Synchronizačné úlohy procesov

The Little Book of Semaphores
Second Edition
Version 2.1.5, Allen B. Downey

serializácia

randezvous

multiplex

bariera

niektoré ďalšie synchronizačné úlohy

Serializácia

- Serializácia:

  proces1: a1 proces2: b1

  a2 b2

  inštrukcia a2 sa musí vykonať pred inštrukciou b1

- Riešenie:

  Jedným semafórom inicializovaným na 0

proces1:	proces2:
a1	sem.wait()
a2	b1
sem.signal()	b2

Serializácia v cykle

- Serializácia:

```
 proces1:                    proces2:
while(1){                        while(1){
            a1                                b1
            a2                                b2
}                                }
```

inštrukcia a2 sa musí vždy vykonať pred inštrukciou b1

- Kedy je nasledovný kód riešením?:
  Jedným semafórom inicializovaným na 0

```
proces1:                    proces2:
while(1){                    while(1){
        a1                          sem.wait()
        a2                          b1
        sem.signal()                b2
 }                           }
```

- Riešenie:

  Dvoma semafórmi inicializovanými *sem1* na 1 a *sem2* na 0

```
proces1:                    proces2:
while(1){                    while(1){
    sem1.wait()                 sem2.wait()
    a1                          b1
    a2                          b2
    sem2.signal()               sem1.wait()
 }                           }
```

Rendezvous

- Rendezvous

  proces1: proces2:

  1 statement a1 1 statement b1

  2 statement a2 2 statement b2

  Treba zabezpečiť aby a1 nastalo pred b2 a b1 pred a2

  Synchronizačný problém sa volá rendezvous a znamená, že dva procesy sa musia začať vykonávať **súčasne**

- Riešenie:

  Dvoma semafórmi inicializovanými na 0

proces1:	proces2:
1 a1	1 b1
2 sem1.signal()	2 sem2.signal()
3 sem2.wait()	3 sem1.wait()
4 a2	4 b2

- alebo takto

proces1:	proces2:
1 a1	1 b1
2 sem2.wait()	2 sem2.signal()
3 sem1.signal()	3 sem1.wait()
4 a2	4 b2

- ale nikdy nie takto spôsobí deadlock

proces1:	proces2:
1 a1	1 b1
2 sem2.wait()	2 sem1.wait()
3 sem1.signal()	3 sem2.signal()
4 a2	4 b2

Mutual exclusion

- Vzájomné vylúčenie nad kritickou oblasťou
  semafór mutex sa inicializuje na 1

proces1:	proces2:
1 mutex.wait()	1 mutex.wait()
2 CS	2 CS
3 mutex.signal()	3 mutex.signal()

Multiplex

- Multiplex

  V CS môže byť najviac *n* procesov

  Semafór multiplex sa inicializuje na *n*

  $proces_i$

  1 multiplex.wait()

  2 CS

  3 multiplex.signal()

Bariera

- Rendezvous *n* procesov

  Každý proces vykonáva kód:

  ...

  statement r // rendezvous

  statement a1

  ...

  Synchronizačná podmienka:

  Žiaden proces nevykoná príkaz a1 pokiaľ všetky procesy nevykonali príkaz r

- Riešenie:

  n – počet procesov

  count – zdieľaná premenná určujúca koľko procesov vykonalo príkaz r, je inicializovaná na 0

  mutex – semafór pre vzájomné vylúčenie procesov pri prístupe k count, inicializovaný na 1

  barrier – semafór na ktorom procesy čakajú aby mohli vykonať a1, je inicializovaný na 0

Kód procesov-riešenie

statement r

mutex.wait()

count += 1

if (count = = n) barrier.signal()

mutex.signal()

barrier.wait()

barrier.signal()

statement a1

Bariera v cykle

```
while (1){
   statement r
   statement a1
}
```

Synchronizačná podmienka:

Kým všetky procesy nevykonajú niektorý z príkazov (r alebo a1) nesmie žiaden proces vykonať ďalší

Prečo je nie riešením nasledovný kód?

```
while(1){
  statement r
  mutex.wait()
  count += 1
  if (count = = n) barrier.signal()
  mutex.signal()
  barrier.wait()
  barrier.signal()
  statement a1
  mutex.wait()
  if (count = = n) {
    barrier.wait()
    count = 0
  }
  mutex.signal()
}
```

- Riešenie:

  n – počet procesov

  count – zdieľaná premenná určujúca koľko procesov vykonalo príkaz r, je inicializovaná na 0

  mutex – semafór pre vzájomné vylúčenie procesov pri prístupe k *count*, inicializovaný na 1

  barrier1 – semafór (inicializovaný na 0), na ktorom procesy čakajú aby mohli vykonať a1

  barrier2 – semafór (inicializovaný na 1), na ktorom procesy čakajú pokiaľ nebude uzamknutý semafór *barrier1*

```
while(1){
  statement r
  mutex.wait()
  count += 1
  if (count = = n) {
          barrier2.wait()
          barrier1.signal()
  }
  mutex.signal()

  barrier1.wait()
  barrier1.signal()
  statement a1
```

```
 mutex.wait()
 count -= 1
 if (count = = 0) {
        barrier1.wait()
        barrier2.signal()
 }
 mutex.signal()

 barrier2.wait()
 barrier2.signal()
}
```

Efektívnejšie riešenie

count – zdieľaná premenná určujúca koľko procesov vykonalo príkaz r, je inicializovaná na 0

mutex – semafór pre vzájomné vylúčenie procesov pri prístupe k *count*, inicializovaný na 1

barrier1 – semafór (inicializovaný na 0), na ktorom možno vykonať súčasne *n* operácií signál(*n*)

barrier2 – semafór (inicializovaný na 0), na ktorom možno vykonať súčasne *n* operácií signál(*n*)

```
while(1){
  statement r
  mutex.wait()
  count += 1
  if (count = = n) barrier1.signal(n)
  mutex.signal()
  barrier1.wait()
  statement a1
  mutex.wait()
  count -= 1
  if (count = = 0) barrier2.signal(n)
  mutex.signal()
  barrier2.wait()
}
```

Niektoré ďalšie synchronizačné úlohy a ich modely

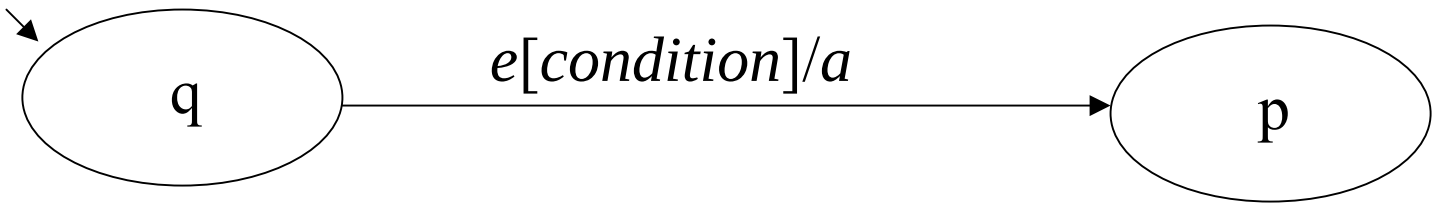
Prostriedky tvorby konkurentných udalostných modelov

- textové
- grafické (Petriho siete, statecharts, UML diagramy)
- konečné automaty (aj časové)
- mnoho iných špecificky orientovaných podľa aplikácie

UML syntax stavovo-prechodových diagramov

Stavy – tvoria ich hodnoty premenných (objektov) alebo stavy procesov

Prechody – vyjadrujú zmeny medzi stavmi



Prechod zo stavu q do stavu p sa uskutoční ak vznikne udalosť *e* je splnená podmienka *condition* a potom sa generuje akcia *a*

↘ je označený počiatočný stav

Producer-consumer

- Problém výrobca – konzument
- Zásobník s konečnou kapacitou *n*
- Podmienky:

  Proces výrobca vkladá do zásobníka

  Proces konzument vyberá zo zásobníka

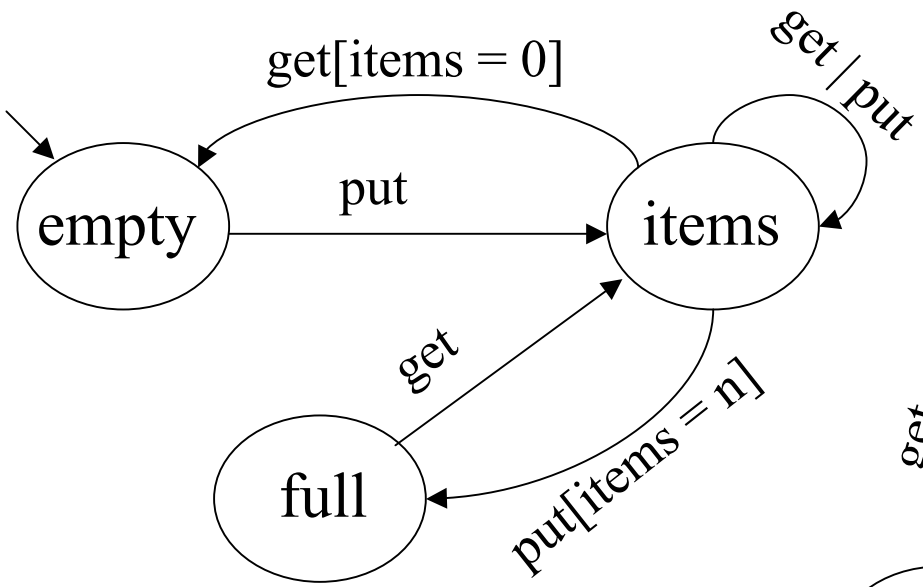
  Požiadavky:

  - Zásobník je zdieľaný preto musia mať oba procesy k nemu exkluzívny prístup

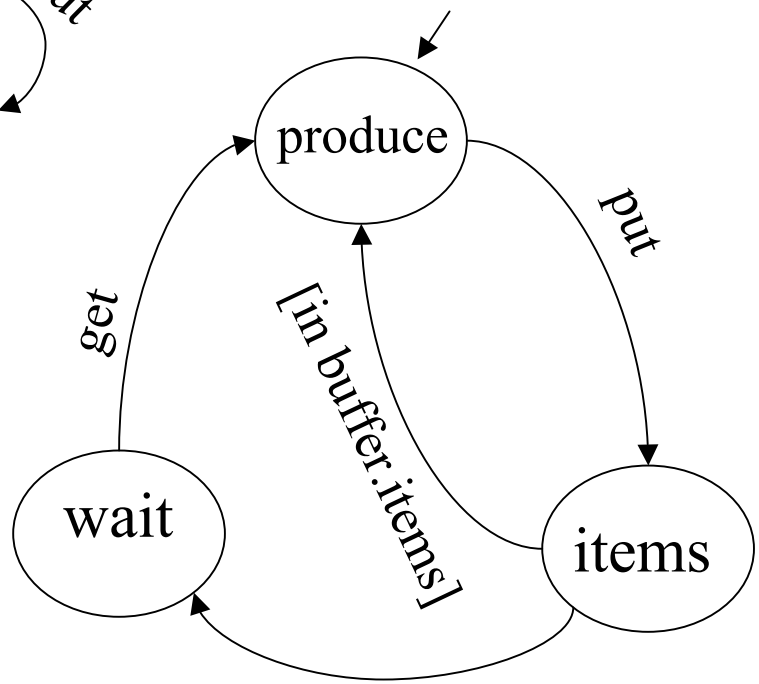
  - Konzument môže zo zásobníka vyberať len ak je nie prázdny

  - Výrobca môže do zásobníka vkladať len ak je nie plný

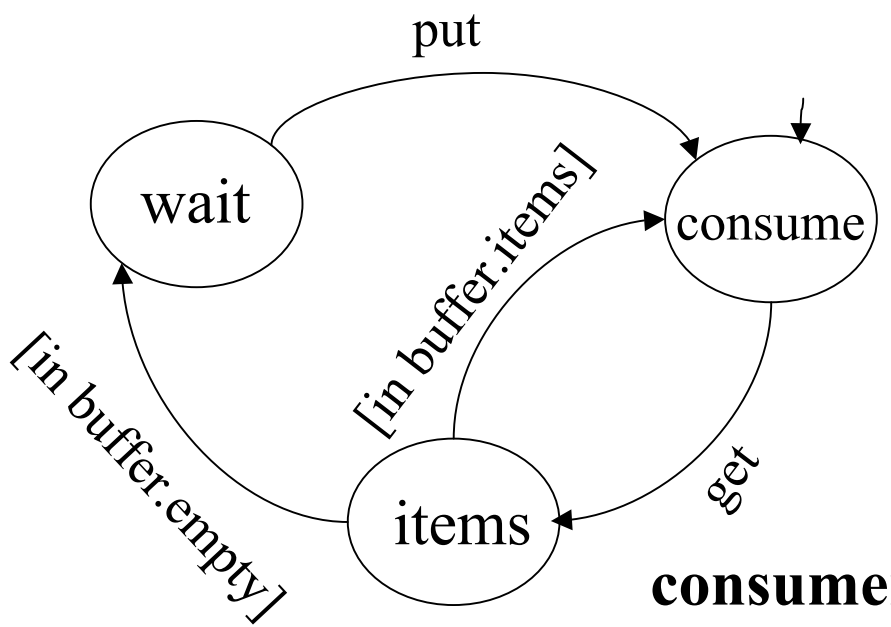
- Označenie udalostí:
  put – vkladanie do bufra
  get – výber z bufra
- Označenie akcií:
  items++ inkrementovanie prvkov v bufri
  items-- dekrementovanie
- Označenie podmienok: [in object.state] – model *object* je v stave *state*



buffer



producer



consumer

put – items++
get – items--

- Riešenie:

  mutex – semafór na vzájomné vylúčenie pri prístupe k zásobníku. Inicializovaný na 1

  semempty – semafór inicializovaný na 0

  semfull – semafór inicializovaný na veľkosť zásobníka *n*

Proces Konzument	Proces Výrobca
1 sempty.wait()	1 sfull.wait()
2 mutex.wait()	2 mutex.wait()
3 buffer.get()	3 buffer.add()
4 mutex.signal()	4 mutex.signal()
5 sfull. signal()	5 sempty. signal()

Readers-writers

- Problém čitatelia-zapisovatelia

  Podmienky:

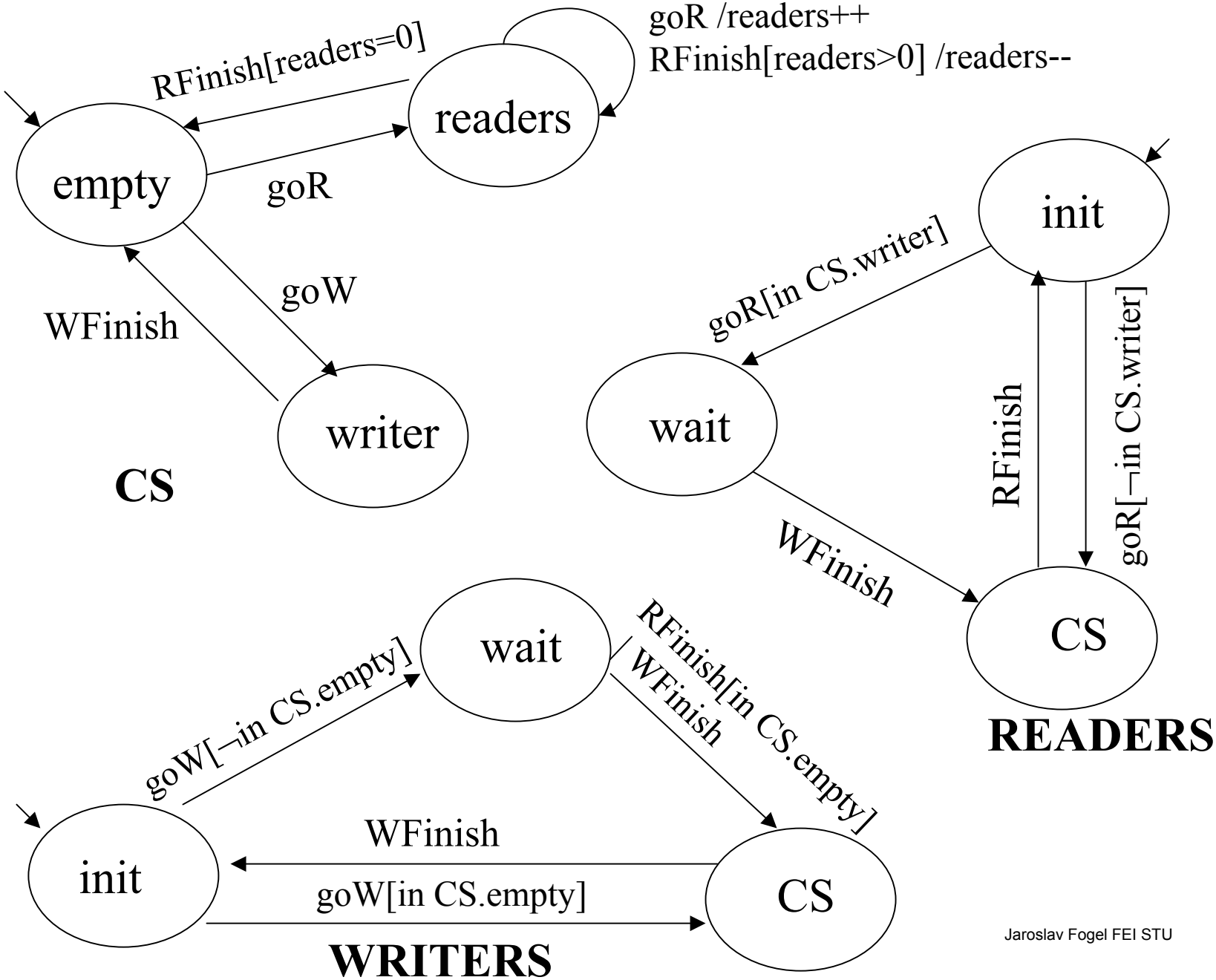
  Zapisovať môže len jeden proces

  Čítať môže súčasne viac procesov

  Ak proces zapisuje žiaden proces nesmie čítať

  Oblasť (pamäť, súbor), ku ktorej procesy pristupujú je kritická oblasť

- Označenie udalostí:
  goR – štart čítania
  goW – štart zapisovania
  RFinish – ukončenie čítania
  WFinish – ukončenie zápisu
- Označenie akcií:
  readers++ inkrementovanie počtu čitateľov
  readers-- dekrementovanie počtu čitateľov
- Označenie podmienok: [in object.state] – model *object* je v stave *state*


goR /readers++
RFinish[readers>0] /readers--
RFinish[readers=0]
readers
empty
goR
goW
WFinish
writer
CS
init
goR[in CS.writer]
wait
RFinish
goR[¬in CS.writer]
WFinish
CS
READERS
wait
goW[¬in CS.empty]
RFinish[in CS.empty]
WFinish
init
WFinish
goW[in CS.empty]
CS
WRITERS

- Riešenie:

  count_readers – zdieľaná premenná obsahujúca počet čítajúcich procesov. Inicializovaná na 0.

  mutex – semafór na ochranu pri prístupe k zdieľanej premenej readers. Inicializovaný je na 1.

  CsEmpty – semafór s hodnotou 1 ak v CS nie je žiaden proces inak 0. Inicializovaný je na 1.

Proces Writers:

1 CsEmpty.wait()

2 CS

3 CsEmpty.signal()

Proces Readers

1 mutex.wait()

2 count_readers +=1

3 if count_readers = = 1 CsEmpty.wait()

4 mutex.signal()

5 CS

6 mutex.wait()

7 count_readers -=1

8 if count_readers = = 0 CsEmpty.signal()

9 mutex.signal()

- Poznámky ku kódu

  - Ak je CS prázdna proces Writers do nej vstúpy čím vylúči vstup všetkých ostatných procesov

  - Kód pre Readers má navyše chránený prístup k premennej count_readers pomocou mutex a tiež

  CS zamkýna pred procesmi Writers ak je v nej aspoň jeden proces Readers a odomkýna keď posledný proces Readers skončí čítanie

  - Program má jednu zlú vlastnosť proces Writers sa do CS v prípade, že stále prichádzajú procesy Readers nemusí nikdy dostať (starvation – vyhladovanie)

Spôsob ako starvation vylúčiť

- Riešenie:

  count_readers – zdieľaná premenná obsahujúca počet čítajúcich prosesov

  mutex – semafór na ochranu pri prístupe k zdieľanej premenej readers. Inicializovaný je na 1.

  CsEmpty – semafór s hodnotou 1 ak v CS nie je žiaden proces inak 0. Inicializovaný je na 1.

  turn – semafór inicializovaný na 1

Proces Writers:	Proces Readers
1 turn.wait()	1 turn.wait()
	2 turn.signal()
2 CsEmpty.wait()	3 mutex.wait()
3 CS	4 count_readers +=1
4 turn.signal()	
5 CsEmpty.signal()	5 if count_readers = = 1 CsEmpty.wait()
	4 mutex.signal()
	5 CS
	6 mutex.wait()
	7 count_readers -=1
	8 if count_readers = = 0 CsEmpty.signal()
	9 mutex.signal()

- Poznámky ku kódu

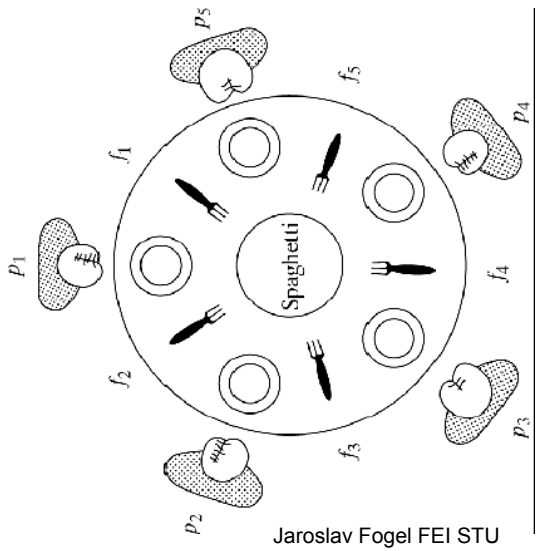
  - Ak Writers má záujem o vstup zatiaľ čo je už v CS nejaký Readers potom zostane čakať na riadku 2 s uzamknutým semafórom turn, takže ďalší Readers bude blokovaný na riadku 1 a do CS sa nedostane prv ako Writers

  - Ak posledný Readers odomkne semafór CsEmpty tým odblokuje čakujúceho Writera, ktorý do CS okamžite vstúpy, pretože žiaden z čakajúcich Readers sa nemôže dostať cez semafór turn

Obedujúci fylozofi

- Každý fylozof potrebuje na jedenie 2 vidličky
- Požiadavky
  –Zabrániť deadlocku
  –Zabezpečiť spravodlivosť: žiaden nesmie zostať hladný
  –Zabezpečiť konkurentnosť:

  susedia nemôžu jesť súčasne



- Riešenie:
- Stavy fylozofa

  state[i]: hungry (hladný)

  eating (práve je)

  thinking (rozmýšľa)
- *mutex* - semafór pre prístup do CS (nastavenie a testovanie stavov susedných fylozofov)
- s[i] - semafór pre každého fylozofa na signalizaciu získania oboch vidličiek

```
#define N                5                  /* number of philosophers */
#define LEFT             (i+N−1)%N          /* number of i's left neighbor */
#define RIGHT            (i+1)%N            /* number of i's right neighbor */
#define THINKING         0                  /* philosopher is thinking */
#define HUNGRY           1                  /* philosopher is trying to get forks */
#define EATING           2                  /* philosopher is eating */
typedef int semaphore;                      /* semaphores are a special kind of int */
int state[N];                               /* array to keep track of everyone's state */
semaphore mutex = 1;                        /* mutual exclusion for critical regions */
semaphore s[N];                             /* one semaphore per philosopher */

void philosopher(int i)                     /* i: philosopher number, from 0 to N−1 */
{
      while (TRUE) {                        /* repeat forever */
            think( );                       /* philosopher is thinking */
            take_forks(i);                  /* acquire two forks or block */
            eat( );                         /* yum-yum, spaghetti */
            put_forks(i);                   /* put both forks back on table */
      }
}
```

```
void take_forks(int i)                  /* i: philosopher number, from 0 to N–1 */
{
    down(&mutex);                       /* enter critical region */
    state[i] = HUNGRY;                  /* record fact that philosopher i is hungry */
    test(i);                            /* try to acquire 2 forks */
    up(&mutex);                         /* exit critical region */
    down(&s[i]);                        /* block if forks were not acquired */
}

void put_forks(i)                       /* i: philosopher number, from 0 to N–1 */
{
    down(&mutex);                       /* enter critical region */
    state[i] = THINKING;                /* philosopher has finished eating */
    test(LEFT);                         /* see if left neighbor can now eat */
    test(RIGHT);                        /* see if right neighbor can now eat */
    up(&mutex);                         /* exit critical region */
}

void test(i)                            /* i: philosopher number, from 0 to N–1 */
{
    if (state[i] == HUNGRY && state[LEFT] != EATING && state[RIGHT] != EATING) {
        state[i] = EATING;
        up(&s[i]);
    }
}
```

Problém spiaceho holiča

- Podmienky:

  - u holiča je *n* stoličiek pre čakanie zákazníkov

  - ak nie je žiaden zákazník holič spí

  - ak sú obsadené všetky stoličky zákazník nečaká a odíde

  - ak je holič obsadený a zákazník má voľnú stoličku sadne si a čaká kým príde na rad

  - ak holič spí zákazník ho zobudí

- Treba napísať program, ktorý koordinuje činnosť holiča a zákazníkov

Riešenie

zdielaná premenná *customers* určuje počet čakajúcich zákazníkov

mutex – semafór inicializovaný na 1 na ochranu pri prístupe k premennej customers

customer – semafór inicializovaný na 0, na ktorom čaká holič na príchod zákazníka

barber – semafór inicializovaný na 0, na ktorom čaká zákazník kým bude holič voľný

Kód zákazníka

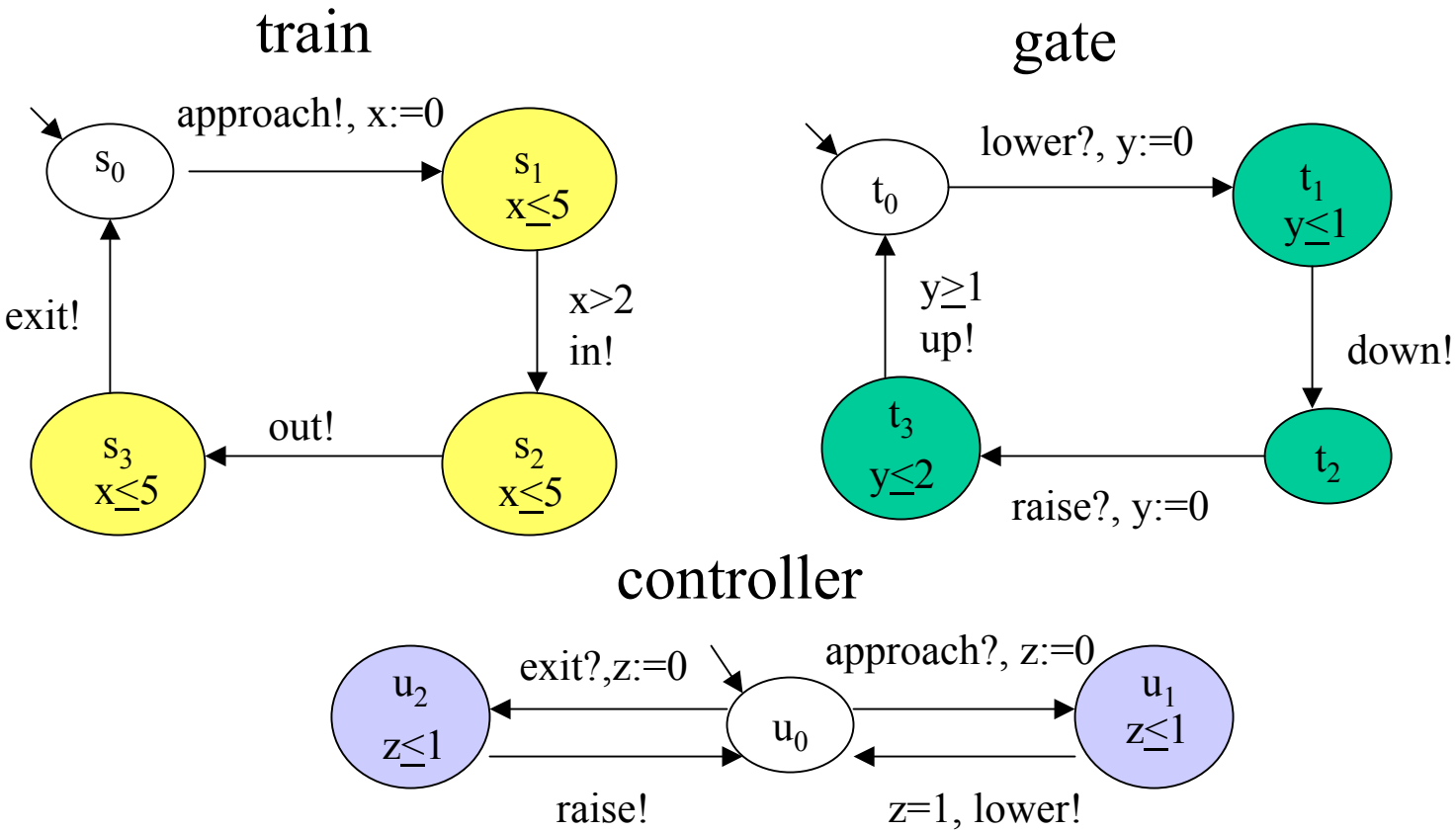
```
mutex.wait()
   if customers = = n {
            mutex.signal()
            go_away()
   }
   customers += 1
mutex.signal()
customer.signal()
barber.wait()
getHairCut()

mutex.wait()
customers -= 1
mutex.signal()
```

Kód holiča

```
1 customer.wait()
2 barber.signal()
3 cutHair()
```

Niektoré ďalšie reálne aplikácie

Train-gate example



Udalostný model železničného priecestia popísaný časovými automatmi.

Úloha

- Železničné priecestie, na ktorom riadi spúšťanie závor RS tak aby boli dodržané príslušné časové ohraničenia
- Signály:

  approach – priblíženie sa vlaku na danú vzdialenosť

  in – vlak vchádza na priecestie

  out – vlak priecestie opúšťa

  exit – vlak je mimo priecestia

  lower – povel na spustenie závor

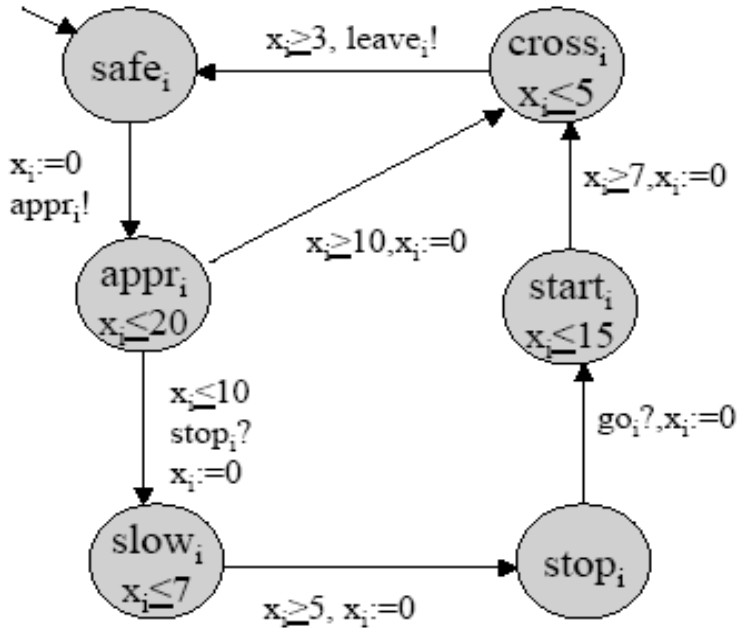
  down – závory sú spustené

  raise – povel na zdvihnutie závor

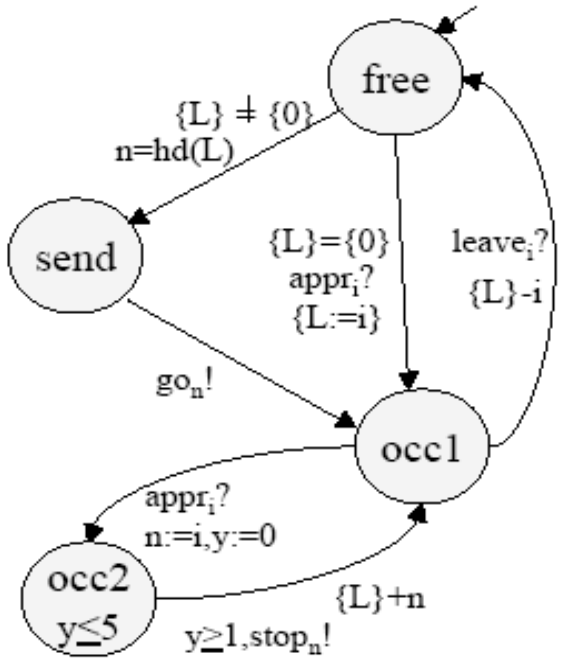
  up – závory sú zdvihnuté

A Railway Control System

Train$_i$, i=1,…,k



Controller



($||_i$Train$_i$||Controller)

Safety:

$$AG\neg(cross_i \wedge cross_j)$$

Úloha

- Do križovaky môže prísť *n* vlakov. Úlohou RS je riadiť prechod vlakov križovatkou tak, aby ich prechod bol bezpečný a boli dodržané dané časové ohraničenia
- Programová implementácia RS križovatky vlakov procesmi (alebo vláknami)
- Označenia signálov:

  $approach_i$ – i-ty vlak sa ku križovatke priblížil

  $leave_i$ – i-ty vlak križovatku opustil

  $stop_i$ – zastavenie i-teho vlaku

  go_i – i-ty vlak môže prejsť križovatku

  L – fronta vlakov

  hd(L) – funkcia, vyberie vlak, ktorý čaká na vstup najdlhšie