Operačné systémy

5. Klasické IPC problémy

Ing. Martin Vojtko, PhD.

Fakulta informatiky a informačných technológií
STU v Bratislave

2021/2022

Vzájomné vylučovanie

Vzájomné vylučovanie - N procesov

Funguje bez nutných zmien

- Busy Waiting - špeciálne inštrukcie
- Sleep and WakeUp - semafór, mutex, monitor

Problem

- Busy Waiting - softvérové riešenia
- Predstavili sme si zatial len Petersona

Peterson 2 procesy

```
#define TRUE 1
#define FALSE 0
#define A 0
#define B 1
int inA, inB = FALSE;
int turn = A;

void procesA()
{
    while (TRUE)
    {
        inA = TRUE;
        turn = A;
        while(turn == A && inB == TRUE) { ; }
        critical_region();
        inA = FALSE;
        noncritical_region();
    }
}
```

```
stav
inA  = FALSE
inB  = FALSE
turn = A
```

```
void procesB()
{
    while (TRUE)
    {
        inB = TRUE;
        turn = B;
        while(turn == B && inA == TRUE) { ; }
        critical_region();
        inB = FALSE;
        noncritical_region();
    }
}
```

Peterson 2 procesy

```
1  int inA, inB = false;
2  int turn = A;
3
4  void procesA()
5  {
6      inA = true;                              //
7      turn = A;                                //enter_section
8      while (turn == A && inB == true) { ; } //
9      critical_region();
10     inA = false;                             //exit_section
11     noncritical_region();
12 }
```

Peterson 2 procesy

```
1  int inA, inB = false;
2  int turn = A;
3
4  void enter_section()
5  {
6      inA = true;
7      turn = A;
8      while (turn == A && inB == true) { ; }
9  }
10
11 void exit_section()
12 {
13     inA = false;
14 }
```

- Čo bude treba spraviť aby sme podporovali N procesov?

Peterson N procesov - pokus

```
1  #define N 100
2  int proc_in[N];
3  int proc_turn = 0;
4
5  void enter_section(int pid)
6  {
7      proc_in[pid] = true;
8      proc_turn = pid;
9      while (proc_turn == pid && other_in(pid)) { ; }
10 }
11
12 void exit_section(int pid)
13 {
14     proc_in[pid] = false;
15 }
```

- other_in - loop: ma nejaky iny proces zaujem?
- Nefunguje to prečo?

Peterson N procesov

```
1  #define N 100
2  #define MAX_LEVEL N - 1
3  int proc_in_level[N];
4  int proc_turn[MAX_LEVEL];
5
6  void enter_section(int pid)
7  {
8      for (int level = 0; level < MAX_LEVEL; level++)
9      {
10         proc_in_level[pid] = level;
11         proc_turn[level] = pid;
12
13         while (proc_turn[level] == pid &&
14                other_proc_has_higher_level(pid, level))
15         {;}
16     }
17 }
18
19 void exit_section(int pid)
20 {
21     proc_in_level[pid] = -1;
22 }
```

Peterson N procesov

```
1  #define N 100
2  #define MAX_LEVEL N - 1
3  int proc_in_level[N];
4  int proc_turn[MAX_LEVEL];
5
6  bool other_proc_has_higher_level(int pid, int level)
7  {
8      for (int oPid = 0; oPid < N; oPid++)
9      {
10         if (oPid == pid)
11             continue;
12
13         if (proc_in_level[oPid] >= level)
14             return true;
15     }
16     return false;
17 }
```

- Tak toto funguje ale za akú cenu?
  - s rastúcim N rastie čas vykonania kvadraticky (N^2)

Bakery Algorithm

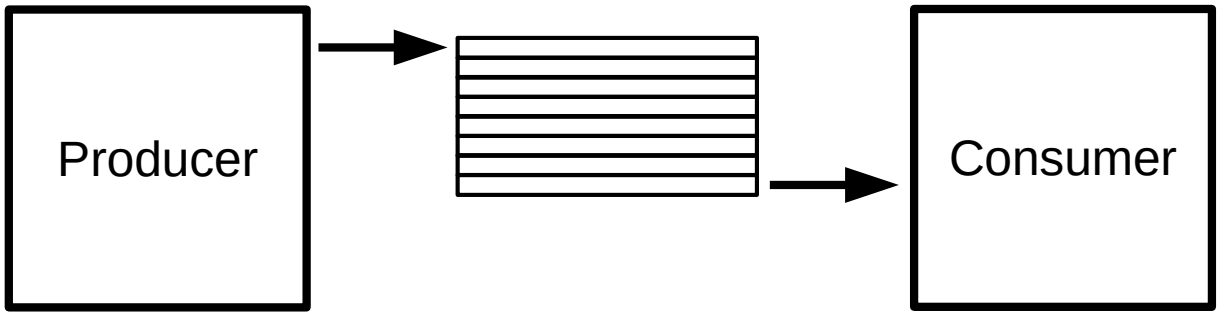
```
1  #define N = 100
2  int ticketN = 0; //zanedbajme ze tiket moze pretiect
3  int choosing[N];
4  int ticket[N];
5
6  void enter_section(int pid)
7  {
8      choosing[pid] = true;    //cisc OK, risc ?
9      ticket[pid] = ++ticketN; //sanca, ze viac procesov dostane rovnake cislo
10     choosing[pid] = false;
11
12     for (int oPid = 0; oPid < N; oPid++)
13     {
14         while (choosing[oPid] == true) {;}
15         while (ticket[oPid] != 0 && (ticket[oPid] < ticket[pid] ||
16                                      (ticket[oPid] == ticket[pid] && oPid < pid)))
17         {;}
18     }
19 }
20
21 void exit_section(int pid)
22 {
23   ticket[pid] = 0;
24 }
```

Klasické IPC problémy

Producent-Konzument (Producer-Consumer)

- Majme Rad o veľkosti N prvkov.
- Majme producenta, ktorý produkuje produkty vždy na koniec radu.
- Majme konzumenta, ktorý konzumuje produkty vždy zo začiatku radu.
- Ak je rad plný producent musí čakať.
- Ak je rad prázdny konzument musí čakať.
- Navrhnite program producenta a program konzumenta tak, aby nedošlo k uviaznutiu, súpereniu alebo vyhladovaniu.

Producent-Konzument (Producer-Consumer)



Producent-Konzument (Producer-Consumer)

```
#define N 100
int count = 0;

void producer()
{
  int item = 0;
  while (true)
  {
    produce_item(&item);
    if (count == N) sleep();

    push_item(&item);
    count++;

    if (count == 1) wake(consumer);

  }
}
```

```
1
2
3
4  void consumer()
5  {
6    int item = 0;
7    while (true)
8    {
9
10     if (count == 0) sleep();
11
12     pop_item(&item);
13     count--;
14
15     if (count == N-1) wake(producer);
16     consume_item(&item);
17   }
18 }
```

- Riešenie s kritickým Race condition.
  - count
  - vlakadanie a vyberanie do radu
  - sleep a wake

Producent-Konzument - Semafór

```
#define N 100
typedef int tSem;
tSem mutex = 1, empty = N, full = 0;

void producer()
{
  int item = 0;
  while (true)
  {
    produce_item(&item);
    down(&empty);

    down(&mutex);
    push_item(&item);
    up(&mutex);

    up(&full);

  }
}
```

```
1
2
3
4
5  void consumer()
6  {
7    int item = 0;
8    while (true)
9    {
10
11     down(&full);
12
13     down(&mutex);
14     pop_item(&item);
15     up(&mutex);
16
17     up(&empty);
18     consume_item(&item);
19   }
20 }
```

Producent-Konzument - POSIX threads

```
1  #include <pthread.h>
2
3  pthread_mutex_t mutex;
4  pthread_cond_t full, empty;
5
6  int main(int argc, char *argv[])
7  {
8      pthread_t pro, con;
9      pthread_mutex_init(&mutex, 0);
10     pthread_cond_init(&full, 0);
11     pthread_cond_init(&empty, 0);
12     pthread_create(&pro, 0, producer, 0);
13     pthread_create(&con, 0, consumer, 0);
14
15     pthread_join(&pro, 0);
16     pthread_join(&con, 0);
17     pthread_cond_destroy(&full);
18     pthread_cond_destroy(&empty);
19     pthread_mutex_destroy(&mutex);
20 }
```

Producent-Konzument - POSIX threads

```
void producer()
{
  int item = 0;
  while (true)
  {
    produce_item(&item);
    pthread_mutex_lock(&mutex);

    while (isFull())
      pthread_cond_wait(&full, &mutex);

    push_item(&item);
    pthread_cond_signal(&empty);

    pthread_mutex_unlock(&mutex);

  }
}
```

```
1  void consumer()
2  {
3    int item = 0;
4    while (true)
5    {
6
7      pthread_mutex_lock(&mutex);
8
9      while (isEmpty())
10       pthread_cond_wait(&empty, &mutex);
11
12     pop_item(&item);
13     pthread_cond_signal(&full);
14
15     pthread_mutex_lock(&mutex);
16     consume_item(&item);
17   }
18 }
```

Producer-Consumer problem - Monitor(Java)

```
1  public class ProducerConsumer
2  {
3    static Monitor mon = new Monitor();
4    static Producer pro = new Producer();
5    static Consumer con = new Consumer();
6
7    public static void main(String args[])
8    {
9      pro.start();
10     con.start();
11   }
12
13   /*Producer*/
14
15   /*Consumer*/
16
17   /*Monitor*/
18 }
```

Producer-Consumer problem - Monitor(Java)

```
1  static class Producer extends Thread
2  {
3    public void run()
4    {
5      int item;
6      while (true) {
7        produce_item(item);
8        mon.push_item(item);
9      }
10   }
11 }
12
13 static class Consumer extends Thread
14 {
15   public void run()
16   {
17     int item;
18     while (true) {
19       mon.pop_item(item);
20       consume_item(item);
21     }
22   }
23 }
```

Producer-Consumer problem - Monitor(Java)

```
1  static class Monitor
2  {
3    static Queue q = new Queue();
4    public synchronized void push_item(int item)
5    {
6      while (isFull()) go_wait();
7      q.push_item(item);
8      if (wasEmpty()) notify();
9    }
10
11   public synchronized void pop_item(int item)
12   {
13     while (isEmpty()) go_wait();
14     q.pop_item(item);
15     if (wasFull()) notify();
16   }
17
18   private go_wait()
19   {
20     try {
21         wait();
22     }
23     catch(InterruptedException e) {}
24   }
25 }
```

Producer-Consumer problem - Message Passing

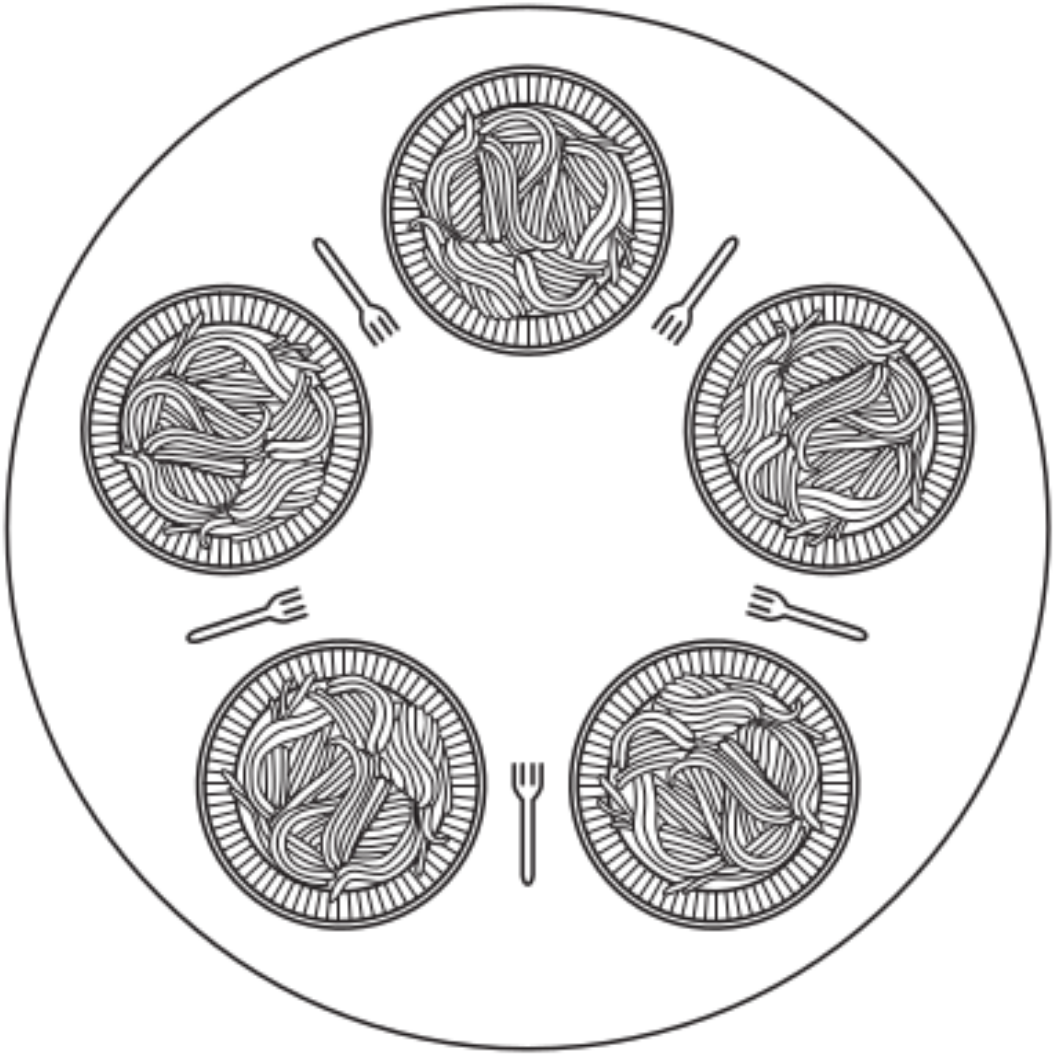
```
void producer(void)
{
  int item;
  message m;



  while (true)
  {
    produce_item(&item);
    receive(consumer, &m);
    build_message(&m, &item);
    send(consumer, &m);

  }
}
```

```
1  void consumer(void)
2  {
3    int item, i;
4    message m;
5    for (i = 0; i < N; i++)
6      send(producer, &m);
7
8    while (true)
9    {
10
11     receive(producer, &m);
12     extract_item(&m, &item);
13     send(producer, &m);
14     consume_item(&item);
15   }
16 }
```

Večerajúci filozofi (Dining Philosophers)

- Majme 5 filozofov sediacich okolo okrúhleho stola.
- Každý filozof má pred sebou tanier so špagetami.
- Medzi dvoma taniermi je práve jedna vidlička.
- Špagety sú klzké a preto sú potrebné dve vidličky na nabranie sústa.
- Filozof vykonáva len dve činnosti: premýšľa a konzumuje.
- Ak chce filozof jesť pokúsi sa vziať postupne obe vidličky, ktoré má vedľa taniera.
- Ak má filozof obe vidličky chvíľu konzumuje a potom odloží vidličky a znovu premýšľa.
- Navrhnite program pre každého filozofa tak, aby nedošlo k uviaznutiu, súpereniu alebo vyhladovaniu.

Večerajúci filozofi (Dining Philosophers)



Večerajúci filozofi (Dining Philosophers) - pokus

```
1  #define N 5
2  tSem fork[N];
3
4  void philosopher(int id) {
5    while (true) {
6      think();
7      take_fork(id);
8      take_fork((id+1) % N);
9      eat();
10     put_fork(id);
11     put_fork((id+1) % N);
12   }
13 }
14
15 void take_fork(int id) {
16     down(&fork(id));
17 }
18
19 void put_fork(int id) {
20     up(&fork(id));
21 }
```

- Hrozí uviaznutie. Ak si v jednom momente vezme každý ľavú vidličku nikdy sa nedostane k pravej.

Večerajúci filozofi (Dining Philosophers) - Semafór

```
1  #define N 5
2
3  void philosopher(int id)
4  {
5    while (true) {
6      think();
7      take_forks(id);
8      eat();
9      put_forks(id);
10   }
11 }
```

- take_forks a put_forks su KO ako by sme ich implementovali?

Večerajúci filozofi (Dining Philosophers) - Semafór pokus

```
1  tSem mutex = 1;
2
3  void take_forks(int id)
4  {
5      down(&mutex);
6      take_fork(id);
7      take_fork((id+1)%N);
8      up(&mutex);
9  }
10
11 void put_forks(int id)
12 {
13     down(&mutex);
14     put_fork(id);
15     put_fork((id+1)%N);
16     up(&mutex);
17 }
```

- Kde je tu problem?

Večerajúci filozofi (Dining Philosophers) - Semafór pokus 2

```
1  tSem mutex = 1;
2
3  void take_forks(int id)
4  {
5      down(&mutex);
6      take_fork(id);
7      take_fork((id+1)%N);
8      up(&mutex);
9  }
10
11 void put_forks(int id)
12 {
13     put_fork(id);
14     put_fork((id+1)%N);
15 }
```

- Kde je tu problem?

Večerajúci filozofi (Dining Philosophers) - Semafór

```
1 #define LEFT (id+N-1)%N
2 #define RIGHT (id+1)%N
3 tSem mutex = 1;
4 tSem pSem[N];
5 int state[N];
```

```
void take_forks(int id)
{
  down(&mutex);
  state[id] = HUNGRY;
  test(id);
  up(&mutex);
  down(&pSem[id]);
}
```

```
1  void put_forks(int id)
2  {
3    down(&mutex);
4    state[id] = THINKING;
5    test(LEFT);
6    test(RIGHT);
7    up(&mutex);
8  }
```

```
1 void test(int id)
2 {
3   if (state[id] == HUNGRY && state[LEFT] != EATING && state[RIGHT] != EATING) {
4     state[id] = EATING;
5     up(&pSem[id]);
6   }
7 }
```

Čitatelia-Zapisovatelia (Readers-Writers)

- Majme spoločnú pamäť prípadne databázu.
- Majme množinu zapisovateľov, ktorý sú oprávnený pridávať položky do databázy.
- Majme množinu čitateľov, ktorý sú oprávnený čítať položky z databázy.
- V jeden moment môže iba jeden zapisovateľ pristupovať do databázy.
- Ak nie je žiaden zapisovateľ v databáze, tak v jeden moment môže čítať databázu neobmedzené množstvo čitateľov.
- Navrhnite program čitateľa a program Zapisovateľa tak, aby nedošlo k uviaznutiu, súpereniu alebo vyhladovaniu.

Čitatelia-Zapisovatelia (Readers-Writers)

```
tSem mutex = 1, db = 1;
int rc = 0; // reader-count




void writer()
{
  while (true) {
    down(&db);
    write_db();
    up(&db);
  }
}
```

```
1  void reader()
2  {
3    while(true) {
4      down(&mutex);
5      if (++rc == 1) down(&db);
6      up(&mutex);
7
8      read_db();
9
10     down(&mutex);
11     if (--rc == 0) up(&db);
12     up(&mutex);
13   }
14 }
```

- Riešenie uprednostňujúce čitateľov. Zapisovatelia môžu hladovať.

Zhrnutie

Zhrnutie

- Softvé riešenia vzájomného vylučovania N procesov nie sú zadarmo.
  - Peterson kvadratická zložitosť N^2
  - Bakery lineárna zložitosť N
- Producer-Consumer
- Dining Philosophers
- Readers-Writers

Čo robiť do ďalšej prednášky

- Pripravte sa na test!!!
- Prečítať kapitolu 6. z Tanenbauma.