Procesai-gijos

Java gijos

Java – lygiagrečiojo programavimo kalba

- Gijos įtrauktos į bazinę Java kalbos specifikaciją.
- Gija – lengvasvoris procesas, kurio viduje veiksmai vykdomi nuosekliai.
- Gijų naudojimas sudaro galimybę tuo pačiu metu (lygiagrečiai) vykdyti kelias programos užduotis.
- Programoje visada yra bent viena gija: `main` funkcija – taip pat gija.
- Kiekviena Java virtualioji mašina (JVM) privalo palaikyti gijas.

Gijų kūrimo būdai

1. Panaudojant (paveldint) `java.lang.Thread` klasę.
2. Panaudojant (realizuojant) `java.lang.Runnable` sąsają.

java.lang.Thread klasė

```
public class Thread extends Object
                    implements Runnable {
  public Thread();
  public Thread(String name);
  public Thread(Runnable target);
  public Thread(Runnable target, String name);
  public Thread(Runnable target, String name,
                long stackSize);

  public void start();
  public void run();
  ...
}
```

Pagrindiniai gijų klasių metodai

- **`public void start()`**

  Causes this thread to begin execution; the Java Virtual Machine calls the `run` method of this thread.

  The result is that two threads are running concurrently: the current thread (which returns from the call to the `start` method) and the other thread (which executes its `run` method).

- **`public void run()`**

  If this thread was constructed using a separate `Runnable` run object, then that `Runnable` object's `run` method is called; otherwise, this method does nothing and returns.

  Subclasses of `Thread` should override this method.

Gijos sukūrimas ir vykdymas (1)

```
class Gija extends Thread
{
   . . .
   public void run() {
      . . .
   }
}
   . . .
Gija g = new Gija();
g.start();
```

Gijos sukūrimas ir vykdymas (2)

```
class Gija implements Runnable
{
   . . .
   public void run() {
      . . .
   }
}
   . . .
Thread g = new Thread(new Gija());
g.start();
```

Gijų vykdymas

- Programa lygiagrečiai (*concurrent*) gali vykdyti bet kokį skaičių gijų.
- Gijos vykdymas pradedamas, iškvietus gijos-objekto metodą `start`.
- Gijos vykdymo metu nuosekliai atliekami veiksmai, užrašyti gijos klasės metode `run`.
- Visos programos gijos gali atlikti skirtingus veiksmus (jei jos yra skirtingų gijų klasių objektai).
- Jei programuotojas netaiko papildomų priemonių, gijos-objekto vykdymas nepriklauso nuo kitų gijų vykdymo.

Atsiminkite:

- Programose `run` metodas neturėtų būti kviečiamas tiesiogiai. Sistema iškvies jį pati.
- Jei `run` metodas kviečiamas tiesiogiai, programos kodas vykdomas ne lygiagrečiai, bet nuosekliai.

Naudingi gijų metodai

- `public final boolean isAlive()`
  Tests if this thread is alive. A thread is alive if it has been started and has not yet died. Returns: `true` if this thread is alive; `false` otherwise.
- `public final void join()`
  `throws InterruptedException`
  Waits for this thread to die.

Pirmoji programa[1]. Gijų klasė

```
class Gija extends Thread {
  private String vardas;
  public Gija(String vardas) {
    this.vardas = vardas;
  }
  public void run() {
    System.out.println(vardas + ": vienas");
    System.out.println(vardas + ": du");
    System.out.println(vardas + ": trys");
  }
}
```

[1] čia ir toliau pateikiamos ne pilnos programos, bet tik programų fragmentai

Pirmoji programa. Pagrindinė klasė

```
class Pirma {
  public static void vykdytiGijas() {
    Gija g1 = new Gija("Pirma");
    Gija g2 = new Gija("Antra");
    g1.start(); g2.start();
  }
  public static void main(String[] args)
                          throws IOException {
    vykdytiGijas();
    System.out.println(" Programa baigė darbą");
  }
}
```

Pirmoji programa. Galimi rezultatai

```
Pirma: vienas
 Programa baigė darbą
Antra: vienas
Antra: du
Pirma: du
Antra: trys
Pirma: trys
```

Pirmoji programa. Papildyta pagrindinė klasė

```
class Pirma {
  public static void vykdytiGijas() {
    Gija g1 = new Gija("Pirma");
    Gija g2 = new Gija("Antra");
    g1.start(); g2.start();
    while (g1.isAlive()||g2.isAlive()) { }
  }
  public static void main(String[] args)
                          throws IOException {
    vykdytiGijas();
    System.out.println(" Programa baigė darbą");
  }
}
```

Papildyta programa. Galimi rezultatai

```
Pirma: vienas
Antra: vienas
Antra: du
Antra: trys
Pirma: du
Pirma: trys
 Programa baigė darbą
```

C++11 gijos

C++11 gijų klasė `std::thread`

- Skirta kurti objektus, kurių veiksmai gali būti vykdomi lygiagrečiai su kitų gijų veiksmais.
- Gijos naudojasi ta pačia adresų erdve.
- Kiekviena gija turi savo unikalų `id`.
- `main()` funkcija vykdoma atskiroje gijoje.

Reikalingi metodai

- konstruktorius su parametrais:
  `thread(fn, args)`, kur
  `fn` – gijoje vykdoma funkcija,
  `args` – funkcijai perduodamų argumentų sąrašas;
- gijos darbo pabaigos laukimo metodas:
  `join()`

Elementari programa (1)

```
void vykdyti(string vardas){
    cout << vardas << ": vienas\n";
    cout << vardas << ": du\n";
    cout << vardas << ": trys\n";
}
```

Elementari programa (2)

```
#include <thread>
. . .
int main(){
    thread g1(vykdyti, "Pirma");
    thread g2(vykdyti, "Antra");
    g1.join();
    g2.join();
    cout << " Programa baigė darbą";
}
```

Elementari programa. Galimi rezultatai

```
Pirma: vienas
Antra: vienas
Pirma: du
Antra: du
Pirma: trys
Antra: trys
 Programa baigė darbą
```

OpenMP gijos

OpenMP – kas tai?

- OpenMP – tai papildomos priemonės, suteikiančios galimybes kurti lygiagrečias programas Fortran, C, C++ kalbomis;
- OpenMP skirta kurti lygiagrečias programas, kuriose procesai (gijos) naudojasi **bendra** atmintimi;
- OpenMP sudaro: kompiliatoriaus direktyvos, specialių funkcijų biblioteka, aplinkos kintamųjų rinkinys.

OpenMP gijų rinkinys

- Nuo programos darbo pradžios iki pabaigos vykdoma pagrindinė gija (*master thread*).
- Naujos gijos kuriamos kompiliatoriaus direktyvomis.
- Vienu metu vykdomų gijų rinkinys sudaro lygiagrečią sritį (*parallel region*).

OpenMP C++ programos struktūra

```
#include <omp.h>
...
main(int argc, char **argv) {
   ...  // Nuoseklus kodas
   #pragma omp parallel
   {
      ...  // Lygiagretus kodas
   }
   ...  // Nuoseklus kodas
   return 0;
}
```

Lygiagreti sritis

```
#pragma omp parallel [Atrib. sakinys]
  Struktūrinis blokas
```

// Struktūrinis blokas – iš naujos eilutės

Gijų vykdymas

- Lygiagrečioje srityje vykdomų gijų skaičių galima nurodyti programos vykdymo metu.
- **Visos lygiagrečios srities gijos vykdo tą patį struktūrinį bloką**.
- Visos lygiagrečios srities gijos pradedamos vykdyti tuo pačiu metu ir vykdomos lygiagrečiai.
- Lygiagreti sritis baigiama vykdyti, kai baigiamos vykdyti visos tos srities gijos.
- Kiekvienai lygiagrečios srities gijai suteikiamas unikalus numeris; *master* gijos numeris – 0.
- Programos vykdymo metu nėra galimybių keisti gijos numerį.

Gijų vykdymo valdymas

- Lygiagrečioje srityje vykdomų gijų skaičius nurodomas `pragma omp parallel` atributu `num_threads()` arba prieš lyg. sritį funkcijos `omp_set_num_threads()` parametru.
- Vykdomų gijų skaičių grąžina funkcija `omp_get_num_threads()`.
- Gijos numerį grąžina funkcija `omp_get_thread_num()`.

Elementari programa (1)

```
void vykdyti(string vardas){
    cout << vardas << ": vienas\n";
    cout << vardas << ": du\n";
    cout << vardas << ": trys\n";
}
```

Elementari programa (2)

```
int main(){
    int gijuSk = 2, gijosNr   = 0;
    omp_set_num_threads(gijuSk);
    // ------ Lygiagretus kodas --------
    #pragma omp parallel private(gijosNr)
    {
        gijosNr = omp_get_thread_num();
        if (gijosNr == 0)
            vykdyti("Pirma");
        else
            vykdyti("Antra");
    }
    cout << " Programa baigė darbą";
}
```

Elementari programa. Galimi rezultatai

```
PirmaAntra: vienas
Pirma: du
Pirma: trys
: vienas
Antra: du
Antra: trys
 Programa baigė darbą
```

CUDA gijos

CPU ir GPU gijos

- CPU gijos vykdomos pagrindiniame procesoriuje, GPU (grafinio procesoriaus) gijos vykdomos grafinėje plokštėje;
- CPU efektyviai gali būti vykdoma tik nedidelis kiekis gijų, o GPU - tūkstančiai;
- sąvokos: *host* = CPU, *device* = GPU, *kernel* – funkcija, kuri vykdoma GPU;
- daug GPU gijų tuo pačiu metu gali vykdyti tą pačią *kernel*'io funkciją.

Elementari programa (1)

```
#include "cuda_runtime.h"
#include <cuda.h>
#include "device_launch_parameters.h"
. . .
int main(int argc, char** argv)
{
    vykdytiGPUGijoje<<<1, 2>>>();  // dvi gijos
    cudaDeviceSynchronize();

    cout << " Programa baigė darbą\n";
}
```

Elementari programa (2)

Gijos funkcija, kviečiama iš CPU, vykdoma GPU. Pradedama raktažodžiu `__global__`:

```
__global__ void vykdytiGPUGijoje() {
    if (threadIdx.x == 0)
        vykdyti("Pirma");
    else
        vykdyti("Antra");
}
```

Elementari programa (3)

Funkcija, kviečiama iš GPU, vykdoma GPU. Pradedama raktažodžiu `__device__`:

```
__device__ void vykdyti(char vardas[]) {
    printf("%s: vienas\n", vardas);
    printf("%s: du\n", vardas);
    printf("%s: trys\n", vardas);
}
```

Elementari programa. Galimi rezultatai

```
Antra: vienas
Antra: du
Antra: trys
Pirma: vienas
Pirma: du
Pirma: trys
 Programa baigė darbą
```

Tipiniai programos veiksmai:

1. GPU atminties skyrimas (`cudaMalloc`);
2. duomenų kopijavimas iš CPU į GPU (`cudaMemcpy`);
3. lygiagretus GPU gijų vykdymas (`<<< >>>`);
4. duomenų kopijavimas iš GPU į CPU (`cudaMemcpy`);
5. GPU atminties atlaisvinimas (`cudaFree`).

GPU atminties skyrimas ir atlaisvinimas

`cudaError_t` **`cudaMalloc`**`(void **devPtr, size_t size)`
Skiria GPU atmintį.
`devPtr` - pradžios rodyklė, `size` - dydis baitais.

`cudaError_t` **`cudaFree`**`(void **devPtr)`
Atlaisvina GPU atmintį.
`devPtr` - pradžios rodyklė.

Duomenų kopijavimas iš CPU į GPU ir iš GPU į CPU

`cudaError_t` **`cudaMemcpy`**`(void *dst, const void *src, size_t count, enum cudaMemcpyKind kind)`
Kopijuoja iš `src` į `dst`.
`kind` - viena iš reikšmių: cudaMemcpyHostToHost, cudaMemcpyHostToDevice, cudaMemcpyDeviceToHost, arba cudaMemcpyDeviceToDevice.

2 pvz. Vektorių sudėtis (1)

```
int a[N], b[N], c[N];
int *dev_a, *dev_b, *dev_c;
. . .
int dydis = N * sizeof(int);
cudaMalloc((void**)&dev_a, dydis);
cudaMalloc((void**)&dev_b, dydis);
cudaMalloc((void**)&dev_c, dydis);
```

2 pvz. Vektorių sudėtis (2)

```
cudaMemcpy(dev_a,a,dydis,cudaMemcpyHostToDevice);
cudaMemcpy(dev_b,b,dydis,cudaMemcpyHostToDevice);

add<<<1, N>>>(dev_a, dev_b, dev_c);

cudaMemcpy(c,dev_c,dydis,cudaMemcpyDeviceToHost);

cudaFree(dev_a);
cudaFree(dev_b);
cudaFree(dev_c);
```

2 pvz. Vektorių sudėtis (3)

```
__global__ void add(int *a, int *b, int *c ) {
    int tid = threadIdx.x;
    if (tid < N)
       c[tid] = a[tid] + b[tid];
}
```

Klausimai pakartojimui

1. Kaip kuriamos Java (C++11, OpenMP, CUDA) gijos?
2. Kada pradedamos vykdyti Java (C++11, OpenMP, CUDA) gijos?
3. Kaip vykdomi veiksmai, užrašyti tos pačios Java (C++11, OpenMP, CUDA) gijos viduje?
4. Kaip vykdomi veiksmai, užrašyti skirtingų Java (C++11, OpenMP, CUDA) gijų viduje?
5. Kaip pakeisti gijų skaičių Java (C++11, OpenMP, CUDA) programoje?
6. Reikia, kad tam tikri veiksmai būtų atlieka po to, kai visos gijos baigs darbą. Kaip tai padaryti Java (C++11, OpenMP, CUDA) programoje?
7. Palyginkite tarpusavyje Java, C++11, OpenMP, CUDA gijas: privalumai ir trūkumai?