# 4. Souběžnost, atomičnost a synchronizace

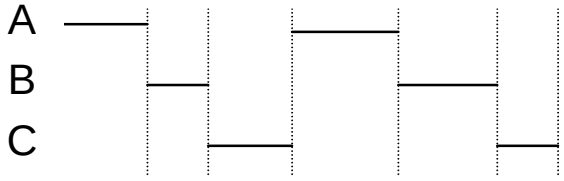
Virtualizace procesoru u multiprogramování → vlastní procesor
Přepínání kontextu → souběžný běh více procesů

**Jednoprocesorový systém** - pseudosouběžnost

- Preemptivní přepínání kontextu - kdykoli, proces nemůže ovlivnit okamžik přepnutí kontextu
- Nepreemptivní - kooperativní, proces se musí vzdát procesoru

**Víceprocesorový systém** - fyzická souběžnost


multiprocessing




multiprogramování

**Nezávislé procesy** - bez interakce, neinterferující, deterministické

**Kooperující procesy** - sdílená paměť, nedeterministické

**Nastává souběžnost v normálním programu?**

**signály** - funkce obsluhy signálu běží pseudosouběžně s programem

**vlákna - paralelně prováděné části programu**

Problém – souběžné procesy (vlákna) = **paralelní systém**.
Co s nedeterminismem? Nedeterminismus celého systému nevadí (externí asynchronní vstupy), důležitá je konzistence stavu objektů, datových struktur, operací.

**Problém analýzy - příklad:**

```
P1:                          P2:

i=0;                         i=0;
while (i < 10) i=i+1;        while (i > -10) i=i-1;
printf("P1 wins\n");         printf("P2 wins\n");
```

1. Kdo vyhraje?
2. Je zaručeno, že někdo vyhraje?
3. Je zaručeno, že při stejně rychlých procesorech poběží systém nekonečně dlouho?
4. Co v jednoprocesorových systémech?

**Příklad:**

| **M** | **Ž** |
|---|---|
| 8:00 v lednici není mléko | |
| 8:05 odchod do obchodu | |
| 8:10 | v lednici není mléko |
| 8:15 nákup mléka | odchod do obchodu |
| 8:25 návrat domů, mléko do lednice | nákup mléka |
| 8:35 | návrat domů, **2** mléka |

Proč v reálném životě problém paralelní „inkrementace čítače“ většinou nenastává?

**Příklad:**

„Zámek“ pomocí souboru:

```
/* dokud existuje, čekáme */
while (access(lockfile, F_OK) == 0) sleep(1);
/* vytvoříme soubor zámku */
fd = creat(lockfile, S_IRUSR|S_IWUSR);
```

Proč nebude fungovat?

**Def.: Časově závislé chyby** (souběh, race conditions) - chyby vznikající díky interferenci při různé relativní rychlosti provádění procesů v paralelním systému. Obvykle jsou spojeny se sdílenými proměnnými.

*Herb Sutter: The Free Lunch Is Over: A Fundamental Turn Toward Concurrency in Software*

> Probably the greatest cost of concurrency is that concurrency really is hard: The programming model, meaning the model in the programmer's head that he needs to reason reliably about his program, is much harder than it is for sequential control flow. Everybody who learns concurrency thinks they understand it, ends up finding mysterious races they thought weren't possible, and discovers that they didn't actually understand it yet after all.

**Příklad:**
Obousměrně vázaný seznam - vyjmutí položky:

```
void delete_entry(struct list_entry *entry)
{
    entry->prev->next = entry->next;
    entry->next->prev = entry->prev;
}
```

Nezáleží nám na pořadí provedení *delete_entry()* z různých procesů, ale seznam musí být mimo operaci *delete_entry()* vždy v konzistentním stavu.

**Def.: Atomická operace** - nedělitelná operace, nemůže být přerušena uprostřed.
**Význam:** Pokud jsou prováděny všechny operace nad sdílenou datovou strukturou atomickými operacemi, zůstává stav struktury konzistentní i při paralelním přístupu.

**Co je atomická operace na úrovni procesoru?**

**4.1) Přístup do paměti** (Read, Write) - atomický, ale pouze pro slovo zarovnané na hranici přístupu do společné paměti! Pokud procesor podporuje přístup ke slovu přes hranici zarovnání, nemusí být přístup atomický!

**Příklad: Intel x86**
*The Intel486 processor (and newer processors) guarantees that the following basic memory operations will always be carried out atomically:*

- *Reading or writing a byte*
- *Reading or writing a word aligned on a 16-bit boundary*
- *Reading or writing a doubleword aligned on a 32-bit boundary*

*The Pentium processor (and newer processors):*

- *Reading or writing a quadword aligned on a 64-bit boundary*
- *16-bit accesses to uncached memory locations that fit within a 32-bit data bus*

*The P6 family processors (and newer processors):*

- *Unaligned 16-, 32-, and 64-bit accesses to cached memory that fit within a cache line.*

**Ne každá proměnná je přístupná atomicky!**

**Jazyk C99:** proměnná nesmí být optimalizovaná do registru, každý přístup k ní musí směrovat do paměti - datový typ *volatile sig_atomic_t*

```
volatile sig_atomic_t shared; /* obvykle int */
int val = shared;  /* čtení sdílené proměnné */
shared = newval;   /* zápis sdílené proměnné */
```

## Komplikace - modely přístupu k paměti

Čtení a zápis různých proměnných nemusí probíhat na úrovni společné paměti ve stejném pořadí jak jsou v programu:

- překladač – může přeuspořádat příkazy (*volatile*)
- procesor – spekulativní provádění, write buffer
- paměťová sběrnice – rozprostřená paměť

**Ostatní procesory mohou vidět čtení i zápisy různých proměnných v jiném pořadí! Pořadí zápisů stejné proměnné z jednoho procesoru musí být zachováno.**

1. Silná konzistence / sekvenční konzistence (Intel po Pentium) – čtení a zápisy jsou vidět ve stejném pořadí ve všech procesorech (neefektivní).
2. Total Store Ordering (TSO – SPARC, Intel od P4, AMD64) – čtení může předběhnout zápis.
3. Partial Store Ordering (PSO – SPARC) – navíc je povoleno přehození zápisů.
4. Relaxed Memory Order (RMO – IBM Power, SPARC, ARM, Intel IA64) – povoleno přehození čtení se čtením i zápisy.

**Příklad:**

```
shared1 = 10;
shared2 = 20;
if (flag != 0) func1();
```

Pro PSO a RMO architektury mohou ostatní procesory vidět nejprve zápis *shared2* a pak teprve *shared1*. V okamžiku čtení *flag* nemusí být pro TSO, PSO a RMO ještě vidět zápis *shared2*.

**Řešení:** instrukce BARRIER, STBAR, MEMBAR, SFENCE, LFENCE, apod. – pozastaví provádění instrukcí do ukončení paměťových transakcí (zápisu, čtení) – lze dosáhnout modelu sekvenčně konzistentního:

```
shared1 = 0;
instrukce stbar;
shared2 = 0;
instrukce stbar;
```

Tato synchronizace obsahu sdílené paměti je normálně skryta v synchronizačních nástrojích typu zámek a semafor, protože ji musí interně použít a tím synchronizuje i ostatní data.

Co když ale chceme použít atomické instrukce přímo?**! Zde je minové pole - přiřazovací příkaz nestačí!**

**Až do standardu jazyka C z roku 2011 bylo nutno použít asembler!**

**Jazyk C11:** doplňuje podporu pro atomické operace, viz hlavičkový soubor <stdatomic.h> (test viz kap. 3)

- paměťové modely *memory_order*:

`memory_order_relaxed, consume, acquire, release, acq_rel, seq_cst`

- datové typy:

`atomic_flag, atomic_bool, atomic_char, ...` (ekvivalentní `_Atomic _Bool, _Atomic char, ...`)

- operace load a store pro atomické datové typy:

```
char atomic_load(atomic_char *c, char v);
char atomic_load_explicit(atomic_char *c, char v,
memory_model m);
void atomic_store(atomic_char *c, char v);
void atomic_store_explicit(atomic_char *c, char
v, memory_model m);
```

**Příklad:** ekvivalentní normálnímu přiřazení výše

```
volatile atomic_int shared;
int val = atomic_load_explicit(&shared,
                               memory_order_relaxed);
atomic_store_explicit(&shared, newval,
                               memory_order_relaxed);
```

**Jazyk C++11:** doplňuje podporu pro atomické operace <atomic> podobně jako C11, navíc jsou dostupné i jako metody v*al.load()*, *val.store(newval)*.

**Java:** java.util.concurrent.atomic (JDK 5.0)

```
AtomicInteger, AtomicLong, AtomicBoolean, ...

AtomicInteger val = new AtomicInteger();
int i = val.get();
val.set(newvalue);

val.getAndIncrement(), val.compareAndSet(expect,
newval), ...
```

**Paměťové modely C11/C+11:**
**a) Sekvenčně konzistentní (SC) model – seq_cst.** Je implicitní pro atomické operace C11 a C++11, ale ne pro **normální přiřazovací příkazy (ty jsou relaxované)**.

Pořadí zápisů i čtení je zachováno:
Proces1:

```
atomic_store(&shared1, 10);
atomic_store(&shared2, 20);
```

Proces2:

```
if (atomic_load(&shared2) == 20)
    assert(atomic_load(&shared1) == 10);
```

Proces2 nemůže vidět hodnotu shared2=20 bez hodnoty shared1=10. Sekvenčně konzistentní model je nejjednodušší na použití, ale je nejpomalejší (atomická instrukce musí být obvykle obalena bariérami). **Příklady synchronizačních algoritmů obvykle předpokládají tento model!**

**b) Volný model (relaxed)**
Není nijak omezeno přeuspořádání čtení a zápisů (jako pro normální čtení/zápis u relaxovaných modelů).

Proces1:

```
atomic_store_explicit(&shared1, 10,
                      memory_order_relaxed);
atomic_store_explicit(&shared2, 20,
                      memory_order_relaxed);
```

Proces2:

```
if (atomic_load_explicit(&shared2,
                  memory_order_relaxed) == 20) {
    /* může být nepravda */
    assert(atomic_load_explicit(&shared1,
         memory_order_relaxed) == 10);
}
```

Toto je povoleno pro různá paměťová místa, pro stejné musí být pořadí zachováno:

Proces1:

```
atomic_store_explicit(&shared, 10,
                      memory_order_relaxed);
atomic_store_explicit(&shared, 20,
                      memory_order_relaxed);
```

Proces2:

```
int v1 = atomic_load_explicit(&shared,
                      memory_order_relaxed);
int v2 = atomic_load_explicit(&shared,
                      memory_order_relaxed);
assert(v1<=v2);
/* nemůže nastat v1=20 a v2=10 */
```

**c) Uvolnění/získání (release/acquire)**

Dvojice operací, atomický zápis s uvolněním (*release*) zabraňuje přesunu předchozích operací na pozdější dobu (dřívější čtení/zápis nemůže být vidět později), atomické čtení se získáním (*acquire*) zabraňuje přesunu následujících operací před tuto operaci (pozdější čtení/zápis nemůže být vidět dříve). Dvojice *release*/*acquire* zajišťuje konzistentní stav proměnných mezi dvěma procesy – první při ukončení modifikace provede poslední zápis typu *release*, druhý při zahájení čtení dat provede první čtení typu *acquire*. Překladač pak zajistí vložení patřičných paměťových bariér. Pouze tento způsob použití atomických operací je myšlenkově přijatelný a zároveň dostatečně efektivní!

Proces1:

```
atomic_store_explicit(&shared1, 10,
                      memory_order_relaxed);
atomic_store_explicit(&shared2, 20,
                      memory_order_release);
```

Proces2:

```
    if (atomic_load_explicit(&shared2,
                    memory_order_acquire) == 20) {
        /* nemůže být nepravda */
        assert(atomic_load_explicit(&shared1,
             memory_order_relaxed) == 10);
    } else {
        /* pokud nevidí zápis do shared2, pak je
            stav shared1 neznámý */
    }
```

Použití relaxovaného modelu je důležité pro efektivitu (menší počet synchronizačních bariér a tím zastavování provádění).

### 4.2) Složitější atomické instrukce

- Pokud mají instrukce zdroj a cíl v paměti, nejsou většinou atomické (typu R-op-W, např. inc mem), pouze speciálně synchronizované instrukce (x86 xchg mem, lock cmpxchg, bts, btr, btc, inc/dec mem, add/sub mem) jsou atomické (musí je podporovat hardware víceprocesorového systému),
- Instrukce typu R-op nebo op-W mohou být atomické stejně jako load/store, záleží opět na zarovnání operandu.

**Jazyk C99:** není standardní podpora pro speciální atomické instrukce (nutný asembler).

**Jazyk C11 a C++11:**

- úroveň dostupnosti pro všechny datové typy:

```
_Bool atomic_is_lock_free(const atomic_xxx *var);
...
```

- makra ATOMIC_BOOL_LOCK_FREE, ATOMIC_CHAR_LOCK_FREE, ...

```
#define ATOMIC_XXX_LOCK_FREE 0-2
    0=never/1=sometimes/2=always
```

- operace *exchange, compare_exchange, fetch_add, fetch_sub, fetch_or, fetch_xor, fetch_and*)

```
bool atomic_compare_exchange_weak(atomic_xxx
*var, xxx *newval, xxx expect);
bool atomic_compare_exchange_strong(atomic_xxx
*var, xxx *newval, xxx expect);
```

- operace *test_and_set*:

```
_Bool atomic_flag_test_and_set(volatile
atomic_flag *f);
```

V C++ jsou dostupné jako funkce i metody.

**Příklad:** atomický čítač pomocí C11

```
/* nezaručuje uspořádání */
atomic_int counter;
atomic_init(&counter, 0);

#ifdef ATOMIC_INT_LOCK_FREE
counter++;      /* pro ++, --,  +=, -= je
                   zaručena atomičnost */

/* X86-64 vygeneruje lock addl $1, xx(%rip) */
atomic_fetch_add(&counter, 1);    /* seq_cst */

atomic_fetch_add_explicit(&counter, 1,
        memory_order_relaxed);
#endif
```

**Problém:** atomické instrukce jsou příliš jednoduché, jak vytvořit složitější atomickou operaci pomocí toho, co máme k dispozici?

## 4.3 Synchronizace

**Synchronizace**: zajištění kooperace mezi paralelně (souběžně) prováděnými procesy.

Atomická instrukce provede bezpečně pouze jednu operaci, co když je třeba na zpracování dat provést dvě a více?
**Vzájemné vyloučení (*mutual exclusion*)** – základní úloha synchronizace: pouze jeden může provádět dané operace. Vytváříme tím složitější nedělitelnou atomickou operaci.

Hledáme algoritmus pro INIT, ENTRY a EXIT, který by vzájemně vyloučil provádění kritických sekcí K1, K2, ..., Kn:

```
    INIT
    start                       start
A:  while (1) {                 while (1) {
B:      ENTRY;                      ENTRY;
D:      kritická sekce K1           kritická sekce K2
E:      EXIT;                       EXIT;
F:      výpočet                     výpočet
    }                           }
```

**Kritická sekce** – kód, jehož provádění je vzájemně vyloučené = atomicky prováděný kód

### Omezující podmínky:

1. Sekce *start* a *výpočet* mohou být libovolně dlouhé, trvat libovolnou dobu, včetně nekonečné (proces zde může skončit).
2. Kritická sekce je provedena vždy v konečném čase (proces zde nesmí zůstat čekat, ani skončit).
3. Proces musí mít zaručen nekonečně krát vstup do kritické sekce v konečné době (*liveness*). Jinak formulováno: každý proces kdykoli čekající na vstup do kritické sekce, se musí do ní někdy dostat.
4. Přidělování procesoru je nestranné, spravedlivé (*weak fairness*).

**Fairness (spravedlnost) přidělování procesoru**:

- **unconditional fairness** (nepodmíněná) - každý aktivní nepodmíněný atomický příkaz bude někdy proveden (může být dlouhodobě prováděn pouze jeden proces)
- **weak fairness** = unconditional fairness + každý aktivní podmíněný atomický příkaz bude proveden za předpokladu, že podmínka nabude hodnoty TRUE a nebude se měnit (pokud se podmínka mění během čekání na přidělení procesoru, nemusí být příkaz proveden nikdy)
- **strong fairness** = unconditional fairness + každý podmíněný atomický příkaz, jehož podmínka se nekonečně častokrát mění, bude nekonečně krát proveden (prakticky nerealizovatelný plánovací algoritmus, musel by střídavě provádět po jedné atomické instrukce jednotlivých procesů).

**Příklad:**

```
    P1                          P2
A:  shared int continue = 1, try = 0;
B:  while (continue) {      atomic {
C:      try = true;             if (try)
D:      try = false;                continue=0;
F:  }                       }
```

**Formální požadavky na hledané řešení:**

- **bezpečnost** (*safeness*), v daném případě zaručuje vyloučení
- **živost** (*liveness*):
  - nedochází k uváznutí (*deadlock*)
  - nedochází k blokování (*blocking*)
  - nedochází k hladovění (*starving*)

**Historie:** Leslie Lamport (1977), Lehman (scheduling policy, 1981) , Pnueli (temporal logic), Owicki&Lamport - formální definice liveness a postup dokazování pomocí temporální logiky

## Různá chybná řešení pro 2 procesy

## Příklad 1 – safeness (bezpečnost):

```
    P1                          P2
A:  shared int flag1 = 0, flag2 = 0;
B:  while (1) {                 while (1) {
C:      while (flag2) ;             while (flag1) ;
D:      flag1 = 1;                  flag2 = 1;
E:      kritická sekce              kritická sekce
F:      flag1 = 0;                  flag2 = 0;
G:      výpočet                     výpočet
    }                           }
```

INIT: A
ENTRY: C,D
EXIT: F

**Vlastnosti:**
1. Nezaručuje vzájemné vyloučení:
Posloupnost provádění (viz determinismus): ((C,C), (0,0)), ((D, C), (1,0)), (D,D), (1,1)), dva procesy jsou v kritické sekci

**Verifikace: PROMELA/spin**

1. Přepsat algoritmus do jazyka PROMELA (viz materiály ke cvičení)
2. Vygenerovat verifikátor (pan.*):
   ```
   spin -a -v mutex0
   ```
3. Přeložit s parametry pro testování bezpečnosti (ověření tvrzení)
   ```
   gcc -O -o pan -DREACH -DSAFETY pan.c
   ```
4. Spustit verifikátor:
   ```
   ./pan -i
   ```

```
01  /* vzájemné vyloučení - nesprávné řešení */
02
03  byte flag[2];
04  byte count;
05
06  proctype P(byte i)
07  {
08  end:
09  /* vstup do kritické sekce */
10          do
11          :: (flag[1-i] == 0) -> break;
12          :: else ;
13          od ;
14          flag[i] = 1;
15  /* začátek kritické sekce */
16  progress:
17          count++;
18          assert(count == 1);
19          count--;
20  /* výstup z kritické sekce */
21          flag[i] = 0;
22  /* ostatní kód */
23          timeout;
24  /* znovu pokus o vstup do KS */
25          goto end;
26  }
27
28  init
29  {
30          run P(0); run P(1);
31  }
```

```
./pan -i

pan: assertion violated (count==1) (at depth 9)
pan: wrote mutex0.trail
pan: reducing search depth to 9
pan: wrote mutex0.trail
pan: reducing search depth to 8
pan: wrote mutex0.trail
pan: reducing search depth to 8
(Spin Version 4.1.2 -- 21 February 2004)
        + Partial Order Reduction

Full statespace search for:
   never claim           - (none specified)
   assertion violations  +
   cycle checks          - (disabled by -DSAFETY)
   invalid end states    +

State-vector 24 byte, depth reached 20, errors: 3
      44 states, stored
      31 states, matched
      75 transitions (= stored+matched)
       0 atomic steps
(max size 2^18 states)

unreached in proctype P
        line 25, state 14, "-end-"
        (1 of 14 states)
unreached in proctype :init:
        (0 of 3 states)
```

5. V souboru *mutex0.trail* je posloupnost vedoucí k porušení tvrzení, můžeme si ji odkrokovat:

```
spin -v -t mutex0
```

```
1: proc 0 (:init:) line 29 "mutex0" [(run P(0))]
2: proc 0 (:init:) line 29 "mutex0" [(run P(1))]
3: proc 2 (P) line 11 "mutex0"  [((flag[(1-i)]==0))]
4: proc 1 (P) line 11 "mutex0"  [((flag[(1-i)]==0))]
5: proc 2 (P) line 14 "mutex0"  [flag[i] = 1]
6: proc 2 (P) line 17 "mutex0"  [count = (count+1)]
7: proc 1 (P) line 14 "mutex0"  [flag[i] = 1]
8: proc 1 (P) line 17 "mutex0"  [count = (count+1)]
spin: line 18 "mutex0", Error: assertion violated
spin: text of failed assertion: assert((count==1))
9: proc 1 (P) line 18 "mutex0"  [assert((count==1))]
spin: trail ends after 9 steps
#processes: 3
                flag[0] = 1
                flag[1] = 1
                count = 2
9: proc 2 (P) line 18 "mutex0" (state 9)
9: proc 1 (P) line 19 "mutex0" (state 10)
9: proc 0 (:init:) line  30 "mutex0" (state 3) <valid
end state>
3 processes created
```

**Příklad 2 – deadlock (uváznutí):**

```
    P1                          P2
A:  shared int flag1 = 0, flag2 = 0;
B:  while (1) {              while (1) {
C:      flag1 = 1;               flag2 = 1;
D:      while (flag2) ;          while (flag1) ;
E:      kritická sekce           kritická sekce
F:      flag1 = 0;               flag2 = 0;
G:      výpočet                  výpočet
    }                        }
```

INIT: A
ENTRY: C,D
EXIT: F

**Vlastnosti:**
1. Bezpečný
2. Posloupnost: ((B,B), (0,0)), ((B, C), (0,1)), (C,C), (1,1)), ((D,C), (1,1)), ((D,D), (1,1)), ... nekonečné opakování

**Uváznutí** (deadlock) - procesy čekají v synchronizaci na stav, který by mohl nastat, kdyby jeden z nich mohl pokračovat (vzájemné *blokování* - viz dále).
Porušuje podmínku proces se dostane do kritické sekce v konečném čase.
Přesněji: pro každou posloupnost provádění platí, pokud je nějaký proces v ENTRY, pak je dále v posloupnosti provádění stav, kdy je nějaký proces uvnitř kritické sekce.

**Nebezpečí:**
- nemusí nastat vždy (to, že program jednou běží, neznamená, že poběží vždy!)
- obtížná detekce (zvláště při aktivním čekání)

Verifikace živosti – hledání cyklů, které trvale neprochází přes označená místa algoritmu (v daném případě přes kritickou sekci):

Generování verifikátoru je stejné, jen se musí jinak přeložit:

```
gcc -O -o pan -DREACH -DNP pan.c
```

Spuštění (pro velké stavové prostory vyžaduje **hodně** paměti):

```
./pan -l -f
         (-l = hledat non-progress, -f = weak fairness)
pan: non-progress cycle (at depth 10)
pan: wrote mutex1.trail
...

spin -t -v mutex1
2: proc  0 (:init:) line  29 "mutex1" [(run P(0))]
4: proc  0 (:init:) line  29 "mutex1" [(run P(1))]
6: proc  2 (P) line  10 "mutex1" <valid end state>
[flag[i] = 1]
8: proc  1 (P) line  10 "mutex1" <valid end state>
[flag[i] = 1]
10: proc  2 (P) line  13 "mutex1" [else]
  <<<<<START OF CYCLE>>>>>
12: proc  2 (P) line  13 "mutex1" [else]
14: proc  1 (P) line  13 "mutex1" [else]
16: proc  2 (P) line  13 "mutex1" [else]
spin: trail ends after 16 steps
#processes: 3
                flag[0] = 1
                flag[1] = 1
                count = 0
16: proc  2 (P) line  11 "mutex1" (state 5)
16: proc  1 (P) line  11 "mutex1" (state 5)
16: proc  0 (:init:) line  30 "mutex1" (state 3) <valid
end state>
```

**Příklad 3 – blocking (blokování):**

```
    P1                          P2
A:  shared int turn = 1;
B:  while (1) {                 while (1) {
C:      while (turn==2) ;           while (turn==1) ;
D:      kritická sekce              kritická sekce
E:      turn = 2;                   turn = 1;
F:      výpočet                     výpočet
    }                           }
```

INIT: A
ENTRY: C
EXIT: E

**Vlastnosti:**

1. Bezpečný
2. Nedochází k uváznutí
3. Pokud je některý proces ve stavu F, trvale *blokuje* opakovaný vstup do kritické sekce druhému procesu.

**Blokování** (blocking) - proces čeká v synchronizaci na stav, který generuje jiný proces, a toto čekání není nutné z hlediska synchronizace (kritická sekce je volná). Postup procesu je blokován jiným procesem.
Porušuje podmínku každý proces se dostane do kritické sekce v konečném čase.
Přesněji: pro každou posloupnost provádění platí, pokud je nějaký proces v ENTRY, pak je dále v posloupnosti stav, kdy je stejný proces uvnitř kritické sekce.

**Nebezpečí:**

- blokování znamená vždy potenciální problém (co, když proces ve stavu F skončí)
- obvykle neefektivní

**Příklad 4 – starving (hladovění, stárnutí):**

```
    P1                          P2
A:  shared int flag1 = 0, flag2 = 0;
B:  while (1) {             while (1) {
C:      flag1 = 1;              flag2 = 1;
D:      while (flag2) {         while (flag1) {
E:          flag1 = 0;              flag2 = 0;
F:          čekání                  čekání
G:          flag1 = 1;              flag2 = 1;
H:      }                       }
I:      kritická sekce          kritická sekce
J:      flag1 = 0;              flag2 = 0;
K:      výpočet                 výpočet
    }                       }
```

INIT: A
ENTRY: C-H
EXIT: J

**Vlastnosti:**
1. Bezpečný
2. Nedochází k uváznutí
1. Nedochází k blokování
2. Cyklus D-H se může potenciálně opakovat nekonečně krát, není definována žádná horní mez - stárnutí, hladovění

**Stárnutí** (starving) - proces může čekat v synchronizaci na stav, který nemusí být nikdy pravdiný v okamžiku testování. Není striktně omezena horní mez čekání (jinak jako blokování).
- v praxi se obvykle toleruje, závisí na plánovacím algoritmu

**Živost** (liveness) - algoritmus je živý, pokud je bezpečný a nedochází ke uváznutí, blokování a stárnutí (je zaručeno jeho dokončení v konečné době).

## Petersonův algoritmus (pro 2 procesy, 1981)

```
    shared int flag1 = 0;
    shared int flag2 = 0;
    shared int turn = 1;
A:  while (1) {              while (1) {
B:      flag1 = 1;               flag2 = 1;
C:      turn = 1;                turn = 2;
D:      while (flag2 &&          while (flag1 &&
                turn == 1);              turn == 2);
E:      kritická sekce           kritická sekce
F:      flag1 = 0;               flag2 = 0;
G:      výpočet                  výpočet
    }                        }
```

**Vlastnosti:**

1. Bezpečný
2. Nedochází k uváznutí (D přeruší turn)
3. Nedochází k blokování (pokud není P v KS, flag(P) je 0)
4. Nedochází k stárnutí (pokud je P v KS a druhý čeká, dá P při příštím vstupu přednost druhému - C)

Co když to napíšu v C a pustím na víceprocesorovém systému:

- Nebude fungovat!
- Viz paměťové modely (je třeba přidat write bariéru mezi C/D a E/F!)

**Podobná řešení:**

Dekker - 2 procesy (oproti Petersonovu komplikovanější)
Dijkstra - *n* procesů se stárnutím, 1965
Knuth - *n* procesů, 1966
Eisenberg&McGuire - *n* procesů bez stárnutí
Lamport - bakery algorithm, pro *n* procesů, 1974

## Lamportův bakery algoritmus pro N procesů

```
    shared int flag[N];
    shared int ticket[N];
    int j, max;
    for (j = 0; j < N; ++j) ticket[j] = 0;

Proces i=0..N-1:
A:  while (1) {
B:      flag[i] = 1;        /* choosing ticket */
C:      for (j = 0; j < N; ++j) {  /* select */
            if (ticket[j] > max) max = ticket[j];
        }
        ticket[i] = max + 1;  /* take a ticket */
D:      flag[i] = 0;
E:      for (j = 0; j < N; ++j) {
F:          while (flag[j]) /* selecting tckt */;
G:          if (ticket[j] > 0 &&  /* entering */
        /* lower ticket or equal and lower id */
E:              (ticket[j] < ticket[i] ||
F:              (ticket[j] == ticket[i] && j<i)){
G:                  while (ticket[j] > 0)
                    /* wait until leaves CS */;
            }
        }
E:      kritická sekce
F:      ticket[i] = 0;
G:      výpočet
    }
```

**Vlastnosti:**

1. Bezpečný
2. Nedochází k uváznutí (F, G)
3. Nedochází k blokování (pokud i není v KS, num(i) je 0 - G)
4. Nedochází k stárnutí (vždy vstupuje i s nejstarším lístkem - E, pokud si vyberou stejný, rozhoduje číslo procesu - F)

## Původní Dekkerův algoritmus pro 2 procesy

Proces i = 0,1:

```
shared int flag[0] = 0, flag[1] = 0, turn = 0;
A:  while (1) {
B:      flag[i] = 1;
C:      while (flag[!i]) {
D:          if (turn == !i) {
E:              flag[i] = 0;
F:              while (turn == !i) /* loop */;
G:              flag[i] = 1;
            }
        }
H:      kritická sekce
I:      flag[i] = 0;
J:      turn = !i;
L:      výpočet
    }
```

## Dijkstrův algoritmus pro n procesů

```
    int turn = 0;
    int flag[N];
Proces i:
A:  while (1) {
B:  restart:
        flag[i] = ENTER;
C:      while (turn != i) {
D:          if (flag[turn] == OUT) turn = i;
        }
E:      flag[i] = IN;
F:      for (j = 0; j < N; j++) {
G:          if (j == i) continue;
H:          if (flag[j] == IN) goto restart;
        }
I:      kritická sekce
J:      flag[i] = 0;
K:      výpočet
```