HATÉKONY JAVA PROGRAMOZÁS

5. forduló

MSCI

A kategória támogatója: MSCI

RENDELKEZÉSRE ÁLLÓ IDŐ:

10:00

Ismertető a feladathoz

Felhasznált idő: 02:08/10:00 **Elért pontszám:** 0/15

1. feladat 0/3 pont

Az alábbi algoritmus futási idejét mértük egyforma hosszú, azonos számokat tartalmazó tömbökre, csak a számok sorrendje különbözött. Azt tapasztaltuk, hogy akkor fut le a leggyorsabban, ha rendezett a tömb. Mi lehet ennek az oka?

```
public long testAlgorithm() {
    long sum = 0;
    for (int i : input) {
        if (i > 0) {
            sum += i;
        }
    }
    return sum;
}
```

Válasz

- ○ rendezett tömb esetén a sum változó hamarabb túlcsordulhatott, így kisebb számokra gyorsabb lett a számítás
- ○ nagyobb hatékonysággal működik a processzor elágazásbecslése (branch prediction)
- ○ ez nem fordulhat elő, a mérés hibás (pl. más párhuzamos számítás lassíthatta vagy másban is különbözött a bemenet)

- ○ a tömörítés miatt a rendezett tömbök kevesebb memóriát foglalnak, így hatékonyabb az ezeken végzett számítás

Magyarázat

A processzor elágazásbecslése akkor működik hatékonyan, ha az elágazások kimenetele megjósolható (pl. rendezett tömb esetén). Amikor az elágazásbecslés téved, akkor tovább tart a számítás.

2. feladat 0/3 pont

Adott az alábbi egyszerűsített algoritmus alábbi két megvalósítása (test1 és test2 metódusok). Feltételezve, hogy a Java 11 JIT fordító elvégez minden lehetséges optimalizációt, mit mondhatunk el a test1 és test2 metódusok automatikus szemétgyűjtésre (GC) való hatásáról?

```
private static class Calculator {
    private int a;
    private int b;

    public Calculator(int a, int b) {
        this.a = a;
        this.b = b;
    }

    public int calculate() {
        return a + b;
    }
}

public int test1(int a, int b) {
    return new Calculator(a, b).calculate();
}

public int test2(int a, int b) {
    return a + b;
}
```

Válasz

- ○ a test1 metódus allokál rövid életű Calculator objektumokat a heap memóriaterületen, így jobban terheli a szemétgyűjtőt mint a test2 metódus
- ○ a test2 metódus is létrehoz a Calculator osztályhoz nagyon hasonló rejtett objektumokat, így szemétgyűjtést tekintve nem lesz különbség a kettő futása között
- ◯ a test1 metódus Calculator objetumai a stacken jönnek létre, így szemétgyűjtést tekintve nem lesz különbség a kettő futása között
- ○ mivel a test1 csak rövid életű objektumokat hoz létre, érdemes lenne programozottan felszabadítani (finalize hívással), mert akkor nem terhelné az automatikus szemétgyűjtőt

Magyarázat

Az Escape analysis optimalizáció miatt ezek a rövid életű objektumok a stacken jönnek létre és azonnal felszabadulnak.

3. feladat 0/3 pont

Java alapú szerveralkalmazásunk működése során azt tapasztaljuk, hogy a futtatás első órájában lényegesen lassabban sikerül kiszolgálni a kéréseket mint a későbbiekben. Mi lehet ennek az oka?

Válasz

- ○ Java 11-ben vezették be a mesterséges intelligencia alapú szemétgyűjtő algoritmust, melynek időre van szüksége ahhoz, hogy betanuljon és hatékonyabban működjön
- ◯ A JIT fordító csak a gyakran használt kódrészleteket fordítja hatékony kódra, így időre van szüksége a statisztika gyűjtésére
- ○ Linux alatt ez minden alkalmazás esetében megfigyelhető tendencia, melyet egy jól ismert kernel bug okozott. Már elérhető a javítás.
- ○ Az első órában még minden bizonnyal nem kell működnie a GC algoritmusának, mely a futása során optimalizálja a futtatott kódot

Magyarázat

Ezt a jelenséget a JIT compiler okozza, melynek időre van szüksége a hatékony fordításhoz.

4. feladat 0/3 pont

Szövegek összefűzésére adott az alábbi két megvalósítás. Melyik a hatékonyabb implementáció és miért?

```
public String concat1(String a, String b) {
    return a + b;
}

public String concat2(String a, String b) {
    return new StringBuilder().append(a).append(b).toString();
}
```

Válasz

- ◯ a két implementáció megegyező futási teljesítményt fog nyújtani, így a concat1 az olvashatósága miatt hatékonyabb

- a concat2 jobb futási teljesítményt fog nyújtani, így azt érdemes használni még akkor is, ha kevésbé olvasható a kódja
- a concat2 jobb futási teljesítményt fog nyújtani, de a különbség annyira elenyésző, hogy a concat1 megvalósítását érdemes használni, mivel az jobban olvasható
- a concat2 jobb futási teljesítményt fog nyújtani, de csak addig, amíg a JIT compiler nem végzi el az optimalizációt, így a concat1 az olvashatósága miatt hatékonyabb

Magyarázat

A helyes válasz megadja a magyarázatot. A megegyező futási teljesítményt az adja, hogy a két kódrészlet ugyanazt a bytecode fordítást fogja eredményezni.

5. feladat 0/3 pont

Milyen esetekben választottunk jól GC algoritmust Java 17 alatt futó alkalmazásunkhoz?

Válaszok

- [x] egy processzorral és 1GB memóriával rendelkező virtuális gépen batch folyamatok futtatása: -XX:+UseSerialGC
- [ ] egy processzorral és 1GB memóriával rendelkező virtuális gépen batch folyamatok futtatása: -XX:+UseG1GC
- [ ] konzisztensen alacsony számítási időre kiemelten érzékeny tőzsdei alkalmazás 8 processzorral és 32GB memóriával: -XX:+UseConcMarkSweepGC
- [x] konzisztensen alacsony számítási időre kiemelten érzékeny tőzsdei alkalmazás 8 processzorral és 32GB memóriával: -XX:+UseZGC
- [ ] átlagos webalkalmazás 2 processzorral és 4GB memóriával: -XX:+UseConcMarkSweepGC
- [ ] átlagos webalkalmazás 2 processzorral és 4GB memóriával: -XX:+UseEpsilonGC

Magyarázat

Egy processzoron batch műveletekhez a SerialGC a leghatékonyabb.

A ConcMarkSweepGC Java 17 alatt már nem működik.

Az EpsilonGC nem alkalmas webalkalmazásokhoz, mert hamar meg fog telni a memória.

Alacsony válaszidőhöz és nagy memóriaterülethez a ZGC a leghatékonyabb a beépített szemétgyűjtők közül.

KÉSZÍTETTE

Megjelenés

Világos