1.Programare multithread

Programarea multithread constituie o caracteristică de bază a limbajului Java, ce permite crearea de obiecte multiple care pot executa simultan diferite operații, oferind astfel suport pentru concurență și paralelism în aplicații.

1.1.Obiective

- Studiul mecanismelor de programare cu threaduri multiple în Java
- Implementarea unor aplicații ce presupun concurență/paralelism prin crearea și utilizarea mai multor threaduri

1.2. Concepte

Firele de execuţie (engl. threaduri) fac trecerea de la programarea secvenţială la programarea concurentă/paralelă. Java integrează mecanismele programării concurente sub forma unor clase şi interfeţe specifice, oferind astfel suportul necesar programării aplicaţiilor moderne ce necesită threaduri multiple. Se oferă astfel suport pentru creşterea performanţelor aplicaţiilor prin implementarea *paralelismului real* pentru sisteme *multiprocesor*, respectiv a *paralelismului prin întreţeserea threadurilor* pentru sisteme monoprocesor.

Suportul multithread este implementat de maşina virtuală Java (JVM) , iar manifestarea threadurilor este direct influenţată de modul în care sistemul de operare gestionează threadurile multiple,diferit pentru diverse platforme.Un sistem de operare monotasking nu este capabil să execute decât un singur proces la un moment dat, în timp ce un sistem de operare multitasking poate rula oricâte procese în acelaşi timp concurent, alocând periodic *cuante* din timpul de lucru al CPU fiecărui proces. Am reamintit acest lucru deoarece noţiunea de fir de execuţie nu are sens decât în cadrul unui sistem de operare multitasking. Un fir de execuţie este similar unui proces secvenţial în sensul că are un început, o secvenţa de execuţie şi un sfârşit. Un program îşi poate defini însă nu doar un fir de execuţie ci oricâte, ceea ce înseamnă că în cadrul unui proces se pot executa simultan mai multe fire de execuţie, permiţând astfel execuţia concurentă a sarcinilor independente ale programului.

Execuţia simultană a firelor de execuţie în cadrul unui proces este similară cu execuţia concurentă a proceselor: sistemul de operare va aloca ciclic cuante din timpul procesorului fiecărui fir de execuţie până la terminarea lor. Deosebirea majoră între proces și thread constă în faptul că acestea din urmă nu pot exista decât în cadrul unui proces, astfel la crearea unui nou proces fiu este realizată o copie a procesului părinte: cod + date, iar la crearea unui fir de execuţie nu este copiat decât codul procesului părinte; toate firele de execuţie au deci *acces la aceleaşi date,* datele procesului original. Deasemenea, comutarea de context și comunicarea între threaduri este mult mai simplă și mai rapidă.

Proiectarea şi implementarea aplicaţiilor folosind threaduri oferă o serie de avantaje, astfel:

- simplificarea programării unor aplicații mai complexe şi o mai bună structurare a codului
- eficienţă în utilizarea resurselor sistemului (modelul multithread poate înlocui soluţiile clasice de comunicare între procese de tip IPC pentru anumite probleme)
- ascunderea latenţei acceselor la memorie, I-O sau comunicaţiei şi astfel câştig de performanţă prin suprapunere comunicaţie-procesare
- îmbunătăţirea interacţiunii la nivel de aplicaţie prin proiectarea unor interfeţe performante, responsive controlate printr-un thread GUI distinct
- sistemele moderne de obiecte distribuite sunt sisteme multithread (orice server de obiecte distribuite este multithread)
- suport pentru planificare şi echilibrarea încărcării de execuţie prin maparea dinamică a taskurilor (asociate unor threaduri) la procesoare (pentru sistemele actuale cu procesoare multiple)

1.2.1.Clasele programării Java multithread

Aplicațiile sunt structurate în procese şi astfel un proces master creează alte procese explicit pentru a putea separa logic funcționalitățile aplicației. Dacă procesele pot fi considerate construcții ce deteremină arhitectura aplicației, threadurile sunt construcții specifice limbajului.

Clasa Obiect . Următoarele metode ale clasei Object şi anume *wait(), notify() şi notifyAll(),* sunt utilizate în programarea multithread, având următoarea semnificaţie:

- *wait()* - pune obiectul în aşteptare până la apariţia unui eveniment (notificare) cu/fără indicarea duratei maxime de aşteptare
- *notify()* - permite anunţarea altor obiecte de apariţia unui eveniment
- *notifyAll()* - implementează mod broadcast notificarea mai multor obiecte la apariţia unor evenimente
- *finalize()* - poate fi suprascrisă în clase şi defineşte diversele acţiuni ce vor fi executate de "colectorul de deşeuri" JVM înaintea distrugerii obiectului

Clasa Thread. Maşina virtuală Java permite definirea mai multor threaduri concurente, orice thread este un obiect Java ce posedă metode şi poate fi transferat ca parametru, plasat într-un tablou, etc. Maşina virtuală Java realizează maparea obiectului Java de tip runnable unei implementări de thread dependentă de sistem, iar sistemul de operare îi alocă resurse. Orice obiect de tip thread posedă o metodă *run()* prin intermediul căreia se implementează manifestarea threadului, definind o buclă ce se execută până la terminarea acestuia.

Prin intermediul atributelor unui thread se pot defini caracteristici specifice acestuia şi anume: *identificatorul de thread, numele, politica de planificare, prioritatea*, cine este responsabil cu controlul execuţiei threadului (procesul utilizator sau nucleul sistemului de operare). Un thread există în cadrul unui proces, el este compus din context, structura utilizator conţinând copii ale valorilor regiştrilor generali, stiva, *zona de date private* şi *setul de instrucţiuni*. Un thread este o entitate planificabilă, ea poate fi întreruptă preemptiv şi poate utiliza în funcţie de numărul de procesoare din sistem concurenţa reală (multiprocesor) sau logică (monoprocesor). Cele mai importante proprietăți ale unui thread sunt:

- threadurile posedă nume pentru identificare, însă pot exista mai multe threaduri cu acelaşi nume, dacă numele nu a fost specificat la creare, se va genera un nume nou, implicit, acesta putând fi citit cu metoda getName
- orice nou thread începe execuţia cu *metoda run()* similar începerii execuţiei unui program cu metoda main()
- orice thread posedă **o** *anumită prioritate* (valoare întreagă între Thread.MIN_PRIORITY şi Thread.MAX_PRIORITY,în rangul 1-10,în mod curent este utilizat Thread.NORM_PRIORITY,având valoarea 5), prioritatea threadului nou creat este identică cu a threadului părinte. Maşina virtuală mapează priorităţile threadurilor nivelurilor de prioritate ale platformei care poate dispune de mai puţine niveluri de prioritate (sub Windows, anumite niveluri JVM vor fi mapate aceloraşi niveluri de sistem).

Algoritmul de planificare favorizează threadurile cu prioritate mai mare, diverse *politici de planificare* trebuie utilizate doar pentru eficientizarea unui anumit algoritm, care însă trebuie să se poată executa corect şi în absenţa acestuia.

Principalele câmpuri şi metode ale clasei Thread sunt:

- *void start()* - lansează în execuţie noul thread, moment în care execuţia programului este controlată de cel puţin două threaduri: threadul curent ce execută metoda start şi noul thread ale cărui instrucţiuni sunt definite în metoda run ().

- *void run()* – defineşte corpul threadului nou creat, întreaga activitate a threadului va fi descrisă prin suprascrierea acestei metode
- *static void sleep ()* – pune în aşteptare threadul curent pentru un anumit interval de timp (*msecs*)
- *void join ()* - se aşteaptă ca obiectul thread ce apelează această metodă să se termine
- *suspend()* - suspendare temporară a threadului (*resume() este metoda* dual ce relansează un thread suspendat (implementările JDK ulterioare versiunii 1.2 au renunţat la utilizarea lor)
- *yield()* - realizează cedarea controlului de la obiectul thread, planificatorului JVM pentru a permite unui alt thread să ruleze
- *void interrupt()* – trimite o întrerupere obiectului thread ce o invocă (setează un flag de întrerupere a threadului activ). Metodele isInterrupted() şi interrupted() permit testarea stării de întrerupere a threadului apelant. Metoda interrupted() modifcă starea threadului curent (la un apel secund al ei starea threadului revine la cea iniţială).
- *static boolean interrupted()* - metodă statică, testează dacă threadul curent a fost întrerupt, resetează starea interrupted a threadului current
- *boolean isInterrupted* () - testează dacă un thread a fost întrerupt fără a modifica starea threadului
- *static Thread current Thread()* - returnează obiectul reprezentând threadul curent în execuţie
- *boolean isAlive()* - permite identificarea stării obiectului thread, astfel metoda returnează *true* dacă threadul a fost pornit şi nu a murit încă,respectiv *false* dacă threadul nu a fost pornit sau a murit, fără a putea diferenţia între un thread ce nu a fost încă pornit, respectiv unul ce a murit.
- *void SetDaemon(boolean on)* - apelată imediat înainte de start permite definirea threadului ca daemon. Un thread este numit daemon, dacă metoda lui run conţine un ciclu infinit, astfel încat acesta nu se va termina la terminarea threadului părinte.
- *getPriority()* - returnează prioritatea threadului curent
- *setPriority(newPriority)* -permite atribuirea pentru threadul curent a unei priorităţi dintr-un interval.

Metodele stop(), suspend() şi resume(),definite în versiuni anterioare au fost eliminate deoarece în cazul unei proiectări defectuoase a codului pot provoca blocarea acestuia (nu sunt thread-safe, altfel spus nu asigură execuţie corectă). O clasă aparte de thread-uri sunt cele Daemon care sunt thread-uri de serviciu (aflate în serviciul altor fire de execuţie). Când se porneşte maşina virtuală Java, există un singur fir de execuţie care nu este de tip Daemon şi care apelează metoda main(). JVM rămâne pornită cât există activ un thread care să nu fie de tipul Daemon. Metoda setDaemon poate fi utilizată pentru transformarea unui thread în daemon şi invers.

1.2.2.Controlul şi ciclul de viaţă al unui thread

Fiecare fir de execuţie are propriul său ciclu de viaţă: este creat, devine activ prin lansarea sa în execuţie şi la un moment dat, se termină. În continuare vom vedea mai îndeaproape stările în care se poate găsi un fir de execuţie. Diagrama din Figura 1.1. ilustrează aceste stări precum şi metodele care provoacă tranziţia dintr-o stare în alta. Aşadar, un fir de execuţie se poate găsi în una din următoarele stări ilustrate în Figura 1.1, acestea vor fi descrise în continuare.

Starea "New Thread" .
Un fir de execuţie se găseşte în această stare imediat după crearea sa, cu alte cuvinte după instanţierea unui obiect din clasa Thread sau dintr-o subclasă a sa.

```
Thread counterThread = new Thread ( this );
//counterThread se găseşte în starea New Thread
```

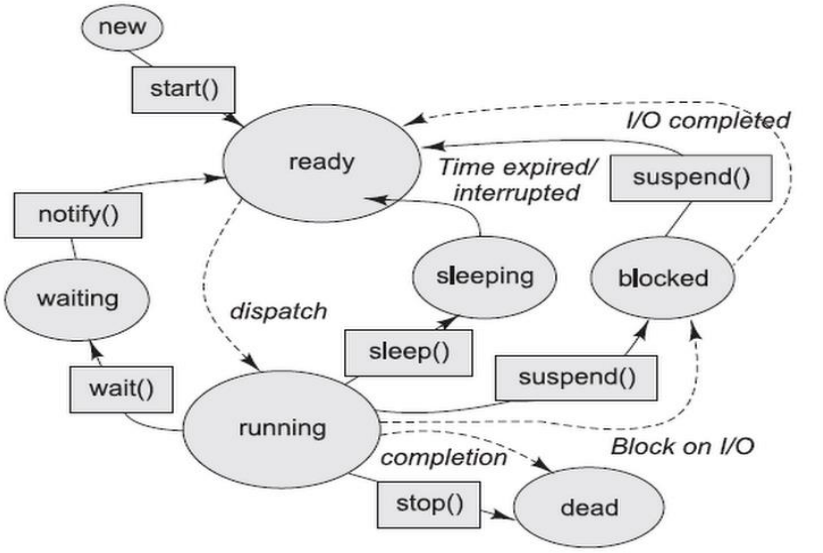

Figura 1.1. Stările unui thread Java [4]

În această stare firul de execuţie este "vid", el nu are alocate nici un fel de resurse sistem şi singura operaţiune pe care o putem executa asupra lui este lansarea în execuţie, prin metoda start(). Apelul oricărei alte metode în afară de start nu are nici un sens şi va provoca o excepţie de tipul IllegalThreadStateException.

Stări de tip "Runnable"(ready/running)

După apelul metodei start un fir de execuţie va trece în starea "Ready", adică se găseşte în execuţie.

```
counterThread.start();
//counterThread se găseşte în staarea ready
```

Metoda *start* realizează următoarele operaţiuni necesare rulării firului de execuţie:

- alocă resursele sistem necesare
- planifică firul de execuţie la CPU pentru a fi lansat
- apelează metoda run a obiectului reprezentat de firul de execuţie

Stări de tip "Not Runnable" (sleeping/blocked/waiting)

Un fir de execuţie ajunge în această stare în una din următoarele situaţii:

- este "adormit" prin apelul metodei *sleep.*
- a apelat metoda *wait,* aşteptând ca o anumită condiţie să fie satisfăcută
- este blocat într-o operaţie de intrare/ieşire

Metoda *sleep* este o metodă statică a clasei Thread care provoacă o pauză în timpul rulării firului curent aflat în execuţie, cu alte cuvinte îl "adoarme" pentru un timp specificat. Lungimea acestei pauze este specificată în milisecunde/ nanosecunde.

```
public static void sleep( long millis ) throws InterruptedException
public static void sleep( long millis, int nanos ) throws InterruptedException
```

Întrucât poate provoca excepţii de tipul InterruptedException apelul acestei metode se face într-un bloc de tip try-catch:

```
try { Thread.sleep(1000);
                    //face pauza de o secunda
                            } catch (InterruptedException e) {
                                    . . . }
```

După expirarea acestui interval threadul revine în starea Running, iar dacă procesorul este în continuare disponibil, îşi continua execuţia. Pentru fiecare tip de intrare în starea "Not Runnable", există o secvenţă specifică de ieşire din starea respectivă, care readuce firul de execuţie în stări Runnable. Acestea sunt:

- dacă un fir de execuţie a fost "adormit", atunci el devine Runnable doar după scurgerea intervalului de timp specificat de instrucţiunea sleep.
- dacă un fir de execuţie aşteaptă o anumită condiţie, atunci un alt obiect trebuie să îl informeze dacă acea condiţie este îndeplinită sau nu; acest lucru se realizează prin instrucţiunile *notify sau notifyAll*
- dacă un fir de execuţie este blocat într-o operaţiune de intrare/ieşire atunci el redevine Running atunci când acea operaţiune s-a terminat.

Starea "Dead"

Este starea în care ajunge un fir de execuţie la terminarea sa. Un fir de execuţie nu poate fi oprit din program printr-o anumită metodă, ci trebuie să se termine în mod natural la terminarea metodei run pe care o execută. Spre deosebire de versiunile curente ale limbajului Java, în versiunile mai vechi exista metoda stop a clasei Thread care termină forţat un fir de execuţie, însă ea a fost eliminată din motive de securitate.

1.2.3.Operaţii cu threaduri Java

a. Crearea şi pornirea unui thread

Există două tehnici pentru crearea unui nou thread de execuţie
şi implementarea metodei run() şi anume: extinderea clasei Thread (implementarea unei clase derivate) şi suprascrierea metodei run(), respective, definirea unei clase ce implementează o interfaţă Runnable

Pentru a decide care din cele două metode este necesară unui anumit context de programare este utilă următoarea observaţie: dacă clasa este derivată din altă clasă este de preferat să implementeze interfaţa Runnable (datorită lipsei mecanismului de moştenire multiplă în limbajul Java), iar în caz contrar, se utilizează extinderea clasei Thread, ambele soluţii fiind implementate în pachetul java.lang.

Varianta1.Extinderea clasei java.lang.Thread

Etapele de creare a threadului folosind subclasarea sunt :

1. se creează o clasă derivată din clasa Thread
2. se suprascrie metoda public void run() moştenită din clasa Thread
3. se instanţiază un obiect thread folosind new
4. se porneşte thread-ul instanţiat, prin apelul metodei start() moştenită din clasa Thread. Apelul acestei metode face ca maşina virtuală Java să creeze contextul de program necesar unui thread după care să apeleze metoda run().

```
public class Fir
{ public static void main(String args[])
  {
     FirdeExecutie fir=new FirdeExecutie();
     fir.start();
     System.out.println("Revenim la main"); }}

class FirdeExecutie extends Thread
{
   public void run()
   {
      for(int i=0;i<10;i++)
         System.out.println("Pasul "+i);
      System.out.println("Run s-a terminat");}}
```

Variata2.Implementare folosind interfaţa java.lang.Runnable.

Pentru această metodă de implementare, etapele specifice sunt:

1. se creează o clasă care implementează interfaţa Runnable
2. se implemetează metoda run() din interfaţă
3. se instanţiază un obiect al clasei folosind new()
4. se creează un obiect din clasa Thread folosind un constructor care are ca parametru un obiect de tip Runnable (un obiect al clasei ce implementează interfaţa)
5. se porneşte thread-ul creat la pasul anterior prin apelul metodei start().

```
public class Fir
{
   public static void main(String args[])
  {
     FirdeExecutie fir=new FirdeExecutie();
     Thread thread=new Thread(fir);
     thread.start();
     System.out.println("Revenim la main");}}
 class A  {
  public void afis()
 {
   System.out.println("Este un exemplu simplu"); } }
class FirdeExecutie extends A implements Runnable
{
   public void run()
   {
      for(int i=0;i<5;i++)
         System.out.println("Pasul "+i);
           afis();
         System.out.println("Run s-a terminat");}}
```

Aceasta este o modalitate extrem de utilă atunci când clasa de tip Thread care se doreşte a fi implementată moşteneşte o altă clasă (Java nu permite moştenirea multiplă). Interfaţa Runnable descrie o singură metodă run().

Metoda start()creează resursele sistem necesare execuţiei threadului şi apelează metoda run. După revenire din această metodă, threadul se află în starea runnable, sistemul runtime Java implementează o schemă de planificare ce partajează procesorul (sau procesoarele) între toate threadurile active, astfel încât la un moment dat un thread aşteaptă accesul planificat la resursele sistem CPU.

Paralelismul real pentru sisteme multiprocesor pe care se execută un program multithread depinde de modul în care procesoarele sunt alocate threadurilor. Pentru un thread metoda *start ()* se poate apela o singură dată. Se poate pune întrebarea de ce sunt necesare două metode de creare a threadurilor. Argumentele sunt următoarele: dacă programul necesită control asupra cicluluid eviaţă a threadului , extinderea clasei Thread constituie abordarea acorectă, iar dacă este necesară mai multă flexibilitate în exinderea altor clase se va prefera implementarea interfeței Runnable.

b. Transformarea threadului în Not Runnable

Threadul devine *Not Runnable* în situaţia în care apare unul din următoarele evenimente:

- este invocată metoda *sleep(),* astfel deşi procesorul este disponibil threadul este pus în aşteptare un anumit interval de timp, dacă threadul revine în starea Runnable şi procesorul redevine disponibil threadul îşi va reîncepe execuţia.
- threadul apelează metoda wait() pentru a aştepta satisfacerea unei anumite condiţii, iar pentru a deveni Runnable, un alt thread trebuie să notifice threadul în aşteptare despre o modificare a condiţiei apelând *notify () sau notifyAll().*
- threadul e blocat la o operaţie I/O şi devine Runnable la terminarea operaţiei.

c.Terminarea unui thread

JVM va executa un thread cât şi threadurile create de acesta până la întâlnirea uneia din situaţiile: apelul metodei *exit()* a clasei Runtime iar managerul de securitate a permis îndeplinirea cu succes a acestui apel, respectiv până s-au terminat toate threadurile ce nu sunt daemon, prin revenirea metodei *run(),* apelul metodei *stop()*, respectiv recepţia unei întreruperi *interrupt()* din partea altui thread. După cum am văzut, un fir de execuţie nu poate fi terminat forţat de către program ci trebuie să-şi programeze singur terminarea sa. Acest lucru poate fi realizat în două modalităţi :

1.**prin scrierea unor metode run care să-şi termine execuţia** în mod natural; la terminarea metodei run se va termina automat şi firul de execuţie, acesta intrând în starea Dead.

2.**prin folosirea unei variabile de terminare.** În cazul când metoda run trebuie să execute o buclă infinită atunci aceasta trebuie controlată şi printr-o variabilă care să oprească această buclă atunci când dorim ca firul de execuţie să se termine. Uzual, aceasta este o variabilă membră a clasei care descrie firul de execuţie care fie este publică, fie este asociată cu o metoda care îi schimbă valoarea.

Metoda run() nu se termină natural, ea rulează la infinit aşteptând să fie terminată forţat folosind de exemplu metoda stop. Această metodă este însă "învechită" (deprecated) iar la compilarea programului vom obţine un mesaj de avertizare în acest sens. Putem evita metoda stop prin folosirea unei variabile de terminare.

Metoda *isAlive* () este folosită pentru a vedea dacă un fir de execuţie a fost pornit şi nu s-a terminat încă. Metoda returnează:

- true - dacă firul este în una din stările Runnable sau Not Runnable
- false - dacă firul este în una din stările New Thread sau Dead

Între stările Runnable sau Not Runnable, respectiv New Thread sau Dead nu se poate face nici o diferenţiere.

Nu este necesară distrugerea explicită a unui fir de execuţie,sistemul se ocupa de acest lucru. El poate fi fortat să dezaloce resursele alocate unui thread prin atribuirea cu null a variabilei care referea instanţa firului de execuţie: myThread = null .

Aşteptarea terminării unui thread. Dacă este necesară aşteptarea terminării unui thread se poate utiliza metoda *join()* a obiectului thread după care se aşteaptă, funcţie ce are acelaşi efect ca şi combinaţia *sleep()* şi *isAlive().* Apelul this.join() referă o aşteptare perpetuă, astfel threadul curent nu se va termina. Funcţia *join()* pentru un thread ce nu şi/a început execuţia sau deja s-a terminat va returna fără să aştepte.

d.Întreruperea unui thread

Un thread se termină la revenirea din metoda run(). Funcţia interrupt () solicită unui thread abandonarea activităţilor sale (din starea ready). Dacă threadul este blocat *(în stările wait* sau *sleep*) nu se poate autoinspecta pentru întrerupere. Dacă threadul este în aşteptare, va genera o excepţie Interrupted Exception (flagul corespunzător nu este setat), iar dacă threadul este blocat prin sincronizare, nu va genera excepţie, însă flagul corespunzător va fi setat.

Preemptiv vs. non-preemptiv **.**Sistemul runtime Java nu va întrerupe threadul curent pentru un alt thread de aceeaşi prioritate, este posibil însă ca sistemul de operare să implementeze propria politică de preempţie (întrerupere) a threadurilor. Un thread se consideră că se manifestă corect dacă este capabil să elibereze periodic CPU altor threaduri. Dacă acesta nu se poate auto-suspenda prin aşteptare la o condiţie, prin apelul metodelor *sleep* sau invocarea unor operaţii de intrare-ieşire, este necesară invocarea metodei *yield().* Prin această metodă, threadul renunţă de bună voie la serviciile procesorului şi maşina virtuală Java este anunţată că poate preda controlul unui alt thread pe care planificatorul decide să îl treacă în starea ready. În orice moment este posibilă verificarea execuţiei unui thread folosind funcţia *isAlive().*

e. Planificarea threadurilor

Execuţia într-o anumită ordine a mai multor fire de execuţie pe un singur procesor se numeşte *planificare (engl. scheduling)*. Sistemul Java de execuţie a programelor implementează un algoritm simplu, determinist de planificare, cunoscut sub numele de *planificare cu prioritati fixate*. Fiecare fir de execuţie Java primeşte la crearea sa o anumită prioritate. O prioritate este de fapt un număr întreg cu valori cuprinse între MIN_PRIORITY şi MAX_PRIORITY. Implicit prioritatea unui fir de execuţie nou creat are valoarea NORM_PRIORITY. Aceste trei constante sunt definite în clasa Thread:

public static final int MIN_PRIORITY - prioritatea minimă
public static final int MAX_PRIORITY - prioritatea maximă
public static final int NORM_PRIORITY – prioritatea implicită

Schimbarea ulterioară a priorităţii unui fir de execuţie se realizează cu metoda setPriority() a clasei Thread. Planificatorul Java lucrează în modul următor: dacă la un moment dat sunt mai multe fire de execuţie în starea Runnable, adică sunt pregătite pentru a fi executate, planificatorul îl va alege pe cel cu prioritatea cea mai mare pentru a-l executa. Doar când firul de execuţie cu prioritate maximă se termină sau este suspendat din diverse motive va fi ales un fir de execuţie cu o prioritate mai mică. În cazul în care toate firele au aceeaşi prioritate ele sunt alese după un algoritm simplu de tip "round-robin". De asemenea, planificarea este complet *preemptivă*: dacă un fir cu prioritate mai mare decât firul care se execută la un moment dat solicită procesorul, atunci firul cu prioritate mai mare este imediat trecut în execuţie ,iar celălalt trecut în aşteptare.

Planificatorul Java nu va întrerupe un fir de execuţie în favoarea altuia de aceeaşi prioritate, însă acest lucru îl poate face sistemul de operare în cazul în care acesta alocă procesorul în cuante de timp. Aşadar, un fir de execuţie cedează procesorul în una din situaţiile:

- un alt fir de execuţie cu o prioritate mai mare solicită procesorul
- metoda sa run se termină
- vrea să facă explicit acest lucru apelând metoda yield
- timpul alocat pentru execuţia sa a expirat

În nici un caz corectitudinea unui program nu trebuie să se bazeze pe mecanismul de planificare a firelor de execuţie, deoarece acesta poate fi imprevizibil şi depinde de la un sistem de operare la altul. Un fir de execuţie de lungă durată şi care nu cedează explicit procesorul la anumite intervale de timp astfel încât să poată fi executate şi celelalte fire de execuţie se numeşte fir de execuţie egoist şi trebuie evitată utilizarea lor, întrucât acaparează pe termen nedefinit procesorul, blocând efectiv execuţia celorlalte fire de execuţie până la terminarea sa.

f. Grupuri de threaduri

Gruparea firelor de execuţie pune la dispoziţie un mecanism pentru manipularea acestora ca un tot şi nu individual,astfel putem să pornim sau să suspendăm toate firele dintr-un grup cu un singur apel de metodă. Gruparea firelor de execuţie se realizează prin intermediul clasei *ThreadGroup.*

Fiecare fir de execuţie Java este membru al unui grup, indiferent dacă specificăm explicit acest lucru. Afilierea unui fir de execuţie la un anumit grup se realizează la crearea sa şi devine permanentă, în sensul că nu vom putea muta un fir de execuţie dintr-un grup în altul, după ce acesta a fost creat. În cazul în care creăm un fir de execuţie fără a specifica în constructor din ce grup face parte, el va fi plasat automat în acelaşi grup cu firul de execuţie care l-a creat. La pornirea unui program se creează automat un obiect de tip ThreadGroup cu numele *main*, care va reprezenta grupul tuturor firelor de execuţie create direct din program şi care nu au fost ataşate explicit altui grup. Cu alte cuvinte, putem să ignoram complet plasarea firelor de execuţie în grupuri şi să lăsam sistemul să se ocupe cu aceasta.

Exista situaţii când programul creează multe fire de execuţie, iar gruparea lor poate simplifica substanţial manevrarea lor. Un grup se compune dintr-o mulţime de threaduri şi poate conţine alte grupuri, formând un arbore, un thread poate accesa doar informaţia referitoare la propriul grup fără a avea acces la alte grupuri sau la grupul părinte. Orice thread aparţine unui grup specificat ca un parametru constructor sau moştenit. Threadurile unui grup pot fi enumerate, pentru un grup poate fi setată prioritatea maximă respectiv grupul de threaduri poate fi întrerupt din execuţie. Grupurile de threaduri sunt folosite deoarece permit manipularea mai multor threaduri la un singur apel de funcţie şi oferă bazele mecanismului de securitate folosit pentru lucrul cu threaduri. În mod implicit toate threadurile create în program aparţin unui grup definit de maşina virtuală Java, însă din motive de securitate doar programele standalone pot opera cu grupuri de threaduri.

Unele dintre metodele de manipulare a grupurilor de threaduri sunt preluate de la clasa Thread, prin extinderea semnificaţiei pentru grupuri. Un constructor de thread, în forma cea mai generală ataşează noul thread unui grup de threaduri, dacă argumentul lipseşte atunci este ataşat grupului curent. Grupul unui thread este stabilit la crearea acestuia şi nu mai poate fi modificat pe parcursul execuţiei sale, la terminare threadul este automat eliminat din grup Pentru a afla cărui grup aparţine un anumit fir de execuţie putem folosi metoda getThreadGroup a clasei Thread. Un grup poate avea ca părinte un alt grup, ceea ce înseamnă că firele de execuţie pot fi plasate într-o ierarhie de grupuri, în care rădăcina este grupul implicit main.

g.Comunicarea prin fluxuri de tip "pipe"

O modalitate deosebit de utilă prin care două fire de execuţie pot comunica este realizată prin intermediul *canalelor de comunicatii (pipes)*. Acestea sunt implementate prin fluxuri descrise de clasele
PipedReader, PipedWriter - pentru caractere, respectiv
PipedOutputStream, PipedInputStream - pentru octeţi
Constructorii acestor clase sunt:

```
public PipedReader( )
public PipedReader( PipedWriter pw ) throws IOException  public PipedWriter( )
public PipedWriter(PipedReader pr) throws IOException
```

În cazul în care este folosit constructorul fără argument conectarea unui flux de intrare cu un flux de ieşire se face prin metoda **connect**:

```
public void connect(PipedWriter pw) throws IOException
public void connect(PipedReader pr) throws IOException
```

Întrucât ,fluxurile care sunt conectate printr-un pipe trebuie să execute simultan operaţii de scriere/citire folosirea lor se va face în cadrul unor fire de execuţie. Funcţionarea obiectelor care instanţiază PipedWriter şi PipedReader este asemănătoare cu a canalelor UNIX (pipes). Fiecare capăt al unui canal este utilizat dintr-un fir de execuţie separat. La un capăt al pipeline-ului se scriu caractere, la celalalt se citesc. La citire, dacă nu sunt date disponibile firul de execuţie se va bloca. Se observă că acesta este un comportament tipic producător-consumator, firele de execuţie comunicând printr-un canal. Realizarea conexiunii se face astfel:

```
PipedWriter pw1 = new PipedWriter();
PipedReader pr1 = new PipedReader(pw1);
sau
PipedReader pr2 = new PipedReader();
PipedWriter pw2 = new PipedWriter(pr2);
sau
PipedReader pr = new PipedReader();
PipedWriter pw = new PipedWriter();
pr.connect(pw) //echivalent cu
pw.connect(pr);
```

Scrierea şi citirea pe/de pe canale se realizează prin metodele uzuale read şi write în toate formele lor.

Exemplu de comunicare pipe:

Thread Scriitor

```
import java.io.*;
class FirScriitor extends Thread{private PipedOutputStream po;
FirScriitor(){po = new PipedOutputStream();}
public void run()
{try{while (true)
       {int d = (int)(10*Math.random());
           System.out.println("Fir scriitor trimite : "+d);
              po.write(d);
              sleep(400);}
       }catch(Exception e){}}
PipedOutputStream getPipe(){return po;}}
```

Thread Cititor

```
import java.io.*;
class FirCititor extends Thread
{private PipedInputStream pi;
FirCititor()
{pi = new PipedInputStream();     }
public void run()
{try
       {while (true)
       {if (pi.available()>0)
              {System.out.println("Fir cititor a primit : "+pi.read());}}
       }catch(Exception e){}}
void conect(PipedOutputStream os)throws Exception
{pi.connect(os);}}
```

Comunicarea Scriitor-Cititor

```
public class Test
{public static void main(String args[])
       {FirCititor fc = new FirCititor();
              FirScriitor fs = new FirScriitor();
              try
              {fc.conect(fs.getPipe());
                     fc.start();
                     fs.start();
              }catch(Exception e)
              {e.printStackTrace();}}}
```

1.2.4.Sincronizarea threadurilor

În secţiuena precedent s-a prezentatcum pot fi create fire de execuţie *independente şi asincrone,* care nu depind în nici un fel de execuţia sau de rezultatele altor fire de execuţie. Există însă numeroase situaţii când fire de execuţie separate, dar care rulează concurent, trebuie să comunice între ele pentru a accesa diferite resurse comune sau pentru a-şi transmite dinamic rezultatele "muncii" lor, operând astfel în manieră cooperativă.

Mecanismul monitor Java. Dacă mai multe threaduri operează simultan asupra unui obiect, datele acestuia pot fi corupte. Deoarece comutarea de context între threaduri poate surveni în orice moment, prevenirea accesului concurent este necesară chiar dacă sistemul este uniprocesor. Blocul de cod ce accesează acelaşi obiect din două threaduri separate este numit *secţiune critică*. Pentru a permite unui singur thread accesul la secţiunea critică, este necesar un mecanism de excludere mutuală. Excluderea mutuală presupune utilizarea unor algoritmi, implementarea de semafoare, respectiv monitoare. Mecanismul de sincronizare de bază în Java este *monitorul*. În Java orice obiect este un potenţial monitor având asociată o coadă de aşteptare şi mecanisme de semnalizare.

Monitorul încapsulează *datele* obiectului partajat, *procedurile de acces sincronizat* la aceste obiecte şi un *constructor* de iniţializare a obiectului monitor. Obiectele monitor aplică principiul excluderii mutuale pentru grupul de proceduri sincronizate, astfel *accesul sincronizat presupune serializarea threadurilor.*

Referitor la utilizarea metodelor-secţiunilor de program sincronizate cât şi pentru a evita condiţiile de tip "cursă" în accesul la date, asigurând astfel integritatea acestora se impun următoarele precizări:

- *secţiune critică* poate reprezenta o *metodă* sau *un bloc* de cod şi este identificată de cuvântul cheie synchronized. În primul caz mecanismul de sincronizare este asociat obiectului curent (monitorului său), iar în cel de al doilea se poate alege obiectul de sincronizare ce poate fi un obiect static sau cel curent, evitând alegerea unui obiect a cărui valoare este modificată în cadrul blocului
- Java asociază un lacăt (monitor) oricărui (any) obiect, acapararea şi eliberarea lacătului este realizată în mod *automat şi atomic* de către sistemul runtime la intrarea, respectiv ieşirea din blocul de cod declarat *sincronizat.*
- modificările efectuate într-o metodă sincronizată devin *vizibile* celorlalte metode sincronizate referitor la acelaşi obiect
- apelul unei metode sincronizate determină ca threadul ce a apelat metoda să plaseze un *lacăt* asupra obiectului a cărui metodă a fost apelată
- alte threaduri nu pot executa o metodă sincronizată asupra unui obiect pană la eliberarea lacătului, acestea vor fi blocate.
- atunci când threadul ce deţine lacătul va reveni din metoda sincronizată lacătul obiectului va fi în mod automat eliberat
- unul din threadurile ce aşteaptă acapararea lacătului îl va prelua şi va apela metoda sincronizată.
- citirile şi scrierile unor câmpuri volatile sunt atomice şi vizibile altor câmpuri

Exemplu de declarare a metodelor sincronizate pentru un cont bancar:

```
class Account {
private double balance;
public Account(double initialDeposit) { balance = initialDeposit;   }
public synchronized double getBalance() {  return balance;   }
public synchronized void deposit(double amount) { balance += amount;   }  }
```

- declararea metodei ca synchronized presupune ca lacătul obiectului curent (this) să fie acaparat înaintea execuţiei metodei
- mecanismul de sincronizare a blocurilor de cod, permite specificarea explicită a lacătului unui anumit obiect ce trebuie acaparat de un thread înaintea execuţiei blocului respectiv (poate fi un obiect Java de tip any, sau este posibil să fie un obiect ce nu este utilizat în cadrul blocului sincronizat)
- creşterea gradului de paralelism poate fi implementată prin declararea unui întreg bloc de cod ca synchronized.

Mecanismul wait-notify. Un thread ce deţine exclusivitatea asupra unui obiect de tip monitor poate ceda temporar exclusivitatea accesului. Apelul metodei *wait(),* are drept rezultat pentru threadul curent oprirea temporară a execuţiei şi trecerea în starea *waiting(),* iar reluarea accesului exclusiv are loc atunci când threadul se trezeşte ca urmare a unui mesaj de anunţ generat de un alt thread activ ce va executa una din metodele *notify()* sau *notifyAll().* În urma acestui eveniment threadul recapătă exclusivitatea cedată şi continuă execuţia oprită anterior după apelul *wait().* Un obiect posedă două cozi asociate şi anume :

- o coadă pentru *excluderea prin sincronizare* a altor threaduri
- o coadă pentru *aşteptare la condiţie*, însă se poate afla la un moment dat doar într-una din cele două cozi.

Anunţul (notificarea) efectuat cu metoda *notify()* este adresat tuturor threadurilor aflate în starea Waiting, dar unul singur îşi va relua activitatea, iar pentru un anunţ efectuat cu *notifyAll()* toate threadurile în aşteptare îşi vor relua activitatea. Este posibil ca un thread să solicite intrarea sa în starea Waiting pentru un interval de timp bine delimitat deoarece metoda *wait ()* permite specificarea duratei maxime de aşteptare. Dacă un alt thread apelează metoda *interrupt()* a threadului în aşteptare, execuţia acestuia este reluată cu secvenţa *catch* ce interceptează întreruperea.

Cele două metode *wait()- notify()* trebuie apelate *în interiorul unei metode sau bloc* ***sincronizat.*** Tehnica integrării acestui mecanism cu metodele de sincronizare este similară metodelor de sincronizare de nivel jos, caz în care variabilele condiţionale au asociate variabile tip mutex pe care le blochează-deblochează funcţie de faza aşteptării. Astfel în Java înaintea intrării în aşteptare prin wait(),threadul curent eliberează obiectul de sincronizare blocat şi îl obţine din nou când threadul este anunţat că a fost realizată condiţia aşteptată. Pentru o utilizare corectă a acestui mecanism sunt necesare următoarele precizări :

- dacă *notify()*, notifyAll() este apelată dintr-un thread fără a exista un thread care să aştepte, se va reveni fără eroare
- metoda *wait ()* eliberează obiectul de sincronizare la intrarea în aşteptare şi îl reobţine la recepţionarea notificării
- dacă există mai multe threaduri ce aşteaptă o anumită condiţie nu se poate şti care thread va primi notificarea, maşina virtuală Java şi mecanismul de planificare a threadurilor determină ordinea precisă
- dacă se doreşte apelul funcţiilor wait()-notify() din metode statice este necesară declararea în cadrul clasei a unui obiect de sincronizare static pentru care vor fi apelate aceste funcţii
- diferenţa între metodele wait() şi sleep() constă în aceea că metoda sleep *nu necesită apelul obligatoriu dintr-un bloc sincronizat* şi nu implică blocarea sau deblocarea unui obiect de sincronizare.
- *NotifyAll* nu eliberează lacătul
- nici un thread nou nu poate intra în coada wait asociată pe durata cât threadul notificator deţine încă lacătul. Funcţia va transfera toate threadurile aflate în aşteptare din coada de aşteptare în coada de blocare aşteptând momentul în care pot continua.

Evitarea metodelor stop() şi suspend(). Metodele *stop, resume şi suspend()* lasă obiectele în stări inconsistente. Un thread suspendat forţat poate deţine infinit un lacăt, iar dacă va apela ulterior resume() va genera blocaj datorat aceluiaşi lacăt. Threadul trebuie să verifice dacă a fost intrerupt. Versiuni sigure pentru operaţiile suspend() şi resume() pot fi implementate bazat pe operaţiile *wait() şi notifyAll(),* astfel threadul va verifica cererea de suspendare şi va controla momentul în care îşi poate întrerupe în siguranţă activităţile.

Detectarea şi evitarea blocării în programele concurente este realizată prin proiectarea corectă a algoritmilor. Blocajele sunt generate de dependenţele circulare între threaduri şi lacătele pe care acestea încearcă să le acapareze. O definire simplă a blocajului este următoarea: un thread deţine un lacăt (resursă) necesar unui alt thread, el la rândul său încercând să obţină un lacăt deţinut de un alt thread.Evitarea blocajului este dificilă deoarece presupune interacţiuni corecte între obiecte multiple cooperante.

Următorul set de reguli permit evitarea blocajelor în secvenţele de cod ce necesită concurenţă:

- evitarea apelurilor blocante (operaţii de intrare-ieşire) în cadrul unui bloc sincronizat
- evitarea apelurilor imbricate sincronizate
- structurarea liniară a fluxului de control prin aplicarea unor strategii şi/sau euristici între care reordonarea resurselor, creşterea granularităţii (obţinerea prealabilă a lacătului exterior)
- utilizarea unor mecanisme de *tip timeout –retry* (poate genera blocaje de tip livelock)
- implementarea unor mecanisme de tip *transfer de jeton* ce permit evitarea unor sincronizări dinamice a accesului la obiectul resursă
- *planificare explicită* a threadurilor
- modificări de semantică a cerinţelor de atomicitate totală (ex. propagarea modificărilor şi actualizarea datelor nu necesită tratare atomică)

1.3. Exemple

Se vor testa exemplele ce ilustrează modul de utilizare a API-ului Java multithreading pentru următoarele mecanisme specifice : utilizare diverse metode ale clasei thread, blocuri sincronizate, prioritizare și planificare threaduri, exemple pe care le puteți descărca de la adresa *https:/ftp.utcluj.ro /~civan/CPD/1_LABORATOR/01_JavaMT*

1.4. Întrebări teoretice

1.4.1. Definiți conceptul de thread. Explicați succint diferențe conceptuale și de utilizare între proces și thread.

1.4.2. Explicați conceptul de comutare de context.

1.4.3. Care sunt avantajele multitaskingului realizat prin intermediul threadurilor vs al proceselor.

1.4.4. Descrieți ciclul de viață al unui thread, identificând stările și tranzițiile dintre acestea.

1.4.5. Explicați conceptul de prioritate prin intermediul unui exemplu.Cum se asigură prioritizarea threadurilor ?

1.4.6. Descrieți modalitățile prin care pot fi create threaduri și identificați cazurile de aplicare.

1.5. Probleme propuse

1.5.1. Să se implementeze problema *producător – consumator* utilizând două mecanisme de sincronizare diferite. O primă soluţie poate fi considerată utilizarea unei cozi intermediare, se cunoaşte faptul că producătorul necesită 1000 ms pentru a produce o valoare, respectiv consumatorul necesită 2000 ms pentru a o consuma.

1.5.2. Să se implementeze folosind facilităţile de sincronizare specifice limbajului Java problema accesului concurent a scriitorilor şi cititorilor la un fisier. Realizaţi o soluţie ce oferă acces *concurent* pentru citire şi exclusiv pentru scriere.

1.5.3. Intr-un birou sunt 8 functionari care din când în când tipăresc la imprimantă documente, nu toți elaborează documentele în același ritm. Fiindcă au o singură imprimantă în birou, poate tipări doar o singura persoană la un moment dat. Să se simuleze functionarea biroului.

1.5.4. Scrieţi un program Java care generează două fire de execuţie pentru parcurgerea unui String de la cele două capete. Folosiţi doi pointeri a căror valoare se incrementează, respectiv se decrementează într-o funcţie din memoria comună (în memoria comună se află String-ul.)

1.6.Referințe bibliografice

1. Tutorial Oracle https://docs.oracle.com/javase/tutorial/essential/concurrency/procthread.html
2. Tutorial cu exemple https://www.tutorialspoint.com/java/java_multithreading.htm
3. Tuli S, Multithreading in Java https://dzone.com/articles/java-thread-tutorial-creating-threads-and-multithr
4. Buyya R., http://www.buyya.com/java/Chapter14.pdf