Operační systémy

Synchronizace vláken

Jan Trdlička

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

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

Obsah přednášky

Paralelní výpočet – problémy

- **Bez použití synchronizace**
  - Časově závislé chyby (race conditions): situace, kdy více vláken používá (čte/zapisuje) společné sdílené prostředky během deterministického algoritmu a jeho výsledek je závislý na rychlosti vláken.
- **Při použití synchronizace**
  - Uváznutí (deadlock): situace, kdy několik vláken čeká na událost, kterou může vyvolat pouze jedno z čekajících vláken.
  - Livelock: situace, kdy několik vláken vykonává neužitečný výpočet (mění svůj stav), ale nemohou dokončit výpočet.
  - Hladovění (starvation): situace, kdy je vlákno ve stavu "Ready" předbíháno a nedostane se po "dlouhou" dobu k prostředkům.

Přehled synchronizačních nástrojů

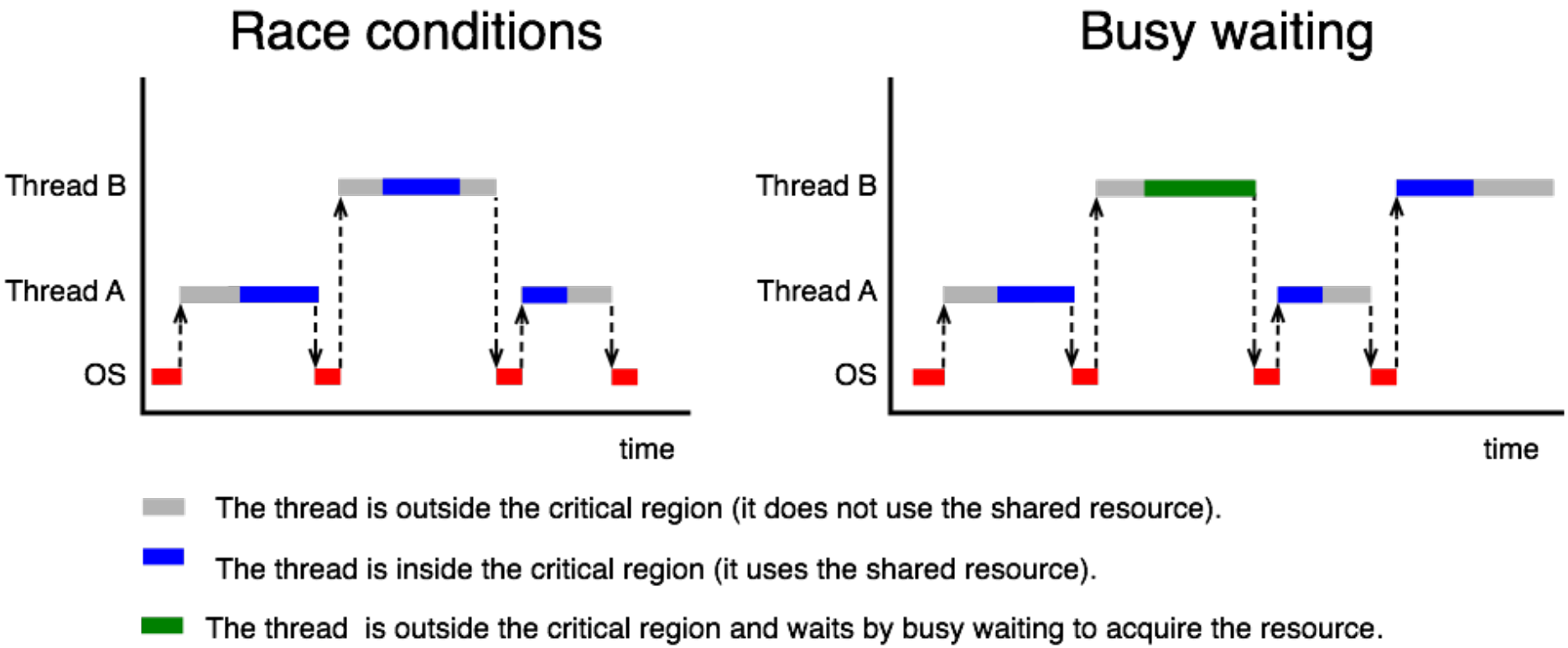
Hardware	SPARC v9: instrukce cas, ldstub, swap, . . . x86-64: instrukce xchg, cmpxchg,. . .
Jádro OS	Linux: atomic operation, spin locks, reader-writer locks, . . . Solaris: spin locks, adaptive locks, reader-writer locks, . . . MS Windows: executive dispatcher objects, slim reader-write locks, . . .
Aplikace	Unix OS: pipes, signals, System V semaphores, message queues, . . . MS Windows: mutexes, reader-writer locks, semaphores, events, . . . POSIX Thread: mutexes, condition variables, semaphores, . . . Languages: C++11, Java, . . .

Synchronizace kritické sekce – aktivní čekání

- Pouze jedno vlákno může být uvnitř kritické sekce
  $\Rightarrow$ ostatní vlákna musí počkat dokud se kritická sekce neuvolní.
- Toho lze docílit dvěma základními způsoby
  - pomocí aktivního čekání,
  - pomocí blokujících systémových volání/knihovních funkcí.
- **Aktivní čekání (busy waiting, spinning)**
  - Sdílená proměnná indikuje obsazenost kritické sekce (zamčená/odemčená).
  - Před vstupem do
    - ★ zamčené sekce: vlákno ve smyčce "aktivně" testuje aktuální hodnotu proměnné do okamžiku než se sekce uvolní,
    - ★ odemčené sekce: vlákno změní hodnotu sdílené proměnné (zamkne kritickou sekci) a vstoupí do sekce.
  - Po opuštění kritické sekce
    - ★ Vlákno změní hodnotu sdílené proměnné (odemkne sekci).

Synchronizace kritické sekce – aktivní čekání

- Předpokládejme, že dvě vlákna A a B přistupují ke společnému sdílenému prostředku.
- Následující obrázek ukazuje přístup bez synchronizace a synchronizaci pomocí aktivního čekání.



Příklad: Aktivní čekání pomocí proměnné `lock`

- Vzájemné vyloučení pomocí sdílené proměnné `lock`, kterou vlákno nastaví když vstupuje do kritické sekce.
- Kritická sekce je
  - odemčená, pokud `lock` má hodnotu 0,
  - zamčená, pokud `lock` má hodnotu 1.

```
1  int lock = 0;                                   /* control access to critical region(CR)*/
```

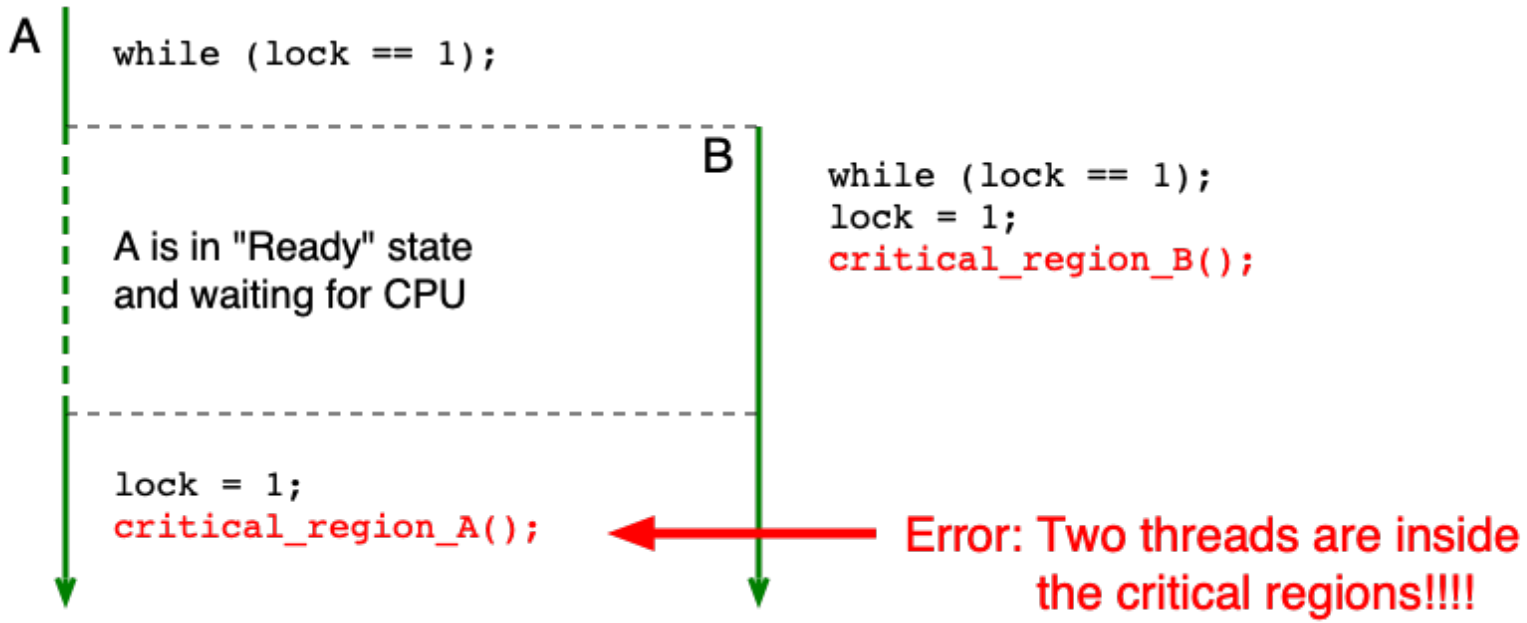
```
1  void thread_A(void)
2  {
3    while (TRUE)
4    {
5      ...
6      while ( lock == 1 );   /* busy waiting */
7      lock = 1;              /* enter CR */
8        critical_region_A();
9      lock = 0;;             /* leave CR */
10     ...
11   }
12 }
```

```
1  void thread_B(void)
2  {
3    while (TRUE)
4    {
5      ...
6      while ( lock == 1 );   /* busy waiting */
7      lock = 1;              /* enter CR */
8        critical_region_B();
9      lock = 0;;             /* leave CR */
10     ...
11   }
12 }
```

- Je to správné řešení?

Příklad: Aktivní čekání pomocí proměnné `lock`

- **Proč předchozí řešení je špatné?**
- Předpokládejme, že v kritické sekci není žádné vlákno (`lock` má hodnotu 0).

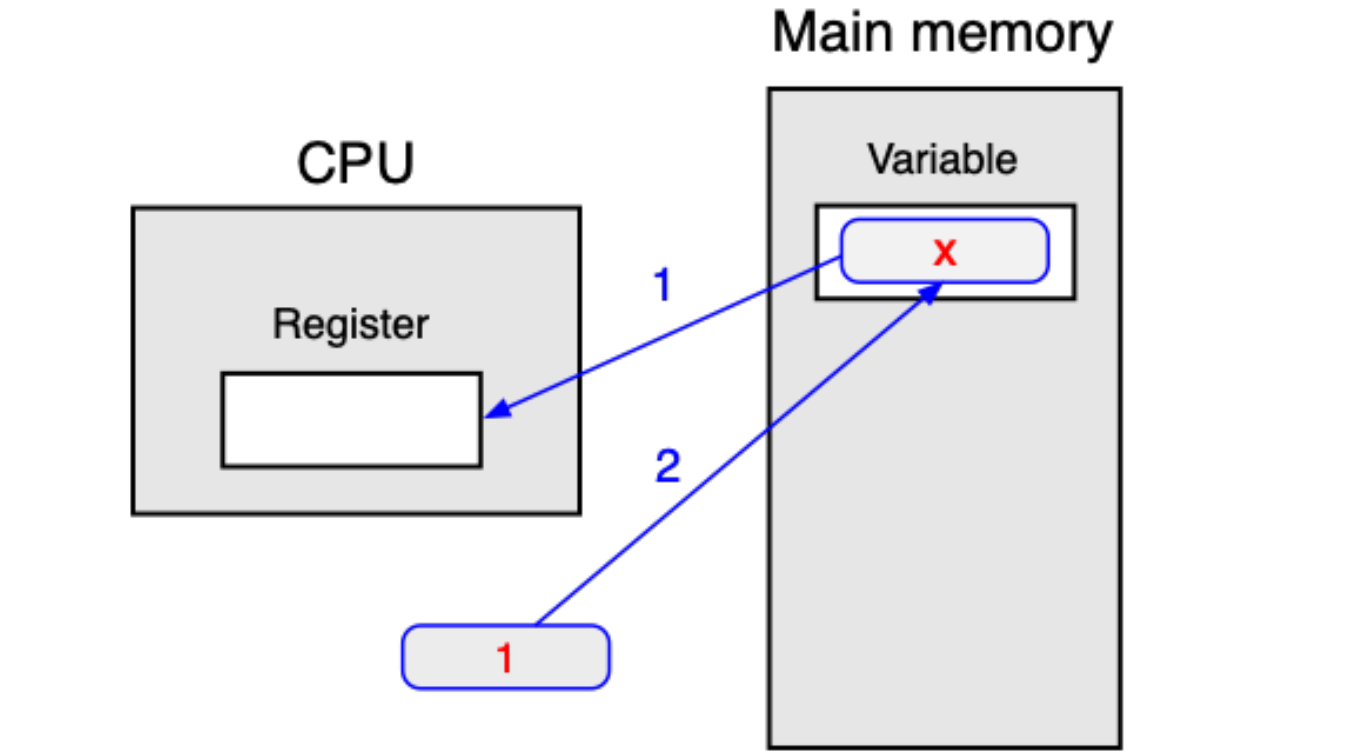


Aktivní čekání: Instrukce TSL

- **Instrukce TSL (test-and-set-lock)**
  - Hypotetická instrukce sloužící ke korektní implementaci aktivního čekání na hardwarové úrovni.
  - Samotná instrukce se skládá ze dvou kroků
    1. Načte obsah slova z dané adresy v paměti do registru.
    2. Nastaví obsah slova v paměti na nenulovou hodnotu (zamkne kritickou sekci).
  - Instrukce je atomická
    - ⋆ Žádné jiné vlákno během provádění této instrukce nemůže přistupovat ke slovu v paměti.
    - ⋆ Její implementace závisí na konkrétní hardwarové architektuře (např. zamčení paměťové sběrnice, . . .).

    $\Rightarrow$ korektní hardwarové řešení ve více-jádrových systémech se sdílenou pamětí.

Aktivní čekání: Instrukce TSL



The pair of operations 1 and 2 is executed atomically.

Aktivní čekaní: Instrukce TSL

Průchod kritickou sekcí se skládá z následujících kroků.

1. Před vstupem do kritické sekce vlákno testuje, zda je sekce odemčená a pokud není, tak aktivně čeká.

```
enter_region:
   tsl REGISTER, LOCK    | copy lock to register and set lock to 1
   cmp REGISTER, #0      | was lock zero?
   jne enter_region      | if it was nonzero, lock was set, so loop
   ret                   | return to caller; critical region entered
```

2. V kritické sekci vlákno bezpečně používá sdílený prostředek.
3. Po upuštění kritické sekce vlákno musí sekci odemknout.

```
leave_region:
   store LOCK,#0         | store a 0 in lock
   ret                   | return to caller
```

Aktivní čekání: Instrukce TSL

- ISA reálných procesorů většinou neobsahuje přímo instrukci TSL, ale nabízí nějakou její variantu, pomocí které můžeme implementovat aktivní čekání (různé varianty aktivních zámků).
  - SPARC v9
    - Load-store unsigned byte: `ldstub`,
    - Compare and swap: `cas`,
    - Swap register with memory: `swap`, ...
  - x86-64
    - Exchange register / memory with register: `xchg`,
    - Compare and exchange: `cmpxchg`, ...

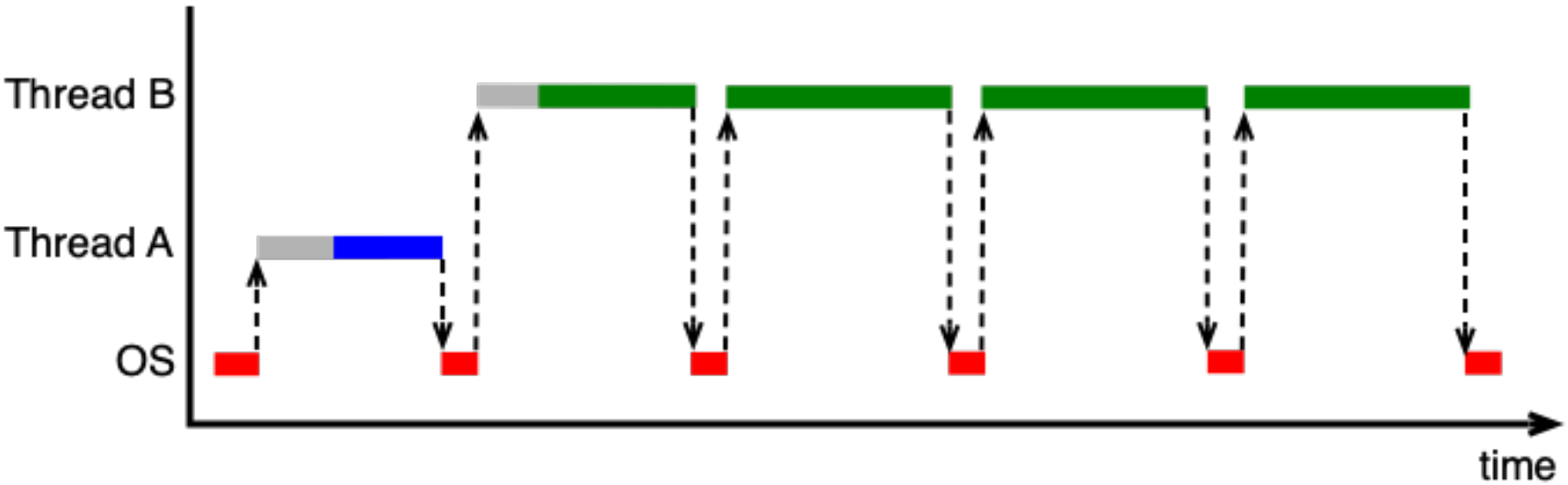
Aktivní čekání: Vlastnosti

- **Výhody**
  - Minimální režie, pokud vlákno nemusí čekat nebo čeká krátkou dobu před vstupem do kritické sekce.
- **Nevýhody**
  - Vlákno, které čeká před vstupem do kritické sekce, zatíží jedno jádro CPU na 100%.
  - V jistých situacích může aktivní čekání skončit uváznutím
    $\Rightarrow$ inverzní prioritní problém.
- **Inverzní prioritní problém**
  - Nutné podmínky
    - OS používá prioritní plánování vláken s fixní prioritou (priorita přiřazená vláknu se během jeho existence nemění).
    - CPU má omezený počet *n* jader.
  - Uváznutí nastane pokud
    - vlákno *A* má nízkou prioritu a nachází se v kritické sekci,
    - vlákno B má vyšší prioritu a čeká pomocí aktivního čekání na vstup do kritické sekce,
    - všechna jádra v systému jsou obsazena vlákny s vyšší prioritou.

$\Rightarrow$ Aktivní čekání se vyplatí, pokud se očekává krátká doba čekání na prostředek a nehrozí inverzní prioritní problém (např. synchronizace v jádru OS).

Aktivní čekání: Inverzní prioritní problém

1) Assume two threads A and B and shared critical region.
2) System has only one CPU with one core and uses the cheduling with fix priority.
3) The thread B has higher priority than the thread A.



The thread is outside the critical region (it does not use the shared resource).

The thread is inside the critical region (it uses the shared resource).

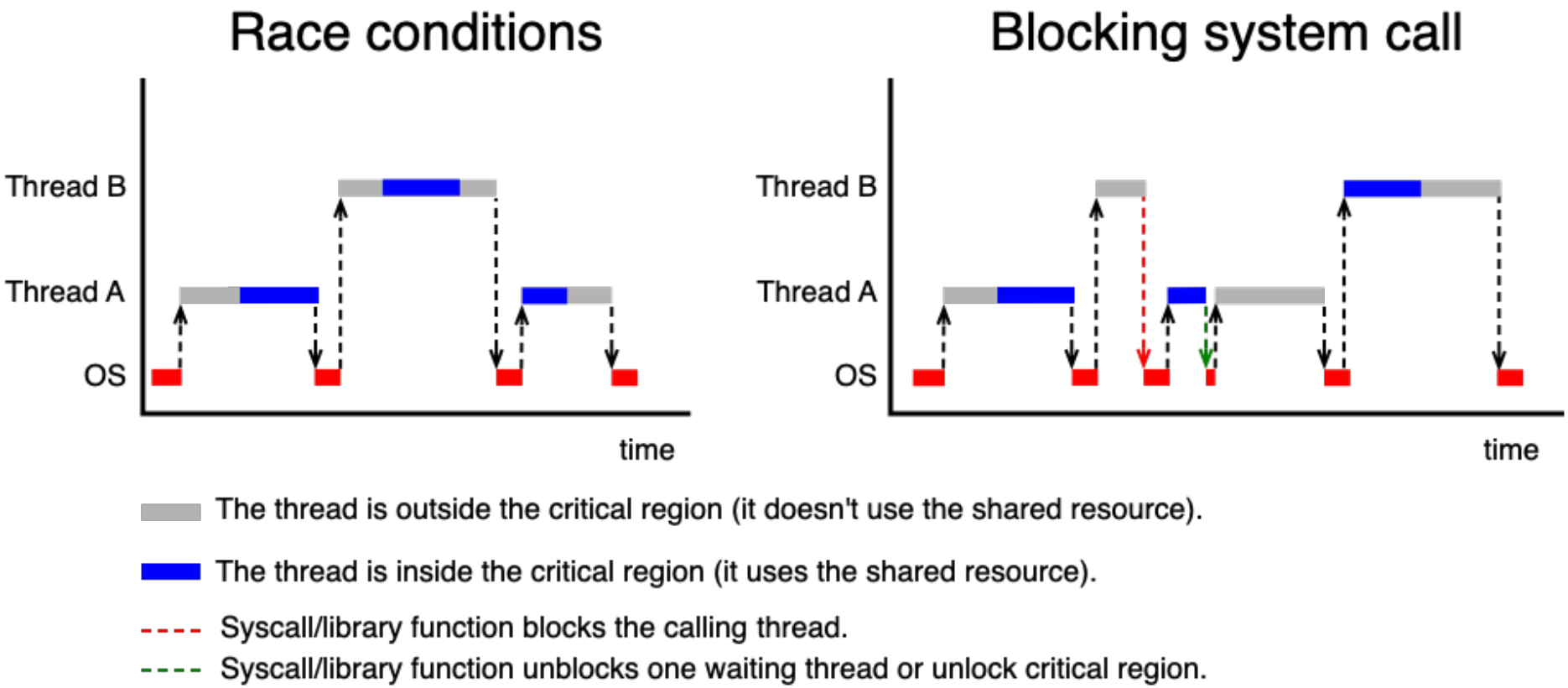
The thread is outside the critical region and waits by busy waiting to acquire the resource.

Synchronizace kritické sekce – blokující volání

- **Blokující systémové volání/knihovní funkce** je implementováno obvykle pomocí datových struktur, které umožňují
  - si pamatovat stav kritické sekce (odemčená/zamčená),
  - udržovat seznam vláken, která čekají na vstup do kritické sekce.
- **Před vstupem do kritické sekce**
  - zamčená sekce: vlákno provede systémové volání/knihovní funkci, které ho zablokuje (přepne jeho stav do stavu "Blocked"), a tím pádem vláknu přestane být přidělován procesor $\Rightarrow$ pasivně čeká na uvolnění sekce,
  - odemčená sekce: vlákno provede systémové volání/knihovní funkci, které ho nezablokuje ale pouze si zapamatuje, že sekce je zamčená, a vlákno "vstoupí" do sekce.
- **Po opuštění kritické sekce**
  - vlákno pomocí systémového volání/knihovní funkce probudí čekající vlákno/vlákna,
  - v případě, že již žádná vlákna nečekají, pak si zapamatuje, že sekce je odemčená.

Synchronizace kritické sekce – blokující volání

- Předpokládejme, že dvě vlákna A a B přistupují ke společnému sdílenému prostředku.
- Následující obrázek ukazuje přístup bez synchronizace a synchronizaci založenou na blokování.



Blokující volání: Zámky

- Zámek je často označován jako mutex (MUTual EXclusion lock) a pamatuje si
  - svůj stav (zamčený/odemčený),
  - kdo je jeho vlastníkem (vlákno, které ho zamklo),
  - množinu vláken, která jsou na něm blokovaná.
- Nad zámkem `mutex` jsou definovány atomické operace
  - `mutex_lock(mutex_t *mutex)`
    - ★ Pokud je `mutex` odemčený, tak ho zamkne. Volající vlákno se stane vlastníkem zámku.
    - ★ Pokud je `mutex` zamčený, tak zablokuje volající vlákno.
  - `mutex_unlock(mutex_t *mutex)`
    - ★ Měl by volat pouze vlastník zámku (jinak skončí chybou).
    - ★ Pokud jsou nějaká vlákna blokována zámkem, tak se jedno z nich probudí.
    - ★ Pokud již žádné vlákno není blokováno, tak se odemkne `mutex`.
- **Příklady**
  - POSIX: datový typ `pthread_mutex_t` s funkcemi `pthread_mutex_lock()`, `pthread_mutex_trylock`, `pthread_mutex_unlock`,...
  - C++: třídy `std::mutex`, `std::lock_guard`, `std::unique_lock`.

Blokující volání: Zámky

```
1  mutex_t mutex;                                   /* control access to critical region(CR)*/
```

```
1  void thread_A(void)
2  {
3    while (TRUE)
4    {
5      ...
6      mutex_lock(&mutex);          /* enter CR */
7        critical_region_A();
8      mutex_unlock(&mutex);        /* leave CR */
9      ...
10    }
11 }
```

```
1  void thread_B(void)
2  {
3    while (TRUE)
4    {
5      ...
6      mutex_lock(&mutex);          /* enter CR */
7        critical_region_B();
8      mutex_unlock(&mutex);        /* leave CR */
9      ...
10   }
11 }
```

Blokující volání: Vlastnosti

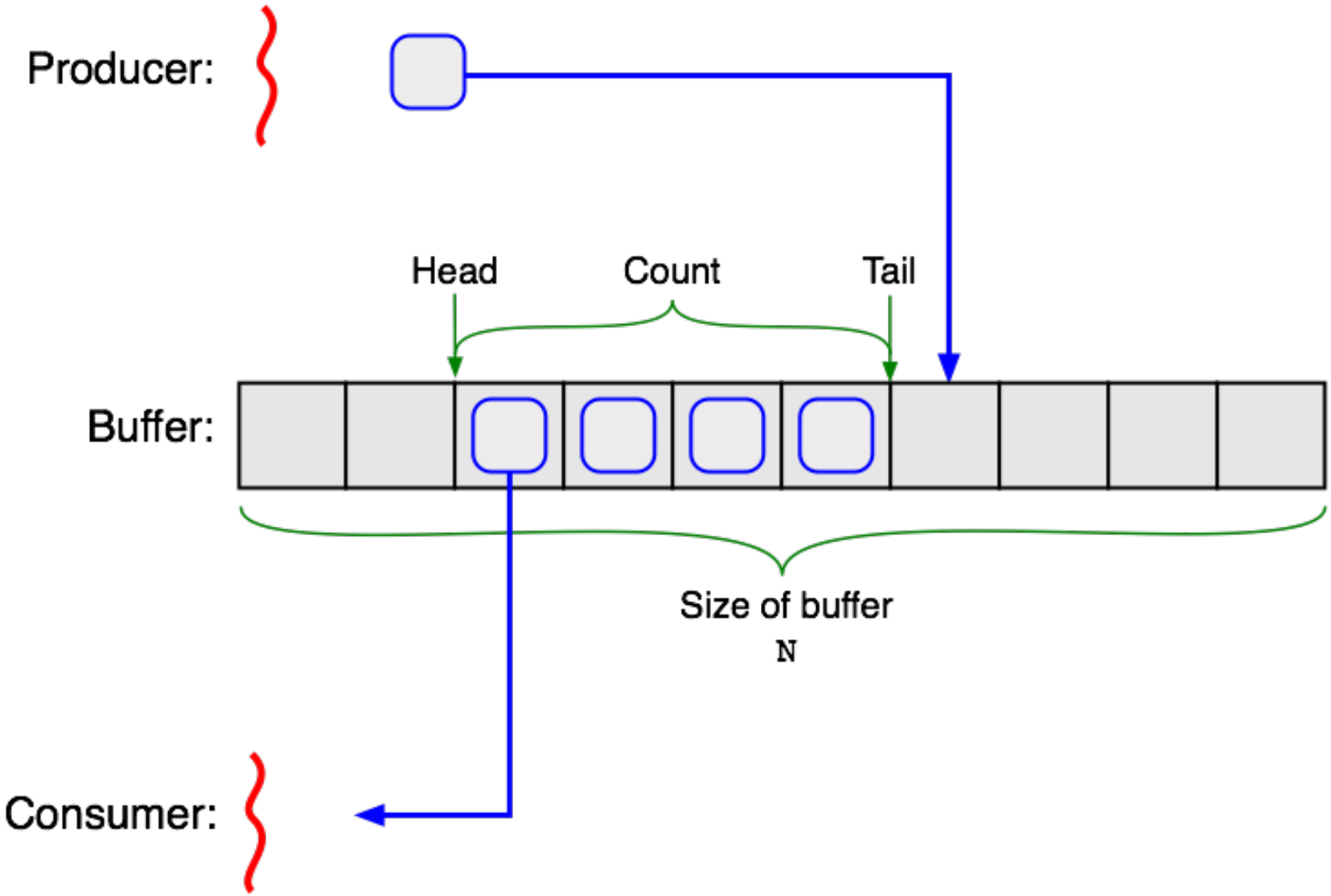
- **Výhody**
  - Čekání na vstup do kritické sekce nepředstavuje žádnou režii.
- **Nevýhody**
  - Začátek a ukončení představují určitou režii, neboť je nutné změnit stav vlákna v jádře OS.

$\Rightarrow$ Vyplatí se, pokud se očekává delší doba čekání na prostředek.

Synchronizační problém: Producent-konzument

- Klasický synchronizační problém, který přestavuje situaci, kdy si několik vláken vyměňuje data prostřednictvím např. sdílené paměti s omezenou velikostí.
- **Producent**
  - produkuje data a vkládá je do sdílené paměti (fronty) s omezenou velikostí.
- **Konzument**
  - vybírá data ze sdílené paměti (fronty).
- **Problémy**
  1. Musíme zajistit výlučný přístup při vkládání/vybírání dat z fronty.
  2. Pokud je fronta prázdná $\Rightarrow$ musíme zablokovat konzumenta.
  3. Pokud je fronta plná $\Rightarrow$ musíme zablokovat producenta.

Synchronizační problém: Producent-konzument



Příklad: Blokující volání `wait()` a `signal()`

- Předpokládejme, že máme k dispozici následující hypotetické funkce
  - `wait()`
    - ★ Systémové volání, které zablokuje volající vlákno (nastaví jeho stav na "Blocked").
  - `signal(thread_t *thread)`
    - ★ Pokud je vlákno `thread` ve stavu "Blocked", tak se jeho stav změní na "Ready" (bude mu opět přidělováno CPU).
- V následujícím příkladě se pokusíme vyřešit problémy 2 a 3 v úloze producent-konzument pomocí těchto funkcí.
- Je řešení správné?

Příklad: Producent-konzument s `wait()` a `signal()`

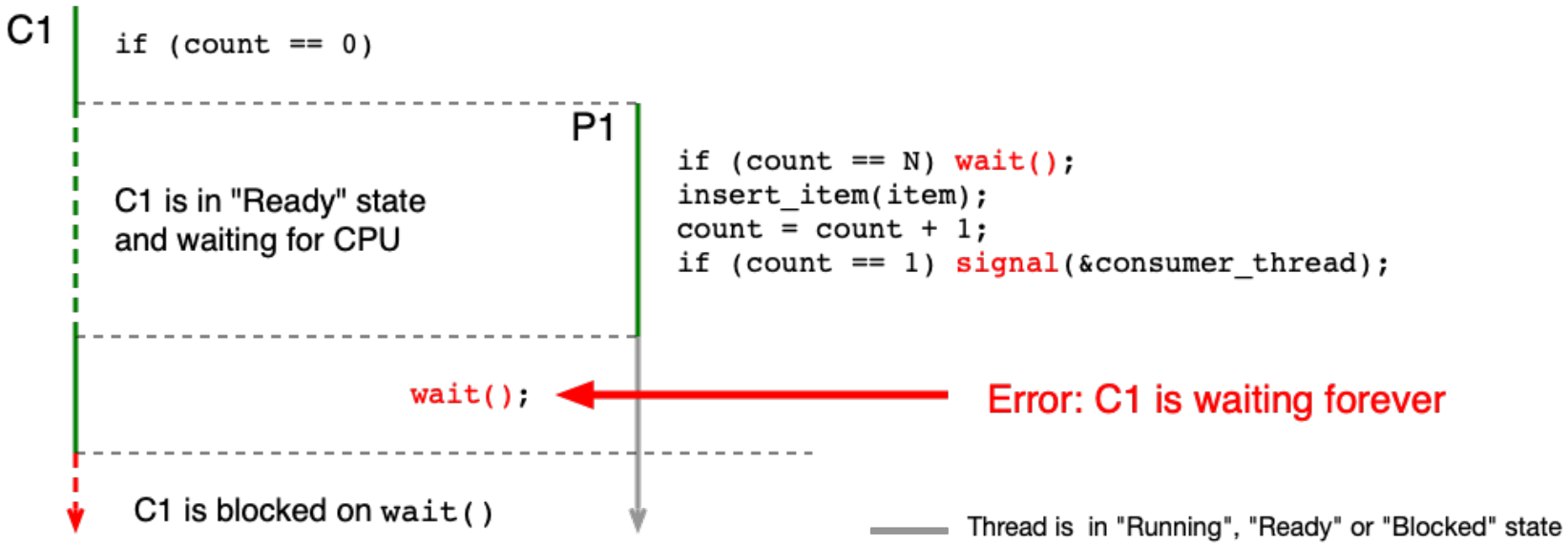
```
1  #define  N 100                                    /* number of slots in the buffer */
2  int      count = 0;                               /* number of items in buffer */
```

```
1  void producer(void)  {
2     item_t item;
3     while (TRUE)    {
4        item = produce_item();
5        if (count == N) wait();                     /* if buffer is full, go to sleep */
6        insert_item(item);
7        count = count + 1;
8        if (count == 1) signal(&consumer_thread);   /* was buffer empty? */
9     }
10 }
```

```
1  void consumer(void)  {
2     item_t item;
3     while (TRUE)    {
4        if (count == 0) wait();                     /* if buffer is empty, go to sleep */
5        remove_item(&item)
6        count = count - 1;
7        if (count == N - 1) signal(&producer_thread);   /* was buffer full? */
8        consume_item(&item);
9     }
10 }
```

Příklad: Producent-konzument s `wait()` a `signal()`

- **Proč předchozí řešení je špatné?**
- Předpokládejme, že fronta je prázdná (`count=0`) a existuje jeden producent a jeden konzument.



- **Vyřešil by se problém pokud bychom se pokusili vždy**
  - při vložení prvku probudit konzumenta,
  - při odebrání prvku probudit producenta?

Blokující volání: Podmíněné proměnné

- Podmíněná proměnná si pamatuje, která vlákna jsou na ní blokovány.
- Nad podmíněnou proměnnou `var` jsou typicky definovány operace
  - `cond_wait(cond_t *var, mutex_t *mutex)`
    - Funkce musí být volána se zámkem `mutex`, který je zamčený volajícím vláknem.
    - Funkce automaticky uvolní `mutex` a zablokuje volající vlákno, dokud nebude proměnná opět signalizována (předchozí signály nejsou ukládány).
    - Po odblokování (návratu z funkce) je `mutex` opět zamčen.
  - `cond_signal(cond_t *var)`
    - Odblokuje aspoň jedno ze zablokovaných vláken.
- **Příklady**
  - POSIX: datový typ `pthread_cond_t` s funkcemi `pthread_cond_wait()`, `pthread_cond_signal()`, `pthread_cond_broadcast()`,...
  - C++: třída `std::condition_variable`.

Producent-konzument: Podmíněné proměnné

```
1  #define  N 100                                        /* number of slots in the buffer */
2  int      count = 0;                                   /* number of items in buffer */
3  mutex_t  mutex;                                       /* mutex */
4  cond_t   cv_empty, cv_full;                           /* condition variables */
```

```
1  void producer(void)   {
2     item_t item;
3     while (TRUE)  {
4        item = produce_item();
5        mutex_lock(&mutex);
6           while (count == N) cond_wait(&cv_full, &mutex); /* if buffer is full, go to sleep */
7           insert_item(item);
8           count = count + 1;
9           cond_signal(&cv_empty);                       /* try to wakeup consumer */
10       mutex_unlock(&mutex);
11 }}
```

```
1  void consumer(void)   {
2     item_t item;
3     while (TRUE)  {
4        mutex_lock(&mutex);
5           while (count == 0) cond_wait(&cv_empty, &mutex); /* if buffer is empty, go to sleep */
6           remove_item(&item)
7           count = count - 1;
8           cond_signal(&cv_full);                        /* try to wakeup producer*/
9        mutex_unlock(&mutex);
10       consume_item(&item);
11 }}
```

Producent-konzument: Podmíněné proměnné

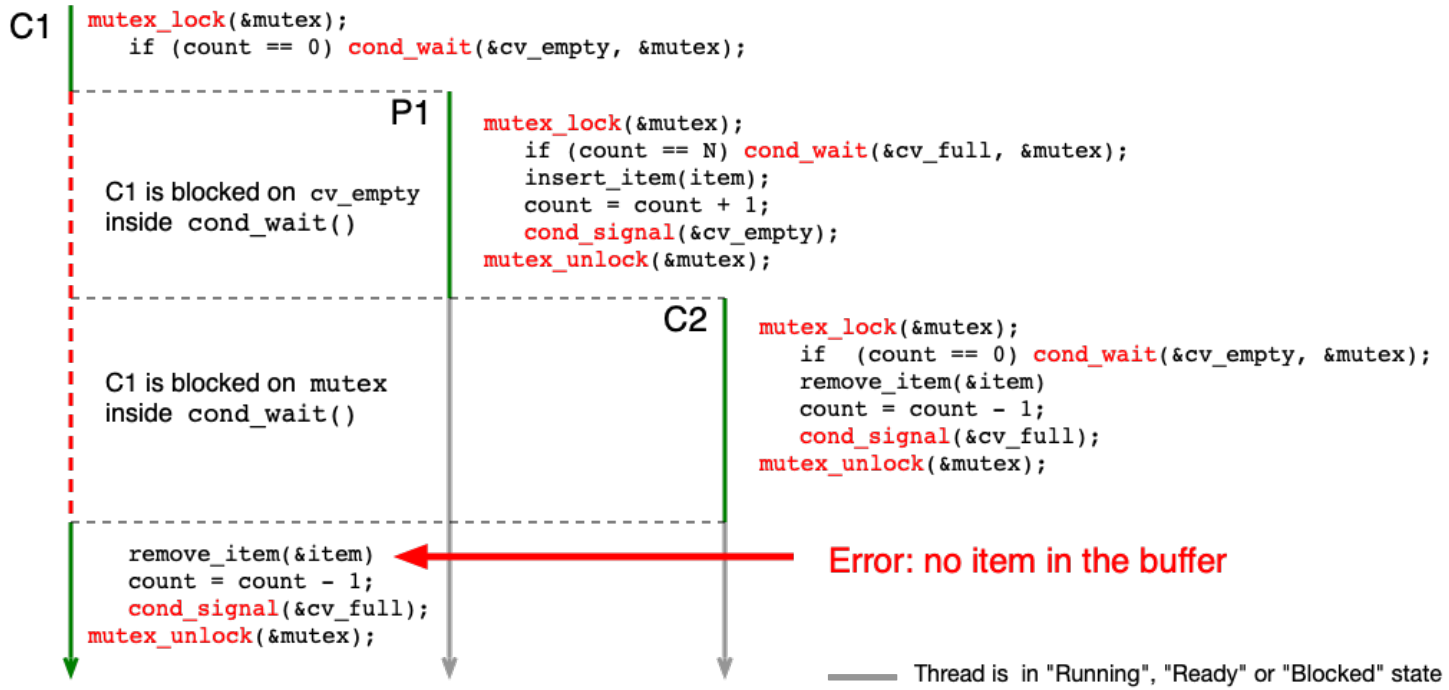
- **Proč v předchozím řešení používáme cyklus `while` místo `if`?**

```
1  void producer(void)   {
2     item_t item;
3     while (TRUE)  {
4        item = produce_item();
5        mutex_lock(&mutex);
6           if (count == N) cond_wait(&cv_full, &mutex); /* if buffer is full, go to sleep */
7           insert_item(item);
8           count = count + 1;
9           cond_signal(&cv_empty);                      /* try to wakeup consumer */
10       mutex_unlock(&mutex);
11 }}
```

```
1  void consumer(void)   {
2     item_t item;
3     while (TRUE)  {
4        mutex_lock(&mutex);
5           if (count == 0) cond_wait(&cv_empty, &mutex); /* if buffer is empty, go to sleep */
6           remove_item(&item)
7           count = count - 1;
8           cond_signal(&cv_full);                       /* try to wakeup producer*/
9        mutex_unlock(&mutex);
10       consume_item(&item);
11 }}
```

Producent-konzument: Podmíněné proměnné

- **Proč v předchozím řešení používáme cyklus `while` místo `if`?**
  - Předpokládejme, že fronta je prázdná a existuje více producentů a konzumentů.



  - Některé API (POSIX Threads, WinAPI,...) umožňují falešné probuzení (spurious wakeup) uspaného vlákna, aniž by jiné vlákno zavolalo odpovídající funkci `cond_signal()`.

Blokující volání: Semafory

- Datový typ semafor
  - obsahuje celočíselný čítač,
  - pamatuje si množinu vláken, která jsou na něm zablokována.
- Nad semaforem `sem` jsou definovány atomické operace
  - `sem_init(sem_t *sem, unsigned value)`
    - ★ Nastaví čítač na hodnotu `value` a vyprázdní frontu čekajících vláken.
  - `sem_wait(sem_t *sem)`
    - ★ Pokud je čítač větší než nula, potom se sníží o jedničku.
    - ★ V opačném případě se volající vlákno zablokuje a uloží do fronty.
  - `sem_post(sem_t *sem)`
    - ★ Pokud nějaká vlákna čekají ve frontě, potom se jedno z nich probudí.
    - ★ V opačném případě se čítač zvětší o jedničku.
- **Příklady**
  - Unix System V: `semget()`, `semctl()`, `semop()`.
  - POSIX: datový typ `sem_t` s funkcemi `sem_init()`, `sem_wait()`, `sem_post()`, ...
  - C++: nejsou implementovány.

Synchronizace kritické sekce – semafory

```
1  sem_t bin_sem;                                  /* control access to critical region(CR)*/
2
3  sem_init(&bin_sem, 1);
```

```
1  void Thread_A(void)
2  {
3    while (TRUE)
4    {
5      ...
6      sem_wait(&bin_sem);     /* enter CR */
7        critical_region_A();
8      sem_post(&bin_sem);     /* leave CR */
9      ...
10    }
11 }
```

```
1  void Thread_B(void)
2  {
3    while (TRUE)
4    {
5      ...
6      sem_wait(&bin_sem);     /* enter CR */
7        critical_region_B();
8      sem_post(&bin_sem);     /* leave CR */
9      ...
10   }
11 }
```

Synchronizace producent-konzument – semafory

```
1  #define  N 100                                   /* number slots in buffer */
2
3  mutex_t  mutex;                                  /* guards critical region (CR) */
4  sem_t empty;                                     /* counts empty slots */
5  sem_t full;                                      /* counts full slots */
6
7  sem_init(&empty, N);
8  sem_init(&full, 0);
```

```
1  void producer(void)
2  {
3     item_t item;
4     while (TRUE)
5     {
6        item = produce_item();
7        sem_wait(&empty);
8           mutex_lock(&mutex);    /* enter CR*/
9              enter_item(item);
10          mutex_unlock(&mutex); /* leave CR*/
11       sem_post(&full);
12    }
13 }
```

```
1  void consumer(void)
2  {
3     item_t item;
4     while (TRUE)
5     {
6     sem_wait(&full);
7        mutex_lock(&mutex);    /* enter CR*/
8           remove_item(&item);
9        mutex_unlock(&mutex); /* leave CR*/
10    sem_post(&empty);
11    consume_item(&item);
12    }
13 }
```

Blokující volání: Bariéry

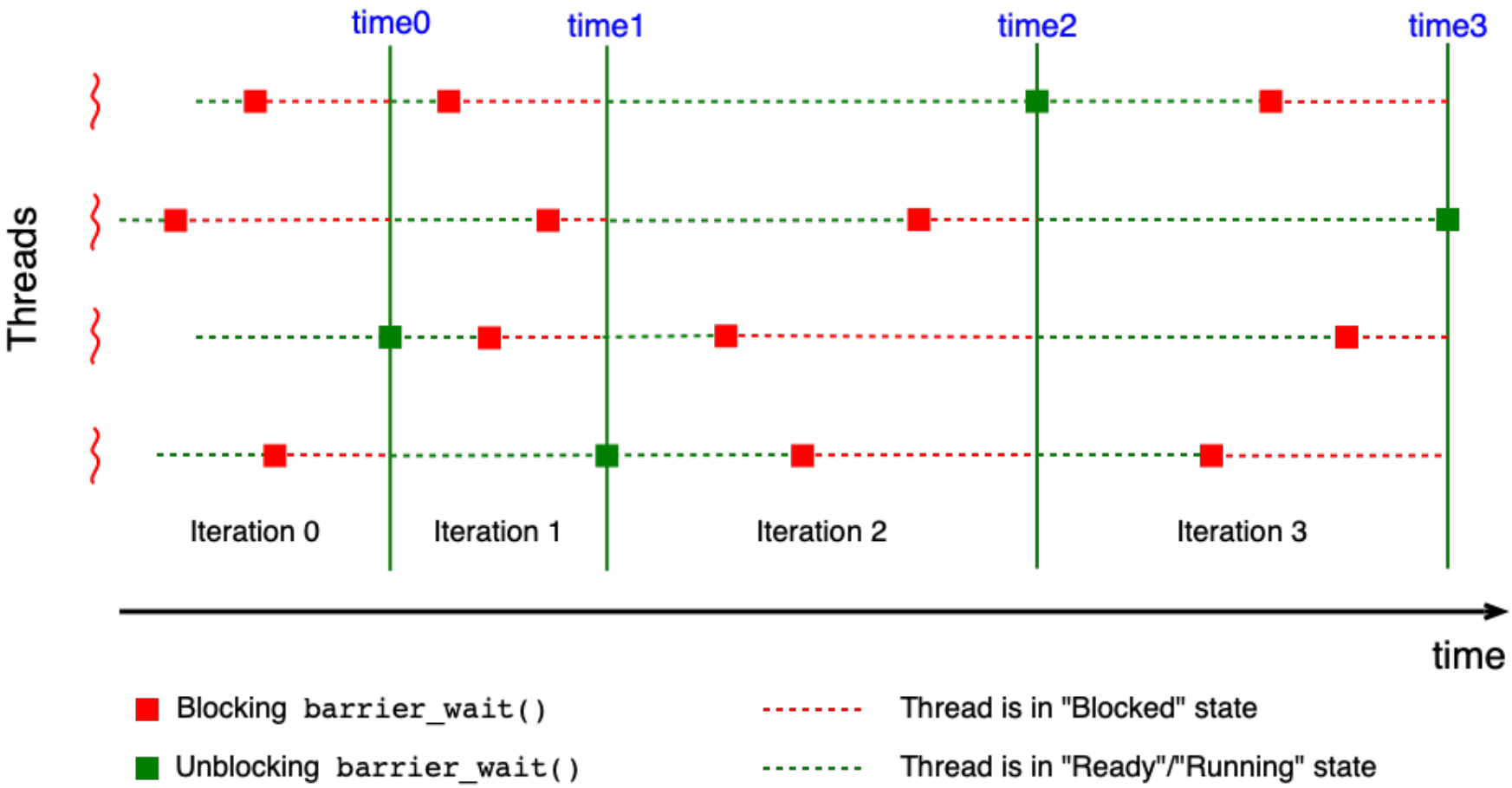
- Bariéra umožňuje jednoduše synchronizovat iterační výpočty (např. výpočet *n*-té mocniny matice pomocí více vláken).
- Bariéra obsahuje
  - čítač, který definuje sílu bariery (počet vláken, který bariéru prolomí),
  - frontu vláken, která jsou na bariéře blokována.
- Pro bariéru jsou definovány atomické operace
  - `barrier_init(barrier_t *bar, int value)`
    - ★ Funkce nastaví čítač bariery na `value` a vyprázdní frontu čekajících vláken.
  - `barrier_wait(barrier_t *bar)`
    - ★ Pokud je čítač bariéry větší než 1, pak se čítač sníží o 1 a volající vlákno se zablokuje.
    - ★ Jinak se všechna blokovaná vlákna probudí a čítač se opět nastaví na `value`.
- **Příklady**
  - POSIX: datový typ `sem_t` s funkcemi `pthread_barrier_init()`, `pthread_barrier_wait()`,...
  - C++: třída `std::experimental::barrier`

Blokující volání: Bariéry

```
1 #define  M 4                          /* number of threads */
2 #define  N 10                         /* number of iterations */
3
4 barrier_t   bar;
5 barrier_init(&bar, M);
```

```
1 void thread_function(void)
2 {
3    int i;
4    for ( i = 0; i < N; i++ )
5    {
6       iteration(i);
7       barrier_wait(&bar);
8    }
9 }
```

Blokující volání: Bariéry



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.
4. R. Marejka: *Atomic SPARC: Using the SPARC Atomic Instructions*, Oracle Doc, 2008.