Operační systémy

Klasické synchronizační úlohy

Jan Trdlička

České vysoké učení technické v Praze, Fakulta informačních technologií
Katedra počítačových systémů

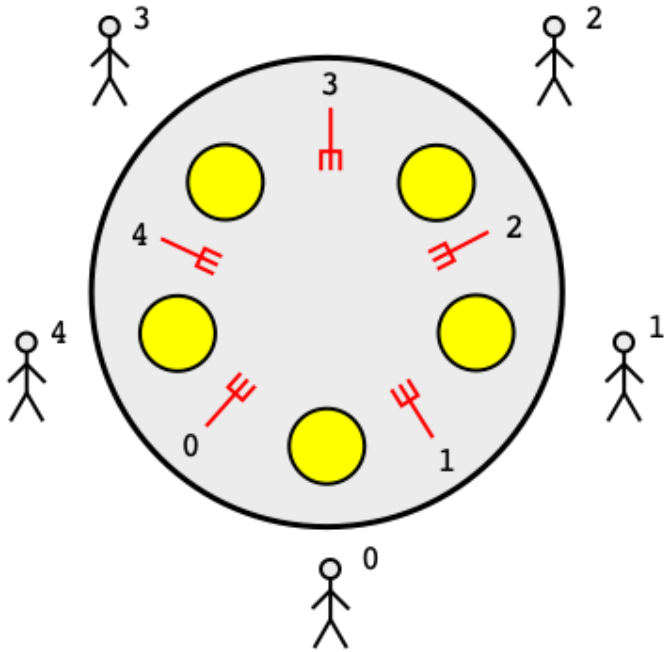
https://courses.fit.cvut.cz/BI-OSY

Obsah přednášky

Večeřící filosofové: Definice

- Klasický synchronizační problém, který reprezentuje situaci, kdy několik vláken soutěží o omezený počet prostředků.
- **Popis problému**
  - V systému je N filosofů, kteří sedí kolem kulatého stolu.
  - Před každým filosofem se nachází talíř s jídlem a mezi sousedními talíři je vždy jedna vidlička (celkem je N vidliček).
  - Pokud dostane filosof hlad, musí získat obě vidličky, které leží nalevo a napravo od jeho talíře.
  - Filosof se může nacházet ve třech stavech
    - ★ přemýšlí (nepotřebuje žádné prostředky),
    - ★ má hlad a pokouší se získat vidličky (snaží se alokovat prostředky),
    - ★ jí (používá prostředky).
- **Správné řešení:** nebude docházet k časově závislým chybám, uváznutí, livelocku, hladovění,...
- **Optimální řešení:** v jeden okamžik může jíst až $\lfloor N/2 \rfloor$ filosofů.
- Existuje několik řešení tohoto problému.

Večeřící filosofové: Naivní řešení



Naivní řešení

```
1  #define  N      5
2  #define  LEFT   (i % N)
3  #define  RIGHT  ((i+1) % N)
4
5  void philosopher(int i){
6     while (TRUE){
7        think();
8        take_fork(LEFT);
9        take_fork(RIGHT);
10       eat();
11       put_fork(LEFT);
12       put_fork(RIGHT);
13    }
14 }
```

- Filosofové jsou simulováni vlákny, která vykonávají funkci `philosopher()`.
- Funkce `take_fork(i)` se pokusí získat vidličku `i`
  - vidlička je volná ⇒ filosof získá vidličku,
  - vidlička je používána ⇒ filosof začne čekat na vidličku.
- Funkce `put_fork(i)` uvolní vidličku `i`.
- Bude toto řešení fungovat?

Večeřící filosofové: Naivní řešení

- Naivní řešení
  - Může selhat v situaci kdy všichni filosofové vezmou levou vidličku současně $\Rightarrow$ potom budou čekat až se uvolní pravá vidlička $\Rightarrow$ uváznutí(deadlock).
- Vylepšené řešení
  - Pokud není pravá vidlička k dispozici, filosof vrátí již alokovanou levou vidličku zpět na stůl a pokus o jídlo zopakuje "později".
  - Toto řešení může selhat v situaci kdy by všichni filosofové prováděli stejné kroky ve stejných okamžicích.
    1. Všichni filosofové vezmou levou vidličku.
    2. Uvolní levou vidličku, protože nemají k dispozici pravou vidličku.
    3. Tento postup budou opakovat po stejné době $\Rightarrow$ livelock.
  - Pravděpodobnost, že filosofové budou takto synchronně fungovat je malá, ale nelze vyloučit.

Večeřící filosofové: Správné řešení

- Na stůl s vidličkami se můžeme dívat jako na kritickou sekci ⇒ budeme synchronizovat známým způsobem.

```
1  mutex_t  mutex;
2
3  void philosopher(int i){
4     while (TRUE){
5        think();
6        mutex_lock(&mutex)
7           take_fork(LEFT);
8           take_fork(RIGHT);
9           eat();
10          put_fork(LEFT);
11          put_fork(RIGHT);
12       mutex_unlock(&mutex)
13    }
14 }
```

- Řešení neobsahuje časově závislé chyby ani uváznutí ⇒ správné.
- Pouze jeden filozof může jíst i když by mohlo jíst současně více filosofů ⇒ není optimální.

Večeřící filosofové: Správné optimální řešení

- Řešení pomocí mutexu a *N* semaforů.
- Je toto řešení správné (bez uváznutí,...) a optimální (jí více filosofů současně)?

```
#define  N     5
#define  LEFT  ((N+i-1) % N)
#define  RIGHT ((i+1)   % N)
typedef  enum  {thinking, hungry, eating} state_t;    /* state of philosopher */
state_t  state[N];
mutex_t  mutex;
sem_t    s[N];
for ( i = 0; i < N; i++ ) { state[i] = thinking; sem_init(&s[i], 0); };
```

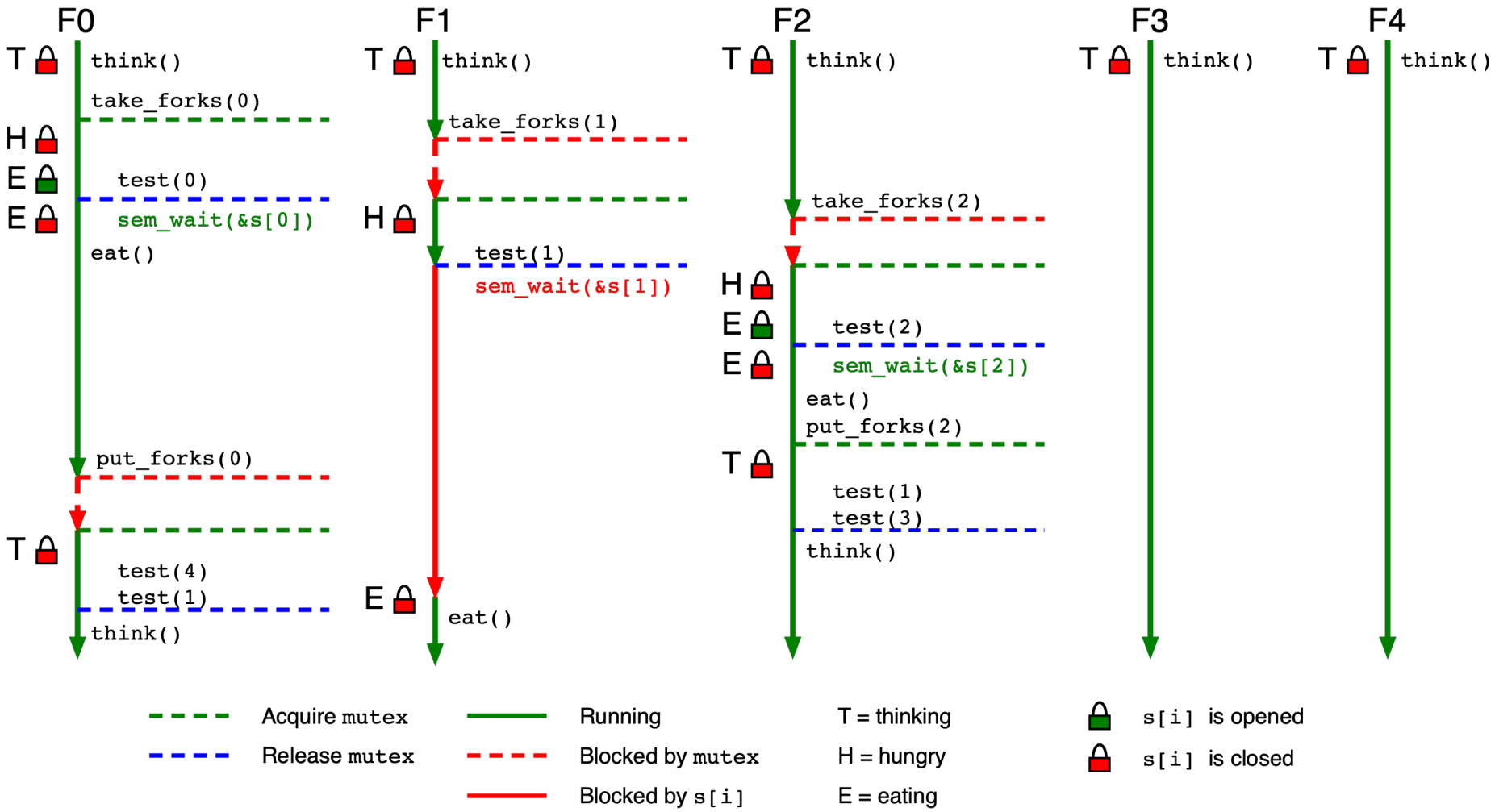
```
void philosopher(int i)  {
   while (TRUE) {
      think();
      take_forks(i);
      eat();
      put_forks(i);
   }
}
```

```
void take_forks(int i) {
   mutex_lock(&mutex);
      state[i] = hungry;
      test(i);
   mutex_unlock(&mutex);
   sem_wait(&s[i]);
}
```

```
void put_forks(int i) {
   mutex_lock(&mutex);
      state[i] = thinking;
      test(LEFT);
      test(RIGHT);
   mutex_unlock(&mutex);
}
```

```
void test(int i) {
   if (state[i] == hungry &&
      state[LEFT] != eating &&
      state[RIGHT] != eating)
   {
      state[i] = eating;
      sem_post(&s[i]);
   }
}
```

Večeřící filosofové: Správné optimální řešení



Večeřící filosofové: Správné optimální řešení

- Řešení pomocí mutexu a *N* podmíněných proměnných.
- Je toto řešení správné (bez uváznutí,...) a optimální (jí více filosofů současně)?

```
#define  N     5
#define  LEFT  ((N+i-1) % N)
#define  RIGHT ((i+1)   % N)
typedef  enum  {available, used} state_t;          /* state of fork */
state_t  fork[N];
mutex_t  mutex;
cond_t   cv[N];
for ( i = 0; i < N; i++ ) { fork[i] = available; }
```

```
void philosopher(int i)  {
   while (TRUE) {
      think();
      take_forks(i);
      eat();
      put_forks(i);
   }
}
```

```
void put_forks(int i) {
   mutex_lock(&mutex);
      fork[LEFT]  = available;
      fork[RIGHT] = available;
      cond_signal(&cv[LEFT]);
      cond_signal(&cv[RIGHT]);
   mutex_unlock(&mutex);
}
```

```
void take_forks(int i) {
   mutex_lock(&mutex);
      while ( ! forks_available(i) )
         cond_wait(&cv[i], &mutex);
       fork[LEFT]  = used;
       fork[RIGHT] = used;
   mutex_unlock(&mutex);
}
```

```
bool forks_available(int i) {
   if (  fork[LEFT]  == available &&
         fork[RIGHT] == available )
   {
      return true;
   }
   return false;
}
```

Čtenáři-písaři: Definice

- Klasický synchronizační problém, ve kterém dva typy vláken soutěží o přístup ke společnému prostředku.
- **Popis problému**
  - V systému je jeden sdílený prostředek a dva typy vláken
    - ★ čtenáři $R_0, \ldots, R_{M-1}$, kteří používají prostředek pouze pro čtení.
    - ★ písaři $W_0, \ldots, W_{N-1}$, kteří mohou obsah prostředku modifikovat.
  - Pouze jeden písař může modifikovat obsah prostředku v jednom okamžiku.
  - **Optimální řešení:** Více čtenářů může číst současně pokud žádný písař nepřistupuje k prostředku.
  - **Spravedlivé řešení:** Pokud písař/čtenář čeká na sdílený prostředek, pak by ho žádný jiný čtenář ani písař neměl předběhnout.

Čtenáři-písaři: Správné řešení

- Čtenáři/písaři jsou simulováni vlákny, která vykonávají funkce `reader()`/`writer()`.
- Na sdílený prostředek se můžeme dívat jako na kritickou sekci ⇒ budeme synchronizovat známým způsobem.

```
1  mutex_t  mutex;
```

```
1  void reader(void)
2  {
3     while (TRUE)
4     {
5        mutex_lock(&mutex);
6           read_data();
7        mutex_unlock(&mutex);
8        use_data();
9     }
10 }
```

```
1  void writer(void)
2  {
3     while (TRUE)
4     {
5        prepare_data();
6        mutex_lock(&mutex);
7           write_data();
8        mutex_unlock(&mutex);
9     }
10 }
```

- Řešení neobsahuje časově závislé chyby ani uváznutí ⇒ správné.
- Pouze jeden čtenář nebo pouze jeden písař může přistupovat ke sdílenému prostředku v jednom okamžiku. ⇒ není optimální

Čtenáři-písaři: Problém s hladověním

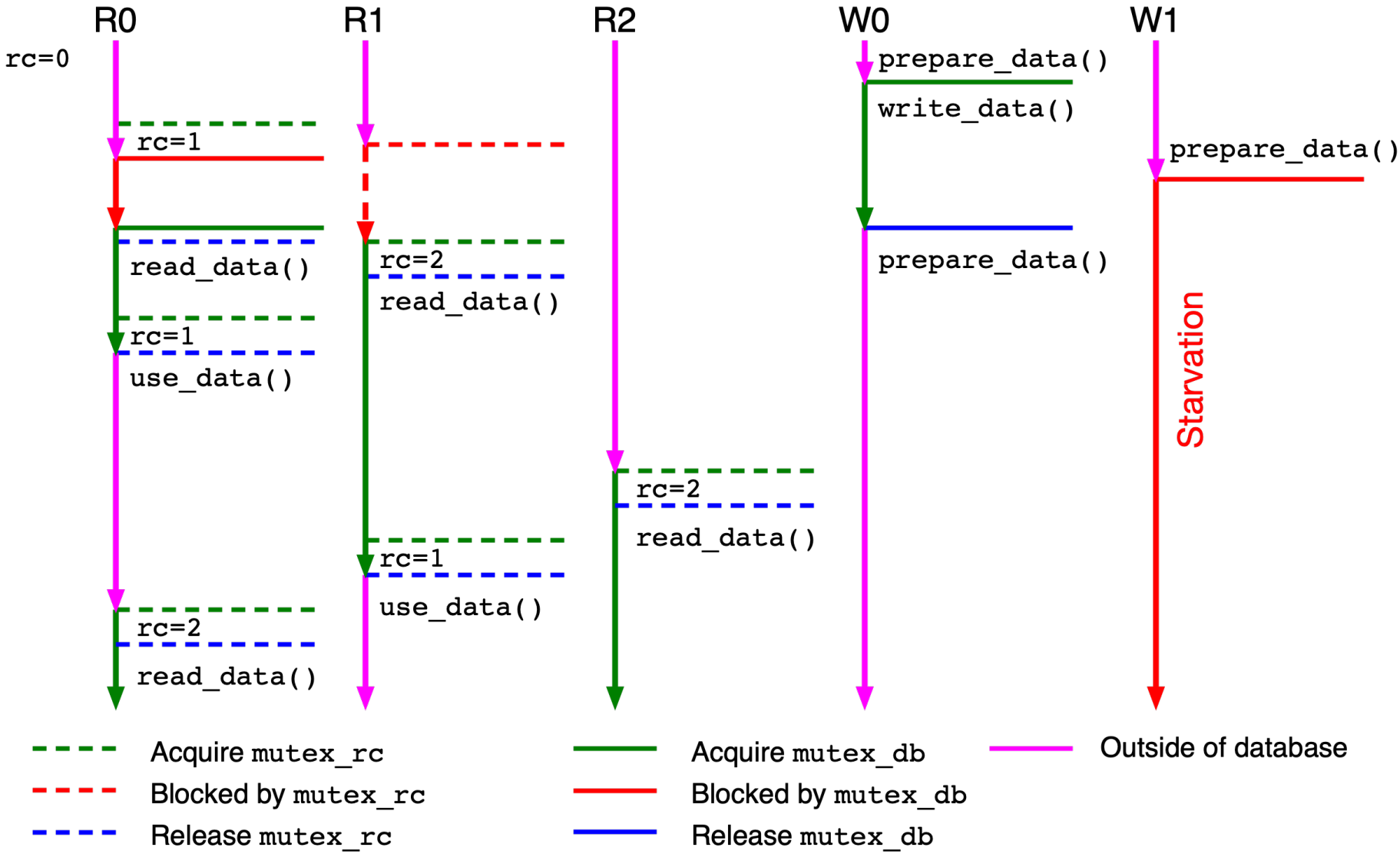
- Řešení pomocí dvou mutexů.
- Je to řešení
  - správné (bez časově závislých chyb, uváznutí,...),
  - optimální (více čtenářů může číst současně),
  - spravedlivé (čekající vlákna se nepředbíhají)?

```
1  int       rc = 0;          /* reader counter */
2  mutex_t   mutex_rc;        /* access to read counter */
3  mutex_t   mutex_db         /* access to database */
```

```
1  void reader(void)
2  {
3     while (TRUE)
4     {
5        mutex_lock(&mutex_rc);
6           rc = rc + 1;
7           if (rc == 1) mutex_lock(&mutex_db);
8        mutex_unlock(&mutex_rc);
9        read_data();
10       mutex_lock(&mutex_rc);
11          rc = rc - 1;
12          if (rc == 0) mutex_unlock(&mutex_db);
13       mutex_unlock(&mutex_rc);
14       use_data();
15    }
16 }
```

```
1  void writer(int i)
2  {
3     while (TRUE)
4     {
5        prepare_data();
6        mutex_lock(&mutex_db);
7           write_data();
8        mutex_unlock(&mutex_db);
9     }
10 }
```

Čtenáři-písaři: Problém s hladověním

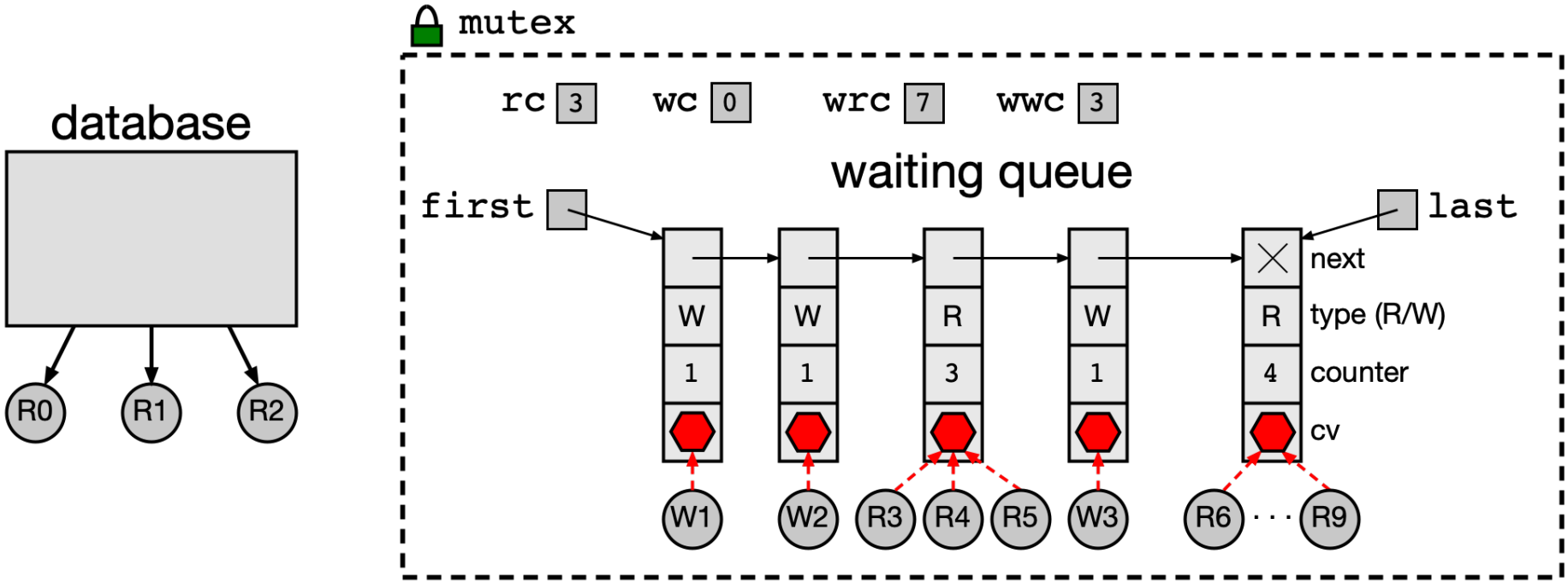


Čtenáři-písaři: Optimální spravedlivé řešení

- **Optimální řešení:** Více čtenářů může číst současně pokud žádný písař nepřistupuje k prostředku.
- **Spravedlivé řešení:** Pokud písař/čtenář čeká na sdílený prostředek, pak by ho žádný jiný čtenář ani písař neměl předběhnout.

  $\Rightarrow$ musíme si pamatovat v jakém pořadí čtenáři a písaři začínají čekat.
- Pokud vlákna jsou blokována na zámku, podmíněné proměnné nebo semaforu, tak většina API negarantuje jejich probuzení v pořadí FIFO.
- Jak to můžeme naimplementovat?

Čtenáři-písaři: Optimální spravedlivé řešení



- `mutex` chrání následující sdílené proměnné
  - reader counter `rc` = počet čtenářů, kteří právě čtou,
  - writer counter `wc` = počet písařů, kteří právě zapisují,
  - waiting reader counter `wrc` = počet čtenářů, kteří čekají na čtení,
  - waiting writer counter `wwc` = počet písařů, kteří čekají na zápis,
  - waiting queue = zřetězený seznam podmíněných proměnných, na kterých jsou blokováni čekající čtenáři/písaři.

Čtenáři-písaři: Optimální spravedlivé řešení

```
1  typedef  enum {
2     writer,
3     reader
4  } type_t;
5
6  typedef  struc {
7     item_t    *next;
8     type_t    type;
9     int       counter;
10    cond_t    cv;             /* condition variable    */
11 } item_t;
12
13 mutex_t  mutex;
14 int      rc, wc, wrc, wwc;
15 item_t   *first, *last;
```

Čtenáři-písaři: Optimální spravedlivé řešení

- V programu budeme používat následující funkce
  - `update_last_item()`
    - ★ Pokud tuto funkci zavolá čtenář a v poslední položce fronty čekají čtenáři, tak čtenář inkrementuje `counter` poslední položky o 1 a uspí se na podmíněné proměnné poslední položky.
    - ★ Jinak vlákno vytvoří novou položku, nastaví `type`, `counter` položky na 1, přidá ji na konec fronty a uspí se na na podmíněné proměnné poslední položky.
  - `update_first_item()`
    - ★ Vlákno dekrementuje `counter` v první položce fronty o 1.
    - ★ Pokud je `counter` roven 0 (nikdo již zde nečeká), tak zruší první položku fronty.
  - `wakeup_first_item()`
    - ★ Postupně probudí čtenáře/písaře, kteří jsou uspaní na podmíněné proměnné v první položce fronty.
    - ★ Buď můžeme v cyklu volat funkci `cond_signal()` nebo můžeme využít vlastností následujících reálných funkcí:
      Posix: `pthread_cond_broadcast()`,
      C++: `std::condition_variable::notify_all()`.

Čtenáři-písaři: Optimální spravedlivé řešení

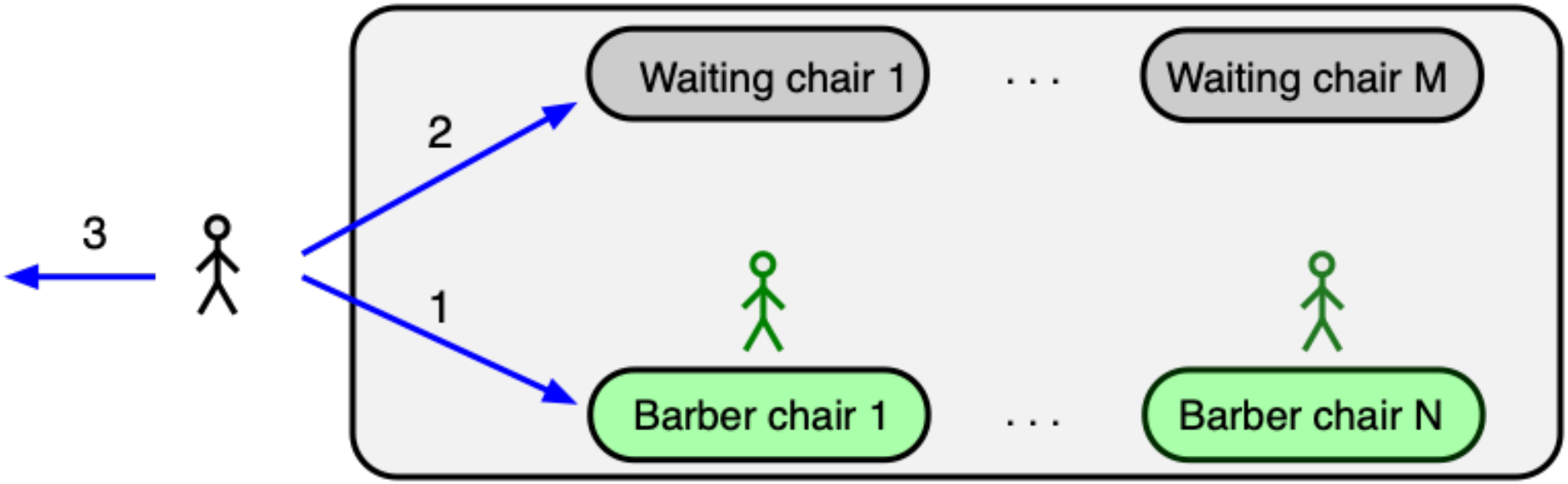
```
1  void reader(void) {
2     item_t   *item;
3     type_t   type = reader;
4
5     while (TRUE) {
6        mutex_lock(&mutex);
7           if ( wc > 0 || wwc > 0 ) {                    /* writer is there => go to wait */
8              wrc = wrc + 1;
9              item = update_last_item(last, type);       /* update the last item of the queue */
10             while ( item != first || wc > 0  )
11                cond_wait(&item->cv, &mutex);
12             wrc = wrc - 1;
13             update_first_item(first);                 /* update the first item of the queue */
14          }
15          rc = rc + 1;
16       mutex_unlock(&mutex);
17
18       read_data();
19
20       mutex_lock(&mutex);
21          rc = rc - 1;
22          if ( rc == 0 ) wakeup_first_item(first);    /* wake up writer in the first item */
23       mutex_unlock(&mutex);
24       use_data();
25    }
26 }
```

Čtenáři-písaři: Optimální spravedlivé řešení

```
1  void writer(void) {
2     item_t   *item;
3     type_t   type = writer;
4
5     while (TRUE) {
6        prepare_data();
7        mutex_lock(&mutex);
8           if ( rc > 0 || wc > 0 || wrc > 0 || wwc > 0 ) { /* anybody is there => go to wait */
9              wwc = wwc + 1;
10             item = update_last_item(last, type);      /* update the last item of the queue */
11             while ( item != first || rc > 0 || wc > 0 )
12                cond_wait(&item->cv, &mutex);
13             wwc = wwc - 1;
14             update_first_item(first);              /* update the first item of the queue */
15          }
16          wc = wc + 1;
17       mutex_unlock(&mutex);
18
19       write_data();
20
21       mutex_lock(&mutex);
22          wc = wc - 1;
23          wakeup_first_item(first);     /* wake up writer/readers in the first item */
24       mutex_unlock(&mutex);
25    }
26 }
```

Spící holiči: Definice

- V holičství je *N* holičů (barbers), *N* křesel k holení(barber chairs) a *M* křesel k čekání (waiting chairs) pro zákazníky.
- Pokud není žádný zákazník v holičství, holič sedne do křesla k holení a usne.
- Pokud přijde zákazník, potom mohou nastat tři situace
  1. holič je volný (holič spí), tak ho probudí a nechá se ostříhat,
  2. holič není volný, ale je volné místo v čekárně, tak počká,
  3. jinak opustí holičství.



Spící holiči: Správné optimální řešení

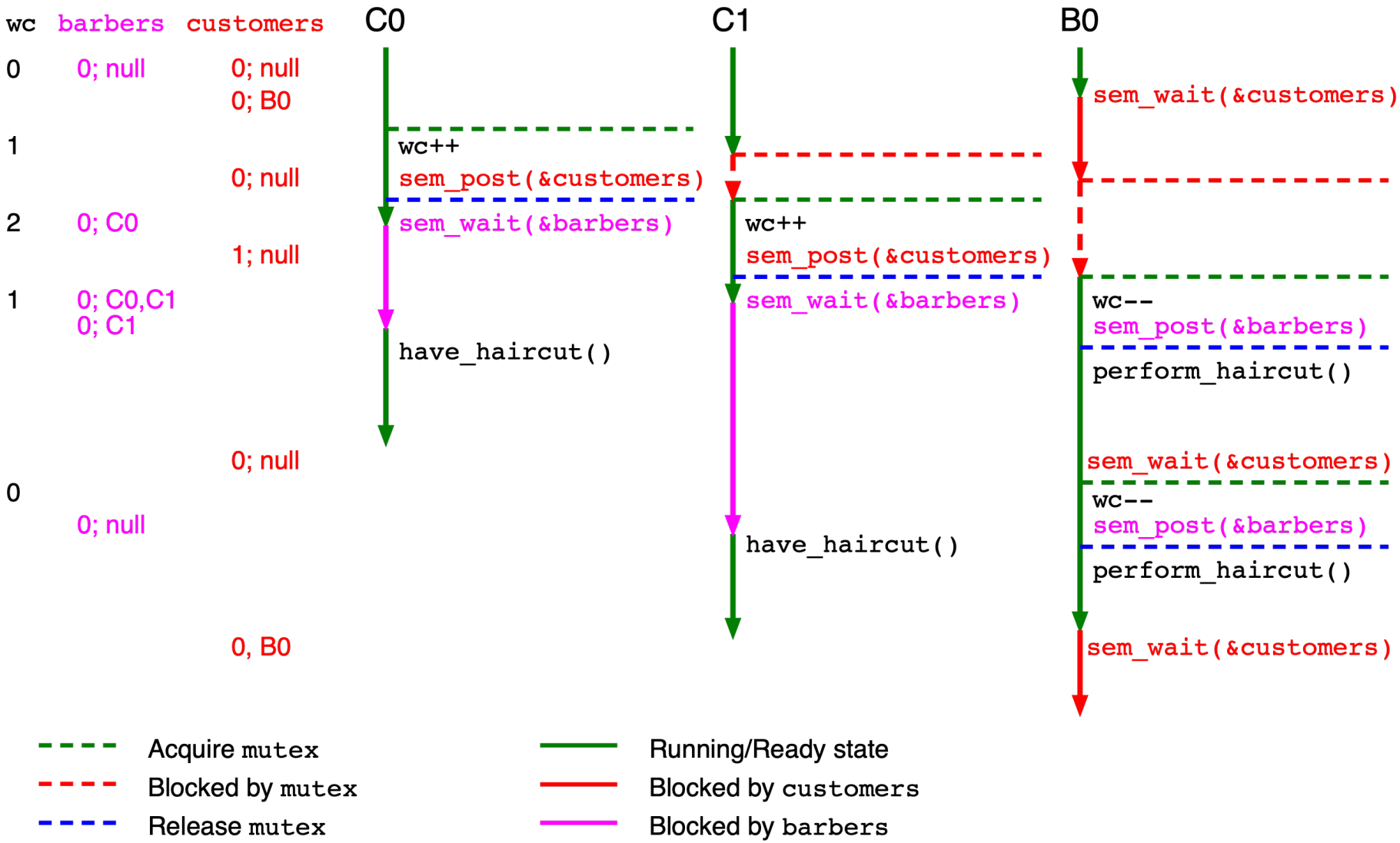
- Řešení pomocí jednoho mutexu a dvou semaforů.
- Zákazníci/holiči jsou simulováni vlákny, která vykonávají funkce `customer()`/`barber()`.
- Bude toto řešení
  - správné (neobsahuje časově závislé chyby, uváznutí,...),
  - optimální řešení (všichni holiči pracují, pokud je více zákazníků)?

```
1  #define  M  5                        /* waiting chairs                          */
2  int      wc = 0;                     /* customers are waiting (not being cut) */
3  mutex_t  mutex;                      /* for mutual exclusion                    */
4  sem_t    customers, barbers;
5  sem_init(customers, 0);              /* # of customers waiting for service    */
6  sem_init(barbers, 0);                /* # of barbers waiting for customers    */
```

```
1  void customer(void)
2  {
3     mutex_lock(&mutex);
4        if ( wc < M )
5        {
6           wc = wc + 1;
7           sem_post(&customes);
8           mutex_unlock(&mutex);
9           sem_wait(&barbers);
10          have_haircut();
11       }
12       else {
13          mutex_unlock(&mutex);
14       }
15 }
```

```
1  void barber(void)
2  {
3     while (TRUE)
4     {
5        sem_wait(&customers);
6        mutex_lock(&mutex)
7           wc = wc - 1;
8           sem_post(&barbers);
9        mutex_unlock(&mutex);
10       perform_haircut();
11    }
12 }
```

Spící holiči: Správné optimální řešení



Použité zdroje

1. A. S. Tanenbaum, H. Bos: *Modern Operating Systems (4th edition)*, Pearson, 2014.
2. W. Stallings: *Operating Systems: Internals and Design Principles (9th edition)*, Pearson, 2017.
3. A. Silberschatz, P. B. Galvin, G. Gagne: *Operating System Concepts (9th edition)*, Wiley, 2012.