Operačné systémy

5. Komunikácia procesov (IPC) - synchronizácia a Klasické IPC problémy

Ing. Martin Vojtko, PhD.

2024/2025

Synchronizácia

Synchronizácia

Synchronizácia Procesov

Okrem špeciálneho prípadu vzájomného vylučovania používaná na zabezpečenie:

- usporiadania vykonávania procesov v čase.
- podmieneného vykonávania procesov.
- manažmentu prostriedkov.

- Semafór
- Condition Variables
- Bariéra

Synchronizácia - Semafór

Binárny semafór

je vhodným prostriedkom na usporiadanie procesov v čase. Semafór je inicializovaný na 0. Proces B, ktorý čaká na proces A vykoná down() v momente keď je nutné čakať na výsledky z A. Ak je proces A hotový vykoná operáciu up() čím prebudí čakajúci proces B.

Semafór

je vhodným nástrojom na manažment prostriedkov. Ak systém má viacero prostriedkov rovnakého typu je možné ich počet reprezentovať semafórom. Semafór následne púšťa procesy ku prostriedkom kým sa všetky neminú.

Synchronizácia - Semafór

```
typedef int tSem;
tSem consume = 0, produce = 1;
void procesA()
{
  while (true)
  {
    produce.down();
    produce_data();
    consume.up();
  }
}
```

```
1
2
3  void procesB()
4  {
5    while (true)
6    {
7      consume.down();
8      consume_data();
9      produce.up();
10   }
11 }
```

Synchronizácia - Condition Variables

Condition Variables

Je vhodným prostriedkom na usporiadanie thread-ov v čase, podmienené vykonanie thread-ov a nástrojom na manažment prostriedkov. Výlučne sa používa v kombinácií s Mutex-om.

- Použitie na podmienene riadenie vstupu do KO s uspaním thread-u ak podmienka nieje splnená.
- Nasadením condition variable sa zamädzí neužitočnému cykleniu.

Synchronizácia - POSIX Condition Variables

Thread call	Description
Pthread_cond_init	Create a condition variable
Pthread_cond_destroy	Destroy a condition variable
Pthread_cond_wait	Block waiting for a signal
Pthread_cond_signal	Signal another thread and wake it up
Pthread_cond_broadcast	Signal multiple threads and wake all of them

Synchronizácia - Condition Variables

```
#include <pthread.h>
pthread_mutex_t mutex;
bool hasData = false;

void threadA()
{
  while (true)
  {
    while (hasData) { ; }
    pthread_mutex_lock(&mutex);

    produce_data();

    hasData = true;
    pthread_mutex_unlock(&mutex);
  }
}
```

```
1
2
3
4
5  void threadB()
6  {
7    while (true)
8    {
9      while (! hasData) { ; }
10     pthread_mutex_lock(&mutex);
11
12     consume_data();
13
14     hasData = false;
15     pthread_mutex_unlock(&mutex);
16   }
17 }
```

- Chceme aby sme do KO vstupili vylučne ak:
  - thread A môže vyprodukovať dáta.
  - thread B môže konzumovať dáta.
- Toto riešenie však nieje dobré. Prečo?

Synchronizácia - Condition Variables

```
#include <pthread.h>
pthread_mutex_t mutex;
bool hasData = false;

void threadA()
{
  while (true)
  {
    while (hasData) { ; } //!loop
    pthread_mutex_lock(&mutex);

    produce_data();

    hasData = true;
    pthread_mutex_unlock(&mutex);
  }
}
```

```
1
2
3
4
5  void threadB()
6  {
7    while (true)
8    {
9      while (! hasData) { ; } //!loop
10     pthread_mutex_lock(&mutex);
11
12     consume_data();
13
14     hasData = false;
15     pthread_mutex_unlock(&mutex);
16   }
17 }
```

- Chceme aby sme do KO vstupili vylučne ak:
  - thread A môže vyprodukovať dáta.
  - thread B môže konzumovať dáta.
- Toto riešenie však nieje dobré. Prečo?

Synchronizácia - Condition Variables

```
#include <pthread.h>
pthread_mutex_t mutex;
bool hasData = false;

void threadA()
{
  while (true)
  {
    pthread_mutex_lock(&mutex);
    while (hasData) { ; }

    produce_data();

    hasData = true;
    pthread_mutex_unlock(&mutex);
  }
}
```

```
1
2
3
4
5  void threadB()
6  {
7    while (true)
8    {
9      pthread_mutex_lock(&mutex);
10     while (! hasData) { ; }
11
12     consume_data();
13
14     hasData = false;
15     pthread_mutex_unlock(&mutex);
16   }
17 }
```

- Chceme aby sme do KO vstupili vylučne ak:
  - thread A môže vyprodukovať dáta.
  - thread B môže konzumovať dáta.
- Toto riešenie však nieje dobré. Prečo?

Synchronizácia - Condition Variables

```
#include <pthread.h>
pthread_mutex_t mutex;
bool hasData = false;

void threadA()
{
  while (true)
  {
    pthread_mutex_lock(&mutex);
    while (hasData) { ; } //!deadlock

    produce_data();

    hasData = true;
    pthread_mutex_unlock(&mutex);
  }
}
```

```
1
2
3
4
5  void threadB()
6  {
7    while (true)
8    {
9      pthread_mutex_lock(&mutex);
10     while (! hasData) { ; } //!deadlock
11
12     consume_data();
13
14     hasData = false;
15     pthread_mutex_unlock(&mutex);
16   }
17 }
```

- Chceme aby sme do KO vstupili vylučne ak:
  - thread A môže vyprodukovať dáta.
  - thread B môže konzumovať dáta.
- Toto riešenie však nieje dobré. Prečo?

Synchronizácia - Condition Variables

```
#include <pthread.h>
pthread_mutex_t mutex;
bool hasData = false;

void threadA()
{
  while (true)
  {
    pthread_mutex_lock(&mutex);
    if (hasData) {
      pthread_mutex_unlock(&mutex);
      continue;
    }

    produce_data();

    hasData = true;
    pthread_mutex_unlock(&mutex);
  }
}
```

```
1
2
3
4
5  void threadB()
6  {
7    while (true)
8    {
9      pthread_mutex_lock(&mutex);
10     if (! hasData) {
11       pthread_mutex_unlock(&mutex);
12       continue;
13     }
14
15     consume_data();
16
17     hasData = false;
18     pthread_mutex_unlock(&mutex);
19   }
20 }
```

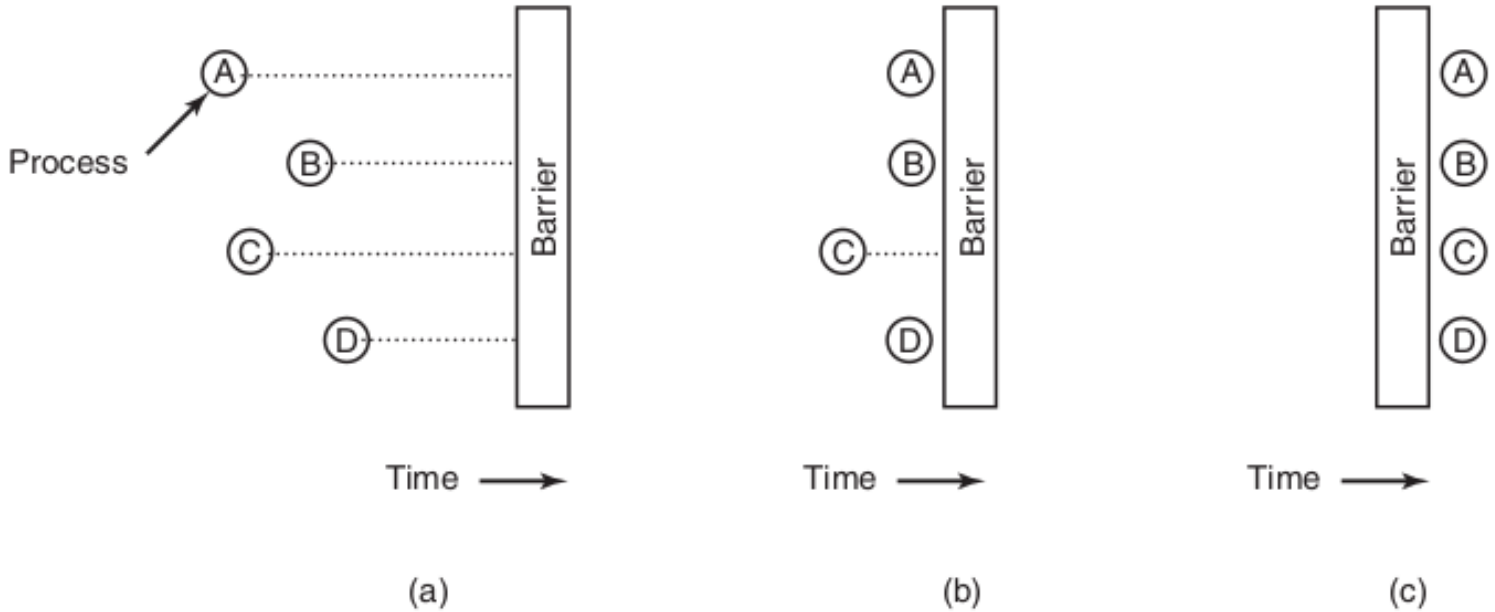
- Chceme aby sme do KO vstupili vylučne ak máme dáta.
- Toto riešenie je funkčné ale nie ideálne prečo?

Synchronizácia - Condition Variables

```
#include <pthread.h>
pthread_mutex_t mutex;
pthread_cond_t cons, prod;
bool hasData = false;

void threadA()
{
  while (true)
  {
    pthread_mutex_lock(&mutex);
    while (hasData)
      pthread_cond_wait(&cons, &mutex);

    produce_data();

    hasData = true;
    pthread_cond_signal(&prod);
    pthread_mutex_unlock(&mutex);
  }
}
```

```
1
2
3
4
5
6  void threadB()
7  {
8    while (true)
9    {
10     pthread_mutex_lock(&mutex);
11     while (! hasData)
12       pthread_cond_wait(&prod, &mutex);
13
14     consume_data();
15
16     hasData = false;
17     pthread_cond_signal(&cons);
18     pthread_mutex_unlock(&mutex);
19   }
20 }
```

Synchronizácia - Bariéra



Message Passing

Message Passing

- Doterajšie metódy komunikácie neumožňujú vzájomnú komunikáciu v rámci distribuovaných systémov.
- Posielanie správ definuje spôsob komunikácie procesov, ktoré môžu byť súčasne vykonávané na rôznych strojoch.
- Posielanie správ definuje dve systémové volania:
  - send(Destination, Message)
  - receive(Source, Message)
- Nakoľko systémové volania sú volaniami OS ich reprezentácie môže zabezpečiť výmenu správ napríklad prostredníctvom internetu.

Message Passing - problémy

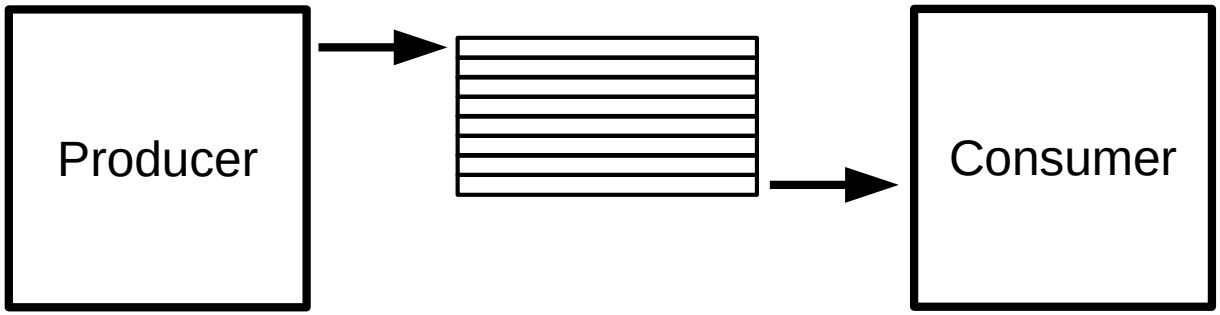
- Strata správy - napraviteľné Ack. správami
- Opakované prijatie správy - napraviteľné identifikátorom správ
- Zmena poradia správ - napraviteľné identifikátorom správ
- Autentickosť správ - nutná autentifikácia a šifrovanie

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 = 0, full = N;

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

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

    up(&empty);

  }
}
```

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

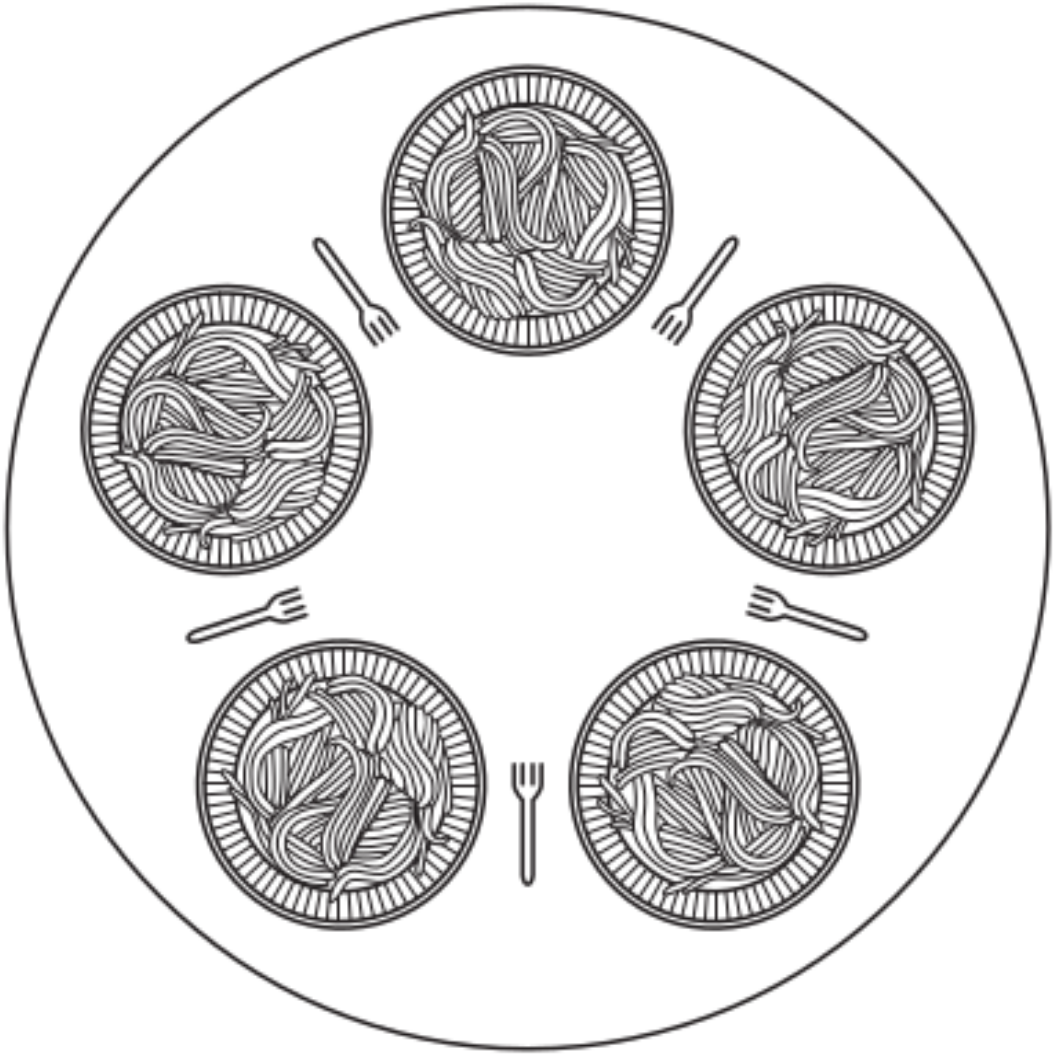
```
void producer(void)
{
  int item;
  message m;



  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 }
```

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 }
18
19 void take_fork(int id) {
20     down(&fork(id));
21 }
```

- 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
14     put_fork(id);
15     put_fork((id+1)%N);
16
17 }
18
19 void take_fork(int id) {
20     down(&fork(id));
21 }
```

- 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] = {0};
5 int state[N] = {THINKING};
```

```
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   {
5     state[id] = EATING;
6     up(&pSem[id]);
7   }
8 }
```

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) - Semafór

```
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.

Čitatelia-Zapisovatelia (Readers-Writers) - Cond. Variable

```
#include <pthread.h>
pthread_mutex_t mutex;
pthread_cond_t rQ, wQ;
int rN, wN, awN = 0;

void reader()
{
  pthread_mutex_lock(&mutex);

  while (wN != 0)
    pthread_cond_wait(&rQ, &mutex);
  rN++;

  pthread_mutex_unlock(&mutex);

  read();

  pthread_mutex_lock(&mutex);

  if (--rN == 0)
    pthread_cond_signal(&wQ);

  pthread_mutex_unlock(&mutex);
}
```

```
1
2  void writer()
3  {
4    pthread_mutex_lock(&mutex);
5
6    wN++;
7    while ((rN != 0) || (awN != 0))
8      pthread_cond_wait(&wQ, &mutex);
9    awN++;
10
11   pthread_mutex_unlock(&mutex);
12
13   write();
14
15   pthread_mutex_lock(&mutex);
16
17   awN--;
18   if (--wN == 0)
19     pthread_cond_broadcast(&rQ);
20   else
21     pthread_cond_signal(&wQ);
22
23   pthread_mutex_unlock(&mutex);
24 }
```

Zhrnutie

Zhrnutie

- Vzájomného vylučovanie je len jedným z prípadov synchronizácie
- Semafór a condition-variable sú príkladmy riešenia keď je potrebné synchronizovať vykonávanie úloh
- príklady použitia synchronizácie v teórii
  - Producer-Consumer
  - Dining Philosophers
  - Readers-Writers

Čo robiť do ďalšej prednášky

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