

Synchronizace procesů

Tomáš Vojnar, Ondřej Lengál
`{vojnar,lengal}@fit.vutbr.cz`

Vysoké učení technické v Brně
Fakulta informačních technologií
Božetěchova 2, 612 00 BRNO

2. dubna 2024

Synchronizace procesů

- **Současný přístup** několika paralelních procesů (vláken, obslužných rutin přerušení či signálů, ...) ke **sdíleným zdrojům** (sdílená data, sdílená I/O zařízení) může vést k nekonzistencím zpracovávaných dat kvůli nevhodnému pořadí provádění různých dílčích operací různými procesy.
- **Časově závislá chyba** (neboli race condition, také souběh): chyba vznikající při přístupu ke sdíleným zdrojům kvůli nevhodnému pořadí provádění jednotlivých paralelních výpočtů v systému, tj. kvůli jejich nevhodné relativní rychlosti.
- Zajištění konzistence dat vyžaduje mechanismy **synchronizace procesů** zajišťující **správné pořadí** provádění spolupracujících procesů.

- **Příklad:** Mějme proměnnou N , která obsahuje počet položek ve sdíleném bufferu (např. $N==5$) a uvažujme provedení následujících operací:

konzument: $N--$ || producent: $N++$

- Na strojové úrovni a při přepínání kontextu může dojít k následujícímu:

```
producent: register1 = N (register1 == 5)
producent: register1 = register1 + 1 (register1 == 6)
konzument: register2 = N (register2 == 5)
konzument: register2 = register2 - 1 (register2 == 4)
producent: N = register1 (N == 6)
konzument: N = register2 (N == 4  !!!!!)
```

- Výsledkem může být 4, 5, nebo 6 namísto jediné správné hodnoty 5!

Kritické sekce

- Máme n procesů soutěžících o přístup ke sdíleným zdrojům. Každý proces je řízen určitým programem.
- **Sdílenými kritickými sekcemi** daných procesů rozumíme ty úseky jejich řídicích programů přistupující ke sdíleným zdrojům, jejichž provádění jedním procesem vylučuje současné provádění libovolného z těchto úseků ostatními procesy.
- Je možný výskyt více sad sdílených kritických sekcí, které navzájem sdílené nejsou (např. při práci s různými sdílenými proměnnými).
- Obecnějším případem pak je situace, kdy sdílené kritické sekce nejsou vzájemně zcela vyloučeny, ale může se jich současně provádět nejvýše určitý počet.
- **Problémem kritické sekce** rozumíme problém zajištění korektní synchronizace procesů na množině sdílených kritických sekcí, což zahrnuje:
 - Vzájemné vyloučení (*mutual exclusion*): nanejvýš jeden (obecně k) proces(ů) je v daném okamžiku v dané množině sdílených KS.
 - Dostupnost KS:
 - Je-li KS volná (resp. opakovaně volná alespoň v určitých okamžicích), proces nemůže neomezeně čekat na přístup do ní.
 - Je zapotřebí se vyhnout:
 - * uvážnutí,
 - * blokování a
 - * stárnutí.

Problémy vznikající na kritické sekci

- **Data race** (časově závislá chyba nad daty, souběh nad daty): dva přístupy ke zdroji s výlučným přístupem ze dvou procesů bez synchronizace, alespoň jeden přístup je pro zápis (zvláštní případ chybějícího vzájemného vyloučení).
- **Uváznutí (deadlock) při přístupu ke zdrojům s výlučným (omezeným) přístupem**: situace, kdy každý proces z určité neprázdné množiny procesů je pozastaven a neomezeně čeká na uvolnění nějakého zdroje s výlučným (omezeným) přístupem vlastněného nějakým procesem z dané množiny, který jediný může tento zdroj uvolnit, a to až po dokončení práce s ním.
(Obecnější pojetí uváznutí viz dále.)
- **Blokování (blocking) při přístupu do KS**: situace, kdy proces, jenž žádá o vstup do kritické sekce, musí čekat, přestože je kritická sekce volná (tj. žádný proces se nenachází v ní ani v žádné sdílené kritické sekci).
- **Stárnutí** (též hladovění, **starvation**): situace, kdy proces čeká na podmínku, která nemusí nastat. V případě kritické sekce je touto podmínkou umožnění vstupu do kritické sekce.
- Při striktní interpretaci jsou uváznutí i blokování zvláštními případy stárnutí.
- Zvláštním případem stárnutí je také tzv. **livelock**, kdy každý proces z určité neprázdné množiny procesů běží, ale provádí jen omezený úsek kódu, ve kterém opakovaně žádá o nějaký zdroj s výlučným přístupem, který vlastní některý z procesů dané množiny a jen ten by ho mohl uvolnit, pokud by mohl pokračovat (situace podobná uváznutí, ale s aktivním čekáním).

Petersonův algoritmus

– Možné řešení problému KS pro dva procesy:

```
bool flag[2] = { false, false }; // shared array
int turn = 0;                      // shared variable

// process i (i==0 or i==1):
do {
    //...

    flag[i] = true;
    turn = 1-i;

    while (flag[1-i] && turn != i) ; // busy waiting
    // critical section
    flag[i] = false;
    // remainder section
} while (1);
```

– Existuje **zobecnění Petersonova algoritmu** pro n procesů.

Bakery algoritmus L. Lamporta

– Vzájemné vyloučení pro n procesů:

- Před vstupem do KS proces získá “přístupový lístek”, jehož číselná hodnota je větší než čísla přidělená již čekajícím procesům (resp. procesu, který je již v KS).
- Držitel nejmenšího čísla a s nejmenším PID může vstoupit do KS (více procesů může lístek získat současně!).
- Čísla přidělovaná procesům mohou teoreticky neomezeně růst.

```
bool flag[N] = {false}; // shared array
int ticket[N] = { 0 }; // shared array
int j, max=0;           // local (non-shared) variables
```

```
// process i
while (1) {
    // ... before the critical section
    flag[i] = true; // finding the max ticket
    for (j = 0; j < N; j++) {
        if (ticket[j] > max) max = ticket[j];
    }
    ticket[i] = max + 1; // take a new ticket
    flag[i] = false;
```

Bakery algoritmus – pokračování

```
// give priority to processes with smaller tickets
//                                     (or equal tickets and smaller PID)
for (j = 0; j < N; j++) {
    while (flag[j]);
    while (ticket[j] > 0 &&
           (ticket[j] < ticket[i] ||
            (ticket[j] == ticket[i] && j < i)));
}
// the critical section
ticket[i] = 0; max = 0;
// the remainder section
}
```

– Pozor na možnost přetečení u čísel lístků!

Využití atomických instrukcí pro synchronizaci

– Založeno na využití instrukcí, jejichž atomicita je zajištěna hardware. Používá se častěji než specializované algoritmy bez využití atomických instrukcí.

– Atomická instrukce typu **TestAndSet** (např. LOCK BTS):

```
bool TestAndSet(bool &target) {  
    bool rv = target;  
    target = true;  
    return rv;  
}
```

– Využití TestAndSet pro synchronizaci na KS:

```
bool lock = false; // a shared variable  
  
// ...  
while (TestAndSet(lock)) ;  
// critical section  
lock = false;  
// ...
```

- Atomická instrukce typu **Swap** (např. LOCK XCHG):

```
void Swap(bool &a, bool &b) {  
    bool temp = a;  
    a = b;  
    b = temp;  
}
```

- Využití Swap pro synchronizaci na KS:

```
bool lock = false; // a shared variable  
  
// ...  
bool key = true;    // a local variable  
while (key == true)  
    Swap(lock, key);  
// critical section  
lock = false;  
// ...
```

- Uvedená řešení vzájemného vyloučení založená na specializovaných instrukcích zahrnují možnost **aktivního čekání**, proto se také tato řešení často označují jako tzv. **spinlock**.
- Lze užít na **krátkých, neblokujících kritických sekcích bez preempce** (alespoň bez preempce na použitém procesoru: proto bývá vlastní použití atomické instrukce uzavřeno mezi zákaz/povolení přerušení).
- Opakovaný zápis sdíleného paměťového místa je problematický z hlediska zajištění **konzistence cache** v multiprocesorových systémech (zatěžuje sdílenou paměťovou sběrnici) – řešením je **při aktivním čekání pouze číst**:

```
// ...  
while (TestAndSet(lock))  
    while (lock) ;  
// ...
```

- Uvedená řešení **nevylučují možnost stárnutí**: bývá tolerováno, ale existují řešení, která tento problém odstraňují.

Semaforey

- Synchronizační nástroj nevyžadující (nebo minimalizující) aktivní čekání – aktivní čekání se v omezené míře může vyskytnout uvnitř implementace operací nad semaforem, ale ne v kódu, který tyto operace používá.
- Jedná se v principu o celočíselnou proměnnou přístupnou dvěmi základními atomickými operacemi:
 - **lock** (také P či down) – zamknutí/obsazení semaforu, volající proces čeká dokud není možné operaci úspěšně dokončit a
 - **unlock** (také V či up) – odemknutí/uvolnění semaforu.

Dále může být k dispozici inicializace, případně různé varianty zmíněných operací, např. neblokující zamknutí (vždy ihned skončí s příznakem úspěšnosti), pokus o zamknutí s horní mezí na dobu čekání, současné zamknutí více semaforů atp.

– Sémantika celočíselné proměnné S odpovídající semaforu:

- $S > 0$ – odemknuto (hodnota $S > 1$ se užívá u zobecněných semaforů, jež mohou propustit do kritické sekce více než jeden proces),
- $S \leq 0$ – uzamknuto (je-li $S < 0$, hodnota $|S|$ udává počet procesů čekajících na semaforu).
 - Někdy se záporné hodnoty neužívají a semafor se zastaví na nule.

- Využití semaforů pro synchronizaci na KS:

```
semaphore mutex; // shared semaphore

init(mutex,1); // initially mutex = 1
// ...
lock(mutex);
// critical section
unlock(mutex);
// ...
```

- POZOR! Semaforey obecně negarantují obsluhu procesů v určitém pořadí (přestože při jejich implementaci bývá využita čekací fronta) ani vyhnutí se stárnutí.

- Konceptuální **implementace semaforu**:

```
typedef struct {
    int value;
    process_queue *queue;
} semaphore;
```

```

lock(S) {
    S.value--;
    if (S.value < 0) {
        // remove the process calling lock(S) from the ready queue
        remove(ready_queue, this_proc_num);
        // add the process calling lock(S) to S.queue
        append(S.queue, this_proc_num);
        // switch context, the current process has to wait to get
        // back to the ready queue
        switch();
    }
}

unlock(S) {
    S.value++;
    if (S.value <= 0) {
        // get and remove the first waiting process from S.queue
        P = get(S.queue);
        // enable further execution of P by adding it into
        // the ready queue
        append(ready_queue, P);
    }
}

```

- Provádění lock a unlock musí být atomické. Jejich tělo představuje rovněž kritickou sekci!!!
- Řešení atomicity lock a unlock:
 - zákaz přerušení,
 - vzájemné vyloučení s využitím atomických instrukcí a aktivním čekáním, tj. s využitím spinlocku:
 - položka reprezentující spinlock je doplněna do struktury reprezentující semafor,
 - spinlock se musí zamknout na vstupu do `lock` a `unlock` a odemknout před výstupem z nich nebo před začátkem čekání v `lock`;
 - používá se u multiprocesorových systémů (spolu se zákazem přerušení pro minimalizaci doby běhu na daném procesoru);
 - čekání pouze na vstup do lock/unlock, ne na dokončení dlouhé uživatelské KS.
- Používají se také:
 - read-write zámky – pro čtení lze zamknout vícenásobně,
 - reentrantní zámky – proces může stejný zámek zamknout opakovaně,
 - mutexy – binární semaforey, které mohou být odemknuty pouze těmi procesy, které je zamkly (umožňuje optimalizovanou implementaci).

– POSIX: Semafore dostupné prostřednictvím volání:

- starší rozhraní (System V): `semget`, `semop`, `semctl`,
- novější (viz `man sem_overview`): `sem_open`, `sem_init`, `sem_post`, `sem_wait`, `sem_getvalue`, `sem_close`, `sem_unlink`, `sem_destroy`,
- POSIXová vlákna: `pthread_mutex_lock`, `pthread_mutex_unlock`, ...

– Linux: futexes – fast user-space locks:

- používá se běžná celočíselná proměnná ve sdílené paměti s atomickou inkrementací/dekrementací v uživatelském režimu na úrovni assembleru,
- při detekci konfliktu se volá pro řešení konfliktu jádro – služba `futex` (hlavní operace `FUTEX_WAIT` a `FUTEX_WAKE`),
- rychlost vyplývá z toho, že při malém počtu konfliktů se zcela obejde režie spojená s voláním služeb jádra.

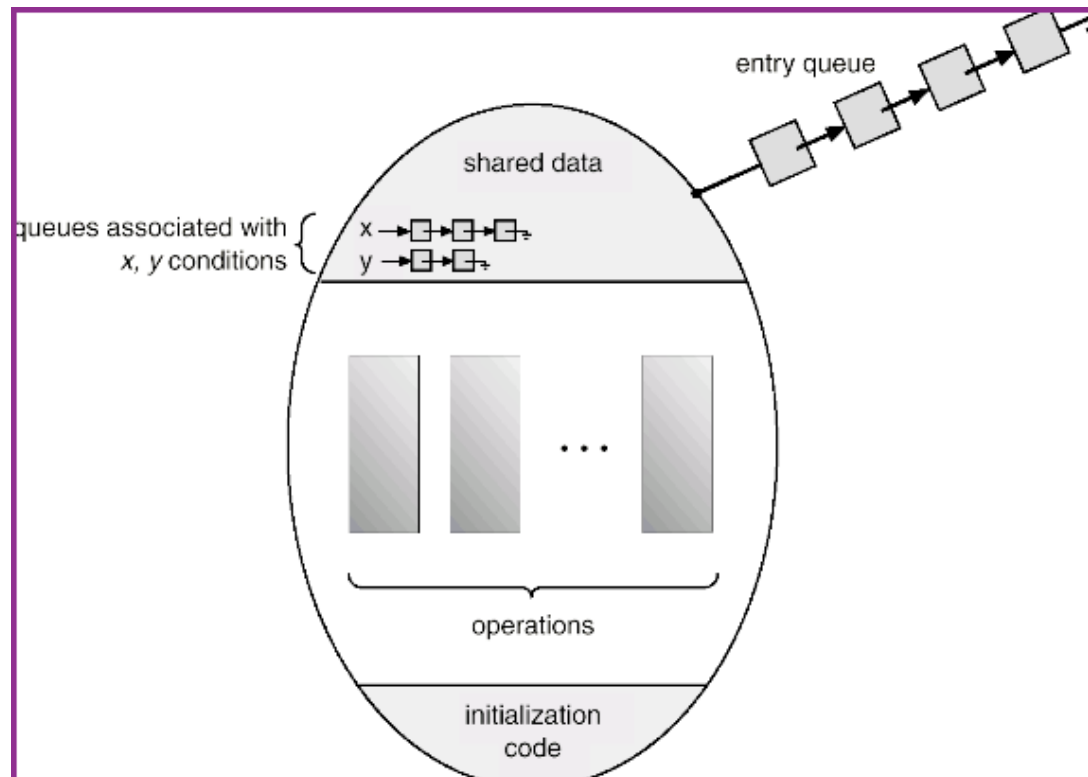
Monitory

– Jeden z vysokoúrovňových synchronizačních prostředků. Zapouzdřuje data, má definované operace, jen jeden proces může provádět nějakou operaci nad chráněnými daty:

```
monitor monitor-name {  
    shared variable declarations  
  
    procedure body P1 (...) {  
        ...  
    }  
    procedure body P2 (...) {  
        ...  
    }  
    {  
        initialization code  
    }  
}
```

– Pro možnost čekání uvnitř monitoru jsou k dispozici tzv. podmínky (*conditions*), nad kterými je možné provádět operace:

- wait() a
- signal(), resp. notify() – pokračuje příjemce (odesílatel má následně prioritu pro návrat), resp. odesílatel signálu; nečeká-li nikdo, jedná se o prázdnou operaci.



– Implementace možná pomocí semaforů: vstupní semafor, semaforey pro podmínky, příp. semafor pro prioritní návrat (signal()).

– Monitory jsou v určité podobě použity v Javě (viz klíčové slovo `synchronized`). Pro POSIXová vlákna jsou k dispozici podmínky `pthread_cond_t` a související funkce `pthread_cond_wait/signal/broadcast`.

Některé klasické synchronizační problémy

- Komunikace **producenta a konzumenta** přes vyrovnávací paměť s kapacitou omezenou na N položek:
- Synchronizační prostředky:

```
semaphore full, empty, mutex;
```

```
// Initialization:
```

```
init(full,0);
```

```
init(empty,N);
```

```
init(mutex,1);
```

– Producent:

```
do {  
    ...  
    // produce an item I  
    ...  
    lock(empty);  
    lock(mutex);  
    ...  
    // add I to buffer  
    ...  
    unlock(mutex);  
    unlock(full);  
} while (1);
```

– Konzument:

```
do {  
    lock(full)  
    lock(mutex);  
    ...  
    // remove I from buffer  
    ...  
    unlock(mutex);  
    unlock(empty);  
    ...  
    // consume I  
    ...  
} while (1);
```

– Problém **čtenářů a písarů**: libovolný počet čtenářů může číst; pokud ale někdo píše, nikdo další nesmí psát ani číst.

– Synchronizační prostředky:

```
int readcount;  
semaphore mutex, wrt;
```

```
// Initialization:  
readcount=0;  
init(mutex,1);  
init(wrt,1);
```

– Písař:

```
do {  
    ...  
    lock(wrt);  
    ...  
    // writing is performed  
    ...  
    unlock(wrt);  
    ...  
} while (1);
```

– Čtenář:

```
do {  
    lock(mutex);  
    readcount++;  
    if (readcount == 1)  
        lock(wrt);  
    unlock(mutex);  
    ...  
    // reading is performed  
    ...  
    lock(mutex);  
    readcount--;  
    if (readcount == 0)  
        unlock(wrt);  
    unlock(mutex);  
    ...  
} while (1);
```

– Čtenář:

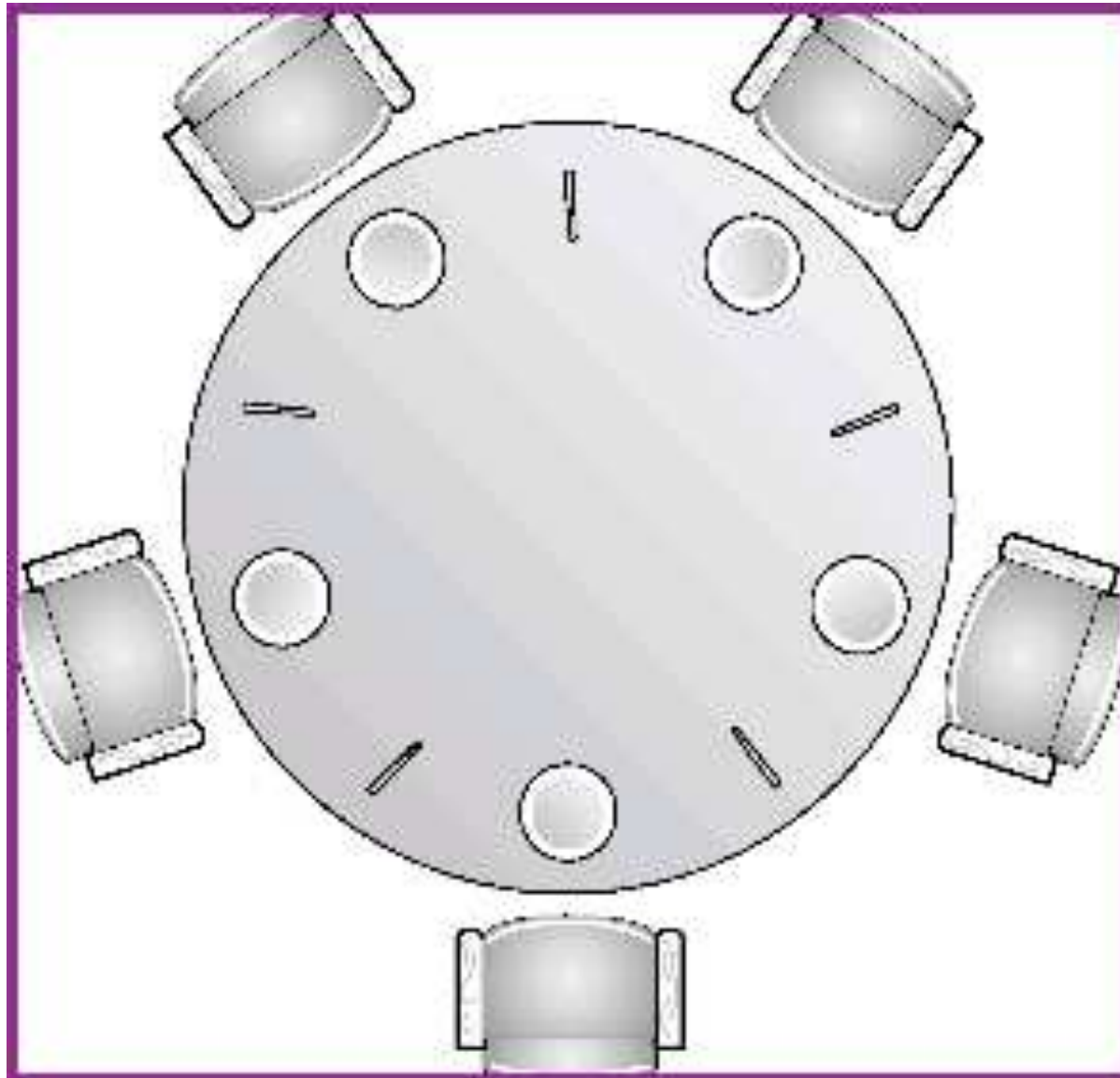
– Písař:

```
do {  
    ...  
    lock(wrt);  
    ...  
    // writing is performed  
    ...  
    unlock(wrt);  
    ...  
} while (1);
```

```
do {  
    lock(mutex);  
    readcount++;  
    if (readcount == 1)  
        lock(wrt);  
    unlock(mutex);  
    ...  
    // reading is performed  
    ...  
    lock(mutex);  
    readcount--;  
    if (readcount == 0)  
        unlock(wrt);  
    unlock(mutex);  
    ...  
} while (1);
```

– Hrozí “vyhladovění” písařů: přidat další semafor.

– Problém **večeřících filozofů**:



– Řešení (s možností uváznutí):

```
semaphore chopstick[5];

// Initialization:
for (int i=0; i<5; i++) init(chopstick[i],1);

// Philosopher i:
do {
    lock(chopstick[i])
    lock(chopstick[(i+1) % 5])
    ...
    // eat
    ...
    unlock(chopstick[i]);
    unlock(chopstick[(i+1) % 5]);
    ...
    // think
    ...
} while (1);
```

– Lepší řešení: získávat obě hůlky současně, získávat hůlky asymetricky, ...

Uváznutí (deadlock)

- **Uváznutí (deadlock) při přístupu ke zdrojům s výlučným (omezeným) přístupem** je situace, kdy každý proces z nějaké neprázdne množiny procesů je pozastaven a neomezeně čeká na uvolnění nějakého zdroje s výlučným (omezeným) přístupem vlastněného nějakým procesem z dané množiny, který jediný může tento zdroj uvolnit, a to až po dokončení jeho použití.
- Typický příklad (v praxi samozřejmě mohou být příslušná volání ve zdrojovém kódu velmi daleko od sebe a v daném pořadí mohou být zamykány jen za určitých podmínek, takže se uváznutí projeví jen zřídka a špatně se odhaluje; uváznutí také může vyžadovat větší počet procesů):

```
semaphore mutex1, mutex2;

init(mutex1,1);  // Initialization:
init(mutex2,1);

...
// Process 1           // Process 2
lock(mutex1);          lock(mutex2);
...
lock(mutex2);          lock(mutex1);
```

- **Obecnější definice** s možností uváznutí i bez prostředků s výlučným přístupem (např. při zasílání zpráv): **Uváznutím** rozumíme situaci, kdy každý proces z nějaké neprázdne množiny procesů je pozastaven a neomezeně čeká na nějakou událost, která by mohla nastat pouze tehdy, pokud by mohl pokračovat některý z procesů z dané množiny.

– Nutné a postačující podmínky uvážnutí při přístupu ke zdrojům s výlučným přístupem (Coffmanovy podmínky):

1. vzájemné vyloučení při používání prostředků,
 2. vlastnictví alespoň jednoho zdroje, pozastavení a neomezené čekání na další,
 3. prostředky vrací proces, který je vlastní, a to po dokončení jejich využití,
 4. cyklická závislost na sebe čekajících procesů.
- (Pozor: Nesouvisí nijak s pojmem aktivního čekání cyklením v čekací smyčce.)

– Řešení:

- prevence uvážnutí,
- vyhýbání se uvážnutí,
- detekce a zotavení.

Prevence uváznutí

– Zrušíme platnost některé z nutných podmínek uváznutí – například:

1. Nepoužívat sdílené prostředky nebo užívat sdílené prostředky, které umožňují skutečně současný sdílený přístup a u kterých tedy není nutné vzájemné vyloučení procesů.
 - Např. použití atomicky manipulovatelných hodnot v paměti.
2. Proces může žádat o prostředky pouze, pokud žádné nevlastní.
 - Vynutit/kontrolovat staticky na úrovni zdrojového kódu, nebo za běhu při nedodržení neúspěšný konec žádosti či zrušení procesu s odebráním jeho zdrojů (možná nekonzistence stavu prostředků – nedokončena operace s nimi, pokud ji nelze vrátit zpět).
3. Pokud proces požádá o prostředky, které nemůže momentálně získat, buď neúspěšný konec žádosti, nebo je proces pozastaven, všechny prostředky jsou mu odebrány a následně je proces buď zrušen, nebo čeká, až mu mohou být všechny potřebné prostředky přiděleny (opět s možností nekonzistence stavu prostředků); případně aplikováno i na jiný proces než žádající.
4. Prostředky jsou očíslovány a je možné je získávat pouze v pořadí vylučujícím vznik cyklické závislosti procesů (např. pouze od nejnižších čísel k vyšším).
 - Opět: Vynutit/kontrolovat staticky, neúspěšná žádost, zrušení a odebrání zdrojů.

– Řešení buď přímo zabudujeme do návrhu implementovaného systému (a ideálně vhodně ověříme, že je opravdu správně použito), nebo ho bude kontrolovat a vynucovat systém přidělování zdrojů.

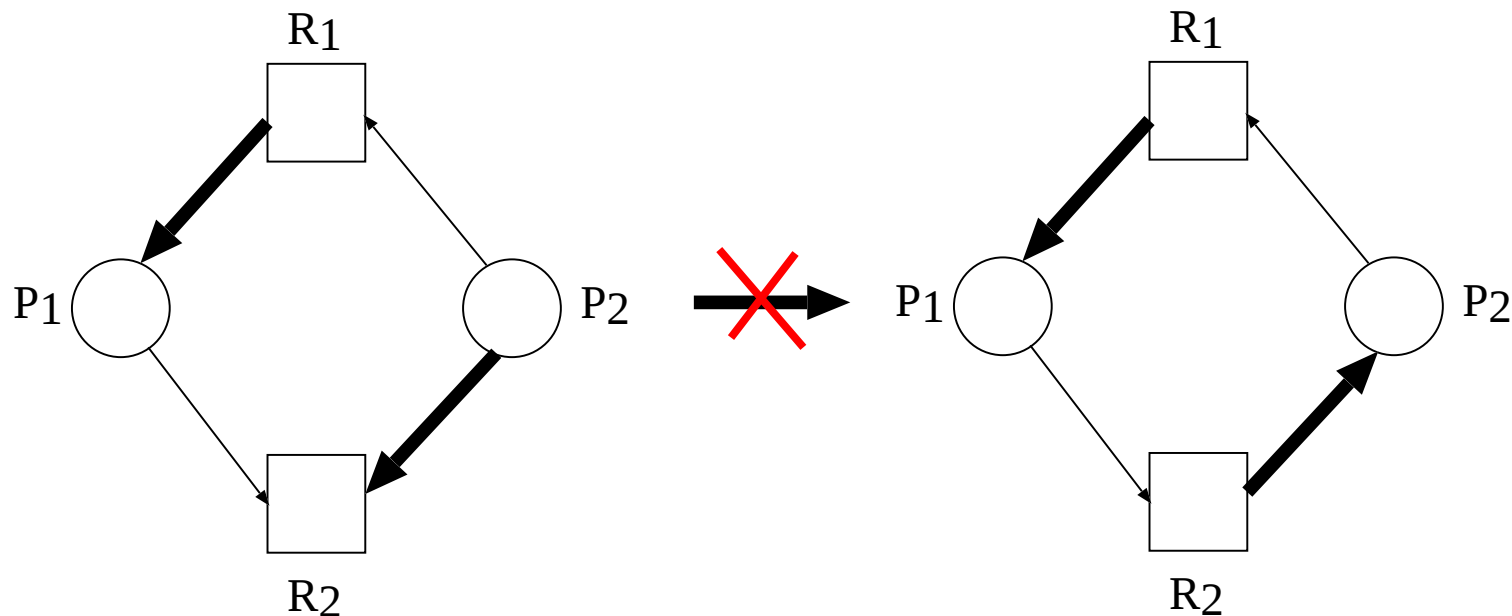
Vyhýbání se uváznutí

– Obecný princip:

- Procesy předem deklarují určité informace o způsobu, jakým budou využívat zdroje: v nejjednodušším případě se jedná o **maximální počet** současně požadovaných zdrojů jednotlivých typů.
- Předem známé informace o možných požadavcích jednotlivých procesů a o aktuálním stavu přidělování se využijí k rozhodování o tom, které požadavky mohou být uspokojeny a které musí počkat, **aby nemohla vzniknout cyklická závislost** na sebe čekajících procesů **ani v nejhorší možné situaci**, která by mohla v budoucnu vzniknout při deklarovaném chování procesů.

– Existují různé **konkrétní algoritmy** pro vyhýbání se uváznutí – např. algoritmus založený na **grafu alokace zdrojů** pro systémy s jednou instancí každého zdroje:

- Systém přidělování zdrojů udržuje graf vztahů mezi procesy a zdroji se dvěma typy uzlů (procesy P_i a zdroje R_j) a třemi typy hran: který zdroj je kým vlastněn ($R_i \Rightarrow P_j$), kdo o který zdroj žádá ($P_i \Rightarrow R_j$), kdo o který zdroj může požádat ($P_i \rightarrow R_j$).
 - POZOR: mluví se zde o typech uzlů/hran, ne o konkrétních uzlech či hranách.
- Zdroj je přidělen pouze tehdy, pokud nehrozí možný vznik cyklické závislosti čekajících procesů, což by se projevilo cyklem v grafu při záměně hrany žádosti za hranu vlastnictví.
 - POZOR: vznik cyklu v grafu alokace neznamena ještě deadlock, ale možnost deadlocku!



– Zobecněním je tzv. bankéřův algoritmus pro práci s více instancemi zdrojů.

Detekce uváznutí a zotavení

– Uváznutí může vzniknout (pomineme-li to, že je vně uváznutých procesů dodán speciální systém k jeho následnému rozřešení); periodicky se přitom detekuje, zda k němu nedošlo, a pokud ano, provede se zotavení.

– Detekce uváznutí:

- graf vlastnictví zdrojů a čekání na zdroje – podobný jako graf alokace zdrojů, ale bez hran vyjadřujících možnost žádat o zdroj;
- cyklus v tomto grafu (na rozdíl od grafu alokace zdrojů) indikuje uváznutí.

– Zotavení z uváznutí:

- Odebrání zdrojů alespoň některým pozastaveným procesům, jejich přidělení ostatním a později umožnění získat všechny potřebné zdroje a pokračovat (případně jejich restart či ukončení).
- Anulace nedokončených operací (*rollback*), nebo nutnost akceptace možných nekonzistencí.

Systematická verifikace

- Pokud použitý systém synchronizace procesů sám spolehlivě nezabraňuje vzniku uváznutí (či jiných nežádoucích chování), je vhodné ověřit, že nad ním navržené řešení je navrženo tak, že žádné nežádoucí chování nehrozí.
- Možnosti odhalování nežádoucího chování systémů (mj. uváznutí či stárnutí) zahrnují:
 - inspekce systému (nejlépe nezávislou osobou),
 - simulace, testování, vkládání „šumu“ do plánování, dynamická analýza (extrapolace sledovaného chování),
 - metody verifikace s formálními kořeny či formální verifikace,
 - nebo kombinace všech uvedených přístupů.
- **Formální verifikace** (na rozdíl od simulace a testování) umožňuje nejen vyhledávat chyby, ale také dokázání správnosti systému s ohledem na zadaná kritéria (což znamená, že žádné chyby nezůstaly bez povšimnutí).
- Experimentuje se i s kombinacemi dynamické analýzy za běhu systému s automatickou opravou, nebo alespoň omezením projevů chyby (např. vložení synchronizace – možnost uváznutí (!), vynucení přepnutí kontextu a získání celého časového kvanta před kritickou sekcí, ...).

– **Proces formální verifikace:**

- vytvoření modelu (lze přeskočit při práci přímo se systémem – případně se vytváří model okolí ověřované části systému),
- specifikace vlastnosti, kterou chceme ověřit (některé vlastnosti mohou být generické – např. absence deadlocku, null pointer exceptions apod.),
- (automatická) kontrola, zda model splňuje specifikaci.

– Základní přístupy k formální verifikaci zahrnují:

- *model checking*,
- *theorem proving*,
- *static analysis*.

– Nad rámec verifikace jde **automatická syntéza** dle zadané specifikace.

– Theorem proving:

- Využívá (typicky) poloautomatický dokazovací prostředek (PVS, Isabel, Coq, ACL/2, ...).
- Vyžaduje obvykle experta, který určuje, jak se má důkaz vést (prestože se objevila řada automatických rozhodovacích procedur pro různé logické fragmenty – lze užít pro automatické ověřování fragmentů kódu bez cyklů (automaticky/ručně dodané anotace cyklů a funkcí) či v kombinaci s jinými přístupy).

– Model checking:

- Využívá obvykle automatický prostředek (Spin, Blast, JPF, CPAchecker, Divine, CBMC, JBMC, ...).
- Využívá typicky generování a prohledávání stavového prostoru.
- Hlavní nevýhodou je problém stavové exploze, kdy velikost stavového prostoru roste exponenciálně s velikostí modelu, případně práce s neomezeným počtem stavů.

– Static analysis:

- Snaha o ověření příslušných vlastností na základě modelu či zdrojového kódu, aniž by se tento prováděl (příp. se provádí jen na určité abstraktní úrovni).
- Různé podoby: data flow analysis, constraint analysis, type analysis, abstract interpretation, symbolic execution, ...
- Řada nástrojů: Facebook Infer, Frama-C, Microsoft SDV, Klee/Symbiotic, AbsInt, Coverity Scan, Klocwork, SpotBugs, cppcheck, rozšíření gcc/clang, ...

Verifikace na FIT – VeriFIT

– Řada témat sahajících od **teoretického výzkumu**, přes **pokročilé algoritmy a datové struktury** potřebné pro efektivní implementaci verifikačních metod po **reálné případové studie**. Konkrétněji:

- statická analýza s formálními kořeny (např. s využitím Facebook Infer či Frama-C),
- formální verifikace programů s ukazateli a dynamickými datovými strukturami (např. různé seznamy či stromy), s řetězci, poli, paralelními procesy, ...,
- pokročilé testování, fuzz testování, testování řízené pokrytím, dynamická analýza,
- automatická syntéza či optimalizace,
- modelem řízený návrh a analýza,
- teorie jazyků a automatů a její využití (nejen ve verifikaci – např. „pattern matching“ ve spolupráci s Microsoft Research), teorie různých logik.

– **Spolupráce**: Uppsala, Academia Sinica, TU Wien, TU Munich, Verimag Grenoble, IRIF Paris, RWTH Aachen, Oxford, ..., Red Hat, Honeywell, Oracle Labs, DiffBlue, UNIS, Microsoft Research, ...

– **Projekty**: GAČR, TAČR, ERC.CZ, H2020 ECSEL, ...

– Související magisterské specializace **Verifikace a testování software** a **Matematické metody v IT**.

– **Zájemci o projektovou praxi, bakalářskou práci, diplomovou práci, disertační práci vítáni!**