nedeljainformatike v8.0

Ослобађање од катанаца: алтернативне технике синхронизације

Igor Pavlović

Matematička gimnazija

19. 05. 2022.

Agenda

1. Osnovni pojmovi
2. Algoritmi sinhronizacije

Osnovni pojmovi

- Sekvencijalno izvršavanje je takvo izvršavanje gde se sledeća računarska operacija ili programska naredba izvršava tek nakon što je prethodna završena, u skladu sa redosledom koji je zadat programom.

- Paralelno izvršavanje označava istovremeno izvršavanje više računarskih operacija, sekvenci operacija, programa ili delova jednog programa.

Osnovni pojmovi

- Konkurentan program sadrži dva ili više procesa koji se izvršavaju sekvencijalno i koji zajedno rade na izvršavanju nekog zadatka.
- Procesi konkurentnog programa komuniciraju pomoću deljenih varijabli ili putem razmene poruka.
- Potrebno je međusobno sinhronizovati procese kako bismo sprečili utrkivanje (race condition).

Problem utrkivanja – race condition

```
x = -1; y = -1;

void P1(){
    x = 1;
    y = 2;
}
```

```
void P2(){
    x2 = x;
    y2 = y;
}
```

Koje su moguće vrednosti za x2 i y2?

Osnovni pojmovi

- Međusobno isključivanje (Mutual exclusion) – sprečavanje više procesa da istovremeno pristupi istom deljenom resursu
- Uslovna sinhronizacija (Conditional synchronization) – blokiranje određenog procesa sve dok neki logički uslov ne bude ispunjen

Kritična sekcija

- Kritična sekcija je sekvenca naredbi koje pristupaju nekom deljenom objektu.
- Neophodno je dizajnirati entry i exit protokole za ulaz i izlaz iz kritične sekcije tako da obezbedimo međusobno isključivanje, izbegnemo Deadlock i Livelock, izbegnemo nepotrebna čekanja i Eventual entry.

```
void process(){
   while (true) {
    entry protocol;
    critical section;
    exit protocol;
    noncritical section;
   }
}
```

Katanci

```
#include <mutex>

std::mutex mtx;

void process() {
    while (true) {
    mtx.lock();
    critical section;
    mtx.unlock();
    noncritical section;
    }
}
```

Test and Set

- Specijalna instrukcija za procesore
- Atomska akcija – vraća staru boolean vrednost i nedeljivo postavlja vrednost na true

```
bool TS(bool lock) {
< bool initial = lock;  lock = true; return initial; > }
```

Test and Set

```
bool lock = false;

void process(){
    while (true) {
    while (TS(lock)) skip;
    critical section;
    lock = false;
    noncritical section;
    }
}
```

Test and Test and Set

- Spin locks – zagušenje memorije i keš memorija
- Upis u lock izaziva invalidaciju keš memorija svih procesora sa deljenom memorijom
- Izmena entry protokola koja dovodi do invaludacije samo kada neko ulazi u kritičnu sekciju

```
while (lock) {
    while (TS(lock)) skip;
}
```

Semafori

```
sem s=4;

void process() {
    while (true) {
    wait(s);
    critical section;
    signal(s);
    noncritical section;
    }
}
```

Agenda

1. Osnovni pojmovi
2. Algoritmi sinhronizacije

Fetch and add

- Specijalna instrukcija za procesore
- Inkrementiranje promenljive sa konstantom kao atomska akcija uz vraćanje stare vrednosti

FA(var, incr):
    <int tmp = var; var = var + incr; return(tmp);>

Ticket algoritam

- Svi koji pokušavaju da uđu u kritičnu sekciju prvo dobiju ticket sa brojem u redosledu dolaska
- Svaki proces čeka da dođe na red za ulazak u kritičnu sekciju
- Pravičan (fair) protokol

Ticket algoritam

```
int number = 1, next = 1, turn[n] = {0};

void process(int id){
    while (true) {
        turn[id] = FA(number,1); /* entry protocol */
        while (turn[id] != next) skip;
        critical section;
        next = next + 1; /* exit protocol */
        noncritical section;
    }
}
```

Andersenov algoritam

```
int slot = 0, flag[n] = { false };
flag[0] = true;

void process(){
    int myslot;
    while (true) {
        /*entry protocol*/
        myslot = (FA(slot,1)+1)%n;
        while (!flag[myslot]) skip;
        critical section;
        /* exit protocol */
        flag[myslot] = false;
        flag[(myslot+1)%n] = true;
        noncritical section;
    }
}
```

Get and Set

- Specijalna instrukcija za procesore
- Dohvata staru vrednost promenljive i postavlja novu vrednost promenljive kao atomsku akciju

GS(var, new):
    <int tmp = var; var = new; return(tmp);>

- Upotreba za <prev = tail; tail = node;>

CLH algoritam

```
Node tail = (false);

void process() {
    while (true) {
        /*entry protocol*/
        Node prev, node = (true);
        prev = GS(tail, node);
        while (prev.locked) skip;
        critical section;
        /* exit protocol */
        node.locked = false;
        noncritical section;
    }
}
```

Bakery algoritam

- Ne koristi FA instrukciju
- Svaki proces izvlači broj za jedan veći od bilo kog drugog procesa do sad
- Proces čeka sve dok njegov broj ne bude manji od broja svih ostalih procesa koji žele da pristupe kritičnoj sekciji
- Ukoliko više procesa dobije isti broj prednost ima proces sa nižim indeksom

Bakery algoritam

```
int turn[n] = { 0 };

void process(int id) {
  while (true) {
    /* entry protocol */
    turn[id] = 1; turn[id] = max(turn[0:n]) + 1;
    for[j = 1 to n st j != id]
      while (turn[j] != 0 && (turn[id], id) > (turn[j], j))
       skip;
    critical section;
    turn[id] = 0; /* exit protocol */
    noncritical section;
  }
}
```

Hvala na pažnji!

Pitanja?