguita da diversi thread: questa è una condizione molto forte, praticamente impossibile da raggiungere in un sistema reale: l'hardware dovrebbe essere in grado di fare tutto quello che facciamo a livello software, e andrebbero eliminate tutte le ottimizzazioni.
Solitamente si usano condizioni meno stringenti, come la **sequential consistency** (introdotta da Lamport): un sistema multithreaded è sequentially consistent se il risultato di ogni sua esecuzione è lo stesso di un'esecuzione sequenziale di tutti i thread e le istruzioni di ogni thread sono eseguite nell'ordine del programma corrispondente. Richiede quindi le proprietà di **program order** (rispetta l'ordine in cui le istruzioni sono scritte nel programma) e di **write atomicity** (tutte le scritture devono apparire a tutti i thread nello stesso ordine)
Nell'esempio dell'algoritmo di Peterson, una possibile violazione della mutua esclusione si ha quando una modifica del flag non è propagata immediatamente al secondo thread: se P1 modifica il suo flag (arrivando per primo) ma P2 non riesce ancora a vedere la modifica riuscirebbero ad entrare entrambi in sezione critica. Questo può verificarsi a causa della bufferizzazione delle scritture: le istruzioni di scrittura vengono "saltate" e messe in coda in un buffer, per farle tutte

insieme ottimizzando così l'esecuzione. Per evitare inconsistenze il processore controlla che scritture sta ottimizzando: ma nel nostro esempio lo fa per flag1 e flag2, che sono due variabili separate, quindi autorizza l'ottimizzazione. Quindi, vengono fatte prima le letture dei flag, che vengono trovati entrambi a zero: le scritture, che dovevano prenotare l'esecuzione, verranno eseguite dopo: ma ormai è troppo tardi, tutti e due i thread hanno trovato il flag dell'altro a 0 e sono entrati contemporaneamente in sezione critica: ciao ciao mutua esclusione.
La consistenza sequenziale a livello hardware è garantita solo da processori fino circa al 1989: adesso per ottenerla bisogna utilizzare strategie software (tipo i lock). Questo perché la scelta ottimizzazione/consistenza è un trade-off: più ottimizzo, meno ho consistenza. Per questo oggi si utilizzano librerie per la programmazione multi-threaded.
Quando abbiamo un memory model weak, sono necessari costrutti di sincronizzazione come volatile di Java, synchronized e librerie atomiche. Esistono anche tecniche miste di compilazione/hardware come le memory barrier, che assicurano che tutte le read successive alla barriera vengano eseguite solo dopo il completamento delle write precedenti.

2.3 Programmazione lock-free

La programmazione concorrente può richiedere la conoscenza dell'architettura sottostante. La programmazione concorrente lock-free è una tecnica usata per ridurre l'overhead che si verifica quando si usano strumenti di sincronizzazione di alto livello (lock, semafori, monitor). Un programma lock-free è un programma multithreaded che usa risorse condivise e che non presenta interleaving che possono bloccare i thread (in deadlock o livelock). Per fare questo vengono usate istruzioni di basso livello come le **memory fence**, che garantiscono la write atomicity. Su x86, l'istruzione mfence garantisce che ogni istruzione di load/store effettuata prima della chiamata ad mfence sia visibile a tutti.
Esistono altre istruzioni hardware per prevenire le race conditions: ci sono istruzioni per disabilitare l'interrupt e istruzioni di atomic swap.
Ci sono istruzioni che permettono al processo di disabilitare tutti gli interrupt hardware (quindi compresi quelli relativi al context switch) all'ingresso della sezione critica. In questo modo la mutua esclusione è garantita, ma è una soluzione pericolosissima: il processo può acquisire il controllo completo del processore, non liberandolo più. Inoltre, non è scalabile a sistemi multiprocessore, a meno di non bloccare tutti i processori. È quindi assolutamente inadatto alla sincronizzazione di processi utente: può essere usato solo per brevissimi segmenti di codice affidabile (solitamente del kernel).
Un meccanismo più ragionevole è quello basato sulle istruzioni di **test and set**: un costrutto che permette di modificare atomicamente il contenuto di una cella di memoria e di ottenere come valore di ritorno il precedente contenuto della cella (l'atomicità è garantita solitamente bloccando il bus di memoria).
Altre istruzioni speciali sono le **compare and swap**, che permettono di scambiare atomicamente due variabili.
Gli esempi di test and set e compare and swap delle slides sono basati su spinlock, che portano a busy wait (attesa attiva, con alto consumo di CPU) e inversione di priorità (un processo a bassa priorità che blocca una risorsa è continuamente ostacolato nella sua esecuzione da un processo a priorità maggiore in busy wait sulla stessa risorsa). Un'idea migliore è cambiare lo stato del processo quando è in attesa di un evento (metterlo in wait), e rimetterlo in ready quando l'evento avviene. Ad esempio il processo può autosospendersi chiamando sleep() quando entra in attesa: quando l'attesa è terminata, un altro processo dovrà risvegliarlo chiamando wakeup(<pid del processo da risvegliare>). Sleep() e wakeup(pid) sono però chiamate di sistema, quindi costose.

3. Semafori e monitor
3.1 Semafori
Introdotti da Dijkstra, i semafori sono un meccanismo di segnalazione che facilita la sincronizzazione tra thread, in un contesto in cui ho due o più processi che comunicano attraverso segnali, in modo che un processo possa essere bloccato finché non riceve un segnale da un altro. Un semaforo è un tipo di dato astratto con due operazioni **V (up)** che viene invocata per inviare un segnale di evento avvenuto e **P (down)**, potenzialmente bloccante, che attende un segnale di evento avvenuto. Può essere implementato come un contatore S intero condiviso, su cui possiamo invocare queste due operazioni: l'invocazione di down(S) (o P(S)) aspetterà che S diventi maggiore di 0, quindi lo decrementerà; l'operazione up(S) (o V(S)) lo incrementerà. Perché il sistema funzioni, le operazioni P e V devono essere atomiche.

Invariante dei semafori:

- Siano:
  - $\mathbf{n_p}$ il numero di azioni P/down completate
  - $\mathbf{n_v}$ il numero di azioni V/up
  - **init** il valore iniziale del semaforo (cioè il numero di risorse disponibili per i processi)
- allora vale la proprietà invariante $\mathbf{n_p <= n_v + init}$
- Se **init = 0** allora $\mathbf{n_p <= n_v}$ (il numero di attese dell'evento non deve essere superiore al numero di volte che l'eveto si è verificato)
- Se **init > 0** allora $\mathbf{n_p < n_v + init}$: il numero di richieste soddisfatte (n_p) non deve essere superiore al numero iniziale di risorse (init) + il numero di risorse restituite (n_v).

Come risolvo il problema della sezione critica con un semaforo?

```
Risorse condivise:
var mutex : semaphore
mutex = 1

processo Pi:

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

I semafori possono essere usati anche per risolvere il problema del produttore-consumatore. In questo caso me ne serviranno 3

```
Risorse condivise:
var mutex : semaphore
var empty : semaphore
var full : semaphore

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

Processo produttore:

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

Vediamo che i semafori non vengono usati solo per la sincronizzazione, ma anche per la comunicazione. Il processo consumatore si comporterà in modo speculare:

Processo consumatore:

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

L'implementazione classica dei semafori è basata su busy wait. Se implementati a livello kernel è possibile limitare l'utilizzo di busy waiting, facendo in modo che l'operazione down sospenda il processo che la invoca se il semaforo è < 0 e che l'operazione up risvegli uno dei processi sospesi in attesa. Il SO dovrà quindi mantenere una struttura dati contenente l'insieme dei processi sospesi, da cui ne verrà selezionato uno quando un processo deve essere svegliato. Come lo scelgo? È possibile adottare una politica FIFO, per garantire l'attesa limitata: il processo che si è messo in attesa per primo sarà il primo ad essere risvegliato. Quindi, una possibile implementazione a livello kernel dei semafori è con un record che ha un campo valore e un campo lista dei processi, e su questo tipo record potrò invocare le operazioni di up e down.
Il tipo del semaforo è definito così:

```
type semaphore = record
     value : integer;
     L : list of process;
end;
```

Le operazioni up e down saranno definite così:

Down(S) / P(S):

```
S.value := S.value - 1;
if S.value < 0
     then begin
          aggiungi questo processo a S.L;
          sleep();
     end;
```

Up(S) / V(S):

```
S.value := S.value +1;
if S.value < = 0
      then begin
            togli il processo Pid da S.L
            wakeup(pid);
      end;
```

In questa implementazione value può avere anche valori negativi: indica quanti processi sono in attesa su quel semaforo. Infatti, quando value è minore di 0 nella down il processo viene inserito nella coda e messo in attesa, nella up invece viene tolto un processo dalla coda e risvegliato. Le due operazioni up e down hanno una sezione critica fino alla sleep/wakeup: si può risolvere con la disabilitazione degli interrupt (ma non in sistemi multiprocessore), con istruzioni test and set o con uno spinlock: quest'ultimo caso è sufficientemente efficiente visto che la sezione critica è molto piccola. Quindi i semafori non eliminano completamente la busy waiting, ma la limitano solamente alle brevi sezioni critiche delle operazioni up e down: senza semafori avremmo busy waiting meno frequenti ma di lunga durata, con i semafori invece abbiamo busy waiting frequenti ma più brevi.

3.2 Monitor

Un **monitor** è un tipo di dato astratto che protegge le strutture dati dei programmi concorrenti, garantendo thread-safeness. Associa un insieme di operazioni ad una struttura dati comune a più thread, ed il compilatore verificherà che quelle operazioni siano le uniche consentite su quella struttura. Ovviamente le operazioni saranno mutuamente esclusive: un solo thread alla volta potrà essere attivo nel monitor. Sono forniti da diverse librerie concorrenti: in C sono dati dai mutex+variabili condizioni nella libreria pthread, in Java dai synchronized methods. Stanno su un livello di astrazione più alto rispetto ai semafori.
Una possibile dichiarazione in pseudo codice può essere:

```
monitor tipo_risorsa {
      <dichiarazioni variabili locali>;
      {<inizializzazione delle variabili locali>};

      public void op1 ( ) {
            <corpo della operazione op1 >;
      }-
      ---------
      public void opn ( ) {
            <corpo della operazione opn>;
      }
}
```

Le variabili comuni mantengono il loro valore tra un'esecuzione e l'altra delle operazioni del monitor: sono variabili permanenti. Le operazioni non pubbliche non potranno essere invocate dall'esterno, ma solo da altre operazioni del monitor. Le operazioni pubbliche di un monitor sono le sole a poter essere usate dai processi per accedere alle variabili comuni. Come nel problema della sezione critica, non dovranno garantire solo la mutua esclusione ma anche l'attesa limitata: servirà quindi qualche politica di gestione dell'ordine delle richieste, solitamente una coda: bisogna serializzare l'accesso alla risorsa condivisa. Ma è sufficiente? Potrebbe ad esempio

succedere che un thread riesca ad entrare nel monitor ma non a completare l'operazione: il monitor infatti non consente neanche di vedere le variabili condivise senza entrarci, quindi un thread che vuole svolgere un'operazione solo se una qualche condizione sulle variabili condivise è verificata dovrà entrare nel monitor per saperlo. Se la condizione non è verificata cosa succede? Due possibilità sono uscire e rientrare o mettersi in attesa della condizione conservando una priorità maggiore dei thread che sono ancora fuori. La seconda strategia è implementata dalle **variabili condizione**: un'altra coda che contiene non i tentativi di eseguire un'operazione ma i tentativi di verificare una qualche condizione.
Su una variabile condizione cond le procedure del monitor potranno agire tramite le operazioni **`wait(cond)`** e **`signal(cond)`.**
Wait viene chiamata dal thread che vuole attendere che una certa condizione diventi vera, signal viene chiamata dal thread che modifica questa condizione, rendendola vera, per segnalare ai thread in attesa che possono risvegliarsi e proseguire nell'esecuzione.
Per garantire l'attesa limitata, un thread che sta usando il monitor e vuole sospendersi dovrà obbligatoriamente rilasciare il monitor prima di farlo, altrimenti gli altri thread in coda non riusciranno mai ad entrare.
Per la signal, esistono diverse strategie per risvegliare i thread in attesa: sia P il thread che riceve la segnalazione e Q il thread che la invia, con la strategia **signal_and_wait** P riprende immediatamente l'esecuzione e Q viene sospeso: questo per evitare che Q, proseguendo nella sua esecuzione, possa nuovamente modificare la condizione rendendola non più vera. Una variante di questa strategia è la **signal_and_urgent_wait,** in cui Q viene sospeso ma mantiene una priorità più alta rispetto agli altri thread: viene messo in una coda interna al monitor (urgent queue) in modo che quando P ha terminato l'esecuzione il controllo venga trasferito immediatamente a Q, senza liberare il monitor.
Con la strategia **signal_and_continue** invece Q prosegue normalmente nella sua esecuzione, mantenendo il controllo del monitor, e P viene spostato dalla coda della variabile condizione alla normale coda di ingresso per poter entrare nel monitor. In questo modo però nulla mi garantisce che tra la signal ed il rientro di P nel monitor la condizione non sia di nuovo stata cambiata: quindi, se sto usando una signal implementata con la strategia signal_and_continue, nel thread P dovrò chiamare la wait in questo modo:

```
while(!<condizione da controllare>)
      wait(cond);
```

così appena risvegliato il thread controllerà di nuovo la condizione: se è vera proseguirà nella sua esecuzione, se è falsa entrerà nel ciclo while e chiamerà nuovamente la wait per rimettersi in attesa.
Una variante della signal_and_continue è la **signal_all**, che risveglia tutti i thread in attesa di una certa condizione e li mette nella normale coda di ingresso, da dove poi entreranno nel monitor uno per volta.

4. Reti di Petri

Le reti di Petri sono un modello computazionale (non Turing completo) basato su grafi, usato per modellare sistemi concorrenti. Abbiamo due tipi di nodi: nodi place, che rappresentano stati, e nodi transition, che rappresentano transizioni tra stati e sincornizzazione. È un

grafo bipartito: abbiamo due classi di nodi e gli archi vanno solo da una classe all'altra: possiamo avere archi da place a transition e da transition a place, ma non da place a place o da transition a transition. Lo **stato** di una rete è rappresentato mettendo dei token dentro i place.
In un modello con N processi non ho bisogno di avere N nodi place, me ne basterà uno, e dentro ci metterò N token.
Una transizione t è abilitata in un marking M se per ogni arco entrante in t da un place p esiste un token distinto dagli altri in M: allora la transizione può essere eseguita (fired) per produrre un nuovo token. L'operazione di firing di una transizione è atomica.
Ma in che ordine vengono sparate le transizioni? Ci sono più ordini possibili, l'esecuzione di una rete di Petri è non deterministica. Sembra strano, ma nella concorrenza il non determinismo è assolutamente normale. Possono essere abilitate diverse transizioni simultaneamente, anche se ne verrà eseguita una sola per volta (sono atomiche).
Esempio dei cinque filosofi: ci sono cinque filosofi orientali attorno ad una tavola rotonda, ed ognuno ha davanti la sua ciotola di riso. Ci sono solo cinque bacchette però, per cui ogni filosofo ha una bacchetta sulla destra e una sulla sinistra. Se tutti i filosofi si comportano allo stesso modo, ad esempio prendendo prima la bacchetta sulla destra e poi quella sulla sinistra abbiamo un deadlock: tutti rimangono in attesa della bacchetta sinistra, che non arriverà mai perché la mia bacchetta sinistra è la bacchetta destra dell'altro. Lo stallo deriva dal fatto che i filosofi prendono prima una bacchetta e poi l'altra, e lo fanno tutti nello stesso ordine.

Ci sono due soluzioni: prendere contemporaneamente le due bacchette solo se ci sono tutte e due, ma potrebbe essere
molto complicato implementare un sistema in cui dobbiamo eseguire due azioni contemporaneamente. Una soluzione più semplice da implementare è diversificare il comportamento delle varie parti: se almeno un filosofo è mancino riuscirà a prendere tutte e due le bacchette.

Il problema può essere modellato con reti di Petri (vedi slides).
Definizione formale:

Una rete di Petri è una quadrupla (P,T,I,O) tale che:

- P è un insieme finito di places
- T è un insieme finito di tansitons
- I: P x T → N è una funzione che determina la molteplicità di ogni arco entrante in una transizione
- O: TxP →N è una funzione che determina la molteplicità di ogni arco uscente da una transizione

L'insieme degli stati che una rete di Petri può assumere determina un grafo di stati chiamato reachablity graph, a partire dalla marcatura iniziale. Questo grafo potrebbe avere cammini infiniti, se la rete può auentare all'infinito il numero di token; altrimenti, se la rete ha un numero finito di marcature possibili (è bounded) allora anche il grafo di raggiungibilità sarà finito.

In caso di reti non bounded, si possono usare tecniche di approssimazione per gestire cammini infiniti, come ad esempio marcare con ∞ una componente che cresce rispetto alle marcature precedenti.

5. Programmazione concorrente in Java

La libreria concorrente di Java è stata una delle prime librerie di concorrenza ad alto livello; uno dei primi tentativi di combinare object orientation e concorrenza. Vengono usate da Android e da svariate librerie a diversi livelli di astrazione per la programmazione distribuita.
Java supporta i thread in maniera indipendente dalla piattaforma sottostante, attraverso la classe Thread. Come in Linux, l'ordine di esecuzione dei thread non è determinato a priori, e abbiamo a disposizione costrutti di sincronizzazione specifici del Java Memory Model.
Per creare un nuovo thread il main userà la new per creare una nuova istanza della classe Thread. La corrispondenza tra thread Java e thread a livello kernel è gestita dalla JVM, che dispone di un thread pool. La classe Thread implementa la classe Runnable, che offre un metodo run(). Abbiamo a disposizione un costruttore che prende un altro oggetto di tipo Runnable
Un oggetto di tipo Thread esegue un task in maniera asincrona. Il metodo run() contiene il codice che sarà eseguito dal thread, e dovremo ridefinirlo nelle sottoclassi di Thread che vogliamo eseguire.
Un thread viene fatto partire col metodo start, che lo avvia in maniera asincrona restituendo subito il controllo al chiamante. Invocare start due volte genera eccezione.
Un oggetto che implementa l'interfaccia Runnable rappresenta un task da eseguire, concorrentemente o no (potrebbe benissimo essere eseguito sequenzialmente invocandone il metodo run); un oggetto Thread rappresenta un thread che esegue un task in maniera asincrona.
Per controllare lo stato di un thread o interromperlo esistono metodi predefiniti ma deprecati (interrupt, isAlive): è meglio usare soluzioni artigianali basate sul controllo di flag booleani, che ogni tanto durante l'esecuzione del thread verranno controllati. La variabile booleana usata come flag dovrà essere dichiarata volatile. Semantica di volatile in Java: le modifiche ad una variabile volatile sono immediatamente visibili a tutti i thread, viene garantita la write atomicity. Non sarà messa in cache: tutte le modifiche ad una variabile volatile avvengono in memoria principale.

Sincronizzazione
L'ordine di esecuzione dei thread è gestito da uno scheduler interno alla JVM, che usa una strategia basata su code multiple di priorità con round robin su ogni coda.
I monitor di Java sono basati sulla keyword **synchronized**, che va anteposta alla dichiarazione di un metodo ed assicura la mutua esclusione nel corpo del metodo: se un thread sta eseguendo un metodo synchronized su un oggetto, altri thread non possono eseguire altri metodi synchronized su quello stesso oggetto.
Quando un thread T invoca un metodo synchronized su un oggetto, ma un altro thread sta usando un altro (o anche lo stesso) metodo synchronized su quell'oggetto, T si blocca finché non riesce ad ottenere il lock.
La semantica è la signal and continue, la stessa dei pthread: //
Oltre ai metodi, posso dichiarare synchronized anche un blocco di istruzioni.
Per i metodi, la keyword synchronized non fa parte della segnatura, è più simile ad un'annotazione: non viene ereditata automaticamente dalle sottoclassi

//manca roba

5.1 High level concurrency: il package java.util.concurrent

L'interfaccia Executor permette di separare un task (un oggetto Runnable) dal modo in cui verrà associato ad un thread (execution policy) L'interfaccia offre un metodo execute con cui mando in

esecuzione l'oggetto Runnable, e nella libreria java.util possiamo trovare diverse politiche predefinite, che istanziamo attraverso factory methods:

- Executors.newSingleThreadExecutor(): un singolo thread background
- Executors.newFixedThreadPool(int): crea un thread pool di dimensione predefinita
- Executors.newCachedThreadPool(): crea un thread pool gestito automaticamente

Oltre ad execute, l'interfaccia Executor offre anche un metodo shutdown() che impedisce all'executor di accettare nuovi task (senza però interrompere quelli già in esecuzione).

5.1.1 ThreadPoolExecutor

Oltre a quelli restituiti dai factory methods, possiamo definire degli executor che usano thread pool personalizzati o usare quelli del package java.util.concurrent, come ad esempio la classe **ThreadPoolExecutor**, il cui costruttore richiede i seguenti parametri:

- `int corePoolSize`
- `int maximumPoolSize`
- `long keepAliveTime`
- `TimeUnit unit`
- `BlockingQueue<Runnable> workqueue`

Il parametro corePoolSize specifica la dimensione minima del pool, maximumPoolSize quella massima. Usando il metodo prestartAllCoreThreads possiamo allocare corePoolSize threads al momento della creazione del pool, e i thread creati rimarranno in attesa di un task da eseguire. Possono anche essere creati on demand, ogni volta che viene sottomesso un nuovo task.
Se tutti i core thread sono in esecuzione ed arriva un nuovo task, questo verrà nella coda workingqueue ricevuta come parametro, e solo quando la coda sarà piena verrà creato un nuovo thread, fino a raggiungere il numero maximumPoolSize.
Riassumendo, ogni volta che 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
- Se la coda è piena e ci sono in esecuzione meno thread di maximumPoolSize viene creato un nuovo thread
- Se la coda è piena e ci sono già maximumPoolSize thread attivi, T viene respinto e viene sollevata un'eccezione.

Quando invece un thread T termina l'esecuzione di un task, e nel pool ci sono altri k threads:

- Se k <= corePoolSize T si mette in attesa di un nuovo task da eseguire
- Se k > corePoolSize, T aspetta il timeout definito da keepAliveTime quindi, se non riceve nessun nuovo task, termina. Il parametro unit determina l'unità di misura rappresentata da keepAliveTime

Ci sono diversi tipi di coda che posso usare in un ThreadPoolExecutor, e il tipo di coda scelto influsice sullo scheduling: la SynchronousQueue è un buffer di capacità zero: un nuovo task in arriva viene eseguito immediatamente se c'è un thread libero o se è possibile crearne uno (numero corrente di thread < maxPoolSize). Una ArrayBlockingQueue è una coda di dimensione limitata, stabilita dal programmatore, che può anche essere maggiore di corePoolSize.

Una LinkedBlockingQueue infine è una coda di dimensione illimitata: è sempre possibile accodare un nuovo task nel caso in cui tutti i thread siano attivi nell'esecuzione di altri task. In questo caso la coda non sarà mai piena, quindi il numero di thread attivi sarà sempre <= di corePoolSize.

5.1.1.1 Terminazione

Per attendere la terminazione abbiamo i metodi booleani isShutdown (ci dice se è stato fatto lo shutdown dell'executor), isTerminated (ci dice se tutti i thread dell'executor hanno terminato la loro esecuzione) e awaitTermination(long timeout, TimeUnit unt) che attende un timeout dopodiché termina i thread. Il metodo shutdown implementa la **graceful termination**: non interrompe nessun thread durante la sua esecuzione, ed anche i task ancora in coda ma non ancora lanciati verranno eseguiti. Se voglio terminare tutto immediatamente c'è il metodo List<Runnable> shutdownNow(), che oltre a bloccare l'accettazione di nuovi task ci restituisce la lista dei task in coda e tenta di terminare a forza quelli in esecuzione attraverso il metodo interrupt().

Blocking Queue

La Blocking Queue è una struttura dati concorrente con operazioni garantite thread-safe. Le operazioni di inserimento e cancellazione sono disponibili in tre versioni: bloccanti (se eseguite in un momento in cui non è possibile eseguirle, bloccano il thread finché la loro esecuzione non diventa possibile), non bloccanti (lanciano eccezione se chiamate in un momento in cui la loro esecuzione non è possibile) e speciali (restituiscono un valore speciale se chiamate in un momento in cui la loro esecuzione non è possibile). Poiché devono garantire l'accesso thread safe ai singoli elementi, saranno implementate attraverso semafori e monitor.

5.1.2 Funzioni asincrone

Un uso molto frequente dei thread è quello di procedure asincrone: il chiamante crea il thread che eseguirà il corpo della procedura e terminerà. A differenza di una normale chiamata a funzione, il controllo viene restituito al chiamante subito dopo la creazione del thread, che procederà in parallelo con la sua esecuzione. In caso di funzioni void non ci sono problemi, ma se la funzione restituisce un risultato come facciamo ad ottenerlo? Sappiamo che in Java i thread sono oggetti Runnable, quindi mi basterà definirvi un getter per il risultato, ma come facciamo a sapere quando è stato calcolato? Per questo abbiamo l'interfaccia **Callable**<T>, dove T è il tipo del risultato, che offre un metodo call in cui andremo a scrivere il corpo della funzione da eseguire. Così come i Runnable, i Callable possono essere messi in un ExecutorService per l'esecuzione.
I **Future** invece sono oggetti che rappresentano il risultato dell'esecuzione di una funzione asincrona: hanno un metodo bloccante get() che restituisce il risultato quando è pronto, ed un overload get(long timeout, TimeUnit unit) che attende il risultato per un tempo specificato, dopodiché solleva una TimeoutException. Quando mettiamo un Callable<T> in un ExecutorService attraverso il suo (di ExecutorService) metodo submit, il valore ritornato è un Future<T>.
Se invece di usare un thread pool maneggiamo direttamente i thread, dovremo creare un FutureTask: il costruttore di questa classe prende come parametro un oggetto Callable.

5.2 Java Memory Model

Il Java Memory Model è la specifica dei costrutti Java per rendere l'implementazione di una JVM robusta anche in caso di thread multipli. I problemi di concorrenza di Java sono dovuti al fatto che gli oggetti non sono tipi primitivi, quindi la maggior parte delle operazioni su di essi non possono essere atomiche. Il memory model specificare anche la semantica delle keyword legate alla concorrenza come synchronized e volatile (la semantica di volatile in C non è per niente chiara, dipende dall'implementazione). Per capire i problemi di definizione di una semantica robusta è utile pensare al design pattern Singleton: una classe di cui si può avere una sola istanza. Una semplice implementazione di questo pattern è una classe con una variabile statica *instance* ed un metodo getter che:

- se instance == null, crea un oggetto e lo restituisce
- altrimenti, si limita a restituire instance

Questa implementazione in ambiente multithreaded causa race conditions: due thread potrebbero eseguire il test instance == null contemporaneamente, lo trovano null e quindi creano non una ma due istanze del singleton. In Java possiamo evitare la race condition dichiarando come synchronized il metodo getInstance: in questo modo un solo thread alla volta può eseguire il metodo. Questa è un'implementazione corretta ma inefficiente: l'accesso al getter è serializzato, ci sono dei lock da prendere e quindi delle code di attesa. Alcune soluzioni proposte spostano il lock più all'interno: non su tutto il metodo ma solo sul codice che crea l'oggetto (all'interno dell'if). In questo modo però ho di nuovo la race condition, potrei comunque creare due oggetti, perché la corsa critica avviene nel test dell'if. Un'altra soluzione è il double checked locking:

```
if(instance == null) {
    synchronized(Singleton.class) {
         if (instance == null)
               instance = new Singleton();
      }
}
```

Ma anche questa soluzione non va bene, perché la new non è un'operazione atomica: quando deve creare un oggetto, prima alloca la memoria richiesta e ne assegna l'indirizzo al reference, poi invoca il costruttore dell'oggetto. Se il thread viene deschedulato dopo aver assegnato il reference, ma prima di aver invocato il costruttore, un altro thread troverà instance diverso da null anche se l'oggetto non è stato ancora creato: ma getInstance non se ne può accorgere, restituisce il reference e qualcuno potrebbe usare l'oggetto che non è ancora stato correttamente inizializzato.
La soluzione definitiva è inizializzare instance direttamente nel codice della classe : **`private static Singleton instance = new Singleton();`** ed eliminare ogni controllo in getInstance che dovrà limitarsi a fare **`return instance`**. La semantica di static infatti mi assicura che l'istanza venga creata dal classloader, e che quindi sia già pronta e visibile a tutti i thread la prima volta che viene chiamato getInstance();

5.3 Interfacce grafiche e concorrenza

Swing è una libreria di componenti grafici e funzionalità per lo sviluppo di interfacce grafiche. E' una libreria platform independent, estendibile e personalizzabile, basata sul pattern MVC (Model/View/Controller).
E' un'estenstione dell'Abstract Window Toolkit (AWT), la vecchia libreria grafica di Java.
Ogni applicazione che utilizza Swing ha almeno un top level container, che sarà un JFrame, un JDialog o una JApplet, all'interno del quale verranno creati gli altri componenti grafici.
L'interfaccia è basata su una gestione ad eventi: ogni azione dell'utente scatena un evento, che sarà ricevuto da un listener che agirà di conseguenza (design pattern observer).
La single thread rule afferma che un solo thread può modificare l'interfaccia grafica: questo però non vuol dire che l'applicazione debba avere un solo thread, l'interfaccia è gestita da un thread separato rispetto al main: nel momento in cui invochiamo frame.setVisible(true) attiviamo il thread che gestisce l'interfaccia. La single thread rule è dovuta al fatto che in decenni di tentativi non si è mai riuscito a trovare un modo efficiente e thread-safe di far gestire le interfacce grafiche a più thread contemporaneamente.
Caratteristica fondamentale di ogni interfaccia grafica è che deve essere sempre reattiva: in particolare su Android, se l'interfaccia non risponde per un certo lasso di tempo il sistema termina l'applicazione.
Un interfaccia Swing è composta da tre tipi di thread:

- i thread iniziali, come il main, che avviano l'interfaccia
- l'**Event Dispatch thread (EDT)** che gestisce l'interazione con l'utente: è questo il thread che esegue il codice dei listener e di risposta agli eventi
- i **worker thread** (o background thread), thread dedicati all'esecuzione di operazioni lunghe o bloccanti, che comprometterebbero la reattività dell'interfaccia se fossero eseguiti nell'event dispatch thread.

L'EDT viene avviato automaticamente dalla JVM alla prima chiamata di un metodo paint() o repaint(): gestisce una coda di eventi e resta in attesa che nuovi eventi da gestire siano immessi nella coda, in modo simile ad un thread pool. Non è un demone: rimane attivo finché ci sono eventi da gestire o finché c'è almeno una finestra visualizzata. Poiché la JVM rimane attiva finché è in esecuzione almeno un thread non demone, un'applicazione basata su Swing resterà attiva finché è attivo l'EDT (ovviamente a meno di altri thread).
Single Thread Rule: *ogni operazione che consiste nella visualizzazione (invocazione del metodo paint()), modifica o aggiornamento di un componente Swing, o che accede allo stato del componente stesso, deve essere eseguita nell'Event Dispatch Thread.*
Sarà quindi l'EDT ad eseguire metodi come actionPerformed, mouseClicked e in generale tutti i metodi di gestione eventi per i quali è stato definito un listener. Per sapere se una porzione di codice verrà eseguita dall'EDT, abbiamo a disposizione il metodo statico SwingUtilities.isEventDispatchThread().
I componenti Swing non sono thread-safe: se i loro metodi vengono invocati da più thread contemporaneamente possono verificarsi deadlock o errori di consistenza della memoria. Per questo si è deciso che possano venire acceduti solo da un thread alla volta, e lasciando che sia solo l'EDT a farlo siamo sicuri che la regola venga rispettata. Ci sono alcune eccezioni, specificate nella documentazione, come il metodo append(String) di JTextArea, i metodi repaint(), revalidate() e invalidate() di JComponent e le liste di ListenerType.
Se abbiamo detto che l'EDT è l'unico thread a poter agire sui componenti Swing, come fanno gli altri thread ad interagire con l'interfaccia dell'applicazione? Ad esempio potremmo avere un

programma di chat che deve visualizzare un messaggio ricevuto da un socket (su cui avevamo messo in attesa un thread apposito). Per questo, la classe SwingUtilities offre due metodi statici che permettono a qualsiasi thread di inviare dei Runnable all'EDT: ne abbiamo uno asincrono, invokeLater(Runnable target) che ritorna immediatamente senza aspettare che il Runnable venga eseguito, ed uno bloccante, invokeAndWait(Runnable target), che attende che l'EDT esegua il codice richiesto prima di ritornare. I Runnable verranno inseriti in una coda, e sarà l'EDT stesso ad eseguirli invocandone il metodo run(), non crea un nuovo thread per ogni Runnable! Per questo motivo, le operazioni inviate all'EDT per l'esecuzione devono essere sempre brevi e non bloccanti, altrimenti perdiamo la responsività dell'interfaccia.

//manca roba

5.4 Programmazione distribuita

RMI (Remote Method Invocation) è una tecnica di comunicazione ad un livello di astrazione superiore rispetto ai socket, che permette di invocare metodi su oggetti remoti. È simile ad RPC (Remote Procedure Call) solo che opera in un mondo OO.
Uno stream è il punto temrinale di un generico canale di comunicazione FIFO, che può essere collegato ad un file aperto o ad un socket: ha due estremi in due oggetti InputStream ed OutputStream che offriranno metodi per apertura, lettura, scrittura e chiusura. InputStream ed OutputStream operano su byte: se vogliamo leggere o scrivere tipi di dati strutturati dobbiamo usare i filtri: classi derivate da InputStream ed OutputStream che permettono di leggere/scrivere stringhe, caratteri, numeri oggetti ecc.

Java RMI è l'API di Java per la remote method invocation: un'evoluzione dell'RPC che permette ad un client di ottenere un riferimento ad un oggetto remoto, in modo che possa invocarne i metodi. Il client riceve dal server uno stub: un oggetto locale al client che fa da proxy per l'oggetto sul server. Ogni invocazione di metodo sullo stub verrà inviata al server. Sul server le invocazioni verranno ricevute dallo skeleton: un segmento di codice presente sul server che riceve le invocazioni remote e le attua sull'oggetto target.
Perché il sistema funzioni è necessario un servizio di naming, per tenere traccia dei nomi degli oggetti distribuiti tra client e server: ci sarà quindi un registro dei nomi, dove gli oggetti remoti faranno bind per associarsi un nome.

6. Tecniche di programmazione thread-safe

Perché del codice funzioni correttamente in ambiente multithreaded, bisogna garantire che tutte le strutture dati siano thread-safe, dobbiamo cioè fare in modo che nessuna di esse sia oggetto di scritture concorrenti da parte di thread diversi. Per raggiungere l'obiettivo esistono diverse tecniche:

- oggetti immutabili
- classi completamente sincronizzate
- confinamento per thread: un solo thread può accedere alla struttura dati
- confinamento per oggetto/metodo/gruppo:

Quella degli **oggetti immutabili**, nonostante sembri abbastanza stupida, è una tecnica interessante: per ogni metodo che richiederebbe una scrittura nello stato dell'oggetto, invece di modificarne lo stato ne creo una copia, esegue le modifiche sulla copia e la restituisco come risultato del metodo. E' questa la tecnica adottata dai linguaggi funzionali, in cui non abbiamo la nozione di stato, ogni azione è sempre senza effetti collaterali. Questa tecnica è semplice ed efficace oltre che concettualmente elegante, ma ha lo svantaggio di portare alla creazione di moltissimi oggetti. Un design pattern basato sulla condivisione di oggetti immutabili è il **flyweight design pattern**: conviene usarlo quando gli oggetti da rendere immutabili sono molto piccoli, come ad esempio i caratteri: se voglio rappresentare un documento ASCII come lista di caratteri non mi serve creare un nuovo oggetto per ogni carattere: sarà più conveniente creare tutti i caratteri e poi avere il documento come lista di reference. Spesso si definiscono due classi gemelle, di cui una è immutabile e l'altra no: ad esempio String (immutabile) e StringBuilder di Java.

Un'altra tecnica di programmazione thread-safe è quella delle **classi completamente sincronizzate**: è una tecnica di sincronizzazione molto large-grain basata sui monitor astratti. Una classe completamente sincronizzata è una classe che non ha campi pubblici (rispetta l'incapsulamento), tutti i metodi pubblici sono syncrhonized e mantengono la consistenza anche in caso di eccezioni. Il costruttore inizializza l'oggetto in uno stato consistente, e bisogna stare attenti che non sfuggano riferimenti ad un oggetto non ancora completamente inizializzato: nessun thread, al di fuori di quello che sta eseguendo il costruttore, deve essere in grado di ottenere un riferimento all'oggetto in corso di costruzione.
Il principale difetto delle classi completamente sincronizzate è l'efficienza: tutti i metodi vengono eseguiti in mutua esclusione, anche nei casi in cui non sarebbe necessario.
Le **tecniche di confinamento** consistono nella creazione di domini all'interno dei quali viene strutturalmente garantito che al più un thread alla volta può accedere ad un oggetto.

Nel **confinamento per metodo** un oggetto è creato all'interno di un metodo, e quindi non corre il rischio di interferenza da parte di altri thread (il suo scope di visibilità è limitato all'esecuzione del metodo in cui viene creato). Bisogna stare attenti però a non farsi scappare reference all'oggetto confinato, evitando l'aliasing. Il confinamento per metodo è un caso particolare di sequenza di hand off intra-thread: una sequenza di hand-off è una consegna di un oggetto ad un thread con la garanzia che solo lui lo può manipolare.

Nel **confinamento per thread** invece vogliamo che l'oggetto sia visibile solamente all'interno del thread che lo ha creato: un esempio di implementazione di questa tecnica sono i Thread Specific Data della libreria pthread del C. Tecniche di confinamento per thread sono usate anche dall'interfaccia grafica javax.swing.

Nel **confinamento per oggetto** abbiamo un oggetto guest confinato all'interno di un oggetto host, che ne disciplina l'accesso. Monitor e vari tipi di collezioni sincronizzate sono esempi di confinamento per oggetto.

Il **confinamento per gruppo** infine consiste nel limitare l'accesso ad un oggetto ad un gruppo ristretto di thread: i thread del gruppo potranno accedere all'oggetto solamente in mutua esclusione. Esempio di confinamento per gruppo è il **token ring**: un token viene fatto girare per un anello di thread, e solo il thread che in quel momento possiede il token potrà accedere alla risorsa condivisa.

7. Sistemi distribuiti

Un algoritmo distribuito non è più un singolo programma, ma vari programmi eseguiti su diversi nodi. Negli esempi che vedremo solitamente ci saranno nodi che possono assumere diversi ruoli, e ad ogni ruolo corrisponde del codice diverso. I programmi eseguiti sui singoli nodi saranno multithreaded.
Un sistema distribuito si dice **sincrono** se esistono e sono noti:

- i limiti inferiore e superiore al tempo di esecuzione di ogni passo di elaborazione
- il limite superiore del tempo di consegna di un messaggio
- il limite superiore alla differenza tra il clock locale e il tempo reale

Sono tutte assunzioni molto forti, e garantirle è molto difficile: per questo solitamente i sistemi distribuiti sono asincroni. In un sistema asincrono invece non ci sono limiti alla velocità di esecuzione o al ritardo dei messaggi: anche Internet è un sistema distribuito asincrono.

Scambio di messaggi
Nel modello a scambio di messaggi la comunicazione è point-to-point attraverso le primitive send(tipo, ricevente, $dato_1$,...,$dato_n$) e receive(tipo, $dato_1$,...,$dato_n$). Il tipo viene usato per identificare il messaggio, in modo da poter scartare quelli che non ci interessano.

7.1 Problema della sincronizzazione di nodi: clock logici

Ogni nodo usa dei timestamp per etichettare le sue azioni, un numero d'ordine che rappresenta la sequenza delle operazioni: in un sistema distribuito, voglio che questi timestamp siano sincronizzati su tutti i nodi della rete. Un sistema di timestamping basato sul clock locale dei nodi non andrebbe bene, perché non sarebbe sincronizzato con gli altri: per questo si usano algoritmi basati su **clock logici**, che possono essere scalari, vettoriali, di versioning o matriciali.

7.1.1 Clock scalari di Lamport

Un algoritmo proposto da Lamport si basa su **clock logici scalari**. Assumiamo di osservare gli eventi che si svolgono nel sistema: agli eventi che si svolgono su un singolo nodo sappiamo dare un ordinamento totale, ma ordinare gli eventi di send e receive è un problema. L'ordinamento totale del sistema sarà un'estensione conservativa dell'ordinamento dei singoli nodi: l'ordine totale non cambierà l'ordine delle operazioni eseguite da un singolo nodo.
Va definita una relazione **happened-before** tra gli eventi: a -> b se a è avvenuto prima di b. Non ha a che fare con i veri tempi di esecuzione, è solo un ordinamento logico: a -> b se:

- a e b sono due eventi nello stesso processo e a avviene prima di b
- a e b avvengono in processi diversi, dove a corrisponde all'invio di un messaggio m e b alla ricezione di m.

E' una relazione transitiva: se a -> b e b -> c allora a -> c. Due eventi a e b sono concorrenti (a || b) se non possono essere messi in relazione happened-before l'uno con l'altro in nessuna direzione: se ¬(a -> b) e ¬ (b -> a) allora a || b.
Per ogni evento non concorrente e vogliamo avere una misura globale del tempo C(e), tale che se a -> b allora C(a) < C(b).
Il nostro algoritmo dovrà garantire che l'ordine dei timestamp corrisponda all'ordine delle relazioni happened-before.
I timestamp hanno senso solo se esiste un'istante 0 comune a tutti i nodi. E' un ordinamento logico, che non sincronizza le velocità di esecuzione sui singoli nodi.

Nell'algoritmo di Lamport ogni nodo ha una variabile locale che rappresenta il suo clock logico corrente, che verrà incluso in ogni messaggio inviato. Tra due eventi il clock deve essere incrementato di almeno un tick. Quando un messaggio viene ricevuto, il clock del ricevente viene aggiornato a tempo di invio del messaggio + 1 tick se il clock locale è minore del timestamp del messaggio, altrimenti viene semplicemente incrementato di un tick.
Il problema di questo algoritmo è che a -> b implica C(a) < C(b), ma C(a) < C(b) non necessariamente implica che a -> b. E' un problema perché non mi permette di capire se due eventi su nodi diversi sono concorrenti. Per questo sono stati introdotti da Mattern e Fridge i clock vettoriali: ogni nodo non conserva solo il clock locale ma un vettore di clock, che rappresenta uno snapshot del sistema: il timestamp degli altri nodi così com'era l'ultima volta che ne ho avuto notizia. Strategie simili sono adottate da molti algoritmi distribuiti: invio la mia conoscenza della rete agli altri nodi, in modo che ogni nodo abbia la conoscenza più vasta possibile, basata non solo su quello che può sapere direttamente ma anche su quello che sanno gli altri.

7.1.2 Clock vettoriali di Mattern e Fridge

In un sistema di N processi il clock logico vettoriale sarà un vettore V di dimensione N dove la posizione V[i] corrisponde al timestamp del processo i-esimo. Ogni processo manterrà una copia locale del vettore, in cui metterà tutti i valori di clock ricevuti dagli altri processi.
L'ordinamento tra due clock vettoriali V e W è definito in questo modo:

- V <= W sse V[i] <= W[i] per ogni i da 1 ad N
- V < W sse V <= W ed esiste un i tale che V[i] < W[i]
- V = W sse V <= W e W <= V

In questo modo la misura C(a) di un evento a è il clock vettoriale del processo dove avviene a, quindi C(a) < C(b) sse a -> b.
Ad ogni evento interno, il processo i incrementerà di 1 il valore in posizione V[i], che rappresenta il suo clock. Ad ogni ricezione di messaggi da un altro processo, confronterà i valori del suo vettore di clock con quello allegato al messaggio e aggiornerà ogni valore al massimo tra la copia locale e quella ricevuta.
I clock logici vettoriali vengono usati nei sistemi distribuiti con optimistic replication: le operazioni concorrenti sono permesse e i conflitti si risolvono solo quando necessario. Un altro possibile utilizzo dei clock vettoriali è come **versioning clock**: i nodi mantengono e si scambiano i vector clock come detto prima: se al momento del confronto i vector clock di due nodi coincidono allora i nodi hanno gli stessi dati, se un vector clock è maggiore di un altro allora un nodo ha eseguito almeno tutte le operazioni dell'altro e quindi quello rimasto indietro può copiare dati e vector clock del nodo più avanti, per arrivare ad una situazione consistente. Se invece i clock di due nodi non sono confrontabili vuol dire che si è verificato un conflitto.

L'uso di clock vettoriali aumenta la conoscenza che i singoli processi hanno del sistema rispetto all'uso di clock scalari. Un'ulteriore espansione di conoscenza si ha con i **clock matriciali** (**matrix clock**), in cui i processi ad ogni messaggio inviato si scambiano una matrice contenente tutti i vettori di tutti i processi: l'elemento [i, j] della matrice rappresenta ciò che il processo Pi sa a proposito del processo Pj. Se Pi nel suo matrix clock ha tutta la colonna relativa a se stesso maggiore di un certo k, allora piò concludere che tutti gli altri processi sanno che Pi è arrivato almeno all'evento k. Questa gestione dei clock è molto utile in situazioni di comunicazione incerta,

con rischio di perdita dei messaggi, poiché permette al mittente di verificare se un messaggio è stato ricevuto da tutti i processi.

7.2 Mutua esclusione distribuita

Obbiettivo della mutua esclusione in un sistema distribuito è far sì che un insieme di processi che condividono una o più risorse possano accedervi senza interferenze e senza portare il sistema a stati inconsistenti. Il problema si complica quando la risorsa non è gestita da un singolo server centrale, ma sono i processi stessi a dover coordinare l'accesso alle risorse condivise.

Su una rete con configurazione ad anello in cui ogni nodo parla solo con i vicini, la mutua esclusione può essere garantita con un algoritmo di token passing: un token viene passato da un processo all'altra e il processo che detiene il token può entrare in sezione critica. Questo protocollo garantisce la mutua esclusione (se il token è unico) ma non permette di decidere l'ordine in cui le richieste verranno esaudite.

7.2.1 Algoritmo di Ricart-Agrawala

Per un sistema con canali FIFO tra ogni coppia di processi, in cui sappiamo che potranno esserci ritardi nella comunicazione ma tutti i messaggi prima o poi verranno ricevuti nel giusto ordine (rete fully connected) è possibile usare una versione dell'algoritmo di Lamport adattata per funzionare in un contesto completamente distribuito, senza bisogno di un nodo che coordini le richieste e assegna i biglietti ai processi: l'algoritmo di **Ricart-Agrawala** in versione distribuita. L'idea su cui si basa è di utilizzare dei sottoprotocolli sia per calcolare il prossimo numero da utilizzare come biglietto sia per testare la condizione di ingresso in sezione critica. Ogni nodo quindi deve svolgere due funzioni contemporaneamente: tentare di entrare in sezione critica e rispondere alle richieste di calcolare o confrontare i ticket che arrivano dagli altri nodi. La fase più delicata è il testing della condizione: se il confronto tra il biglietto B inviato dal processo P che vuole entrare in sezione critica e il biglietto T del processo P' fallisce, allora P' non deve dare subito l'ok a P ma deve ricordarsi la sua richiesta e rispondere appena possibile (non appena esce dalla sezione critica).

Pseudocodice Ricart-Agrawala:

```
int myNum <- 0
set of node IDs deferred <- empty set

main:

//non critical section
myNum <- chooseNumber
for all other nodes N
      send(request, N, myID, myNum);
await reply from all other nodes
//critical section
for all nodes N in deferred
      remove N from deferred
      send(reply, N, myID);
```

receive:

```
receive(request, source, reqNum)

if reqNum < myNum
      send(reply, source, myID)
else
      add source to deferred;
```

Un processo che vuole accedere alla sezione critica manda una richiesta a tutti gli altri processi, si pone in attesa della risposta e una volta ricevuto risposta da tutti entra in sezione critica. Una volta uscito dalla sezione critica invia una risposta a tutti i processi nella sua coda deferred. Quando invece un processo riceve una richiesta da un altro:

- Se è già in sezione critica, non risponde e mette la richiesta nella coda deferred
- Se non è ancora in sezione critica, ma vuole accedervi, confronta il valore dei ticket: se l'altro processo ha il ticket minore gli invia l'OK per l'accesso in sezione critica, altrimenti non risponde e mette la richiesta in coda

I problemi che possono verificarsi con questo algoritmo sono:

- Nodi con ticket uguali
- Scelta del ticket non monotona rispetto a tutti i ticket visti tra i singoli nodi
- Nodi quiescenti
- Race contidions nei singoli nodi

Se due nodi hanno ticket uguali l'algoritmo va in deadlock.

Se la scelta di ticket non è monotona rispetto a tutti i ticket visti dai singoli nodi potrebbe verificarsi una violazione della mutua esclusione (vedi esempio su slides che si capisce meglio, parte Choosing ticket numbers).

Anche i nodi quiescenti sono un problema: se un nodo si blocca in sezione non critica non potrà più inviare le risposte agli altri nodi, che resteranno in attesa per sempre (starvation).

Le race conditions sui singoli nodi infine possono verificarsi perché ogni nodo ha (almeno) due thread: il thread principale (che vuole entrare in sezione critica) e il thread per la ricezione dei messaggi degli altri nodi, che condividono:

- La variabile booleana che rappresenta la volontà di entrare in sezione critica (requestCS)
- Il valore corrente del ticket (myNum)
- Il valore del ticket più alto tra il proprio e tutti quelli ricevuti dagli altri nodi: serve per poter generare un nuovo ticket che sia sempre maggiore di tutti gli altri (highestNum)
- La coda delle richeste in attesa (deferred)

Dovrò quindi svolgere atomicamente le operazioni di requestCS = true/scelta nuovo numero ed il confronto tra il mio ticket e quello del richiedente quando ricevo una richiesta.

7.2.2 Algoritmo di Ricart-Agrawala con token passing

I difetti dell'algoritmo di Ricart-Agrawala sono la necessità il broadcasting delle richieste, inefficiente in una rete con molti nodi, e che richiede sempre l'invio di messaggi a tutti gli altri nodi anche quando non c'è contesa per entrare in sezione critica. Un possibile miglioramento è un algoritmo basato su token passing, in cui il permesso per entrare in sezione critica è associato al possesso di un token, unico in tutta la rete. Il processo che detiene il token può entrare in sezione critica un numero arbitrario di volte, se nessun altro chiede l'accesso.

Una soluzione deadlock-free e starvation-free è quella di usare due array: un'array granted in cui vengono memorizzati i ticket di ogni nodo al momento dell'ultimo ingresso in sezione critica: questo array verrà passato come token. Ogni nodo memorizzerà in un array locale requested i ticket ricevuti dagli altri nodi, ed il token verrà passato solo quando si verifica una contesa.

//pseudocodice nelle slides

7.3 Algoritmo di Neilsen-Mizuno (spanning tree)

Questo algoritmo è sempre basato su token passing, con la differenza che la coda dei processi non è rappresentata dentro il token, ma è rappresentata dai link che collegano i vari nodi in un insieme di alberi di copertura[2] (spanning tree).

I nodi hanno un puntatore parent al padre ed un puntatore deferred. Il nodo che ha il token entra in sezione critica, i nodi che non hanno il token ma vogliono entrare in sezione critica cercano di diventare radice di uno spanning tree: inviano una richiesta al padre e resettano il puntatore parent. Quando un nodo radice riceve una nuova richiesta memorizza un puntatore al richiedente in deferred: in questo modo si costruisce una coda distribuita.

7.4 Algoritmo di Maekawa (quorum based)

In u algoritmo quorum based, invece di inviare le richieste di ingresso in sezione critica ad ogni processo, la si invia a sottoinsiemi di processi (request sets), che dovranno avere intersezione non vuota (deve esserci almeno un processo comune per ogni coppia di request set).

Avremo un insieme di processi P, dei sottoinsiemi di P (i quorum) e degli insiemi di quorum (i request sets) a intersezione non vuota.

[2] Un albero di copertura di un grafo, connesso e con archi non orientati, è un albero che contiene tutti i vertici del grafo e contiene solo gli archi necessari per connettere tra loro tutti i vertici con uno e un solo cammino.