Univerzitet Singidunum

Race Condition

www.singidunum.ac.rs

MSc Miloš Mravik

Primer

- **Primer:**
- Dodeljivanje vrednosti postojećoj varijabli

- x = x + 1

- Koraci u izvršavanju zadataka:

1. Čitanje vrednosti varijable x (desna strana)
2. Računanje sume i dodavanje vrednosti 1
3. Upisivanje vrednosti unutar varijable x (leva strana)

Primer

- Ako se ova izjava dva puta (uzastopno) izvršava u jednoj niti, konačna vrednost promenljive x je uvek za 2 veća od početne vrednosti

- Ako se ova izjava izvršava dva puta, jednom u svakom od dva paralelna niza koji dele promenljivu x, konačna vrednost promenljivex može biti za 1 ili 2 veća od početne vrednosti

Primer

- Scenario za neočekivan rezultat:

1. Prva nit je započela izvršavanje, ali samo prva dva koraka od tri su učinjena pre nego što je nit prekinuta
2. Druga nit je započela izvršavanje i svi koraci njenog izvršenja su završeni
3. Prva nit nastavlja izvršavanje trećeg koraka izjave o dodeli

Primer

- Dolazimo do sledećeg scenarija:

1. S obzirom na to da prva nit nije u potpunosti upisala svoju vrednost broj x, druga nit će da pročita staru vrednost broja x, povećaće je za 1 i takvu će je upisati u memoriju
2. Kada prva nit nastavi da se izvršava stara vrednost broj x koja je povećana za vrednost 1 će ponovo biti učitana kao nova vrednost promenljive x
3. На крају ће се почетна вредност x повећати за 1, а не за 2!

Race Condition problem

- **Primer:**
- Deljeni brojač sa početnom vrednošću 0 i 10 niti (zadataka) svaki povećavajući početnu vrednost brojača za 1
- RaceCondition.java

Race Condition problem

- Rešenje Race Condition problema predstavlja sinhronizaciju niti
- Sinhronizovani metodi ne mogu biti ometeni u toku njihovog izvršavanja

```
class SynchronizedCounter {

  private int counter = 0;

  public synchronized void add1() {
        counter = counter + 1;
  }

  public synchronized int value() {
        return counter;
  }
}
```

Race Condition problem

- Sinhronizacija niti ne rešava u potpunosti problem
  - Dugo sinhronizovani metodi mogu monopolisati programsko izvršavanje
- Potrebna je bolja kontrola sinhronizacije niti, tako da je potreban samo (manji) blok koji ne može da se prekine

Race Condition problem

- **Primer:**

```
if (obj.value() != 0)
  aMethod();
```

- Pri čemu metod aMethod() tačno izračunava svoj rezultat samo ako vrednost objekta nije 0
- Druga nit zasebno postavlja vrednost objekta obj

Race Condition problem

- **Primer:**

```
if (obj.value() != 0)
  aMethod();
```

- Race Condition se može dogoditi ako druga nit postavi nultu vrednost za obj odmah nakon što prva nit izračuna uslov u slučaju da je uslov tačan

Race Condition problem

- Prva nit treba ekskluzivni pristup objektu obj tokom čitavog izvršavanja metoda:

```
synchronized (obj) {
  if (obj.value() != 0)
    aMethod();
}
```

- If uslov sinhronizuje obj objekat
- Sinhronizovani metodi su sinhronizovani nad this objektom

Race Condition problem

- Sintaksa pozivanja sinhronizovanog uslova:

```
synchronized (obj) {
  .
  . // Uslov
  .
}
```

- Kritički region – predstavlja region koda koji ne bi smeo da se prekine u toku svog izvršavanja

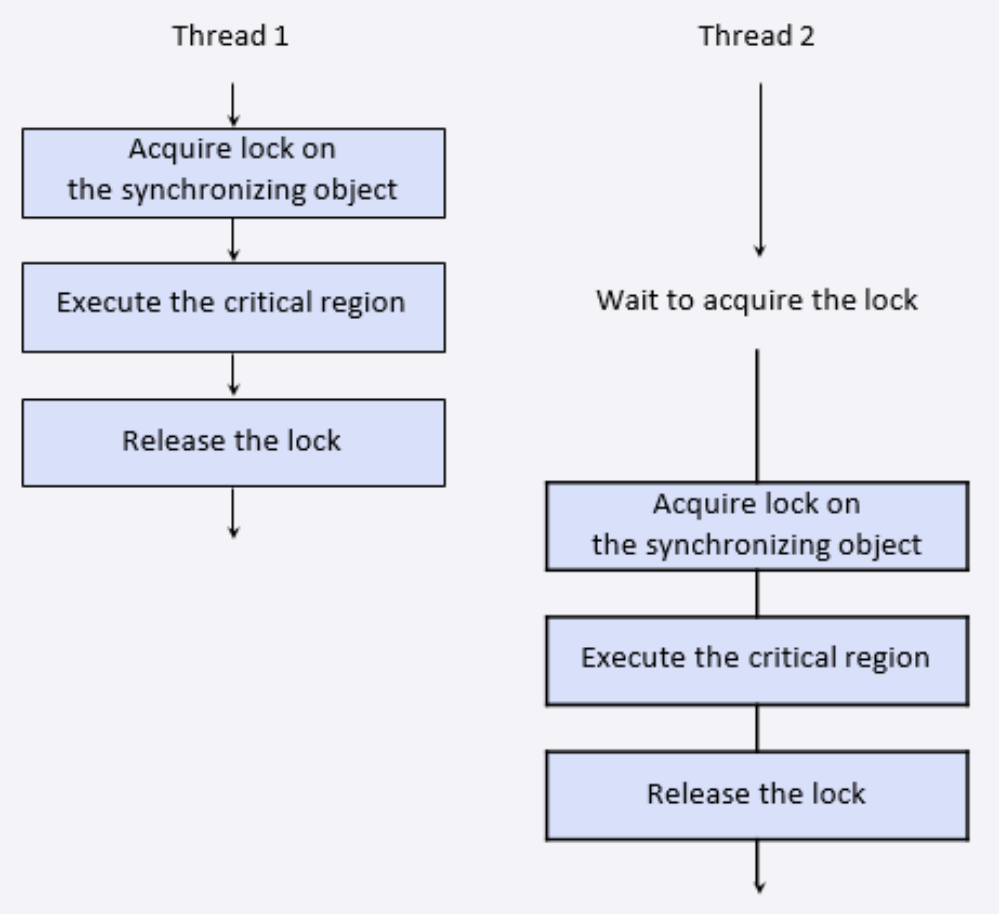
Race Condition problem

- Dve niti ne mogu istovremeno da izvršavaju kritične regione koji su sinhronizovani nad istim objektom
- **Lock** – mehanizam koji daje ekskluzivno pravo niti nad nekim resursom
- Lock mehanizam se koristi za sigurno izvršavanje niti u u nekom programu

Race Condition problem

- Ako nit izvršava sinhronizovan metod nad nekim objektom, prvo se zaključava objekat(**lock**), zatim se izvršava kritična oblast, a nakon toga se oslobađa objekat (**unlock**)
- Ukoliko je neki objekat zaključan od strane neke niti ostale niti moraju da čekaju dok se kritična oblast ne izvrši i dok se objekat ne otključa

Race Condition problem



Deadlock

- Sinhronizacija rešava Race Condition problem
- Sinhronizacija dovodi do novog problema – **deadlock**
- Deadlock se dešava kada se jedna nit sinhronizuje na više objekata

Deadlock

- **Primeri:**

1. Dve niti zaključaju dva objekta
2. Dve niti čekaju otključavanje jedna druge da bi oslobodile resurse nad različitim objektima i nastavile izvršavanje

Deadlock

- Primeri:

nit A

```
synchronized (o1) {
    .
    . // Uslov
    .
    synchronized (o2) {
        .
        . // Uslov
        .
    }
}
```

nit B

```
synchronized (o2) {
    .
    . // Uslov
    .
    synchronized (o1) {
        .
        . // Uslov
        .
    }
}
```

- Deadlock scenario - Nit A je zaključala objekat o1 i odmah nakon toga je Nit B zaključala objekat o2

Hvala na pažnji!

Pitanja?