Multithreading a komunikácia medzi procesmi a threadmi,

- Časová závislosť, kritická sekcia a vzájomné vylúčenie, súvislosti medzi týmito pojmami
- Riešenia vzájomného vylúčenia
  - bez podpory OS (väčšinou busy waiting)
  - s podporou OS
- Klasické príklady z komunikácie

Autor: Peter Tomcsányi

Použité obrázky z učebnice:
Andrew. S. Tanenbaum, Modern operating systems
http://www.cs.vu.nl/~ast/books/

Thready v C vo Windows

súbor: plusminus_windows.cpp

```
#include <iostream>
#include "windows.h"
#define N 1000000  // pocet opakovani
int x = 0;  // globalna zdielana premenna
DWORD WINAPI T1(_In_ LPVOID lpParameter) {
    for (int i = 0; i < N; i++)
        x++;    // N-krat pripocita 1 k x
    return 0;
}
DWORD WINAPI T2(_In_ LPVOID lpParameter) {
    for (int i = 0; i < N; i++)
        x--;    // N-krat odcita 1 od x
    return 0;
}
int main() {
    HANDLE h[2];
    // Vytvor thready, ale nerozbehni ich
    h[0] = CreateThread(NULL, 0, &T1, NULL, CREATE_SUSPENDED, NULL);
    h[1] = CreateThread(NULL, 0, &T2, NULL, CREATE_SUSPENDED, NULL);
    for (int i = 0; i < 2; i++) ResumeThread(h[i]); // Rozbehni ich
    WaitForMultipleObjects(2, h, true, INFINITE); // Pockaj ich
    printf("x=%d\n\n", x);  // Vypis vysledok
}
```

Čo bude program vypisovať?
Vždy nulu?
Alebo nie vždy?
Závisí to od N?

Thready v C v Linuxe

súbor: plusminus_linux.c

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>

#define N 100000000  // pocet opakovani (treba viac nez vo Windows)

int x = 0;  // globalna zdielana premenna

void *T1(void *vargp) {
    for (int i = 0; i < N; i++)
        x++;        // N-krat pripocita 1 k x
    return NULL;
}

void *T2(void *vargp) {
    for (int i = 0; i < N; i++)
        x--;        // N-krat odcita 1 od x
    return NULL;
}

int main() {
    pthread_t h[2];
    pthread_create(&h[0], NULL, T1, NULL);
    pthread_create(&h[1], NULL, T2, NULL);

    for (int i = 0; i < 2; i++) pthread_join(h[i], NULL);

    printf("x=%d\n\n",x);
    return 0;
}
```

Pri kompilovaní treba pridať parameter **-pthread** napr.:

```
gcc plusminus_linux.c -pthread -o plusminus
```

Ak používate v Linuxe CLion, skúste nastavenie podľa tohto linku

Thready v Pythone

súbor: print_problem.cpp

```
from time import sleep
from random import random
import threading
def vlakno(meno,m):
    n = 0
    while n < m:
        print(meno,n)
        n+=1
        sleep(random())
def go(m):
    p1=threading.Thread(target = vlakno,
      args = ('prve',m))
    p2=threading.Thread(target = vlakno,
      args = ('druhe',m))
    p1.start(); p2.start()
    p1.join(); p2.join()
    print('koniec')
```

Thready v Pythone (2)

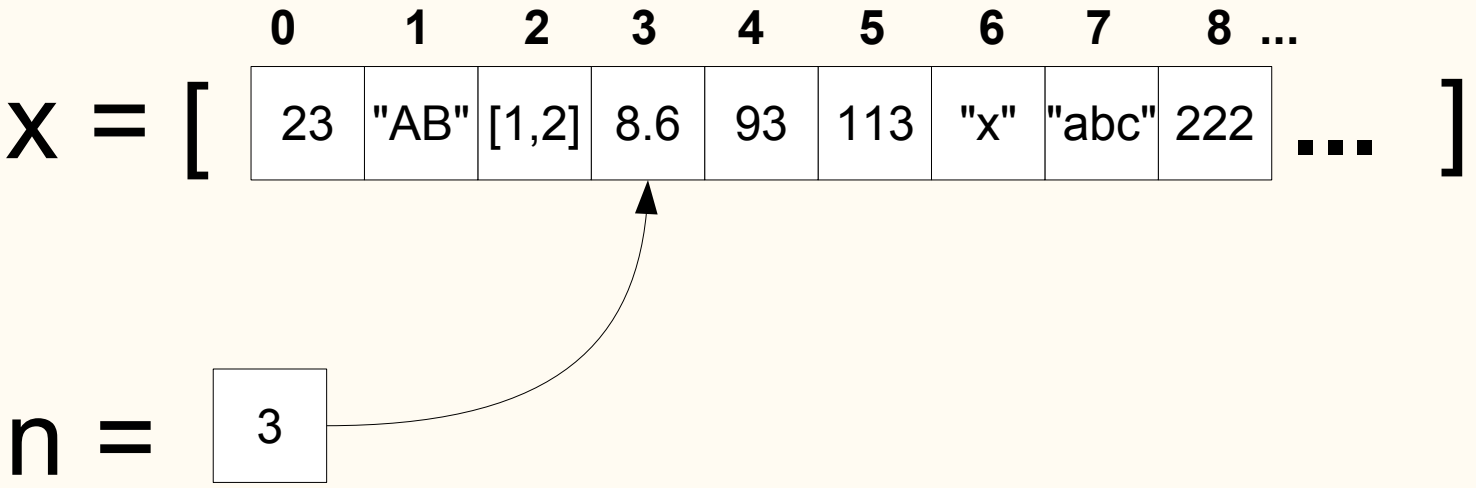
```
>>> go(30)
prvedruhe  00

druhe 1
druhe 2
prve 1
druhe 3
druhe 4
prve 2
...
druhe 10
prve 10
druhe 11
druhe 12
prve 11
prvedruhe  1213

druhe 14
prve 13
```

```
sleep(random()/10)

>>> go(100)
prvedruhe  00

druhe 1
druhe 2
druhe 3
prve 1
druhe 4
druhe 5
prve 2
prve 3
druhe 6
prve druhe4
7
prve 5
prvedruhe  68

druhe 9
prve 7
druheprve  108
```

Efekty vznikajúce z paralelného behu threadov ktoré sa striedajú aj vo vnútri vykonávania funkcie print.

Efekty sa zvýraznia, keď zmenšíme časy v **sleep** vo funkcii vlakno

Komunikácia medzi procesmi
Ukážkový príklad



Vlákna A a B vyberajú zo zoznamu (poľa) x údaje, ktoré chcú spracovať.
x je zoznam
n je index ešte nevybraného prvku

Implementácia – prvý pokus

```
class Data:
    def __init__(self,pocet):
        self.x = [i for i in range(pocet)]
        self.n = 0
    def vyber(self,proces):
        res=self.x[self.n]
        self.n+=1
        return res
```

aby sa nám ľahko testoval výsledok, x naplníme číslami od 0 do pocet-1

druhý parameter je číslo procesu, zatiaľ ho nepotrebujeme, ale v niektorých ďalších pokusoch bude použitý

Ak na tomto mieste je A odobratý procesor a B vtedy zavolá metódu vyber, tak **vznikne problém** – Front bude „pokazený“
Nielen to - aj samotný príkaz n+=1 môže mať problémy keď ho vykonávajú súčasne dve vlákna
(Podrobnejšie vysvetlenia na prednáške)

Testovací program

súbor: kriticke_sekcie.py

```
import threading
from time import time

def vyberaj(mojecislo,obj,kolko,li):
  for i in range(kolko):
    li.append(obj.vyber(mojecislo))

def skus_class(kolko,Data_class):
  ti = time()
  obj = Data_class(2*kolko) # vytvor data
  li1 = []; li2 = []
  try:
    t1=threading.Thread
      (target=vyberaj, args=(0, obj, kolko, li1))
    t2=threading.Thread
      (target=vyberaj, args=(1, obj, kolko, li2))
    t1.start(); t2.start()  # spusti obe vlákna
  except:
    print("Chyba: nepodarilo sa spustit vlakna")
  t1.join(); t2.join()  # počkaj na skončenie
  print(time()-ti,'n=',obj.n)    # čas a n
  test(li1,li2,kolko)            # kontrola správnosti
```

```
def skus(kolko):
  skus_class(kolko,Data)
```

Testovací program (2)

```
def test(li1,li2,kolko):
  # kontrola správnosti oboch zoznamov
  # či obsahujú každý očakávaný prvok práve raz
  s1=set(li1); s2=set(li2) # prerob zoznamy na množiny
  vsetky=set(range(2*kolko))
                          # množina všetkých očakávaných prvkov
  chybajuce=vsetky-s1-s2 # množina chýbajúcich prvkov
  duplicitne=s1 & s2      # množina duplicitných prvkov
  if chybajuce | duplicitne:
    print("ZLE")
    if chybajuce:  # vypíš počet a najviac desať chýbajúcich
      print('chybaju',len(chybajuce),list(chybajuce)[:10])
    if duplicitne: # vypíš počet a najviac desať duplicitných
      print('dvakrat',len(duplicitne),list(duplicitne)[:10])
  else:
    print("OK")
```

Testovací program v C (Windows)

súbor: kriticke_sekcie.cpp

```c
#include <iostream>
#include "windows.h"

#define N 10000
int x[2 * N];
int n;
int vyber() {
  int res = x[n];
  n++;
  return res;
}
DWORD WINAPI MojThread
  (_In_ LPVOID lpParameter) {
  int *li = (int *) lpParameter;
  for (int i = 0; i < N; i++) {
    li[i] = vyber();
  }
  return 0;
}
int li1[N];
int li2[N];

int main() {
  HANDLE h[2];

  for (int i = 0; i < 2*N; i++)
    x[i] = i;

  n = 0;

  h[0] = CreateThread(NULL, 0,
    &MojThread, &li1,
    CREATE_SUSPENDED, NULL);
  h[1] = CreateThread(NULL, 0,
    &MojThread, &li2,
    CREATE_SUSPENDED, NULL);

  for (int i = 0; i < 2; i++)
    ResumeThread(h[i]);

  WaitForMultipleObjects
    (2, h, true, INFINITE);

  if (testuj(li1, li2, n))
    printf("OK\n");
  else
    printf("ZLE\n");

  return 0;
}
```

Testovací program v C (2)

```
bool testuj(int li1[], int li2[],int n) {
  int i1, i2, narade;
  i1 = 0; i2 = 0; narade = 0; FILE *f;
  fopen_s(&f, "d:\\test.txt", "w");
  bool ok = true;
  if (n != 2 * N)
    { fprintf(f, "nespravne n: %d\n", n); ok = false; }
  while (narade < 2 * N) {
    if (i1 < N && li1[i1] == narade)
      i1++;
    else if (i2 < N && li2[i2] == narade)
      i2++;
    else
      { fprintf(f, "chyba: %d\n", narade); ok = false; }
    while (i1 < N && li1[i1] == narade)
      { i1++; fprintf(f,"dvakrat: %d\n", narade); ok = false; }
    while (i2 < N && li2[i2] == narade)
      { i2++; fprintf(f,"dvakrat: %d\n", narade); ok = false; }
    narade++;
  }
  fclose(f);
  return ok;
}
```

Testovací program v C (Linux)

súbor: kriticke_sekcie_linux.c

```c
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <stdbool.h>

#define N 100000

int x[2 * N];
int n;

int vyber() {
  int res = x[n];
  n++;
  return res;
}

void *MojThread(void *vargp) {
  int *li = (int *)vargp;
  for (int i = 0; i < N; i++) {
    li[i] = vyber();
  }
  return NULL;
}

int li1[N];
int li2[N];

int main() {
  pthread_t h[2];

  for (int i = 0; i < 2 * N; i++)
    x[i] = i;
  n = 0;

  pthread_create(&h[0], NULL,
      MojThread, &li1);
  pthread_create(&h[1], NULL,
      MojThread, &li2);

  for (int i = 0; i < 2; i++)
    pthread_join(h[i], NULL);

  printf("--koniec\n");

  if (testuj(li1, li2, n))
    printf("OK\n");
  else
    printf("ZLE\n");

  return 0;
}
```

Testovací program v C (Linux) (2)

```
bool testuj(int li1[], int li2[],int n) {
  int i1, i2, narade;
  i1 = 0; i2 = 0; narade = 0; FILE *f;
  f = fopen("test.txt", "w");
  bool ok = true;

  if (n != 2 * N)
    { fprintf(f, "nespravne n: %d\n", n); ok = false; }

  while (narade < 2 * N) {
    if (i1 < N && li1[i1] == narade)
      i1++;
    else if (i2 < N && li2[i2] == narade)
      i2++;
    else
      { fprintf(f, "chyba: %d\n", narade); ok = false; }

    while (i1 < N && li1[i1] == narade)
      { i1++; fprintf(f,"dvakrat: %d\n", narade); ok = false; }
    while (i2 < N && li2[i2] == narade)
      { i2++; fprintf(f,"dvakrat: %d\n", narade); ok = false; }
    narade++;
  }

  fclose(f);
  return ok;
}
```

Časová závislosť (súbeh) a kritická sekcia - definície

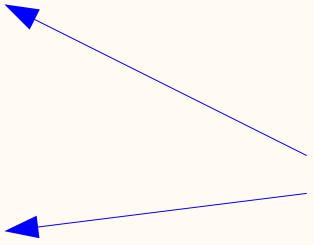
- **Časová závislosť** (Súbeh, Race Condition) nastáva ak výsledok práce viacerých spolupracujúcich procesov alebo threadov závisí od poradia, v ktorom boli plánované plánovačom procesov
- **Kritická sekcia** je úsek programu, ktorý pracuje so zdieľanými údajmi a v jeho vnútri môžu byť údaje nekonzistentné.
- **Vzájomné vylúčenie** dosiahneme ak žiadne dva procesy alebo thready sa neocitnú naraz vo svojich kritických sekciách.

Riešenia vzájomného vylúčenia bez podpory od operačného systému

- Zakázanie prerušení
- Zamykacie premenné
- Striktné striedanie
- Petersonovo riešenie
- Neprerušiteľné "testuj a zamkni"

Zakázanie prerušení

```
def vyber(self,mojecislo):
    zakaz_prerusenia()
    res=self.x[self.n]
    self.n+=1
    povol_prerusenia()
    return res
```



Tieto dve funkcie v skutočnosti v Pythone neexistujú, je to len ilustračný príklad

- Rieši vzájomné vylúčenie lebo znemožní plánovaču procesov dostať sa k slovu
- Tento prístup sa často používa v súčastiach operačného systému napr. v drajvroch
- Zakázanie prerušení je ale tvrdý zásah do chodu celého počítača.
- Preto ho operačné systémy obvykle nedovolia použiť v obyčajných procesoch
- Funguje správne len v prípade jedného procesora

Zamykacie premenné

```
class Data_Zamykacie:
    def __init__(self,pocet):
        ........
        self.volno = True
    def vyber(self,proces):
        while not self.volno: pass
        self.volno = False
        res=self.x[self.n]
        self.n+=1
        self.volno = True
        return res
```

Keď sa vymenia procesy na **tomto** mieste, tak sa môžu dostať dva procesy do kritickej sekcie!
Teda vzájomné vylúčenie sme nevyriešili.

Striktné striedanie

```
class Data_Striktne:
    def __init__(self,pocet):
        ........
        self.narade = 0
    def vyber(self,proces):
        while self.narade != proces:
            pass
        res=self.x[self.n]
        self.n+=1
        self.narade = 1-proces
        return res
```

Tento kód dosiahne vzájomné vylúčenie.
Ale:

- Funguje len pre dva procesy s číslami 0 a 1, pre viac procesov by sa kód musel zmeniť
- Keď prvý proces vojde do kritickej sekcie a potom z nej vyjde a bude chcieť znova vojsť, tak zostane blokovaný a pritom proces 2 vôbec nemusí byť vo svojej kritickej sekcii. A to nechceme.
- Preto je použiteľný len v špeciálnych prípadoch keď vieme, že procesy sa pri vstupoch do svojich kritických sekcií pravidelne striedajú.

Dobré riešenie vzájomného vylúčenia

1. Žiadne dva procesy nie sú naraz vo svojich kritických sekciách
2. Nepredpokladáme nič o vzájomnej rýchlosti a/alebo počte procesorov
3. Žiadny proces mimo svojej kritickej sekcie nesmie brániť iným procesom vstúpiť do svojej kritickej sekcie
4. Žiadny proces nesmie nekonečne dlho čakať na vstup do kritickej sekcie

Petersonovo riešenie

súbor: kriticke_sekcie.py

```
class Data_Peterson:
    def __init__(self,pocet):
        ........
        self.narade = 0
        self.zaujem = [False,False]

    def vstup_do_ks(self,proces):
        druhy = 1-proces
        self.zaujem[proces] = True
        self.narade = proces
        while self.narade == proces and \
        self.zaujem[druhy]:
            pass

    def vystup_z_ks(self,proces):
        self.zaujem[proces] = False
```

Petersonovo riešenie (2)

```
def vyber(self,proces):
    self.vstup_do_ks(proces)
    res=self.x[self.n]
    self.n+=1
    self.vystup_z_ks(proces)
    return res
```

- Rieši správne vzájomné vylúčenie
- Je ale komplikované a potrebuje poznať číslo procesu - uvedený kód je použiteľný len pre dva procesy, pre viac procesov ho treba upraviť
- V CPythone beží príliš pomaly (podrobnejšie vysvetlenie neskôr)

Nepreružiteľné "testuj a zamkni"

- Vráťme sa k myšlienke zamykacích premenných
- Keby sme vedeli v jednej strojovej inštrukcii skopírovať stav zamykacej premennej a zároveň ju nastaviť na 0 (false resp. zamknuté), tak by to mohlo fungovať.
- Áno, ale len keď máme jeden procesor (jadro), lebo v prípade viac procesorov môžu premennú zmeniť dvaja „naozaj naraz“.
- Pomohlo by mať inštrukciu, ktorá zároveň zablokuje zbernicu a cache pamäť pre ostatné procesory (jadrá).
- Strojové kódy zvyknú obsahovať takú inštrukciu – **nepreružiteľné testuj a zamkni**, alebo sa dá jej efekt dosiahnuť **inými inštrukciami**.
- Lenže na každom procesore je to inak...

Neprerušiteľné "testuj a zamkni" v procesoroch Intel

```
int odomknute = 1;

int vyber_shr() {
    __asm {
    l1: shr odomknute, 1
        jnc l1
    }
    int res = x[n];
    n++;
    odomknute = 1;
    return res;
}
```

V jednej inštrukcii zapíšem nulu do premennej a jej pôvodnú hodnotu (jeden bit, ktorý ma zaujíma) skopírujem do bitu CF.

súbor: kriticke_sekcie.cpp

- Registre má každý thread vlastné, preto ani výmena procesov tesne po vykonaní inštrukcie SHR nespôsobí problémy.

- Zabezpečí vzájomné vylúčenie, **ale len v prípade jednojadrového procesora**
- Problémom je keď dva thready *naozaj naraz* urobia svoju operáciu SHR keď je premenná odomknute=1 - jeden prenesie premennú do procesora potom druhý potom jeden urobí SHR, potom druhý a obaja zistia, že je odomknuté.
- Ale môže zostať aj *navždy zamknuté* keď je odomknute = 0 a jeden thread ide robiť SHR a prenesie premennú do pamäti, medzitým druhý thread do nej priradí 1, potom prvý thread spraví shr a zapíše do pamäti 0 a aj v CF bude mať nulu. Potom už nie je nikto, kto by dal do premennej jednotku.

Nepreružiteľné "testuj a zamkni" v procesoroch Intel (2)

```
int vyber_xchg() {
    __asm {
    l1: mov eax, 0
        lock xchg odomknute, eax
        cmp eax, 1
        jne l1
    }
    int res = x[n];
    n++;
    odomknute = 1;
    return res;
}
```

V jednej inštrukcii zapíšem nulu do premennej a jej pôvodnú hodnotu skopírujem do EAX.

súbor: kriticke_sekcie.cpp

Prefix LOCK hovorí, že počas vykonávania jednej inštrukcie nie je dovolený prístup do pamäti iným procesorom (jadrám).
Dá sa použiť len na niektoré inštrukcie (nie na SHR) a len ak majú pamäťový operand.
Ale: inštrukcia XCHG s pamäťovým operandom sa vždy vykonáva akoby bol pred ňou LOCK (takže tu LOCK vlastne netreba napísať).

- **Rieši správne vzájomné vylúčenie**
- Registre má každý thread vlastné, preto ani výmena procesov tesne po vykonaní inštrukcie XCHG nespôsobí problémy
- Vďaka LOCK sa nemôžu inštrukcie XCHG dvoch threadov vykonať *naozaj naraz*

Aktívne čakanie (Busy waiting)

- Petersonovo riešenie a Neprerušiteľné testuj a zamkni dosiahnu vzájomné vylúčenie
- Ale majú predsalen jeden problém: Aktívne čakanie (Busy waiting)
- Aktívne čakanie nastáva keď **proces alebo thread čaká na nejakú udalosť tak, že neodíde do stavu Čakajúci ale dookola testuje splnenie nejakej podmienky**
- Tým vlastne "zbytočne míňa" čas procesora
- Aktívne čakanie môžeme eliminovať len keď nám v tom pomôže operačný systém

Aktívne čakanie (Busy waiting) (2)

- Spomalenie vďaka aktívnemu čakaniu je oveľa výraznejšie na jednojadrovom procesore.
- Vtedy jeden thread počas svojho časového kvanta možno miliónkrát otestuje, či nie je "odomknuté", ale odomknúť mu môže len iný proces, ktorý sa dostane k procesoru až po ňom.
- Jednojadrový procesor je už rarita, ale **CPython** (Implementácia Pyhonu z Python.org) považuje jadro interpretra za kritickú sekciu - viď Global Interpreter Lock (GIL)
- Tým vlastne degraduje každý procesor na jednojadrový.

Aktívne čakanie (Busy waiting) (3)

- Iné implementácie Pythonu, napr. Iron Python alebo Jython, nemajú GIL.
- Skúsme spustiť ukážku Petersonovho riešenia z predošlých strán (súbor kriticke_sekcie.py, príkaz `skus_p(10000)`) v CPythone a v Iron Pythone:
- CPython: 86 sekúnd
- Iron Python: 0,07 sekundy (1228-krát rýchlejšie)

Riešenia vzájomného vylúčenia s podporou operačného systému alebo programovacieho jazyka

- Binárny semafor
- Všeobecný semafor
- Monitor

- Všetky tieto riešenia vyžadujú nové služby operačného systému, ktoré zabezpečia, že proces, ktorý čaká na vstup do kritickej sekcie nebude v stave "Busy waiting"
- Takéto služby nazývame **synchronizačné služby**.

Binárny semafór (Mutex)

- Autor: **E. W. Dijkstra** 1965
- Analógia s vlakovými oddielovými návestidlami (semaformi)
- Semafor je objekt, ktorý má vnútornú hodnotu 0 alebo 1 a pozná dve operácie **Down** a **Up**.
- Obe operácie sú implementované ako nedeliteľné v jadre operačného systému

	0	1
Down	Proces zostane čakať. čaká kým iný proces nevykoná Up	Hodnota:=0 Proces pokračuje v behu
Up	Ak niekto čaká na tento semafór, tak ho pusti ďalej (len jeden proces) Ak nikto nečaká, tak Hodnota:=1	Chybné použitie Binárneho semafóru

Binárny semafor v Pythone Lock

súbor: kriticke_sekcie.py

```
import threading
class Data_Semafor:
  def __init__(self,pocet):
    self.x = [i for i in range(pocet)]
    self.n = 0
    self.mutex = threading.Lock()
  def vyber(self):
    self.mutex.acquire()
    try:
      res=self.x[self.n]
      self.n+=1
    finally:
      self.mutex.release()
    return res
```

Lock je binárny semafor

znamená Down

znamená Up

finally prikazuje vykonať príkazy aj keby vznikla v bloku **try** výnimka

```
def vyber(self):
  with self.mutex:
    res=self.x[self.n]
    self.n+=1
  return res
```

Príkaz **with** nahrádza konštrukciu **try...finally** s volaniami acquire() a release().

Telo príkazu **with** je kritická sekcia ochránená semaforom, ktorého meno je uvedené za slovom with

V Pythone je aj trieda RLock, ktorá funguje ako binárny semafor, ale umožňuje zavolať veľakrát acquire v jednom threade bez zablokovania.

Binárny semafor v C

Windows, súbor: kriticke_sekcie.cpp

```
CRITICAL_SECTION cs;

int vyber_semafor() {
   EnterCriticalSection(&cs);
   int res = x[n];
   n++;
   LeaveCriticalSection(&cs);
   return res;
}

int main() {
   HANDLE h[2];

   InitializeCriticalSection(&cs);
     // treba pridat pre pouzitie vyber_semafor
....
}
```

Linux, súbor: kriticke_sekcie_linux.c

```
pthread_mutex_t mutex =
  PTHREAD_MUTEX_INITIALIZER;

int vyber_semafor() {
  pthread_mutex_lock(&mutex);
  int res = x[n];
  n++;
  pthread_mutex_unlock(&mutex);
  return res;
}
```

Všeobecný semafor

- Jeho hodnota je ľubovoľné celé nezáporné číslo
- Na riešenie vzájomného vylúčenia je zbytočne zložitý, ale môže sa použiť
- Používa sa hlavne na iné druhy synchronizácie

	0	>0
Down	Proces zostane čakať. čaká kým iný proces nevykoná Up	Hodnota:=Hodnota - 1 Proces pokračuje v behu
Up	Ak niekto čaká na tento semafór, tak ho pusti ďalej (len jeden proces) Ak nikto nečaká, tak Hodnota:=1	Hodnota:=Hodnota+1

Všeobecný semafor (2)
Spolupráca Producent-Konzument

súbor: producent_konzument.py

```
class Producent_konzument:
    def __init__(self,maxp,prod,konz):
        self.pole = []
        self.mutex = threading.Lock()
        self.prazdne = threading.Semaphore(value=maxp)
        self.plne = threading.Semaphore(value=0)
        self.produkuj = prod
        self.konzumuj = konz
```

```
def producent(self):
  while True:
    prvok = self.produkuj()
    self.prazdne.acquire()
    self.mutex.acquire()
    try:
      self.pole.append(prvok)
    finally:
      self.mutex.release()
    self.plne.release()
    if prvok == None: break
```

```
def konzument(self):
  while True:
    self.plne.acquire()
    self.mutex.acquire()
    try:
      prvok = self.pole.pop(0)
    finally:
      self.mutex.release()
    self.prazdne.release()
    if prvok == None: break
    self.konzumuj(prvok)
```

Deadlock (uviaznutie)

```
def konzument(self):
  while True:
    self.mutex.acquire()
    self.plne.acquire()
    ....

def producent(self):
  while True:
    prvok = self.produkuj()
    self.prazdne.acquire()
    self.mutex.acquire()
    ....
    self.plne.release()
```

Čo sa stane, ak spravíme v programe chybu a prehodíme dva riadky s volaniami Down?

Ak je pole prázdne tak konzument vojde do kritickej sekcie cez volanie self.mutex.acquire() a zostane čakať na self.plne.acquire().
Producent potom nemôže vojsť do svojej kritickej sekcie - zostane čakať na self.mutex.acquire() a teda nikdy nevykoná self.plne.release() na ktoré čaká konzument.

Takýto stav nazývame **Deadlock**

Množina procesov je v deadlocku ak každý z nich čaká na udalosť, ktorú môže spôsobiť len iný proces z tejto množiny.

Deadlock je *odvrátenou stranou* vzájomného vylúčenia a synchronizácie. Pri synchronizovaní procesov musíme vždy myslieť na možnosť jeho vzniku a presvedčiť sa, či nemôže vzniknúť.

Monitor a signal/wait

Autori: C. A. R. Hoare (1974) a Per Brinch Hansen (1975)

Má nahradiť "nízkoúrovňové" semafory konštrukciami vyššieho programovacieho jazyka. Programátor len označí kritické sekcie a kompilátor zabezpečí správne použitie synchronizačných mechanizmov daného operačného systému. Navyše pre čakanie a signalizovanie používa nový mechanizmus signal/wait

```
monitor ProducerConsumer
        condition full, empty;
        integer count;
        procedure insert(item: integer);
        begin
                if count = N then wait(full);
                insert_item(item);
                count := count + 1;
                if count = 1 then signal(empty)
        end;
        function remove: integer;
        begin
                if count = 0 then wait(empty);
                remove = remove_item;
                count := count - 1;
                if count = N - 1 then signal(full)
        end;
        count := 0;
end monitor;
```

```
procedure producer;
begin
        while true do
        begin
                item = produce_item;
                ProducerConsumer.insert(item)
        end
end;
procedure consumer;
begin
        while true do
        begin
                item = ProducerConsumer.remove;
                consume_item(item)
        end
end;
```

Ukážka je v jazyku Concurrent Pascal, ktorý navrhol Per Brinch Hansen

Monitor (2)

- Per Brinch Hansen implementoval monitory do ním navrhnutého jazyka Concurrent Pascal
- Z rozšírených programovacích jazykov má podobné prostriedky na implementáciu monitorov Java.
  V Jave môže mať metóda triedy atribút **synchronized**.
  Všetky takto označené metódy jednej triedy sa vykonávajú ako kritické sekcie.
- Python má triedu **Condition**, ktorá umožňuje posielať signály a čakať na signál.
  Spolu s príkazom **with** umožňujú implementovať štruktúry podobné monitorom (viď nasledujúca strana).

Monitory a signály v Pythone

súbor: producent_konzument.py

```
class Producent_konzument_monitor:
  def __init__(self,maxp,prod,konz):
    self.pole = []
    self.maxp = maxp
    self.mutex = threading.RLock()
    self.vybralsom = threading.Condition(self.mutex)
    self.vlozilsom = threading.Condition(self.mutex)
    self.produkuj = prod
    self.konzumuj = konz
```

```
def producent(self):
  while True:
    prvok = self.produkuj()
    with self.mutex:
      if len(self.pole) ==
            self.maxp:
        self.vybralsom.wait()
      self.pole.append(prvok)
      if len(self.pole) == 1:
        self.vlozilsom.notify()
    if prvok == None: break
```

```
def konzument(self):
  while True:
    with self.mutex:
      if len(self.pole) == 0:
        self.vlozilsom.wait()
      prvok = self.pole.pop(0)
      if len(self.pole) ==
            self.maxp-1:
        self.vybralsom.notify()
    if prvok == None: break
    self.konzumuj(prvok)
```

Monitory a signály v Pythone - problém

- Riešenie z predošleho slajdu funguje, pre jedného producenta a jedného konzumenta. **Pri väčšom počte je ale problém.**
- Keď je front prázdny, v kritickej sekcii je producent *p*, na vstup do kritickej sekcie čaká konzument *k1* a konzument *k2* čaká na self.vlozilsom.wait().

```
def konzument(self):
  while True:
    with self.mutex:
      if len(self.pole) == 0:
        self.vlozilsom.wait()
      prvok = self.pole.pop(0)
      if len(self.pole) ==
            self.maxp-1:
        self.vybralsom.notify()
    if prvok == None: break
    self.konzumuj(prvok)
```

súbor: producent_konzument.py

- Ak teraz *p* vykoná self.vlozilsom.notify(), tak k2 skončí s čakaním a bude s snažiť vrátiť do kritickej sekcie. Po vyjdení *p* z kritickej sekcie budú *k1* aj *k2* súperiť o vstup do nej. Ak sa to náhodou podarí skôr k1 tak zoberie z frontu vyprodukovaný prvok a keď neskôr vojde do kritickej *k2*, tak bude front prázdny.

Monitory a signály v Pythone – riešenie problému

súbor: producent_konzument.py

```
def producent(self):
  while True:
    prvok = self.produkuj()
    with self.mutex:
      while len(self.pole) ==
            self.maxp:
        self.vybralsom.wait()
      self.pole.append(prvok)
      if len(self.pole) == 1:
        self.vlozilsom.notify()
    if prvok == None: break
```

```
def konzument(self):
  while True:
    with self.mutex:
      while len(self.pole) == 0:
        self.vlozilsom.wait()
      prvok = self.pole.pop(0)
      if len(self.pole) ==
           self.maxp-1:
        self.vybralsom.notify()
  if prvok == None: break
  self.konzumuj(prvok)
```

- Odporúčané riešenie problému je použiť pri **while** namiesto **if**.
- Tým sa problém vyrieši, ale už program nie je taký elegantný, ako navrhovali tvorcovia monitorov.
- Zaviedli sme totiž doň prvok aktívneho čakania (ako veľký, to závisí od konkrétnej implementácie notify a wait).
- **Toto platí aj pre Javu** a iné známe implementácie notify/wait

Klasické príklady z komunikácie

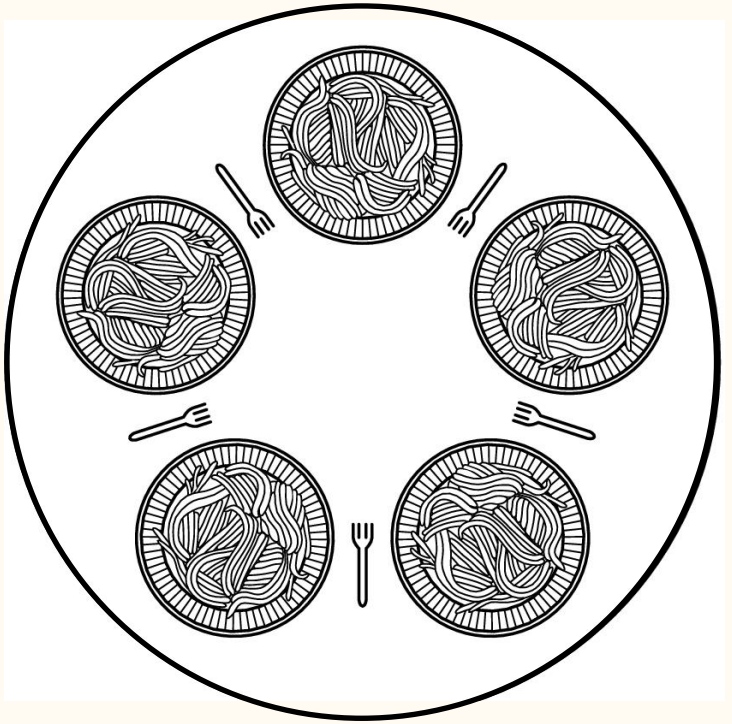
- Producent - Konzument
- Problém večerajúcich filozofov
- Čítanie - Zápis
- Problém spiaceho holiča

Riešenie sme ukazovali v podkapitole o všeobecných semaforoch

súbor: ukazky_algoritmy.py

Problém večerajúcich filozofov (1)

- Filozofi striedavo rozmýšľajú a jedia
- Na jedenie treba dve vidličky
- Berie najprv jednu potom druhú
- Ako zabránime vyhladovaniu (starvation) niektorého filozofa?
- Ako zabránime deadlocku?
- Ako zabezpečíme maximálne možné využitie vidličiek?



Problém večerajúcich filozofov (2)

```
def filozof(i):
    while True:
        rozmyslaj()
        zober(i)
        zober((i+1)%5)
        jedz()
        pusti(i)
        pusti((i+1)%5)
```

Toto riešenie spôsobí deadlock ak všetci filozofi naraz zoberú svoji prvú vidličku a nikto nebude môcť zobrať druhú

Filozof môže zobrať prvú vidličku a ak sa nepodarí zobrať druhú, tak prvú pustiť a (náhodnú) chvíľu počkať. To bude fungovať, ale chovanie algoritmu je ťažko predvídateľné.

Alebo, ak sa nepodarí zobrať druhú vidličku, tak obe položiť a skúsiť ich zobrať v opačnom poradí:
http://rosettacode.org/wiki/Dining_philosophers#Python

Môžeme dovoliť naraz jesť len jednému (kritická sekcia), ale to je plytvanie prostriedkami – vidličkami.

Môžeme aj modifikovať branie vidličiek tak, aby každý najprv zobral vidličku s menším číslom a potom čakal na vidličku s väčším číslom. Vtedy deadlock nenastane (prečo?), ale tiež klesne efektivita využitia vidličiek keď je veľa filozofov naraz hladných.

Problém večerajúcich filozofov (3)

súbor: ukazky_algoritmy.py

```
class Filozof(threading.Thread):

  zijeme = True

  def __init__(self, meno, mutex):
    threading.Thread.__init__(self)
    self.meno = meno
    self.mutex = mutex
    self.mozem = threading.Lock()
    self.mozem.acquire() # zacina na 0
    self.stav = "rozmysla"

  def nastav_susedov(self,la,pr):
    self.susedL = la
    self.susedP = pr
```

```
def run(self):
  while self.zijeme:
    self.rozmyslaj()
    self.zober()
    self.jedz()
    self.pusti()
```

Problém večerajúcich filozofov (4)

```
def zober(self):
    with self.mutex:
        self.stav = "hladny"
        self.test()
    self.mozem.acquire()
```

```
def pusti(self):
    with self.mutex:
        self.stav = "rozmysla"
        self.susedL.test()
        self.susedP.test()
```

```
def test(self):
    if self.stav == "hladny" and \
    self.susedL.stav != "je" and self.susedP.stav != "je":
        self.stav = "je"
        self.mozem.release()
```

```
def filozofuj():
  mutex = threading.Lock()
  mena = ('Aristoteles','Kant','Buddha','Marx', 'Russel')
  f = [Filozof(mena[i], mutex) for i in range(5)]
  for i in range(5): f[i].nastav_susedov(f[(i-1)%5],f[(i+1)%5])
  Filozof.zijeme = True
  for p in f: p.start()
  time.sleep(100)
  Filozof.zijeme = False
  for p in f: p.join()
```

súbor: ukazky_algoritmy.py

Čítanie - Zápis

- Procesy zdieľajú dáta
- Procesy môžu dáta čítať alebo písať
- Čítať môže naraz ľubovoľný počet procesov ak žiadny proces zároveň nezapisuje
- Písať ale môže naraz len jediný proces a to len vtedy ak nikto nezapisuje
- Podobné situácie bežne vznikajú v databázových aplikáciách

Čítanie - Zápis (2)

súbor: ukazky_algoritmy.py

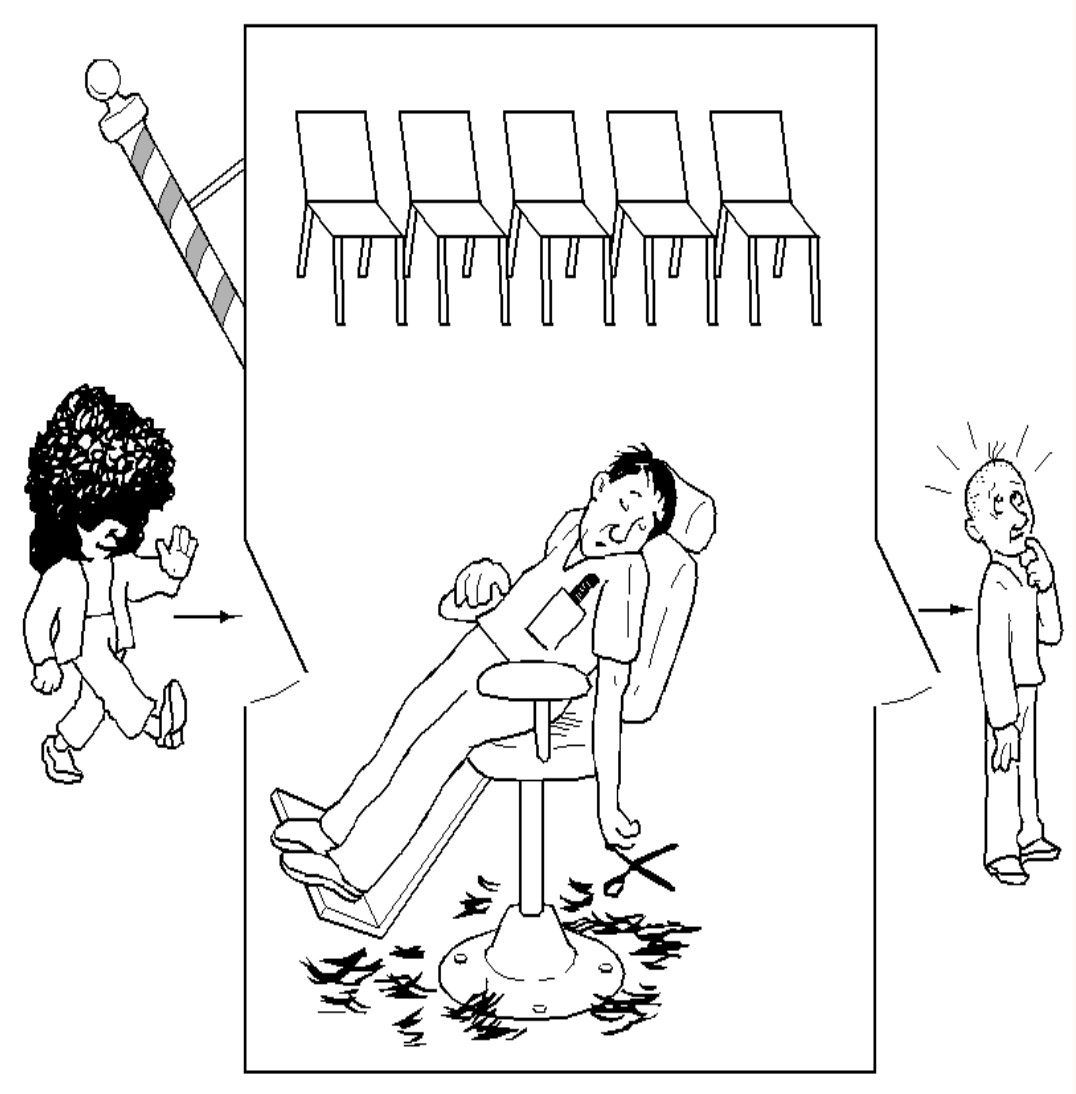
```
class CitanieZapis:
  def __init__(self):
    self.mutex = threading.Lock()
    self.databaza = threading.Lock()
    self.citaju = 0
  def pisatel_vojdi(self):
    self.databaza.acquire()
  def pisatel_vyjdi(self):
    self.databaza.release()
  def citatel_vojdi(self):
    with self.mutex:
      self.citaju += 1
      if self.citaju == 1:
        self.databaza.acquire()
  def citatel_vyjdi(self):
    with self.mutex:
      self.citaju -= 1
      if self.citaju == 0:
        self.databaza.release()
```

```
cz = CitanieZapis()
def pisatel():
  while True:
    vytvor_udaje()
    cz.pisatel_vojdi()
    try:
      zapis_udaje()
    finally:
      cz.pisatel_vyjdi()
def citatel():
  while True:
    cz.citatel_vojdi()
    try:
      citaj_udaje()
    finally:
      cz.citatel_vyjdi()
    spracuj_data()
```

Uvedený kód rieši problém Čítanie-Zápis, ale pisateľ je v nevýhode lebo pokým čaká na vstup do databázy tak ďalší noví čitatelia môžu stále prichádzať. Existujú aj lepšie riešenia pre pisateľa.

Problém spiaceho holiča

- Keď nemá prácu tak spí
- Prvý zákazník ho musí zobudiť
- Ďalší zákazník si sadne na voľnú stoličku
- Ak nie je voľná stolička tak zákazník odíde k inému holičovi



Problém spiaceho holiča (2)

súbor: ukazky_algoritmy.py

```
class SpiaciHolic:
  def __init__(self,stoliciek):
    self.stoliciek = stoliciek
    self.zakaznici = threading.Semaphore(value=0)
    self.holici = threading.Semaphore(value=0)
    self.mutex = threading.Lock()
    self.cakajuci = 0
  def ber_zakaznika(self):
    self.zakaznici.acquire()
    with self.mutex:
        self.cakajuci -= 1
        self.holici.release()
```

```
def cakaj_holica(self):
  self.mutex.acquire()
  if self.cakajuci < self.stoliciek:
    self.cakajuci += 1
    self.zakaznici.release()
    self.mutex.release()
    self.holici.acquire()
    return True
  else:
    self.mutex.release()
    return False
```

Všimnite si, že toto riešenie vyhovuje aj keď bude viac procesov Holic.

```
sh = SpiaciHolic(5)
```

```
def holic():
  while True:
    sh.ber_zakaznika()
    strihaj_zakaznika()
```

```
def zakaznik():
  while True:
    if sh.cakaj_holica():
      daj_sa_strihat();
```