**Domande PCAD**

**Concorrenza e SO**

**1- Cosa prevede il modello di esecuzione sequenziale? E quello concorrente?**

Seq: programmi formati da sequenze di istruzioni (totalmente ordinate) + un’unica memoria (virtuale) per i dati / Conc: più memorie per i dati e nessun ordine in generale per le istruzioni. Rilassiamo dunque il modello di calcolo in modo da prevedere l’esecuzione concorrente di un’insieme di istruzioni (parzialmente ordinate).

**2- Parallelismo reale vs virtuale. Vantaggi della concorrenza?**

Parallelismo reale: multi-processore, sistema distribuito

Parallelismo virtuale: multi-programmazione con scheduling

Vantaggi: sfruttare architetture multi-proc, GUI, giochi, server

**3- Processo vs programma?**

Il modello per la coesistenza di attività contemporanee in un SO prende il nome di modello concorrente ed è basato sul concetto di processo (programma in exec - exec nel tempo)

**4- Processi nei SO? Cos’è il context switch?**

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: lo scheduler gestisce l’avvicendamento dei processi in CPU: decide quale processo deve essere in esecuzione in ogni istante e interviene quando viene richiesta un’operazione di I/O e quando un’operazione di I/O termina, ma anche periodicamente per assicurare il buon funzionamento del sistema.

Context switch: quando la CPU passa ad un altro processo, il sistema deve salvare lo stato del vecchio processo e caricare quello del nuovo processo. Può rappresentare un overhead.

**Processi e Thread**

**5- Processi in Unix? Come si crea un processo? Cos’è la execve?**

Ogni processo UNIX (identificato dal PID) ha uno spazio di indirizzamento separato e quindi non vede le zone di memoria dedicate agli altri. Esso ha: stack, dati (statici, heap) e codice. L’indirizzamento è virtuale (stack nella parte finale, gli altri all’inizio).

I processi si creano con fork (P e F condivideranno il codice e F avrà una copia di dati e PC del padre), che restituisce il PID di F a P, e zero a F. execve sosituisce e dati, codice e stack con quelli del nuovo programma da eseguire nel contesto del processo figlio.

**6- User vs Kernel Mode?**

I processi Unix operano in modo user e kernel: cioè il kernel esegue nel contesto di un processo le operazioni per gestire chiamate di sistema e interrupt. Alla partenza del sistema il codice del kernel viene caricato in memoria principale. Un programma in modo user non può accedere allo spazio di indirizzi del kernel. Quando un processo passa ad eseguire in modo kernel tale vincolo viene rilasciato per eseguire codice kernel nel contesto user.

**7- Cosa sono i livelli di contesto?**

La parte dinamica del contesto di un processo (kernel stack, registri salvati) è organizzata a sua volta come stack con un numero di posizioni che dipende dai livelli di interrupt diversi ammessi nel sistema. Tali passi vengono fatti all’interno dello stesso processo: cambia solo la sua parte di contesto dinamica.

**8- Trap/Interrupt vs Context Switch. 4 scenari di context switch?**

Con esse un processo in modo kernel può svolgere funzioni collegate ad altri processi. La loro gestione si basa su un context switch all’interno del processo: il controllo non passa ad un’altro processo ma è necessario salvare la parte corrente del contesto dinamico del processo all’interno di esso. Gli scenari possibili di context switch sono:

Quando un processo si sospende - 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 ma non è più il processo a più alta priorità.

**9- Quali sono le differenze tra processi e thread? Vantaggi? Svantaggi?**

Processi: rappresentano unità di allocazione risorse: codice eseguibile, dati allocati staticamente (variabili globali) ed esplicitamente (heap), risorse mantenute dal kernel (file, I/O, workind dir), controlli 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,. . .

Thread: (processo leggero) è una unità di esecuzione (PC, registri, stack CPU e di exec) che condivide con i thread fratelli: codice, dati e risorse richieste all’SO. L’esecuzione in parallelo non implica l’esecuzione ripetuta dei sottoprogrammi.

→ Task = unità risorse + i thread che vi accedono

Vantaggi thread: efficienza (creazione, scheduling, throughput, performance)

Svantaggi thread: complessità, < info-hiding, sync, gestione scheduling, data protection

**10- Spiega User-Level Thread vs Kernel-Level vs Ibridi.**

User Level: stack, program counter, e operazioni su thread sono implementati in librerie a livello utente. Vantaggi: efficiente, semplice su sistemi preesisenti, portabile, specifico.

Svantaggi: no auto-scheduling, no prelazione (thread monopolizza CPU), syscall bloccanti bloccano tutti i thread (devo sostituire con routine di libreria: jacketing), accesso seq. al kernel, non sfrutta multi-proc., poco utile per I/O bound

Kernel Level: : il kernel gestisce direttamente i thread. Le operazioni sono ottenute attraverso syscall. 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: < efficienza, necessità di riscrivere vecchie syscall, < portabilità, politica fissata non poi modificabile ( → ES: Unix).

Ibridi: vantaggi di entrambi + flessibilità, MA poca portabilità

**Programmazione concorrente**

**11- Scrivi lo pseudocodice del Merge Sort con COBEGIN e COEND**

Unica accortezza mettere COBEGIN: sort1 || sort2 COEND e solo dopo il merge

**12- Parla del problema del produttore-consumatore.**

Il thread produttore produce informazione che viene consumata da un thread consumatore Soluzione (1) a memoria condivisa: tra i due thread si pone un buffer di comunicazione di dimensione fissata. V. slide per codice

**13- Cos’è un'istruzione atomica? Cos’è una race condition?**

Istruzione atomica: istruzioni che vanno eseguite insieme, pena inconsistenze.

Race Cond: quando 2+ thread entrano in una sezione critica insieme (= risultato dipende dall’ordine di esecuzione).

**14- Parla del problema della sezione critica. Quali 3 criteri bisogna soddisfare?**

è una porzione di codice con dati condivisi in cui può entrare solamente un thread alla volta (dati n thread che competono tra loro per usare tali dati). I criteri sono:

- mutua esclusione: Se il thread Pi sta eseguendo la sua sezione critica, allora nessun altro thread può eseguire la propria sezione critica.

- progresso: Se nessun thread è nella sezione critica ed esiste un thread che desidera entrare nella propria sezione critica, allora l’esecuzione di tale thread non può essere posposta indefinitamente.

- attesa limitata: Se un thread P ha richiesto di entrare nella propria sezione critica, allora il numero di volte che si concede agli altri thread di accedere alla propria sezione critica prima del thread P deve essere limitato superiormente da una costante fissata a priori.

**15- Quali assunzioni sono importanti? Fai un esempio di violazione dei criteri**

Dobbiamo dichiarare a priori quali istruzioni sono atomiche e quali no - Ogni thread deve rimanere nella sezione critica solo un tempo finito - Ogni thread viene eseguito ad una velocità non nulla (quindi un thread non si può fermare da solo) - Non ci sono vincoli sulla velocità relativa dei thread (numero e tipo di CPU).

Una violazione del progresso è un deadlock - del bounded waiting è una starvation

**16- Presenta le soluzioni (al PSC) a flag condiviso e ad alternanza stretta**

La soluzione con flag condiviso non funziona perchè lo scheduler può interrompere il thread subito dopo il ciclo interno e prima dell’istruzione occupato:=1. La soluzione ad alternanza stretta (a turni) non soddisfa il progresso (un thread può decidere di non cedere il turno).

**17- Cos’è il busy waiting? E uno spin-lock?**

è un'attesa attiva di un evento da parte di un thread. è semplice da implementare ma porta a consumi inaccettabili di CPU. In genere, da evitare, ma a volte è preferibile (es. in caso di attese molto brevi). Un thread in busy waiting su una variabile esegue uno spin lock.

**18- Bounded-waiting != Progresso?**

1) Entry nella sez. critica controllata con una variabile condivisa turn. Se Pi e’ nella sezione NON critica ed esegue un loop infinito allora non vale progresso (tutti gli altri thread sono bloccati) . Tuttavia vale bounded waiting. Infatti i thread aspettano al piu’ k − 1 turni.

2) Scegliamo in modo casuale il thread che entra nella sezione critica. Vale progresso, Infatti nessun thread al di fuori della sezione critica puo’ influenzare la scelta di quale thread puo’ entrare. Non vale bounded waiting (nel caso peggiore Pi non entra mai: starvation)

**19- Parla dell’algoritmo di Peterson.**

Algoritmo per 2 thread basato su richiesta e accesso (su spinlock): Per codice v slide

**20- Parla dell’algoritmo di Lamport.**

Algoritmo basato su una coda generata da biglietti di ordine crescente distribuiti ai clienti. 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. Per codice v slide

**Sincronizzazione con semafori**

**21- Cos’è un semaforo? V? P? Invariante?**

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 finchè non riceve un segnale da un altro processo. Dunque è un tipo di dato astratto per il quale sono definite due operazioni atomiche: 1) V chiamata anche up che viene invocata per inviare un segnale, quale il verificarsi di un evento o il rilascio di una risorsa; 2) P, chiamata anche down, che viene invocata per attendere il segnale.

Un semaforo può essere visto come una contatore S condiviso, che assume valori interi e viene inizializzato ad un valore non negativo. L’invariante è np ≤ nv + init

**22- Come risolvo il produttore-consumatore coi semafori?**

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) → Per codice v slide

**23- Come implemento i semafori a livello Kernel? Che conseguenze ci sono?**

L’implementazione della definizione classica dei semafori è basata su busy waiting: procedure down(S): procedure V(S):

while (S≤0) skip endwhile; S:=S+1

S:=S-1

Si può limitare il busy waiting a livello Kernel sospendendo il processo che invoca down e svegliando uno dei processi sospesi al momento della up. Per ogni semaforo il S.O. deve quindi mantenere una struttura dati contenente l’insieme dei processi sospesi, ad esempio con politica FIFO. Si useranno le operazioni sleep e wakeup(pid).

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.

**24- Come implemento i semafori con value negativo?**

In questa implementazione (v. slide) value può avere valori negativi: indica quanti processi sono in attesa su quel semaforo (ad esempio P(S): S - -; if(S.value < 0): add proc; sleep())

**25- Cosa sono i mutex? Kernel vs User level?**

l mutex sono semafori binari con valori: bloccato o non bloccato. Implementare queste funzioni richiede una qualche memoria condivisa. A livello kernel: possono essere mantenute dal kernel e dunque accessibili da tutti con syscall / a livello utente: all’interno dello stesso processo (adatto per i thread) oppure tra processi diversi (shared memory).

**26- Fai un esempio di deadlock con semafori.**

down(S) down(Q)

down(Q) down(S)

… …

up(S) up(Q)

up(Q) up(S)

**27- Cos’è un monitor? E le variabili condition?**

Un monitor è un TDA che fornisce funzionalità di mutua esclusione, aka una collezione di dati privati e procedure per accedervi (un solo processo alla volta). Il programmatore raccoglie quindi i dati condivisi e tutte le sezioni critiche relative in un monitor e questo risolve il problema della mutua esclusione. Essi sono implementati dal compilatore con dei costrutti per mutua esclusione. Per sospendere e riprendere i processi, ci sono le variabili condition con le operazioni wait(c) e signal(c). La signal ha due varianti (nella versione di Hoare chi la esegue si sospende subito) → dopo che p ha lasciato il monitor i risvegliati possono provare ad entrare (scelti dallo scheduler). Eventuali signal non colte vanno perse.

**28- Come risolvo il produttore-consumatore coi monitor?**

Si implementa il monitor (v. slide per codice) e poi si utilizzano semplicemente i metodi. 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.

**Lock-Free Programming**

**29- Cos’è il Lock-Free Programming? Quali sono possibili soluzioni?**

La programmazione concorrente senza lock (lock-free) è una tecnica usata per ridurre l’overhead dovuto all’uso di sincronizzazione tramite lock, semafori e monitor. Possibili soluzioni sono: memory fence, disabilitare le interrupt, istruzioni speciali (ultime 2 per race)

Disabilitare gli interrupt è un modo semplice per garantire la mutua esclusione ma è pericoloso (P può non riabilitarle più), lento e non scala a macchine multiprocessore.

**30- Cos’è m-fence?**

la barriera di memoria è un meccanismo lockfree a basso livello (ad es. mfence per x86) che permette di recuperare la write atomicity inserendo un “flush” di memoria in una specifica zona del codice, assicurandosi che la porzione successiva veda tutti i cambiamenti aggiornati in memoria.

**31- Cosa sono TS e CAS? Come posso usarle?**

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

→ uso: do { TS(lock, old); } until(old == 0); → ok mutua escl. e progr. (ma busy wait)

CAS(x,old,y) := atomico( if old==x then x:=y; ret TRUE else ret FALSE )

→ uso: until(res = CAS(lock, old, 1))

**32- C’è un’idea migliore al busy wait?**

Esso porta a alto consumo di CPU e inversione di priorità (con l’idea migliore, cioè wait/notify, il processo ad alta priorità che tenta di prendere il lock viene messo in attesa e non rischia di “occupare” il tempo di schedule del processo a bassa priorità)

**Memory Model**

**33- Parallelismo vs concorrenza.**

Parallelismo: rendere efficiente l'esecuzione di codice eseguendo in parallelo parti di codice; spesso comportamento deterministico (es. algoritmi paralleli su matrici)

Concorrenza: introduce modularità nel codice, sfrutta quando possibile parallelismo attraverso il multithreading, comportamento non deterministico (diverse esecuzioni possono portare a risultati diversi).

**34- Quali sono gradi di parallelismo delle architetture moderne?**

Multicore: core multipli con diversi livelli di caching (shared e privati) ● Hyperthreading: due o più core logici in un core fisico (tecnologia Intel) ● SIMD: singola istruzione eseguita in parallelo su diverse ALU (nello stesso core) ● ILP: esecuzione in fasi di stream di istruzioni

**35- Cosa sono l’Hardware Out-Of-Order Execution e la Compiler Optimization?**

1) è la possibilità che il processore scambia l’ordine (inizialmente voluto dal programmatore) delle istruzioni. 2) Invece, il compilatore può scambiare l’ordine delle istruzioni perchè, senza pensare l’esistenza di un altro thread che la modifica, potrebbe pensare che una variabile non viene modificata nel mezzo.

**36- Cos’è un Memory Model? Perché ci interessa?**

Specifica formale di come la gestione della memoria appare al programmatore:

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. Ci interessa perchè influenza performance, programmabilità e portabilità di un programma.

**37- Spiega il Single Thread Model, lo Strict Consistency (SC) e il Sequential.**

Single Thread: Accessi in memoria eseguiti uno alla volta ● Read restituisce l'ultima write ● Le operazioni sulla memoria devono seguire l'ordine del programma

StrictC: Ogni operazione in memoria viene vista nell'ordine temporale in cui viene eseguita dai diversi thread (istantaneamente) → utopia

Sequential**:** è 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 sono eseguite nell'ordine del corrispondente programma →

program order: rispetta ordine nel programma (v. es. su slide)

write atomicity: tutte le write devono apparire a tutti i thread nello stesso ordine

**38- Parla dell’algoritmo di Dekker. Come posso violare la mutua esclusione?**

T1 T2 → se NO seq.consist → violo mutua esclusione

Flag1=1 Flag2=1 : ad es. non si propagano le due write del flag:

if(Flag2=0)<crit> if(Flag1=0)<crit> aka entro nelle due critical sections

**39- Come si gestisce la cache nell’SeqC?**

Per processori con cache locali occorre utilizzare un protocollo di consistenza tra memoria e linee di cache. Ad esempio in caso di aggiornamento di una certa locazione di memoria si ha un ciclo di invalidazione o di aggiornamento di tutte le linee associate.

**40- Violazioni su SeqC: read bypassing, no write atomicity & code-optimization**

Potrei violare SC (ad es. in Dekker come detto sopra) se le write venissero messe nel buffer dei thread, avendo read bypassing. Oppure potrei violare write atomicity (ad es. (con cache) se ho if(A==1) then B = 1 nel Thread1; e if(B==1) then tmp = A; nel Thread2 → non posso avere tmp=0 perchè se B=1 (e quindi T2 mette tmp=A) vuol dire che T1 ha visto che A è 1)

La code-optimiz. invece può violare SC per violazione del program order. Ad es:

Thread T1 Thread T2

for(i=0;i<10;i++) while (true)

\*a = i; b = \*a; → Non si può spostare b = \*a fuori dal ciclo in T2

**41- Strong vs Weak memory models**

Nei Weak rilassiamo i requisiti per avere SC: ● Program order (locazioni diverse): si ammettono reorder ● Non si garantisce write atomicity

**Thread in C**

**42- Parla della libreria pthread. Tipo opaco? join?**

Definita in ambito POSIX definisce un insieme di primitive per la programmazione di applicazioni multithreaded realizzate in C. Il thread è l’unità di scheduling, ed è univocamente individuato da un identificatore (intero): pthread\_t tid (tipo opaco: tipo che non espone la sua struttura). Abbiamo diverse funzioni:

int pthread\_create(thread, attr, start\_routine, arg);

int pthread\_join(pthread\_t th, void \*thread\_return);

**43- detach? exit?**

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. Un thread può terminare chiamando: void pthread\_exit(void \*retval) con retval raccolto da join.

**44- Parla dei mutex in pthread. Lock e unlock?**

Astrazione simile al concetto di semaforo binario. Sono utilizzati per risolvere problemi di mutua esclusione. Un mutex è definito dal tipo pthread\_mutex\_t che rappresenta: - lo stato di mutex - la coda dei processi sospesi in attesa. Abbiamo 2 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.

**46- Cos'è la condition variable? Come usarla? Funzioni?**

Strumento di sincronizzazione associato ai lock che consente la sospensione dei thread in attesa che sia soddisfatta. Ad ogni condition variable viene associata una coda di attesa.

Tale coda garantisce la serializzazione, ma se un thread è entrato nel monitor e non riesce ad effettuare un'operazione perchè ad esempio una qualche condizione non è verificata si sospenderà in attesa della condizione con priorità maggiore rispetto a quelli che sono in attesa fuori. Si hanno dunque due code (thread in attesa di entrare nel monitor + thread in attesa dentro il monitor su una condizione → al risveglio tale cond potrebbe non valere più)

Una variabile condizione C viene creata e inizializzata con attributi nel modo seguente: p\_thread\_cond\_t C; p\_thread\_cond\_ init (&C,attr); con fun pthread\_cond\_wait/signal/broad.

**47- Semantica di signal nel dettaglio?**

Il thread T1 esegue signal e il thread T2 è in attesa. T2 viene trasferito dalla coda interna associata alla condition alla entry queue esterna (aka la “precedenza” degli interni si prefigura come l’essere messi per primi sull’esterna quando risvegliati, dato che erano già entrati ma la loro condizione in quel momento non valeva (FIFO ma con priorità)). Altri thread potrebbero entrare nel monitor prima di T2 e modificare le variabili condivise aka la condizione E utilizzata per entrare in attesa andrebbe ricontrollata dopo l’ingresso di T2 nel monitor. Per evitare questo tipo di situazioni l’operazione wait viene quindi solitamente usata con questo pattern: while (E) pthread\_cond\_wait(&C,&m);

**48- Esponi il produttore-consumatore coi pthread**

Ciclo lock/unlock con wait sulla cond da parte del consumatore e risveglio con signal da parte del producer dopo aver prodotto → v.slide per codice

**49- Pthread implementati con: spinlock o wait?**

Un misto: all’inizio spinlock, se passa troppo tempo vado in wait.

**50- Cos'è una barriera? Funzioni di pthread?**

Le barriere sono un meccanismo di sincronizzazione utilizzato ad esempio nel calcolo parallelo Una barriera rappresenta un punto di sincronizzazione per N thread. Si crea con:

int pthread\_barrier\_init o con la macro PTHREAD\_BARRIER\_INITIALIZER(count).

Ci si mette in attesa con pthread\_barrier\_wait e quando N thread sono in wait → via!

**50- Come implementare una barriera senza l'esplicita barriera?**

Un bel while con i join.

**51- Memory Barrier?**

Una memory barrier è una classe di istruzioni fornite dalle architetture per imporre un certo ordine nelle operazioni di lettura e scrittura in memoria. In particolare le operazioni di scrittura prima di una certa barriera vengono consolidate prima di eseguire quelle successive alla barriera (es: memory\_fence() / self\_sync() in C)

**52- Differenza tra dati in processo e thread?**

I thread condividono il segmento dati. Rispetto ai processi, i thread scambiano dati con altri thread con appositi meccanismi per disporre di dati privati (TSD), mentre i processi dispongono di dati privati del processo e di appositi meccanismi per dialogare con altri processi (IPC).

**53- Cosa sono i TSD?**

Può servire disporre di dati che siano globalmente visibili all’interno di un singolo thread ma distinti da thread a thread ● Ogni thread possiede un’area di memoria privata, la TSD area, indicizzata da chiavi ● Essa contiene associazioni tra le chiavi ed un valore di tipo void \* ● diversi thread possono usare le stesse chiavi ma i valori associati variano di thread in thread

**54- Esponi le funzioni per realizzare i TSD in C**

Si crea (es. nel Main) una chiave comune K (con pthread\_key\_create). Il thread la vedrà e genererà la chiave specifica (con pthread\_setspecific) associata a K e ad un puntatore ai dati privati. Userà poi la chiave specifica come se fosse globale (pthread\_getspecific)

**Analisi formale**

**55- Cos’è? Quali sono le difficoltà nelle metodologie tradizionali?**

Dati alcuni problemi nelle metodologie tradizionali (size, complexity, costo del testing, dependability, presenza di sistemi critici, difficoltà con metodologie classiche aka specifiche ambigue, bug, coverage limitato da test, low quality, limiti del peer reviewing, del testing e della simulazione) si può optare per un metodo alternativo che garantisca l’assenza di bug: i metodi formali (aka verifica assiomatica di programmi usando modelli matematici e logica)

**56- Descrivi brevemente i tre problemi formali visti**

Formal specification: Specify system requirements with formal, non ambiguous language. - first step for formal verification and synthesis - Consistency of formal specification may be checked automatically (e.g., theorem proving with logical notations).

Formal synthesis: given a specification S, synthesize a model M (system/program/circuit) which verifies it: M |= S

Formal verification: given a specification S, and a model M (system/program/circuit), check that M verifies S: M |= S → ci sono diversi approcci (ex. theorem proving e model checking)

**57- Descrivi brevemente le due tecnologie di Formal Verification viste**

Theorem Proving: System modeled as a set of logical formulae Γ Properties expressed as Theorems Ψ. Precise, unambiguous semantics Verification via logical reasoning: |= (Γ ⇒ Ψ)

Model Checking: Systems modeled as Finite State Machine M Properties expressed with a formal representation Ψ (e.g Temporal Logic, Automata, MSCs, etc.). ⇒ Precise, unambiguous semantics; Verification via logical reasoning: M |= Ψ

Is M a logical model for Ψ? Yes ⇒ the system verifies the property / No ⇒ a counter-example is returned (representing an execution leading to a bug).

**58- Differenza tra FV e RV**

Formal Verification (v. sopra: sono sicuro di trovare i bug) VS Runtime Verif. (esecuzione)

**Reti di Petri**

**59- Cosa sono? Definizione formale? Da che elementi sono composti?**

Le reti di Petri sono un modello computazionale basato su grafi (il modello di base non è Turing completo), usato per modellare sistemi concorrenti asincroni, distribuiti, paralleli, non deterministici, stocastici. Una rete di Petri è un grafo bipartito con nodi “place” (parti dello stato o risorse) e “transitions” (trans. tra stati e sync) e con archi che collegano i diversi nodi creando il flusso della rete. Lo stato si ha marcando i place con dei token. Una rete di Petri è una quadrupla (P,T,I,O) con P: insieme places, T: insieme trans., I:PxT→N funzione che mi da la molteplicità di ogni arco entrante nelle T, O:TxP→N viceversa gli uscenti.

**60- Definisci le transizioni enabled e fired. L’esecuzione è deterministica?**

Una transizione t è abilitata (enabled) in un marking M se: Per ogni arco da un place p alla transizione t, esiste un token distinto dagli altri in M Una transizione può essere eseguita (fired) e produrre un nuovo marking Il firing di una transizione è un'operazione atomica. L’esecuzione inoltre non è deterministica (simultanee o postposte).

**61- Cos’è il reachability graph? Quando una rete è bounded? Come approssimo?**

Una rete di Petri determina un reachability graph ottenuto a partire dalla marcatura iniziale. Se il numero di marcature possibili è finito (la rete è BOUNDED) allora il grafo di raggiungibilità è finito. Se la rete non è BOUNDED si possono usare tecniche di approssimazione per gestire cammini infiniti aka si “accelera” l'analisi mettendo il valore speciale infinito.

**62- Cos’è il coverability problem? Come si risolve, e con che complessità?**

Date la marcature M (iniziale) ed N (target) vogliamo determinare se esiste una computazione da M ad una marcatura P che in ogni place contiene almeno tanti token quanti N. Si può risolvere costruendo il Karp-Miller Tree a partire da M e poi controllando se esistono marcature (eventualmente con ∞) più grandi di N → esponenziale spazialmente!

**63- Cos’è il reachability problem? Come si risolve, e con che complessità?**

Date la marcatura M (iniziale) ed N (target) vogliamo determinare se esiste una computazione da M ad N. Questo problema è meno interessante di coverability ma molto più difficile da risolvere (ancora decidibile ma indecidibile per le Turing machines)

**Java Thread**

**64- Come realizza i Thread Java? Scheduling? Java Memory Model?**

Java supporta i thread in maniera indipendente dalla piattaforma sottostante attraverso la classe Thread • Tutti gli oggetti (pubblici) possono essere condivisi • L’ordine di esecuzione dei thread Java non è determinato a priori (potenziali race condition) • Il linguaggio fornisce costrutti di sincronizzazione specifiche del Java Memory Model. Il programma non termina fino a quando tutti i thread non hanno finito l’esecuzione • La JVM può associare kernel thread ai thread Java oppure simulare il multithreading.

**65- Quali elementi sono utili?**

Dato un oggetto Thread (oggetto che esegue un task in modo asincrono) abbiamo:

run: codice del Thread che farò partire sequenzialm. chiamandolo o in asincrono chiamando:

start: richiama run lanciando il thread e ritorna al chiamante (non posso chiamare 2 volte).

Per usare i Thread o estendo Thread o implemento Runnable se voglio preservare l’ereditarietà (creerò un nuovo Thread passandogli il Runnable). Distinguo dunque tra Runnable (task da eseguire) e Thread (elemento che esegue il “task”).

interrupt: Viene messo a true il flag di interruzione. Appena il thread entra in stato di blocked riceve una InterruptedException (che potrebbe ignorare!) → oppure posso interrompere con il flag timestop (che si può settare a true con il metodo mystop() dall’esterno) che il thread controllerà periodicamente.

isAlive(): True se il thread non ha ancora terminato l’esecuzione

volatile: Tutte le letture e scritture di un campo volatile devono essere fatte in memoria principale (no caching). Le operazioni su campi volatili devono essere eseguite esattamente nell’ordine richiesto dal thread.

join: Il caller attende che il thread t abbia terminato l’esecuzione

**66- Quali sono gli 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.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: quando un thread chiama wait(), sarà risvegliato da una notify()

**67- Cos’è synchronized in Java? Reentrant? Ereditarietà? Static**

La strategia di schedulazione è basata su code multiple di priorità con round-robin su ogni coda. Per cambiare la priorità (MIN priority=1): void setPriority(int Priority). In caso di context switch da un thread ad un’altro si potrebbero verificare delle race condition sulla parte di oggetto condivisa: i metodi di un oggetto si possono dunque dichiarare synchronized. 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 → aka un monitor: Quando il thread esce dal metodo synchronized rilascia automaticamente il lock, a quel punto gli altri thread ci proveranno.

Singoli blocchi di codice possono essere dichiarati synchronized su un certo oggetto (ma si perde la possibilita’ di documentare a livello di classe la sincronizzazione).

- Un lock reentrant non blocca il thread (in generale riprendere un lock)

- La specifica synchronized non fa parte vera e propria della segnatura di un metodo

- I metodi e blocchi synchronized non assicurano l’accesso mutuamente esclusivo alle parti statiche. Essi sono condivisi da tutti gli oggetti della stessa classe. Per accedere in modo sincronizzato ai dati statici si deve ottenere il lock sull’oggetto Class (metto metodo sync).

**68- Come si usano wait, notify e notifyAll? Ci sono delle eccezioni?**

Un thread può chiamare wait() su un oggetto sul quale ha il lock: il lock viene rilasciato e il thread va in stato di waiting. Altri thread possono ottenere tale lock e poi 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 ri-acquisire il lock. NB: i notify non sono cumulativi. Ad ogni oggetto e’ associato un wait-set.

Wait: – IllegalMonitorStateException - if the current thread is not the owner of the object’s monitor. – InterruptedException - if another thread has interrupted the current thread. • Notify: – IllegalMonitorStateException

**High Level Concurrency**

**69- Perchè introduciamo gli Executor?**

Un oggetto runnable corrisponde alla definizione di un task. La creazione di un thread attraverso il costruttore e i metodi della classe Thread permettono di controllare inizio e fine dell’esecuzione di un task (gestione a basso livello del ciclo di vita che bene per una politica “un task per thread”) → un task x thread però potrebbe essere dispendioso, dunque uso l’interfaccia Executor che fornisce un pattern per separare il task dal modo in cui lo eseguo (la politica definita da execute): executor.execute(new RunnableTask1());

**70- Quali sono 3 tipi di Executors visti con factory methods?**

Executors.newSingleThreadExecutor() (single background thread)

FixedThreadPool(int) (fixed size thread pool)

CachedThreadPool() (unbounded thread pool, with automatic thread reclamation)

**71- Quali parametri sono utili per creare un ThreadPoolExecutor?**

int corePoolSize (dimensione minima del pool: obiettivo riempirla asap), int maximumPoolSize, long keepAliveTime,TimeUnit unit, BlockingQueue workqueue

**72- Quali parametri corrispondono alla creazione manuale dei default?**

Fixed Thread Pool: core N, maxPool qualsiasi, queue: linked, alive: inf

Cached: core 0, maxPool inf, queue: Sync/linked/qualsiasi, alive: valore x

**73- Quali 3 tipi di coda possiamo usare?**

SynchronousQueue: dim 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: dim limitata, stabilita dal programmatore (si puo' andare oltre core)

LinkedBlockingQueue: dim illimitata. E' 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.

**74- Spiega le casistiche di gestione della coda**

Se sono in esecuzione tutti i core thread, un nuovo task sottomesso viene inserito nella coda. I task vengono poi prelevati da Q e inviati ai thread che si rendono disponibili. Solo quando Q risulta piena si crea un nuovo thread attivando così k thread tali che corePoolSize ≤ k ≤ maxPoolSize. 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 exception.

Supponiamo che un thread T termini l'esecuzione di un task e che il pool contenga k thread: Se k <= core: T si mette in attesa di nuovi task da eseguire. L'attesa ė indefinita.

Se k > core, si considera il timeout definito al momento della costruzione del thread pool.

Se nessun task viene sottomesso entro il timeout, T termina la sua esecuzione, riducendo così il numero di thread del pool.

**75- Come si effettua lo shutdown?**

Graduale: Si termina l'esecuzione dei task già sottomessi (compresi quelli in coda), ma non si inizia con nuovi (con shutdown()) VS Istantanea: Terminazione immediata con shutdownNow() che elimina i task in coda e li restituisce in una lista, e che tenta di terminare l’exec dei thread con interrupt() anche se non può garantire.

boolean isShutdown( ): true if this executor has been shut down.

boolean isTerminated( ): 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.

**76- Come potremmo implementare le code e i Pool?**

Le Blocking Queue sono un esempio di struttura dati concorrente con operazioni thread-safe (le op. bloccanti sono put e take). Vanno implementate usando semafori o monitor Java.

→ per le implementazioni v. slide

**Funzioni asincrone**

**77- Cosa sono caller e callee? Cos’è un Callable?**

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. Per ottenere un risultato non basta memorizzarlo nel Runnable e dare un getter (come sincronizzo?): introduco i Callable che permettono di aspettare che il risultato sia pronto: con ExecutorService possiamo usare oggetti Callable che ritornano valori di tipo T. Inoltre il metodo call() dell’interfaccia Callable solleva eccezioni.

**78- Cos’è un Future?**

I Future sono oggetti che rappresentano il risultato di un’esecuzione asincrona • Il metodo get() può essere usato per recuperare il risultato (bloccante se ris non c’è). È possibile definire un tempo massimo di attesa (poi TimeoutException) ed è possibile cancellare il task e verificare se la computazione è terminata oppure è stata cancellata.

**79- Cos’è un 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 poi l'oggetto FutureTask al costruttore del thread

**80- Cosa sono ConcurrentHashMap, CyclicBarrier, ReadWriteLocks e AtomicInt.s?**

ConcurrentHashMap è una versione thread-safe delle HashMap. Si usano lock solo sulle parti modificate in scrittura. Cyclic Barrier implementano le barriere in Java. Read/Write Lock sono lock che implementano la politica multiple readers, single writer. Usati così:

readWriteLock.readLock().lock(); readWriteLock.writeLock().lock();

// multiple readers, no writers // one writer, no readers

readWriteLock.readLock().unlock(); readWriteLock.writeLock().unlock();

**Java Thread il ritorno**

**81- Cosa non è visibile agli altri thread nell’architettura?**

La CPU Cache Memory (es. variabile count locale dentro oggetto)

**82- Le variabili locali sono Thread-Safe? I riferimenti a oggetti locali? A field?**

Le variabili locali sono allocate sullo stack di un thread e dunque sono threadsafe. I riferimenti locali ad oggetti in un metodo sono thread safe. Gli oggetti stessi tuttavia sono creati nell'heap. Se un oggetto creato localmente non esce mai dallo scope del metodo in cui è stato creato allora è threadsafe. MA 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. Se i thread effettuano update su istanze diverse allora non ci sono race condition.

**83- Cos’è la 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. Anche se un oggetto è thread safe, se punta ad un oggetto condiviso allora l'applicazione potrebbe non essere thread safe.

**84- Come si comportano gli oggetti immutabili?**

Possiamo garantire la thread-safeness per oggetti immutabili, una volta creati non vengono più modificati. Per modificare un oggetto immutabile bisogna costruire una nuova istanza.

**85- Cosa sono confinamento per metodo, per Thread e per oggetto?**

Per assicurare thread-safeness si possono usare tecniche di confinamento per metodo (ad es. si crea e si usa un oggetto all’interno del metodo senza poi usarlo altrove).

Un oggetto è confinato in un thread quando solo il thread che lo crea può accedervi Le TSD (che abbiamo visto nei PTHREAD) rientrano in quest’ambito In Java si possono usare oggetti java.lang.ThreadLocal o campi privati di una classe che deriva da java.lang.Thread.

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

**86- Perché Swing è un confinamento per Thread?**

UI è un’oggetto condiviso tra EDT che esegue i listener associati ad eventi di interazione con utente come click, ecc e tutti gli altri thread di un programma. Si ha la Java Single Thread Rule: un solo thread alla volta accede e modifica i campi della UI.

**Completable Future**

**87- Cosa sono i CompletableFuture? Differenze con Future?**

CompletableFuture sono utilizzati (come i Future) per programmare in Java in modo asincrono. Estendono i Future fornendo operazioni addizionali: • completamento manuale di un Future (con complete()) • composizione e aggregazione di Future

**88- Elenca i vari callable methods**

I metodi Async eseguono il task su un thread separato dal thread principale • Quando un metodo con suffisso viene chiamato, si può passare come secondo argomento un executor (dove il task verrà eseguito) • Quando nessun executor viene passato, di default viene usato ForkJoinPool.commonPool(). Abbiamo:

runAsync: eseguire in background task asincroni che non ritornano valori. Prende in input un Runnable e ritorna un CompletableFuture.

supplyAsync: quando vogliamo ritornare un valore da un task eseguito in background in modo asincrono. Prende in input un Supplier e ritorna un CompletableFuture.

thenApply: Prende un Function per concatenare una lambda. Il thread su cui thenApply viene eseguita dipende dallo stato del CompletableFuture su cui viene chiamata (v.slide).

thenAccept: Prende un Consumer e ritorna un CompletableFuture. Ha accesso al risultato del CompletableFuture sul quale è stato attaccato. Rispetto alla Apply non ritorna.

thenRun: Prende un Consumer e ritorna un CompletableFuture. Se non si ha bisogno del risultato della computazione e non si vuole ritornare alcun valore alla fine della catena di chiamate. Si può passare un Runnable.

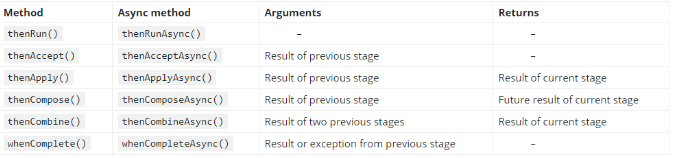
thenCompose: Combina due Future dipendenti. Questo metodo prende in input una funzione e ritorna un CompletableFuture. L’argomento della funzione è il risultato della computazione precedente.

thenCombine: Combina due Future indipendenti e produce qualcosa usando i risultati ottenuti dal loro completamento.

thenAcceptBoth: Viene utilizzato per effettuare una operazione su due Future indipendenti, ma non produce risultato che possa essere concatenato ulteriormente.

allOf: Metodo statico da utilizzare in caso si abbia una lista di Future indipendenti che vogliamo eseguire in parallelo e dove vogliamo fare qualcosa solo una volta che tutte abbiano terminato.

anyOf: Metodo statico che ritorna un singolo CompletableFuture che è completato quando uno dei CompletableFuture dati in input è completato.

Se un’eccezione viene sollevata in un CompletableFuture, l’eccezione viene propagata nella catena di chiamate successive. Ci sono due modi per gestire eccezioni nelle catene di CompletableFuture: exceptionally che recupera da un’eccezione ritornando un valore o eseguendo un’altra fun, o handle (+ generale) che ha result e ex (se c’è ex, result è null)

**JPF e Model Checking**

**89- Model Checking vs Testing vs Debugging**

Model Checking è un rigoroso metodo formale (tutti i cammini possibili) che trova tutti gli errori (che poi andremo a debuggare), mentre il testing non è detto che li esplori tutti.

**90- Cosa sono backtracking e state-matching?**

Backtracking: JPF può restaurare stati appartenenti a esecuzioni passate, per poter controllare se esistono scelte ancora da esplorare.

State matching • JPF controlla se ogni nuovo stato generato è stato già visto in precedenza, in questo caso la ricerca sul cammino corrente può essere interrotta. • Heap e Thread-Stack sono fotografati (snapshots)

**91- Cos’è uno stato? A cosa serve lo State Collapsing?**

Lo stato contiene informazioni riguardo ogni singolo thread: campi statici e dinamici (lock inclusi), …: tutto questo viene salvato in un array di interi.

Si riduce la memoria per salvataggio degli stati: • State collapsing • Premessa: solo una piccola parte dello stato viene modificata in una transizione (es. un singolo stack frame) • Divisione di uno stato in componenti; usare hash table per indicizzare una componente

**92- Cos’è State Reduction? Differenza tra Partial Order Reduction e Abstraction?**

La state reduction non si basa sul concetto “a basso livello” di stato, ma su Partial Order Reduction (ridurre lo spazio degli stati tramite l’identificazione di insiemi di stati di azioni concorrenti → l’interleaving a volte è ridondante, cioè ci sono path diversi in realtà equivalenti) e Abstraction(cioè eliminare i dettagli irrilevanti per la proprietà sotto analisi • Ottenere semplici modelli a stati finiti che siano sufficienti a verificare la proprietà • Svantaggio: • Perdita di precisione: False positives/negatives).

**93- Differenza tra Abstraction esistenziale e universale?**

Si astrae un dominio e si calcolano le transizioni in tale dom astratto, con due diversi approcci: esistenziale (Considerare una transizione da uno stato astratto se almeno uno stato concreto corrispondente possiede tale transizione) o universale (..se tutti..).

**94- Problemi dell’Abstracion? Che garanzie posso avere?**

Potrei avere over-approx nell’esistenziale o under nell’universale. Ho le seguenti garanzie (assumendo M’ astrazione di M):

Strong preservation: una proprietà P è verificata in M’ se e solo se P è verificata in M

Weak preservation: se una proprietà P è verificata in M’ allora P è verificata in M

→ Vale dunque (teorema di preservazione) che se M’ soddisa prop → M soddisfa prop.

Il controes. però può essere spurio (M’ non soddisfa prop !→ M non soddisfa prop)

**95- Perchè dovrei raffinare? E come funziona?**

Problema: Dead End e Bad States sono in qualche stato astratto. Rifinisco così:

Creare modello astratto M’ (da modello concreto M) • Effettuare model checking della proprietà P su M’ • Se la proprietà è verificata da M’, allora M soddisfa P (teorema di preservazione) • Altrimenti, controllare se il controesempio è spurio • Raffinare lo spazio degli stati astratti utilizzando il risultato dell’analisi sul controesempio

**SWING**

**96- Che relazione c’è tra Swing e i thread? Come sfruttare la concorrenza?**

La realizzazione di interfacce utente tramite Swing coinvolge tre tipologie di thread: 1. i thread iniziali: i thread che eseguono il codice iniziale dell’applicazione (es. il main) e/o i thread creati dal programmatore 2. l’event dispatch thread (EDT): il thread che esegue il codice di gestione degli eventi e gestisce l’interazione con l’utente 3. i worker thread (o background thread): thread dedicati che eseguono operazioni “lunghe” (time-consuming) in background, in modo da non compromettere la reattività del thread che gestisce l’interazione con l’utente (cio`e l’EDT). In generale il programmatore non deve creare esplicitamente questi thread: • sono forniti dall’ambiente runtime di Java (il thread iniziale) o dalla libreria Swing (l’event dispatch thread e i worker thread) • ovviamente il programmatore può creare ed utilizzare thread aggiuntivi (cfr. Thread e Runnable)

**97- Descrivi le caratteristiche principali dell’EDT**

L’EDT è il thread che: esegue il drawing dei componenti grafici, esegue il codice per la gestione degli eventi generati dall’interazione tra i componenti grafici e l’utente (pressione di bottoni, ridimensionamento di finestre, click del mouse, etc.). L’EDT è un thread di sistema che: 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 inseriti nella coda, preleva gli eventi dalla coda, li notifica ai Listener corrispondenti ed esegue il codice di gestione degli eventi. In un’applicazione con interfaccia Swing la “vita” dell’EDT contribuisce a determinare la persistenza del programma: • L’event dispatch thread è un thread non demone → persiste finché c'e almeno un evento da gestire nella coda o finchè almeno una finestra si trova nello stato visualizzabile. La JVM resta attiva finchè è attivo almeno un thread non demone

**98- Descrivi la Single Thread Rule. Eccezioni? Come interagire con l’EDT?**

Ogni operazione che consiste nella visualizzazione, modifica o aggiornamento di un componente Swing, o che accede allo stato del componente stesso, deve essere eseguita nell’event dispatch thread. Gran parte dei metodi che operano su oggetti della libreria Swing non sono threadsafe → se mi attengo alla regola no problem, anche se ci sono eccezioni:

- metodi thread safe quali append - interfaccia iniziale nel main (creando obj Runnable con metodo run e poi interagendo con i metodi invoke) - repaint, (re/in)validate anche fuori dall’EDT - liste di Listener modificabili da qualsiasi thread.

La classe SwingUtilities offre due metodi statici che consentono a qualsiasi thread di interagire con l’EDT in modo che esegua del codice definito dal programmatore → il codice deve essere definito nel metodo run del runnable da passare a: invokeLater (async) e invokeAndWait.

**99- Descrivi problema e soluzione dei Long-Running Task**

Se l’EDT è impegnato nell’esecuzione di attività lunghe o nell’esecuzione di codice con istruzioni bloccanti l’applicazione appare “congelata” → uso i worker threads lanciati dall’EDT, che potrà continuare a processare e gestire gli eventi.

**100- Descrivi le caratteristiche principali dello Swing Worker**

è una classe astratta (con param generici T: risultato finale e V: risultati intermedi) che implementa l’interfaccia RunnableFuture (wrapper per le interfacce Runnable e Future) e permette di: definire le attività e la computazione da eseguire in background, produrre aggiornamenti sull’avanzamento della computazione e comunicare risultati intermedi all’EDT per aggiornare la GUI, ritornare all’EDT un oggetto che rappresenta il risultato della computazione, eseguire nell’EDT il codice di aggiornamento della GUI al termine della computazione, definire bound properties: proprietà (variabili di istanza) che quando modificate dal worker thread causano l’invio di un evento all’EDT.

Per usarla si estende SwingWorker, si fa override di doInBackground, si esegue sull’obj il metodo execute, al termine di doInBack verrà eseguito done dall’EDT (che può prendere i risultati della computazione con get()). I risultati intermedi possono essere pubblicati durante la doInBack con publish(), che verranno gestiti dal metodo process() eseguito dall’EDT. Per segnalare ad uno SwingWorker che l’attività in esecuzione in background deve essere interrotta invocare sull’oggetto SwingWorker il metodo cancel (simile a interrupt). Ogni oggetto della classe SwingWorker è caratterizzato da due variabili di stato: progress (int tra 0 e 100 che da lo stato di avanzamento del worker) e state (stato del worker: PENDING (prima di doInBack), STARTED (durante) e DONE). Al cambiamento di tali due variabili viene generato un evento specifico (PropertyChangeListener per monitorarlo).

**Algoritmi Distribuiti**

**101- Parla in generale degli algoritmi distribuiti e delle loro proprietà**

A distributed system is a network that consists of autonomous computers that are connected using a distribution middleware. • They help in sharing different resources and capabilities to provide users with a single and integrated coherent network. • Main features: • Several autonomous nodes, each of which has its own local memory • Nodes communicate with each other by message passing. Properties: location transparency, concurrent/no-shared-memory, no global state, fault-tollerance, heterogeneity.

Also clock may not be accurate and can be out of sync and messages can be delayed for arbitrary period of times.One possible architecture to avoid to create bottlenecks (i.e. serve clients sequentially) is to use multithreaded programming.

**102- Colouring Paths: come funziona l’algoritmo deterministico?**

Ho nodi e 3 colori: voglio non avere lo stesso colore dei vicini. Uso alg con identificatori:

mando msg c ai vicni e ricevo i messaggi M dai vicini. Se c non è in {1, 2, 3} e c > max(M):

allora c = min ( {1, 2, 3} \ M) → raggiunto lo stop state non cambia più stato.

**103- Colouring Paths: come funziona l’algoritmo randomizzato?**

Provo a caso finchè non becco un colore diverso dai miei vicini. Converge con O(logN)

**104- Logical Clocks: Happens Before e clock scalari**

Algorithms based on logical clocks make use of timestamps to sort operations on different nodes of a network. The relation a → b defines the event “a happened before b” in a set of distributed nodes (if a is the event of sending message “m” and b is the event of its reception in a different node then a → b). We would like to find a way to assign to each event “e” a timestamp C(e) such that if a → b then C (a) < C (b) (NON vale il contrario aka non posso usare i timestamp per identificare eventi concorrenti). Aggiungo +1 al clock per ogni evento interno e quando invio, e per la ricezione metto in clock il massimo tra il mio clock e il timestamp ricevuto (+1).

**105- Logical Clocks: clock vettoriali**

Every node maintains a local representation of the logical clocks of all other nodes (for N nodes a vector of N elements) V[i] in node j = what node j knows about the logical clock of process i. Per eventi interni e send aumento V[i] in Pi; e per la receive metto in ogni V[j] il massimo tra V[j] e il T[j] ricevuto (poi aggiorno il mio V[i] a +1). Questa volta vale il sse.

**Programmazione Distribuita in Java**

**106- Cosa sono Stream e filtri?**

Stream = punto terminale di un canale di comunicazione unidirezionale associato ad un File aperto o ad un Socket (FIFO, read e write bloccanti, ho eccezioni). I filtri sono classi derivate da InputStream e OutputStream che permettono di lavorare su dati strutturati e non byte.

**107- Cosa sono i. Socket in Java?**

Socket: astrae il concetto di 'communication endpoint' • Individuato da un indirizzo IP e da un numero di porta • Socket in JAVA: istanza di una di queste classi – Socket – ServerSocket – DatagramSocket – MulticastSocket. Un socket viene chiuso automaticamente: dal garbage collector, se non più raggiungibile – alla terminazione del programma – chiusura esplicita.

**108- Che schema ha un’implementazione di Server multithreaded?**

crea un Listening Socket LS sulla porta associata al servizio pubblicato e si mette in ascolto • quando accetta una richiesta da C crea un nuovo Active Socket e relativi stream su cui avviene la comunicazione con C • quando l’interazione con il client è terminata, chiude il data socket e torna ad ascoltare su CS ulteriori richieste di connessione

→ accept bloccante su ServerSocket che ritorna un Data Socekt. Possiamo usare le tecniche di confinamento e passare il nuovo socket ad un thread (o ad task sottomesso ad thread pool) che gestirà quindi le connessioni. Ho One request One Thread (1 request implica la creazione di 1 thread) e One request One Task (mando al threadPool).