JAVA 2

Concurrency

Teme

- *Multiprocessor vs Multicore processor*
- *Process, Thread, Task*
- *Multi**
- *JVM*
- *Runnable vs Thread, Executors*
- Upravljanje *threadovima*
- Kooperacijski mehanizmi
- Sinhronizacija
- Sinhronizacijski mehanizmi
- Callable i Fork-Join

Multiprocessor vs Multicore processor

- *multiprocessor* – više CPU jedinica koje rade paralelno i mogu istovremeno izvršavati više programa
  - SMP - **S**imultaneous **M**ulti**P**rocessing
  - efikasniji za izvršavanje više programa (*multiprocessing*)
  - omogućuje paralelizam
- *multicore* – jedan CPU sa više repliciranih jedinica (*core*) koje mogu istovremeno izvršavati više instrukcija (istog ili različith programa)
  - CMP – ***C**hip–Level **M**ulti**P**rocessing* – svaka jedinica ima vlastitu jedinicu izvršavanja i L1 *cache,* dok L2 cache dijeli sa ostalim jedinicama
  - omogućuje paralelizam
  - efikasniji za izvršavanje jednog programa (*multithreading*)

Process, Thread, Task

- *process* - instanca programa u izvođenju
  - sadrži kod i sve resurse potrebne za izvođenje
  - sadrži najmanje jedan *thread*
  - primjer: *MS Word process*
- *thread* – najmanji dio procesa, odnosno sekvenci instrukcija
  - dijeli isti memorijski prostor sa procesom i njegovim threadovima
  - primjer: *MS Word process* - *auto-save i spell-check threadovi*
- *task* – programske instrukcije učitane u memoriju
  - *unit of work* – instanca posla

*Multi**

- *multiprocessing* – **paralelno** izvršavanje više programa, na različitim procesorima, dijeleći RAM i periferije
- *multiprogramming* – **konkurentno** izvršavanje više programa, na istom procesoru – jeftinije, ali sporije od *multiprocessinga*
  - OS drži programe u *poolu* i upravlja redovima izvršavanja (CPU, I/O *queue*) – kada jedan program završi sa CPU i traži I/O, ulazi u I/O *queue*, a istovremeno novi program iz CPU *queue* počinje sa izvršavanjem – *process context switching*
- *multithreading* – **konkurentno** izvršavanje *threadova* istog procesa
  - *thread context switching*
- *multitasking* – **konkurentno** izvršavanje programa, procesa, *threadova...*
  - logička ekstenzija *multiprogramminga*, baziran na vremenskom izvršavanju

JVM

- implementira *preemptive*, *priority based scheduler*
  - radi u suglasnosti sa OS *thread schedulerom*
  - *preemptive* – osigurava da će procesorsko vrijeme biti vremenski dijeljeno između različitih *threadova*
    - *time-slicing* osigurava OS
  - *priority based* – svaki od *threadova* ima dodijeljen prioritet, te može biti prekinut kako bi se *thread* većeg prioriteta mogao izvršiti
    - Thread.MIN_PRIORITY, Thread.NORM_PRIORITY, Thread.MAX_PRIORITY
    - kako ne bi izazvao izgladnjivanje (*starvation),* OS *thread scheduler* može odabrati *thread* manjeg (ili istog, izgladnjenog) prioriteta za izvršavanje

Runnable vs Thread, Executors

- kreiranje *low level threada*
  - *extends Thread* – ujedno i ograničenje radi *Single Inheritance Model*
  - *implements Runnable* - omogućuje fleksibilnost jer klasi dodaje novo ponašanje (*mixin*)
  - vrlo skupo i nezgodno za upravljanje
- *Executors*
  - *high level API* – kvalitetno upravljanje *threadovima,* omogućuje ponovno iskorištavanje instanci, a ne ponovno kreiranje po potrebi
    - *Fixed thread pool executor* – kreira *thread pool* sa određenim brojem *threadova*
    - *Cached thread pool executor* – kreira *thread pool* koji po potrebi kreira i *cacheira threadove* za ponovnu uporabu
    - *Scheduled thread pool executor* – kreira *thread pool* koji omogućava pokretanje *threadova* nakon određenog *delaya* ili periodički
    - *Single thread pool executor* – jedinstveni *thread* za sve taskove
    - *Work stealing thread pool executor* – kreira *thread pool* koji održava dovoljno *threadova* da podrži definiran *level paralelizma*

Upravljanje *threadovima*

- *sleep()* – *static,* nema smisla pozivati ga na instanci – *thread* u izvršavanju *spava* određeno vrijeme, držeći *mutex lock*
- *setDaemon()* – *daemon thread* – ne priječi JVM od gašenja premda je još uvijek u stanju izvršavanja (*garbage collector*)
- *join()* – omogućava da jedan *thread* čeka na izvršenje drugoga
- *setPriority(Thread.*_PRIORITY)* – omogućava određenim *threadovima* dati prednost
- *yield()* – *thread* može prepustiti resurse drugim *threadovima*
- *ThreadGroup* – povezivanje *threadova* radi upravljanja

Kooperacijski mehanizmi

- *wait()* – govori *threadu* da prepusti *lock* i čeka dok drugi *thread* ne pozove *notify(), notifyAll()*
  - nalik *sleep()*, ali *wait()* oslobađa *lock*, dok *sleep()* ne oslobađa
- *notify()* – budi jedan *thread* koji je pozvao *wait()* nad istim *lockom*
- *notifyAll()* - budi sve *threadove* koji su pozvali *wait()* nad istim *lockom*
- *producer consumer* problem
- *BlockingQueue* – sakriva komukacijske mehanizme *pod haubom*

Sinhronizacija

- ugrađeni monitor –omogućava *threadovima mutex* (*mutual exclusion*) i kooperaciju
- *synchronized* – kritična sekcija
  - *pesimistic* – prvo provjerava da li *thread* može ući u kritičnu sekciju, a potom osigurava da nijedan *drugi* thread ne može ući
  - *optimistic* - drugi pristup je CAS (*compare and swap*) – dozvoljava se *update*, a potom provjeri da li je bilo *smetnji* – ako je bilo, poništava akciju – *AtomicInteger*, *AtomicLong...*
  - označava kritičnu sekciju – blok, metoda, na izvoru ili klijentu
  - instanca – svi *threadovi* koji koriste isti objekt su sinhronizirani
  - instanca klase – svi *threadovi* koi koriste istu klasu su sinhronizirani
  - definiran kao *re-entrant mutex lock object*
    - ako sinhronizirana metoda pozove drugu sinhroniziranu metodu, slobodno ulazi
- *Lock*
  - može imati *timeout*
  - *lock()* i *unlock()* metode mogu biti pozvane iz raznih metoda, dok je *synchronized* ograničen na jednu metodu, odnosno blok unutar metode
- paziti na *DeadLock!*

Sinhronizacijski mehanizmi

- *Semaphore* – kontrolira broj konkurentnih *threadova* koji pristupaju resursu (*permits*) – ako ih više pristupa, moraju čekati da se resurs oslobodi
- *Count down latch* – koristi se u situaciji kada jedan *thread* očekuje da se drugi *threadovi* izvrše – efektivno zaustavlja izvršenje *threada* dok ostali ne također ne završe
  - primjer – startanje klijenta tek nakon što su pokrenuti server i baza
- *Cyclic barrier* – koristi se da više *threadova* (*parties*) čeka dok svi ne dođu do određenog mjesta (*barrier*)
  - primjer – 2 igrača čekaju da im se pridruži treći

Callable i Fork-Join

- *Callable* – sučelje dizajnirano za izvršavanje u *threadovima*, kao i *Runnable*
  - metoda *call()* - vraća rezultat i baca exception, za razliku od *Runnable run()* metode
- *Fork-Join* – rekurzivno razbija *task* u manje dok nisu dovoljno jednostavni za izvršenje
  - *divide and conquer algorithm*
  - *work stealing* - omogućava da procesor nikada nije *idle*

Demo

- Project

Izvor:http://www.jnhsolutions.com/contact-us/request-a-demo/

Hvala na pažnji!