Paralelní programování

přednášky

Jan Outrata

únor–duben 2011

Literatura

- Ben-Ari M.: *Principles of concurrent and distributed programming*. Addison-Wesley, 2006. ISBN 9780321312839
- Andrews G. R.: *Concurrent programming: principles and practice*. Addison-Wesley, 1991. ISBN 0805300864
- Andrews G. R.: *Foundations of Multithreaded, Parallel, and Distributed Programming*. Addison-Wesley, 2000. ISBN 0201357526
- Schneider F. B.: *On concurrent programming*. Springer, 1997. ISBN 0387949429
- Magee J., Kramer J.: *Concurrency, State Models and Java Programs*. John Wiley and Sons Ltd, 1999.

Úvod do paralelního a distribuovaného programování

- dnešní programy ve své podstatě paralelní nebo distribuované: např. událostmi řízená UI, operační a řídící systémy, víceuživatelské síťové aplikace, ...
- podporováno moderními programovacími jazyky, např. Java, C#
- technologie se mění, ale stále stejné
  - **principy**: prokládání, vzájemné vyloučení, bezpečnost, živost
  - **základní problémy**: kritická sekce, product-konzument, čtenáři a písaři aj.
  - klasické **nástroje řešení**: semafor, monitor, zprávy aj. (vznikají i nové, např. dispatch queues)
- *Co je tedy nové?* Dnes je běžné, ba přímo nutné, vzhledem k vývoji hardware.
- **Problém**: programy nelze „nahackovat“, je potřeba **formálních metod** pro jeho specifikaci a ověření [ponecháno do předmětu navazujícího studia]
- **výuka principů** (ukázky v pseudokódu, konkrétní jazyk na cvičení)

Konkurentní (concurrent) programování

- „klasický“ program – sekvenční, (protože) instrukce jsou vykonávány procesorem sekvenčně
- **paralelní program** = „několik sekvenčních programů“ (procesů) **běžících současně** na samostatných procesorech
- paralelní programování = konstrukce paralelního programu
- **konkurentní program** = „několik sekvenčních programů“ (procesů), které **mohou být vykonávány současně**

= potenciálně paralelní programování – paralelizace může být zdánlivá, řešený sdílením zdrojů (procesoru)

- **konkurence** = abstrakce, předpokládání paralelního zpracování
- např. v OS obsluhy přerušení od hardware vykonávány konkurentně s programy, multitasking, multiprocesoring, distribuované systémy atd.
- **multithreading** v prog. jazycích – konkurentní vlákna v programu, např. UI a výpočet
- terminologie: proces vs. vlákno (thread)

Konkurentní (concurrent) programování

- procesy mohou (musí) **interagovat** a **komunikovat** $\Rightarrow$ potřeba je **synchronizovat**
- (mnohem) těžší než u sekvenčních programů docílit korektnosti a efektivnosti, chyby např. „zamrznutí“, „pády“ aplikací
- problémy **závislé v čase i situaci**, obtížné reprodukovat, diagnostikovat a opravit

Abstrakce konkurentního programu

- **konkuretní program** = (konečná) množina (sekvenčních) procesů
- procesy vykonávají (konečné) množství **atomických akcí**
  - dále nedělitelných = neproložitelných jiným procesem
  - izolovaných = dočasné stavy neviditelné
- **scénář (historie)** = posloupnost **libovolně proložených** atomických akcí procesů, zachováno pořadí akcí v rámci procesu
- následující atom. akce je **(nedeterministicky) vybrána** z následujících atom. akcí procesů, bez omezení (až na jednu výjimku)

Př. 2 procesy A a B s (neřídícími) akcemi $A1 \rightarrow A2$ a $B1 \rightarrow B2$. Možné scénáře:

Abstrakce konkurentního programu

- **konkuretní program** = (konečná) množina (sekvenčních) procesů
- procesy vykonávají (konečné) množství **atomických akcí**
  - dále nedělitelných = neproložitelných jiným procesem
  - izolovaných = dočasné stavy neviditelné
- **scénář (historie)** = posloupnost **libovolně proložených** atomických akcí procesů, zachováno pořadí akcí v rámci procesu
- následující atom. akce je **(nedeterministicky) vybrána** z následujících atom. akcí procesů, bez omezení (až na jednu výjimku)

Př. 2 procesy A a B s (neřídícími) akcemi $A1 \rightarrow A2$ a $B1 \rightarrow B2$. Možné scénáře:

$$A1 \rightarrow B1 \rightarrow A2 \rightarrow B2 \qquad B1 \rightarrow A1 \rightarrow B2 \rightarrow A2$$
$$A1 \rightarrow B1 \rightarrow B2 \rightarrow A2 \qquad B1 \rightarrow A1 \rightarrow A2 \rightarrow B2$$
$$A1 \rightarrow A2 \rightarrow B1 \rightarrow B2 \qquad B1 \rightarrow B2 \rightarrow A1 \rightarrow A2$$

$A2 \rightarrow A1 \rightarrow B1 \rightarrow B2$ není scénář. *Proč?*

- **synchronizace** = omezení počtu scénářů na nevedoucí k chybám, tzn. omezení výběru následující atom. akce

Zápis

Triviální konkurentní program				název programu
int n $\leftarrow$ 0				globální proměnné
				globální atom. akce
A		*B*		označní procesů
	int a $\leftarrow$ 1		int b $\leftarrow$ 2	lokální proměnné
1:	n $\leftarrow$ a	1:	n $\leftarrow$ b	atom. akce

Obrázek: Konkuretní program

Triviální sekvenční program
int n $\leftarrow$ 0
int a $\leftarrow$ 1
int b $\leftarrow$ 2
1: n $\leftarrow$ a
2: n $\leftarrow$ b

Obrázek: Sekvenční program

Zápis

Triviální konkurentní program				název programu
int n ← 0				globální proměnné
				globální atom. akce
A		*B*		označní procesů
	int a ← 1		int b ← 2	lokální proměnné
1:	n ← a	1:	n ← b	atom. akce

Obrázek: Konkuretní program

Triviální sekvenční program	
int n ← 0	
	int a ← 1
	int b ← 2
1:	n ← a
2:	n ← b

Obrázek: Sekvenční program

Stavy

- výpočet definován pomocí stavů a přechodů mezi nimi
- **stav** = n-tice aktuálních následujících atom. akcí procesů a hodnot globálních a lokálních proměnných
- přechody mezi stavy = vykonání následujících atom. akcí procesů
- **stavový diagram** = diagram (dosažitelných) stavů a přechodů mezi nimi

str. 11

Obrázek: Stavový diagram sekvenčního a konkurentního programu

Scénář (historie)

- **scénář** = posloupnost stavů dle přechodů mezi nimi, orientovaná cesta stavovým diagramem z počátečního stavu

A	B	n	$A.a$	$B.b$
1: n ← a	1: n ← b	0	1	2
konec	**1: n ← b**	1	1	2
konec	konec	2	1	2

Obrázek: Zápis scénáře

Paralelní architektury a abstrakce

- **multitasking** (time-sharing): 1 procesor, plánovač, přepnutí kontextu procesů (uložení lokálních proměnných) kdykoliv, tedy je možné lib. proložení instrukcí
- **multiprocesoring**: globální sdílená a lokální paměťi procesorů, (skutečně) paralelní vykonávání instrukcí, při práci s lokální pamětí nerozlišitelné od proložení instrukcí, u globální není přípustný paralelní přístup a je možné lib. pořadí procesorů, tedy i proložení instrukcí
- libovolné proložení atom. akcí = **ignorování času akcí a mezi akcemi**, umožněna formální analýza a nezávislé na aktuálním hardware, software a podmínkách běhu (!), důležité pouze pořadí (spočetného množství) akcí, které může být libovolné
- interakce procesů pomocí **globální sdílené paměti** (obecně sdíleného zdroje)

Distribuované programování

- **distribuovaný systém**: více procesorů (počítačů), žádná globální sdílená paměť, **posílání zpráv** mezi procesory (uzly) skrze kanály (orient. hrany)
- (skutečně) paralelní vykonávání atom. akcí, je možné lib. proložení (jen zpráva je před přijetím odeslána)
- **není globální sdílená paměť**, uzel nemůže přímo poslat zprávu všem ostatním, potřeba uvažovat **topologii uzlů**
- studium chování při chybách uzlů – „obejití“ uzlu
- [ponecháno do předmětu navazujícího studia]

Atomické akce

- dále **nedělitelná** = neproložitelná jiným procesem
- **izolovaná** = dočasné stavy neviditelné

– výsledek „současného“ vykonání = výsledek sekvenčního vykonání (v libovolném pořadí)

– podle potřeby různé úrovně abstrakce:

  - **hrubé (coarse-grained)**: např. volání funkcí
  - **jemné (fine-grained)**: např. příkaz, instrukce procesoru

– ale korektnost algoritmu závisí na zvolené úrovni, tj. specifikaci atom. akcí

Atomická inkrementace

Př.

Atomická inkrementace	
int n $\leftarrow$ 0	
A	*B*
1: n $\leftarrow$ n + 1	1: n $\leftarrow$ n + 1

A	*B*	*n*
1: n $\leftarrow$ n + 1	1: n $\leftarrow$ n + 1	0
konec	**1: n $\leftarrow$ n + 1**	1
konec	konec	2

A	*B*	*n*
1: n $\leftarrow$ n + 1	**1: n $\leftarrow$ n + 1**	0
1: n $\leftarrow$ n + 1	konec	1
konec	konec	2

Obrázek: Atomická inkrementace na hrubé úrovni abstrakce a možné scénáře

Jsou to všechny možné scénáře?

Program je korektní (vzhledem k postkondici $n = 2$).

Problém: Akce $\mathbf{n = n + 1}$ (s glob. proměnnou n) na hardware **nebývá atomická**, instrukce jsou vykonávány na nižší (jemnější) úrovni abstrakce!

Architektury (virtuálního) hardware

- **registrové** – instrukce prováděny s registry v procesoru, data načítána z (load) a zapisována do (store) paměti, registry $\sim$ lokální proměnné (procesor má vlastní sadu nebo kontext)
  Př.

Neatomická inkrementace			
int n $\leftarrow$ 0			
A		*B*	
1:	load R, n	1:	load R, n
2:	add R, 1	2:	add R, 1
3:	store R, n	3:	store R, n

Obrázek: Inkrementace na registrovém stroji

Architektury (virtuálního) hardware

- **zásobníkové** – instrukce prováděny s vrcholem zásobníku „v procesoru“, data načítána z (push) a zapisována do (pop) paměti, položky zásobníku $\sim$ lokální proměnné („procesor“ má vlastní zásobník)
  Př.

Neatomická inkrementace			
int n $\leftarrow$ 0			
A		*B*	
1:	push n	1:	push n
2:	push 1	2:	push 1
3:	add	3:	add
4:	pop n	4:	pop n

Obrázek: Inkrementace na zásobníkovém stroji

– **instrukce**, včetně načítání z a zápisu do paměti, jsou **atomické**
– **abstrakce** registrů nebo vrcholu zásobníku **pomocí lokálních proměnných**

Neatomická inkrementace

Př.

Neatomická inkrementace	
int n ← 0	
A	*B*
int tmp	int tmp
1: tmp ← n	1: tmp ← n
2: n ← tmp + 1	2: n ← tmp + 1

A	*B*	*n*	*A.tmp*	*B.tmp*
1: tmp ← n	1: tmp ← n	0	?	?
2: n ← tmp + 1	1: tmp ← n	0	0	?
konec	**1: tmp ← n**	1		?
konec	**2: n ← tmp + 1**	1		1
konec	konec	2		

A	*B*	*n*	*A.tmp*	*B.tmp*
1: tmp ← n	1: tmp ← n	0	?	?
2: n ← tmp + 1	**1: tmp ← n**	0	0	?
2: n ← tmp + 1	2: n ← tmp + 1	0	0	0
konec	**2: n ← tmp + 1**	1		0
konec	konec	1		

Obrázek: Neatomická inkrementace na jemnější úrovni abstrakce a možné scénáře

Neatomická inkrementace

Kolik je všech možných scénářů?

Výsledek programu je 2, dokud akce 1 a 2 jsou ihned po sobě, neproložené.

Program je nekorektní (vzhledem k postkondici $n = 2$).

- **race condition (chyba souběhu)** = neplatný výsledek výpočru v důsledku (nevhodného) proložení akcí

Př.

Počítadlo	
int n $\leftarrow$ 0	
A	*B*
int tmp	int tmp
1: do 10 times	1: do 10 times
2: tmp $\leftarrow$ n	2: tmp $\leftarrow$ n
3: n $\leftarrow$ tmp + 1	3: n $\leftarrow$ tmp + 1

Obrázek: Demonstrace konkurentního počítadla

Jakých všech možných hodnot může n po výpočtu nabývat?

Kritická reference

Kdy je potřeba analýza na jemnější úrovni abstrakce?

- výskyt proměnné (v akci) je **kritická reference (KR)**, jestliže
  (a) do proměnné je přiřazováno (zapisováno) a je referencována (čtena) v jiném procesu nebo
  (b) proměnná je referencována (čtena) a je do ní přiřazováno (zapisováno) v jiném procesu
- **podmínka kritických referencí (KR)** = každá akce programu obsahuje **nejvýše jednu kritickou referenci**
  – nesplnění podmínky KR pro akci = **interference** akce s akcí (akcemi) v jiném procesu, může vést k chybám souběhu

Př. Akce $n = n + 1$ v příkladu inkrementace nesplňuje podmínku kritických referencí, zatímco akce s lokální proměnnou *tmp* ano. Proč?

Podmínka kritických referencí

Atomičnost akcí programu

Konkuretní program, který splňuje podmínku kritických referencí (KR), vykazuje stejná chování (dosahuje stejných výsledků), ať jsou jeho akce atomické nebo jsou složeny z jednodušších akcí s atomickým přiřazením a referencováním (globální) proměnné.

- **akce splňující podmínku KR** $\sim$ **atomická** akce na jemné abstraktní úrovni (stroje)
- při převodu hrubé atom. akce (programu), která (který) nesplňuje podmínku KR, na posloupnost jemnějších atom. akcí (program) splňujících (splňující) podmínku KR a tím **vyloučení interferencí** může být potřeba další **sychronizace**

Neatomické a volatilní proměnné

Neatomické proměnné

= proměnné (větších) datových typů, ze kterých není čteno a/nebo do kterých není zapisováno strojem atomicky → možnost chyb souběhu
⇒ (v případě globální proměnné) potřeba **synchronizace pro zajištění atomicity**

- ostatní tzv. **atomické proměnné** = proměnné atomického datového typu, např. int (slovo stroje)

Volatilní proměnné

– proměnná, do které je přiřazeno, nemusí být v rámci akce uložena do paměti, může být držena v registrech nebo na vrcholu zásobníku a do paměti uložena později, kvůli **optimalizaci**
– navíc, v rámci optimalizací, může být změněno pořadí akcí v procesu (které nemá vliv na chování procesu)

⇒ v jiném procesu může být hodnota (globální) proměnné **neaktuální** → možnost chyb souběhu

= proměnná, která je **načtena** z paměti **v rámci reference** a **uložena** do paměti **v rámci přiřazení**

Korektnost

- **sekvenční program** se při každém spuštění se stejnými daty na vstupu chová stejně (tzn. vrátí stejný výsledek), tj. má **jediný scénář** vykonávání

$\Rightarrow$ má smysl ladit („debugovat“)

- **konkurentní program** může mít **více scénářů** vykonávání s různými chováními (výsledky)

$\Rightarrow$ nelze klasicky ladit – při každém spuštění může být jiný scénář

$\rightarrow$ **řešení problémů**, které vznikají v důsledku proložení akcí (využívajících globální proměnné)

- **(částečná) korektnost sekvenčního programu**: (jestliže) skončí, a (pak) výstup je správný vzhledem k podmínkám na vstupu (**prekondice**) a výstupu (**postkondice**)

- konkurentní program nemusí skončit (může to být chyba!) a přitom může být „korektní“

Korektnost

= **(korektnost konkuretního programu)** definována pomocí **vlastností** výpočtů (scénářů):

- **bezpečnosti (safety)** = tvrzení (resp. negace tvrzení), které je **vždy pravdivé** (resp. nepravdivé), tzn. v každém stavu každého výpočtu, např. „program nikdy nezatuhne“
- **živosti (liveness)** = tvrzení, které je **někdy pravdivé**, tzn. v nějakém stavu každého výpočtu, např. „program se někdy rozběhne“

- bezpečnost jednoduché dosáhnout, např. triviálně, těžší splnit živost bez porušení bezpečnosti
- bezpečnost a živost jsou **duální** vlastnosti – negace jedné je druhá
- definována pro každý výpočet (dle každého scénáře) $\Rightarrow$ nemožné ukázat testováním programu, analýza všech scénářů náročná $\rightarrow$ **formální metody ověření** [ponecháno do předmětu navazujícího studia]

Férovost

- **výjimka** z požadavku na scénář výpočtu jako posloupnosti libovolně proložených atom. akcí procesů (a tedy výběru následující atom. akce z následujících atom akcí procesů bez omezení) = **scénář zcela bez atom. akcí nějakého procesu**

- scénář je **(slabě) férový**, jestliže každá akce procesu, která je **neustále povolená** (tj. může být vykonána), se někdy objeví ve scénáři (a bude vykonána)
  - akce přiřazení a řídící akce jsou neustále povolené
- scénář je **silně férový**, jestliže každá akce procesu, která je **opakovaně povolená**, se někdy objeví ve scénáři
  - povolení a zakázání akce viz dále
- omezení scénářů na férové závisí na **férovosti plánovací politiky** paralelní architektury, např. cyklická (round-robin) s časovými kvanty je slabě férová, cyklická s výběrem po každé atom. akci je silně férová

$\Rightarrow$ korektnost programu závisí na férovosti

Férovost

Př.

Přerušení cyklu	
int n $\leftarrow$ 0 bool flag $\leftarrow$ false	
A	*B*
1: while flag = false 2: n $\leftarrow$ 1 - n	1: flag $\leftarrow$ true

Obrázek: Demostrace (slabé) férovosti

Zastaví algoritmus vždy (pro všechny scénáře)? Tj. je korektní vzhledem k podmínce (postkondici), že vždy zastaví?

Problém kritické sekce

- Dijkstra, 1965
- centrální problém, nekorektní řešení demonstrují typické chyby konkurentních programů

Definice problému

- N procesů vykonává (v nekonečné smyčce) posloupnost akcí rozdělenou na dvě sekce: **kritickou** a **nekritickou sekci**
- korektnost řešení:
  1. **vzájemné vyloučení** – akce krit. sekcí (dvou a více) procesů nesmějí být proloženy, tj. (svou) krit. sekci může v daném čase vykonávat nejvýše jeden proces
  2. **absence uváznutí (deadlock)** – jestliže se nějaké procesy snaží současně vstoupit do (svých) krit. sekcí, pak jeden z nich musí někdy uspět
     - **uváznutí (deadlock)** = procesy se snaží současně vstoupit do (svých) krit. sekcí, ale nikdy nemohou
  3. **absence vyhladovění (starvation, zaručení vstupu)** – jestliže se nějaký proces snaží vstoupit do (své) krit. sekce, pak musí někdy uspět

Problém kritické sekce

→ synchronizace = další akce před a po krit. sekci – **vstupní** a **výstupní protokol** (**preprotocol** a **postprotocol**)

Problém kritické sekce	
globální proměnné	
A	*B*
lokální proměnné	lokální proměnné
loop forever	loop forever
nekritická sekce	nekritická sekce
vstupní protokol	vstupní protokol
kritická sekce	kritická sekce
výstupní protokol	výstupní protokol

Obrázek: Struktura (řešení) problému kritické sekce pro 2 procesy

Problém kritické sekce

- podmínky řešení:
  - **proměnné** použité v protokolech jsou **různé** od proměnných v sekcích
  - **krit. sekce vždy skončí** – jsou provedeny všechny její akce, nutné pro umožnění krit. sekcí jiných procesů
  - **nekrit. sekce nemusí skončit** – proces se v ní může ukončit (korektně i nekorektně) nebo může nekonečně cyklit
- efektivnost řešení – protokoly co nejjednodušší časově i paměťově
- **použití**: modelování přístupu k datům nebo hardwaru sdílených mezi více procesy, kdy není dovolen vícenásobný přístup, typicky **operace čtení–zápis** a **zápis–zápis**

První pokus

První pokus	
int turn $\leftarrow$ 1	
A	*B*
loop forever	loop forever
1: nekritická sekce	1: nekritická sekce
2: await turn $=$ 1	2: await turn $=$ 2
3: kritická sekce	3: kritická sekce
4: turn $\leftarrow$ 2	4: turn $\leftarrow$ 1

Obrázek: První pokus o řešení problému krit. sekce pro 2 procesy

await

await podmínka

= na implementaci nezávislé **čekání na splnění podmínky** = **podmíněná synchronizace**

- pokud podmínka splňuje podmínku kritických referencí, může být implementováno prázdnou **čekací smyčkou** (**busy-wait loop**) dokud není podmínka pravdivá, tzv. **aktivní čekání**

	Busy-wait loop
1:	while not podmínka
2:	nic

Obrázek: Aktivní čekání

await

- uváznutí (deadlock) $\sim$ zastavení výpočtu, s aktivním čekáním $\sim$ nekonečný cyklus testování podmínky = **livelock**
- při splnění podmínky akce **povolená**, jinak **zakázaná**

Přerušení cyklu 2			
int n $\leftarrow$ 0			
bool flag $\leftarrow$ false			
A		*B*	
1:	while flag = false	1:	await n = 1
2:	n $\leftarrow$ 1 - n	2:	flag $\leftarrow$ true

Obrázek: Demonstrace silné férovosti

Zastaví algoritmus vždy (pro všechny scénáře)? Tj. je korektní vzhledem k podmínce (postkondici), že vždy zastaví?

První pokus

První pokus	
int turn $\leftarrow$ 1	
A	*B*
loop forever	loop forever
1: nekritická sekce	1: nekritická sekce
2: await turn $= 1$	2: await turn $= 2$
3: kritická sekce	3: kritická sekce
4: turn $\leftarrow$ 2	4: turn $\leftarrow$ 1

Obrázek: První pokus o řešení problému krit. sekce pro 2 procesy

Je toto řešení problému krit. sekce korektní?

Dokazování korektnosti pomocí stavových diagramů

= **„neformální“ důkaz**

– pro určení efektu další akce procesu není nutné znát předchozí historii programu, tj. nezáleží na tom, jak se během výpočtu k akci došlo

– u sekvenčních programů je známa jediná další akce jednoho procesu, u konkurentních ne, ale jsou známy všechny možné další akce všech procesů

= **inkrementální konstrukce stavového diagramu** programu z počátečního stavu **a ověření všech dosažitelných stavů** a přechodů mezi nimi

Kolik může být stavů?

– pro N procesů s n_i akcemi v procesu $i, 1 \leq i \leq N$ a M proměnnými s m_j možnými hodnotami proměnné $j, 1 \leq j \leq M$

= počet všech n-tic nezávisle vybraných akcí a proměnných:
$\mathbf{n_1} \times \ldots \times \mathbf{n_N} \times \mathbf{m_1} \times \ldots \times \mathbf{m_M}$

Dokazování korektnosti pomocí stavových diagramů

Př. Počet stavů prvního pokusu

2 procesy s 4 a 4 akcemi, 1 proměnná s 2 možnými hodnotami (1 a 2) $\rightarrow$ počet stavů: $n_1 \times n_2 \times m_1 = 4 \times 4 \times 2 = 32$

- **ne všechny stavy jsou dosažitelné** v jakémkoliv scénáři z počátečního stavu

Fig. 3.1

Obrázek: Stavový diagram prvního pokusu (část)

Př. Počet dosažitelných stavů prvního pokusu: 16

- pro synchronizaci (a důkaz korektnosti) jsou **relevantní jen akce a proměnné v protokolech**, ne v sekcích samotných – používají různé proměnné, neovlivní synchronizaci

$\rightarrow$ **odstranění** (zakomentování) **akcí sekcí** a ponechání jen akcí protokolů

Dokazování korektnosti pomocí stavových diagramů

První pokus (zkrácení)	
int turn $\leftarrow$ 1	
A'	B'
loop forever	loop forever
1: await turn $=$ 1	1: await turn $=$ 2
2: turn $\leftarrow$ 2	2: turn $\leftarrow$ 1

Obrázek: První pokus o řešení problému krit. sekce pro 2 procesy (zkrácení)

Př. Počet stavů prvního pokusu (zkráceného): $2 \times 2 \times 2 = 8$

Fig. 3.2

Obrázek: Stavový diagram prvního pokusu (zkráceného)

Dokazování korektnosti řešení problému krit. sekce

= ověření splnění 3 vlastností korektnosti

Vzájemné vyloučení

- v každém stavu lib. scénáře stavového diagramu je nejvýše 1 následující akce z krit. sekce nějakého procesu, jinak **nežádoucí stav**
- důkaz splnění = absence nežádoucího stavu ve všech scénářích $\Rightarrow$ nutné zkonstruovat celý stavový diagram
- důkaz nesplnění = prezence nežádoucího stavu v lib. scénáři $\Rightarrow$ není nutné zkonstruovat celý stavový diagram

Absence uváznutí a vyhladovění

- absence stavu s akcí vstupního protokolu, ze kterého se nelze dostat do stavu s akcí krit. sekce
- u uváznutí pro více procesů (všechny procesy) zároveň
- u vyhladovění pro každý jeden proces zvlášť

Korektnost prvního pokusu

1. vzájemné vyloučení: ve stavovém diagramu není nežádoucí stav $(A'2, B'2, turn)$, pro lib. hodnotu proměnné *turn* (1 nebo 2), kdy oba procesy jsou v krit. sekci
2. absence uváznutí: proces se snaží vstoupit do krit. sekce ve stavech s akcí await, ze stavu $(A'1, B'1, 1)$ může vstoupit proces A', ze stavu $(A'2, B'1, 1)$ může pokračovat proces A' do stavu $(A'1, B'1, 2)$, ze kterého může vstoupit proces B', zvývající 2 stavy analogicky
3. absence vyhladovění: ze stavu $(A1, B2, 1)$ nemůže proces B vstupit do krit. sekce, dokud proces A nevykoná akci $A4$, ale ten je v nekrit. sekci a ta nemusí skončit!

Korektnost prvního pokusu

- proměnná *turn* funguje jako **„právo“ procesu ke vstupu do krit. sekce**
- proces čeká na „právo“ vstupu do (své) krit. sekce
- vždy nějaký proces má „právo“ $\rightarrow$ není uváznutí
- proces nemusí „právo“ předat, např. zůstane v nekrit. sekci $\rightarrow$ vyhladovění

$\rightarrow$ proces v nekrit. sekci nemůže bránit vstupu jiného do krit. sekce

$\rightarrow$ **procesy nemohou testovat a nastavovat jedinou (glob.) proměnnou**

Korektnost prvního pokusu

- proměnná *turn* funguje jako **„právo“ procesu ke vstupu do krit. sekce**
- proces čeká na „právo“ vstupu do (své) krit. sekce
- vždy nějaký proces má „právo“ $\rightarrow$ není uváznutí
- proces nemusí „právo“ předat, např. zůstane v nekrit. sekci $\rightarrow$ vyhladovění

$\rightarrow$ proces v nekrit. sekci nemůže bránit vstupu jiného do krit. sekce

$\rightarrow$ **procesy nemohou testovat a nastavovat jedinou (glob.) proměnnou**

Druhý pokus

→ každý proces má svoji (glob.) proměnnou

Druhý pokus	
bool isA ← false, isB ← false	
A	*B*
loop forever	loop forever
1: nekritická sekce	1: nekritická sekce
2: await isB = false	2: await isA = false
3: isA ← true	3: isB ← true
4: kritická sekce	4: kritická sekce
5: isA ← false	5: isB ← false

Obrázek: Druhý pokus o řešení problému krit. sekce pro 2 procesy

- proměnné *isA* a *isB* fungují jako **„oznámení“ o vykonávání krit. sekce**, dokud tato neskončí
- proces čeká na zrušení „oznámení“ jiných procesů

Druhý pokus

- proces nebrání vstupu jiného do krit. sekce, např. když zůstane v nekrit. sekci, jeho „oznámení“ zůstává zrušené, a nemůže dojít k uváznutí
- vzájemné vyloučení není splněné: *scénář?*
- stav po čekání, ale před nastavením „oznámení“ o vykonávání krit. sekce, je efektivně součástí krit. sekce, ale „oznámení“ ještě není nastaveno a jiné procesy nemusí čekat

→ **procesy nemohou měnit podmínky vstupu jiných procesů do (jejich) krit. sekcí až v (své) krit. sekci**

Druhý pokus

- proces nebrání vstupu jiného do krit. sekce, např. když zůstane v nekrit. sekci, jeho „oznámení“ zůstává zrušené, a nemůže dojít k uváznutí
- vzájemné vyloučení není splněné: *scénář?*
- stav po čekání, ale před nastavením „oznámení“ o vykonávání krit. sekce, je efektivně součástí krit. sekce, ale „oznámení“ ještě není nastaveno a jiné procesy nemusí čekat

→ **procesy nemohou měnit podmínky vstupu jiných procesů do (jejich) krit. sekcí až v (své) krit. sekci**

Druhý pokus

- proces nebrání vstupu jiného do krit. sekce, např. když zůstane v nekrit. sekci, jeho „oznámení“ zůstává zrušené, a nemůže dojít k uváznutí
- vzájemné vyloučení není splněné: *scénář?*
- stav po čekání, ale před nastavením „oznámení“ o vykonávání krit. sekce, je efektivně součástí krit. sekce, ale „oznámení“ ještě není nastaveno a jiné procesy nemusí čekat

→ **procesy nemohou měnit podmínky vstupu jiných procesů do (jejich) krit. sekcí až v (své) krit. sekci**

Třetí pokus

→ „oznámení“ je nastaveno už před čekáním

Třetí pokus	
bool wantA ← false, wantB ← false	
A	*B*
loop forever	loop forever
1: nekritická sekce	1: nekritická sekce
2: wantA ← true	2: wantB ← true
3: await wantB = false	3: await wantA = false
4: kritická sekce	4: kritická sekce
5: wantA ← false	5: wantB ← false

Obrázek: Třetí pokus o řešení problému krit. sekce pro 2 procesy

- proměnné *wantA* a *wantB* fungují jako **„přání“ vstupu do krit. sekce**, dokud tato neskončí
- proces čeká na zrušení „přání“ jiných procesů

Třetí pokus

- vzájemné vyloučení je splněno a k vyhladovění procesu dojít nemůže
- může dojít k uváznutí: *scénář?*
- proměnné ve skutečnosti fungují jako **„trvání na“ vstupu do krit. sekce**

→ uváznutí = oba procesy současně „trvají na“ vstupu do krit. sekce

→ **procesy si nemohou vynucovat vstup do (své) krit. sekce**

Třetí pokus

- vzájemné vyloučení je splněno a k vyhladovění procesu dojít nemůže
- může dojít k uváznutí: *scénář?*
- proměnné ve skutečnosti fungují jako **„trvání na“ vstupu do krit. sekce**

→ uváznutí = oba procesy současně „trvají na“ vstupu do krit. sekce

→ **procesy si nemohou vynucovat vstup do (své) krit. sekce**

Třetí pokus

- vzájemné vyloučení je splněno a k vyhladovění procesu dojít nemůže
- může dojít k uváznutí: *scénář?*
- proměnné ve skutečnosti fungují jako **„trvání na“ vstupu do krit. sekce**

→ uváznutí = oba procesy současně „trvají na“ vstupu do krit. sekce

→ **procesy si nemohou vynucovat vstup do (své) krit. sekce**

Čtvrtý pokus

→ proces se **„dočasně vzdá trvání na“ vstupu do (své) krit. sekce**, pokud na vstupu do (své) krit. sekce „trvá“ jiný proces

Čtvrtý pokus	
bool wantA ← false, wantB ← false	
A	*B*
loop forever	loop forever
1: nekritická sekce	1: nekritická sekce
2: wantA ← true	2: wantB ← true
3: while wantB	3: while wantA
4: wantA ← false	4: wantB ← false
5: wantA ← true	5: wantB ← true
6: kritická sekce	6: kritická sekce
7: wantA ← false	7: wantB ← false

Obrázek: Čtvrtý pokus o řešení problému krit. sekce pro 2 procesy

Čtvrtý pokus

- akce 4 a 5 za sebou mají v konkuretním programu, vzhledem k lib. proložení akcí, smysl, mezi nimi může druhý proces vstoupit do (své) krit. sekce
- k uváznutí nemůže dojít (není čekání) a vzájemné vyloučení je splněno
- může dojít k vyhladovění procesu: *scénář?*
- scénář vyhladovění „nerealistický“? V našem modelu libovolných proložení je ale **možný**.
- procesy se vyhladoví „navzájem“ $\sim$ „livelock“ ve vstupním protokolu

Čtvrtý pokus

- akce 4 a 5 za sebou mají v konkuretním programu, vzhledem k lib. proložení akcí, smysl, mezi nimi může druhý proces vstoupit do (své) krit. sekce
- k uváznutí nemůže dojít (není čekání) a vzájemné vyloučení je splněno
- může dojít k vyhladovění procesu: *scénář?*
- scénář vyhladovění „nerealistický“? V našem modelu libovolných proložení je ale **možný**.
- procesy se vyhladoví „navzájem“ $\sim$ „livelock“ ve vstupním protokolu

Čtvrtý pokus

- akce 4 a 5 za sebou mají v konkuretním programu, vzhledem k lib. proložení akcí, smysl, mezi nimi může druhý proces vstoupit do (své) krit. sekce
- k uváznutí nemůže dojít (není čekání) a vzájemné vyloučení je splněno
- může dojít k vyhladovění procesu: *scénář?*
- scénář vyhladovění „nerealistický“? V našem modelu libovolných proložení je ale **možný**.
- procesy se vyhladoví „navzájem“ $\sim$ „livelock“ ve vstupním protokolu

Dekkerův algoritmus

Dekkerův algoritmus

bool wantA ← false, wantB ← false
int turn ← 1

	A		*B*
	loop forever		loop forever
1:	nekritická sekce	1:	nekritická sekce
2:	wantA ← true	2:	wantB ← true
3:	while wantB	3:	while wantA
4:	if turn = 2	4:	if turn = 1
5:	wantA ← false	5:	wantB ← false
6:	await turn = 1	6:	await turn = 2
7:	wantA ← true	7:	wantB ← true
8:	kritická sekce	8:	kritická sekce
9:	turn ← 2	9:	turn ← 1
10:	wantA ← false	10:	wantB ← false

Obrázek: Dekkerův algoritmus řešení problému krit. sekce pro 2 procesy

Dekkerův algoritmus

= kombinace prvního a čtvrtého pokusu

– v prvním pokusu se předává „právo“ procesů ke vstupu do krit. sekce – nekorektní v případě, že procesy nesoupeří o vstup

– čtvrtý pokus řeší problém při nesoupeření o vstup, ale v případě soupeření procesy „trvají na“ vstupu do krit. sekce

→ předává se **„právo na trvání na“ vstupu do krit. sekce**

- korektní – důkaz pomocí stavového diagramu?!
- počet stavů (bez akcí sekcí): $8 \times 8 \times 2 \times 2 \times 2 = 512$ → **formální důkaz** pomocí (induktivních a deduktivních) důkazů tzv. **invariantů** v **temporální logice**

Petersonův algoritmus (Tie-breaker)

- Peterson, 1981

Petersonův algoritmus			
bool wantA ← false, wantB ← false int last ← 1			
A		*B*	
	loop forever		loop forever
1:	nekritická sekce	1:	nekritická sekce
2:	wantA ← true	2:	wantB ← true
3:	last ← 1	3:	last ← 2
4:	await wantB = false or last = 2	4:	await wantA = false or last = 1
5:	kritická sekce	5:	kritická sekce
6:	wantA ← false	6:	wantB ← false

Obrázek: Petersonův algoritmus řešení problému krit. sekce pro 2 procesy

Petersonův algoritmus (Tie-breaker)

- založený na Dekkerově algoritmu, cyklus a await složeny do jedné akce **await se složenou podmínkou**
- proměnná *last* indikuje, který proces začal vykonávat vstupní protokol jako poslední a bude (dál) čekat
- podmínka v **await nesplňuje podmínku kritických referencí**

Otázka: *Zůstane algoritmus korektní, i když složená podmínka v await není vyhodnocována atomicky (a await je implementováno aktivním čekáním)?*

Složené atomické akce

- atomické akce při řešení problému kritické sekce byly pouze jednoduché: načtení z a uložení do (sdílené) paměti – těžké problém vyřešit
- lehké, když **atom. akce může číst z i ukládat do (sdílené) paměti**
- implementované v OS nebo hardware

Test-and-set

test-and-set(lock, local)	
1:	local $\leftarrow$ lock
2:	lock $\leftarrow$ true

Obrázek: Složená atomická akce test-and-set

Složené atomické akce

Test-and-set

Problém kritické sekce s test-and-set

bool lock ← false

	A		*B*
	bool localA		bool localB
	loop forever		loop forever
1:	nekritická sekce	1:	nekritická sekce
	repeat		repeat
2:	test-and-set(lock, localA)	2:	test-and-set(lock, localB)
3:	until localA = false	3:	until localB = false
4:	kritická sekce	4:	kritická sekce
5:	lock ← false	5:	lock ← false

Obrázek: Řešení problému krit. sekce pomocí test-and-set

Složené atomické akce

Test-and-set

- proměnná *lock* funguje jako **„zámek“ krit. sekce**, dokud $lock = 1$, není dovoleno vstoupit
- cyklus repeat = **spin lock**
- bývá také implementována tak, že vrací původní hodnotu proměnné *lock*:

test-and-set(lock)	
	bool local
1:	local ← lock
2:	lock ← true
3:	return local

Obrázek: Složená atomická akce test-and-set (vracející hodnotu)

- splnění absence vyhladovění jen v silně férových scénářích

Složené atomické akce

Exchange (atomická výměna hodnot dvou proměnných)

exchange(a, b)	
	bool tmp
1:	tmp $\leftarrow$ a
2:	a $\leftarrow$ b
3:	b $\leftarrow$ tmp

Obrázek: Složená atomická akce exchange

Složené atomické akce

Exchange (atomická výměna hodnot dvou proměnných)

Problém kritické sekce s exchange			
bool lock ← false			
	A		*B*
	bool localA ← true		bool localB ← true
	loop forever		loop forever
1:	nekritická sekce	1:	nekritická sekce
	repeat		repeat
2:	exchange(lock, localA)	2:	exchange(lock, localB)
3:	until localA = false	3:	until localB = false
4:	kritická sekce	4:	kritická sekce
5:	exchange(lock, localA)	5:	exchange(lock, localB)

Obrázek: Řešení problému krit. sekce pomocí exchange

- řešení fungují i pro lib. počet procesů

Složené atomické akce

Fetch-and-add

```
fetch-and-add(counter, local, x)
1:  local ← counter
2:  counter ← counter + 1
```

Obrázek: Složená atomická akce fetch-and-add

Compare-and-swap

```
compare-and-swap(lock, old, new)
    bool tmp
1:  tmp ← lock
2:  if lock = old
3:      lock ← new
4:  return tmp
```

Obrázek: Složená atomická akce compare-and-swap

Složené atomické akce

- každou z uvedených složených atom. akcí lze implementovat pomocí jiné
- místo uvedených složených atom. akcí **lze pro zajištění atomicity** jejich implementace, tj. neproložitelnosti ($\sim$ vzájemného vyloučení) a izolovanosti ($\sim$ použití různých proměnných) jejich akcí, **použít lib. řešení problému krit. sekce s jednoduchými atom. akcemi**!

Problém kritické sekce

- použití aktivního čekání v řešeních problému kritické sekce – **nevýhodné při multitaskingu** (1 procesor, v nepreemptivním dokonce nepoužitelné) **a při větším soupeření procesů o vstup do krit. sekcí** (např. při delších krit. sekcích)

→ tzv. **pasivní čekání** = implementace await pomocí **blokování (pozastavení) procesu plánovačem procesů** v OS

- uvedená řešení problému kritické sekce v praxi nepoužívaná (kromě tzv. **„rychlých“ řešení** dále) → **vyšší synchronizační primitiva**

Semafory

Await

- synchronizace používající await běží na **„železe“** = využívají jen (atomické) instrukce poskytované strojem
- instrukce jsou příliš nízkoúrovňové

Semafor

- Dijkstra, 1968

= klasické synchronizační primitivum implementované v OS

- na vyšší úrovni abstrakce než instrukce
- dvě **atomické operace**: čekání na semaforu, signalizace semaforu
- motivace – semafor na silnici (červená, zelená)

Aktivní implementace semaforu (busy-wait semaphore)

- čekání pomocí **čekací smyčky**

= nezáporné celé **číslo** V

- inicializace na hodnotu $k \geq 0$ – **obecný**, $0 \leq k \leq 1$ – **binární**
- **atomické operace**:

wait(S)	
1:	await $S.V > 0$
2:	$S.V \leftarrow S.V - 1$

Obrázek: Operace *wait*(S) na semaforu S

signal(S)	
1:	$S.V \leftarrow S.V + 1$

signal(S)	
1:	if $S.V = 0$
2:	$S.V \leftarrow 1$

Obrázek: Operace *signal*(S) na obecném a binárním semaforu S

Aktivní implementace semaforu (busy-wait semaphore)

- operace **wait**(**S**) původně (Dijkstra) označovaná **P**(**S**), operace **signal**(**S**) původně označovaná **V**(**S**)
- při operaci wait může proces (aktivně) **čekat na semaforu**
- při operaci signal může **libovolný jeden** proces čekající na semaforu pokračovat (je splněna podmínka await)
- binární semafor je také nazývaný **mutex** – použití pro vzájemné vyloučení (mutual exclusion)
- použití **vhodné při multiprocesoringu** (čekající proces na samostatném procesoru a další volné) **a při menším soupeření procesů o provedení operace wait** (např. při krátkém čekání, ušetření relativně náročného plánování procesů)

Pasivní implementace semaforu

- čekání pomocí **změny stavu procesu** pomocí plánovače procesů v OS na **blokovaný (pozastavený) stav** – **kdykoliv!**:

Obr. 108

Obrázek: Změny stavů procesu

  - při blokovaném stavu je následující akce procesu **zakázaná**, dokud jej jiný proces neodblokuje
  - jen **1** (vybraný) **proces** je v každém čase **běžící**, ostatní připravené – viz jen 1 (vybraná) akce z následujících akcí procesů

= dvojice (V, L), kde V je nezáporné celé **číslo a** L je **množina procesů**

- inicializace na hodnotu $(k, \emptyset)$
- **atomické operace** (A je proces, který operaci vykonává): DALŠÍ SLAJD
- při operaci wait může být proces **blokovaný na semaforu**
- při operaci signal může být **libovolný jeden** proces blokovaný na semaforu odblokován

Pasivní implementace semaforu

wait(S)	
	proces A
1:	if $S.V > 0$
2:	$S.V \leftarrow S.V - 1$
3:	else
4:	$S.L \leftarrow S.L \cup A$
5:	A.state $\rightarrow$ blocked

Obrázek: Operace *wait*(S) na semaforu S

signal(S)	
	proces B
1:	if $S.L = \emptyset$
2:	$S.V \leftarrow S.V + 1$
3:	else
4:	B = libovolný prvek S.L
5:	$B.L \leftarrow B.L \setminus \{B\}$
6:	B.state = ready

signal(S)	
	proces B
1:	if $S.V = 0$
2:	if $S.L = \emptyset$
3:	$S.V \leftarrow 1$
4:	else
5:	B = libovolný prvek S.L
6:	$B.L \leftarrow B.L \setminus \{B\}$
7:	B.state = ready

Obrázek: Operace *signal*(S) na obecném a binárním semaforu S

Problém kritické sekce

– vstupní protokol = čekání na semaforu, výstupní protokol = signalizace semaforu

Pro 2 procesy

Kritická sekce	
binary semaphore $S \leftarrow 1$	
A	B
loop forever	loop forever
1: nekritická sekce	1: nekritická sekce
2: wait(S)	2: wait(S)
3: kritická sekce	3: kritická sekce
4: signal(S)	4: signal(S)

Obrázek: Řešení problému kritické sekce pro 2 procesy

- podobné druhému pokusu o řešení s await, ale operace na semaforu jsou atomické (testování a nastavení hodnoty)

Problém kritické sekce

- důkaz korektnosti pomocí stavového diagramu:

Kritická sekce (zkrácení)			
binary semaphore $S \leftarrow 1$			
A'		B'	
	loop forever		loop forever
1:	wait(S)	1:	wait(S)
2:	signal(S)	2:	signal(S)

Obrázek: Řešení problému kritické sekce pro 2 procesy (zkrácení)

Obr. 6.1

Obrázek: Stavový diagram řešení

Problém kritické sekce

- vzájemné vyloučení: není nežádoucí stav $(A'2, B'2, S)$, pro lib. hodnotu semaforu S, kdy oba procesy jsou v krit. sekci
- absence uváznutí: není stav, ve kterém by oba procesy čekaly/byly blokované
- absence vyhladovění s aktivními semafory: když proces nemůže vstoupit do krit. sekce, je v ní druhý proces, který ale po ukončení krit. sekce do ní může opět vstoupit (je vybrán)! $\rightarrow$ potřeba **silně férové plánování** procesů
- absence vyhladovění s pasivními semafory: když proces nemůže vstoupit do krit. sekce, je v ní druhý proces, který ale při ukončení krit. sekce první proces odblokuje a ten někdy vstoupí do krit. sekce, protože první je při opětovném pokusu o vstup do krit. sekce blokován

Problém kritické sekce

Pro lib. počet procesů

Kritická sekce
binary semaphore $S \leftarrow 1$
loop forever
1: nekritická sekce
2: wait(S)
3: kritická sekce
4: signal(S)

Obrázek: Řešení problému kritické sekce pro lib. počet procesů

- vzájemné vyloučení je splněno a k uváznutí dojít nemůže
- může dojít k vyhladovění procesu: *scénář?*
- proces(y) nemusí být vybrán(y) pro odblokování → potřeba **deterministické férové plánování** procesů, např. cyklický (round robin), z fronty procesů

Problém kritické sekce

Pro lib. počet procesů

Kritická sekce	
binary semaphore $S \leftarrow 1$	
loop forever	
1:	nekritická sekce
2:	wait(S)
3:	kritická sekce
4:	signal(S)

Obrázek: Řešení problému kritické sekce pro lib. počet procesů

- vzájemné vyloučení je splněno a k uváznutí dojít nemůže
- může dojít k vyhladovění procesu: *scénář?*
- proces(y) nemusí být vybrán(y) pro odblokování $\rightarrow$ potřeba **deterministické férové plánování** procesů, např. cyklický (round robin), z fronty procesů

Problém kritické sekce

- **silný (silně férový) semafor**: L je **fronta procesů** a proces pro odblokování v operaci signal není vybrán libovolně, ale např. v pořadí (!) blokování, jinak **slabý (slabě férový) semafor**

Otázka: *Kolik max. procesů může být v krit. sekci při inicializaci hodnoty semaforu na k?*

Problém producenta a konzumenta

– čekání konzumenta na produkci dat = čekání na semaforu, produkce dat = signalizace semaforu

Producent-konzument

dataType buffer[] ← nil
semaphore notEmpty ← 0

	producent		*konzument*
	dataType item		dataType item
	int i ← 0		int i ← 0
	loop forever		loop forever
1:	item = produce()	1:	wait(notEmpty)
2:	buffer[i] ← item	2:	item = buffer[i]
3:	i ← i + 1	3:	i ← i + 1
4:	signal(notEmpty)	4:	consume(item)

Obrázek: Řešení problému producenta a konzumenta s neomezeným bufferem

Problém producenta a konzumenta

– čekání producenta na konzumaci dat (při omezeném bufferu) = čekání na (jiném) semaforu, konzumace data = signalizace (jiného) semaforu

Producent-konzument

```
dataType buffer[N] ← nil
semaphore notEmpty ← 0
semaphore notFull ← N
```

producent	*konzument*
dataType item	dataType item
int i ← 0	int i ← 0
loop forever	loop forever
1: item = produce()	1: wait(notEmpty)
2: wait(notFull)	2: item = buffer[i]
3: buffer[i] ← item	3: i ← i + 1 mod N
4: i ← i + 1 mod N	4: signal(notFull)
5: signal(notEmpty)	5: consume(item)

Obrázek: Řešení problému producenta a konzumenta s omezeným bufferem

Problém producenta a konzumenta

- **rozdělené semafory (split semaphores)** = synch. mechanizmus, kdy **součet hodnot** V dvou a více semaforů je neustále roven nejvýše **pevné hodnotě** N
- operace wait a signal na semaforu jsou v **různých procesech**
- použití pro řešení problémů **pořadí vykonávání procesů**
- např. semafory notEmpty a notFull a N rovno velikosti bufferu
- pro $N = 1 \ldots$ **rozdělené binární semafory**

Problém bariéry

– čekání na bariéře = čekání na semaforu, signalizace příchodu k bariéře = signalizace semaforu

Pro 2 procesy

Bariéra	
binary semaphore $BA \leftarrow 0$	
binary semaphore $BB \leftarrow 0$	
A	B
loop forever	loop forever
1: akce	1: akce
2: signal(SA)	2: signal(SB)
3: wait(SB)	3: wait(SA)

Obrázek: Řešení problému bariéry pro 2 procesy

- semafory SA, SB . . . rozdělené binární semafory

Semafory

- jednodušší na použití než (atomické) instrukce poskytované strojem
- umožňují **pasivní čekání** (změnou stavu procesu na blokovaný)
- **středněúrovňové** synch. primitivum nejčastěji používané v OS a prog. jazycích pro implementaci **zapouzdřených** synch. primitiv na ještě vyšší úrovni

Další řešení kritické sekce

Pekařský (bakery) algoritmus

- Lamport, 1974

= proces, který chce vstoupit do krit. sekce, si musí vzít **číselný lístek (ticket)**, jehož číslo je větší než čísla lístků ostatních čekající procesů, a čeká, až číslo jeho lístku je nejmenší z čekajících

- analogie s „automatem na lístky“ pro obsluhu na principu „první přijde, první obsloužen“

Další řešení kritické sekce

Pekařský (bakery) algoritmus

int $nA \leftarrow 0$, $nB \leftarrow 0$

A		B	
	loop forever		loop forever
1:	nekritická sekce	1:	nekritická sekce
2:	$nA \leftarrow nB + 1$	2:	$nB \leftarrow nA + 1$
3:	await $nB = 0$ or $nA \leq nB$	3:	await $nA = 0$ or $nB < nA$
4:	kritická sekce	4:	kritická sekce
5:	$nA \leftarrow 0$	5:	$nB \leftarrow 0$

Obrázek: Pekařský (bakery) algoritmus řešení problému krit. sekce pro 2 procesy

- proměnné nA a nB obsahují čísla lístků procesů
- pokud jsou čísla lístků procesů, které chtějí vstoupit do krit. sekce, stejná (*scénář?*), vybere se jeden lib. z nich (A)
- předpokládá atomické akce A_2 a B_2!, jinak neplatí vzájemné vyloučení: *scénář?*, *řešení?*

Další řešení kritické sekce

Pekařský (bakery) algoritmus

int $number[N] \leftarrow [0, \ldots, 0]$

```
      loop forever
1:        nekritická sekce
2:        number[i] ← 1 + max(number)
3:        for all other processes j
4:            await (number[j] = 0 or (number[i] ≪ number[j])
5:        kritická sekce
6:        number[i] ← 0
```

Obrázek: Pekařský (bakery) algoritmus řešení problému kritické sekce pro N procesů

for all other processes j:
for j from 1 to N if $j \neq i$

$number[i] \ll number[j]$:
$(number[i] < number[j])$ or $((number[i] = number[j])$ and $(i < j))$

Další řešení kritické sekce

Pekařský (bakery) algoritmus

- nevýhody: čísla lístků mohou růst donekonečna (*scénář?*), proces musí testovat čísla lístků všech ostatních procesů, i když žádný z nich nechce vstoupit do krit. sekce – **neefektivní**, pokud procesy nesoupeří o vstup do krit. sekce a procesů je více
- existuje (originální) řešení korektní i při neatomickém čtení a zápisu do glob. proměnných

Další řešení kritické sekce

„Rychlé“ algoritmy

= při nesoupeření o vstup do krit. sekce proces vykoná vstupní i výstupní protokol sestávající z pevného (a malého) počtu **ne-await akcí**

– proces testuje stav jiných procesů a příp. čeká jen v případě soupeření o vstup do krit. sekce

– první Lamport

Další řešení kritické sekce

„Rychlý“ algoritmus (návrh)

int $gate1 \leftarrow 0$, $gate2 \leftarrow 0$

A		B	
	loop forever		loop forever
1:	nekritická sekce	1:	nekritická sekce
2:	$gate1 \leftarrow idA$	2:	$gate1 \leftarrow idB$
3:	if $gate2 \neq 0$ goto A_2	3:	if $gate2 \neq 0$ goto B_2
4:	$gate2 \leftarrow idA$	4:	$gate2 \leftarrow idB$
5:	if $gate1 \neq idA$	5:	if $gate1 \neq idB$
6:	if $gate2 \neq idA$ goto A_2	6:	if $gate2 \neq idB$ goto B_2
7:	kritická sekce	7:	kritická sekce
8:	$gate2 \leftarrow 0$	8:	$gate2 \leftarrow 0$

Obrázek: „Rychlý“ algoritmus řešení problému krit. sekce pro 2 procesy (návrh)

- proměnné idA a idB jsou číselné nenulové identifikátory procesů A a B
- při absenci soupeření v vstup do krit. sekce jen 3 přiřazení konstanty do glob. proměnné a 2 srovnání glob. proměnné s konstantou

Další řešení kritické sekce

„Rychlý“ algoritmus

Fig. 5.1 – 5.3

Obrázek: Ilustrace rychlého algoritmu

- nesplňuje vzájemné vyloučení: *scénář?*

Další řešení kritické sekce

„Rychlý“ algoritmus

Fig. 5.1 – 5.3

Obrázek: Ilustrace rychlého algoritmu

- nesplňuje vzájemné vyloučení: *scénář?*

Další řešení kritické sekce

„Rychlý“ algoritmus

int $gate1 \leftarrow 0$, $gate2 \leftarrow 0$
bool $wantA \leftarrow false$, $wantB \leftarrow false$

A		B	
	loop forever		loop forever
1:	nekritická sekce	1:	nekritická sekce
2:	$gate1 \leftarrow idA$	2:	$gate1 \leftarrow idB$
3:	$wantA \leftarrow true$	3:	$wantB \leftarrow true$
4:	if $gate2 \neq 0$	4:	if $gate2 \neq 0$
5:	$wantA \leftarrow false$	5:	$wantB \leftarrow false$
6:	goto A_2	6:	goto B_2
7:	$gate2 \leftarrow idA$	7:	$gate2 \leftarrow idB$
8:	if $gate1 \neq idA$	8:	if $gate1 \neq idB$
9:	$wantA \leftarrow false$	9:	$wantB \leftarrow false$
10:	await $wantB = false$	10:	await $wantA = false$
11:	if $gate2 \neq idA$ goto A_2	11:	if $gate2 \neq idB$ goto B_2
12:	else $wantA \leftarrow true$	12:	else $wantB \leftarrow true$
13:	kritická sekce	13:	kritická sekce
14:	$gate2 \leftarrow 0$	14:	$gate2 \leftarrow 0$
15:	$wantA \leftarrow false$	15:	$wantB \leftarrow false$

Další řešení kritické sekce

„Rychlý“ algoritmus pro lib. počet procesů

- pole proměnných $want[N]$ a místo await:

$$\boxed{\begin{array}{l}\text{for all other processes } j \\ \quad \text{await } want[j] = false\end{array}}$$

- nesplňuje absenci vyhladovění procesu, ale jen při velkém soupeření o vstup do krit. sekce

Simulace obecného semaforu

- Barz
- pomocí **dvou binárních semaforů (mutexů) a jedné celočíselné proměnné**

- proměnná *count* obsahuje hodnotu semaforu, mutex *gate* zajišťuje čekání procesů, mutex *M* zajišťuje vzájemné vyloučení při přístupu k proměnné *count*
- inicializace na hodnotu $k \geq 0$: $M \leftarrow 1, gate \leftarrow \min(1, k), count \leftarrow k$

wait

```
1:  wait(gate)
2:  wait(M)
3:  count ← count − 1
4:  if count > 0
5:      signal(gate)
6:  signal(M)
```

signal

```
1:  wait(M)
2:  count ← count + 1
3:  if count = 1
4:      signal(gate)
5:  signal(M)
```

Obrázek: Simulace operace *wait* a *signal*

Monitor

Semafor

- vedle aktivní (čekací smyčka, busy-wait) i „pasivní“ implementace (změna stavu procesu)
- středněúrovňové synch. primitivum – nestrukturované $\rightarrow$ náročné použití pro rozsáhlejší programy

Monitor

- Hoare, Hansen

$=$ **strukturované** synch. primitivum – synchronizace v **„objektu/modulu“**

- zobecnění jádra/supervisoru v OS (centralizace kritických sekcí)
- pro každý „objekt/modul“ vyžadující synchronizaci

Monitor

- **operace na** (stejném) **monitoru prováděné** více procesy **se vzájemným vyloučením**
  - = pouze jeden proces může provádět operaci na monitoru (v daném čase)
    - ostatní procesy čekají na (vstupu do) monitoru
    - řešení problému **kritické sekce**, **atomické provádění operací**
  - = implicitní synchronizace – „zámek“ na vstupu do monitoru
    - není určeno pořadí uvolňování čekajících procesů $\Rightarrow$ může dojít k vyhladovění procesu
- – zobecnění objektu z objektově orientovaného programování (OOP) pro **zapouzdření synchronizačních dat a operací** pomocí třídy
- data monitoru privátní (přístupná pouze z monitoru) = zapouzdření sdílených proměnných
- různé implementace (dat. typu/třídy) napříč prog. jazyky nebo systémy – pozor na sémantiku!

Monitor

Atomická operace monitoru

```
monitor CS
    int n ← 0

    operation incr()
        int temp
        temp ← n
        n ← temp + 1
```

A	*B*
1: CS.incr()	1: CS.incr()

Obrázek: Atomická inkrementace pomocí monitoru

Podmíněná proměnná (condition variable, event)

- monitor = implicitní vzájemné vyloučení
- mnoho synch. problémů vyžaduje explicitní synchronizaci = čekání na splnění podmínky, např. problém producent-konzument

= proměnná, na které proces „čeká“, dokud není splněna podmínka (**čekání na podmínce**); při naplnění podmínky uvolnění procesu – po explicitní **signalizaci podmínky**

- test podmínky a čekání – atomické, **součást operace na monitoru** $\Rightarrow$ hodnota se mezi testováním a čekáním nemůže změnit
- při čekání procesu „opuštění“ monitoru – **atomicky**, pro umožnění operací na monitoru (zejm. signalizace) jiným procesům
- pasivní implementace: proměnná = množina (fronta) blokovaných procesů
- možné implementace i mimo monitor, např. PThreads – potřeba zajistit vzájemné vyloučení monitoru, např. mutexem
- rozdíly oproti semaforu: *waitC* vždy čeká, *signalC* bez efektu, pokud žádný proces nečeká

Podmíněná proměnná (condition variable, event)

(Atomické) operace čekání na podmínce proměnné *cond*, signalizace podmínky a testování na čekající procesy (prováděné procesem *A*):

waitC(cond)
proces A
monitor mon
$cond \leftarrow cond \cup A$
A.state $\leftarrow$ blocked
unlock(mon.lock)

signalC(cond)
proces B
if $cond \neq \emptyset$
B $\leftarrow$ libovolný prvek cond
$cond \leftarrow cond \setminus \{B\}$
B.state $\leftarrow$ ready

emptyC(cond)
return $cond = \emptyset$

Obrázek: Operace *waitC*(*cond*), *signalC*(*cond*) a *emptyC*(*cond*) na podmíněné proměnné *cond*

Podmíněná proměnná (condition variable, event)

Simulace semaforu pomocí monitoru

```
monitor Sem
    int s ← k
    condition notZero

    operation wait()
        if s = 0
            waitC(notZero)
        s ← s - 1

    operation signal()
        s ← s + 1
        signalC(notZero)
```

A	*B*
loop forever	loop forever
nekritická sekce	nekritická sekce
1: Sem.wait()	1: Sem.wait()
kritická sekce	kritická sekce
2: Sem.signal()	2: Sem.signal()

Obrázek: Simulace semaforu (na řešení problému krit. sekce)

Podmíněná proměnná (condition variable, event)

- stavový diagram: celé operace monitoru $\sim$ jediný krok (vzájemné vyloučení, žádné prolnutí)

Obr. 151

Obrázek: Stavový diagram simulace semaforu (na řešení problému krit. sekce)

  - pozn.: uvolněný proces ihned pokračuje, v rámci kroku operace se signalizací

- **problém**: při uvolnění čekajícího procesu pokračování a „vstup“ do monitoru, signalizující proces ale také pokračuje „v“ monitoru = neplatný stav
- **řešení**: jeden z procesů musí počkat na „opuštění“ monitoru druhým procesem – (uvolněný) čekající W nebo signalizující S nebo libovolný?
- navíc ještě čekají procesy E na „vstupu“ do monitoru

Podmíněná proměnná (condition variable, event)

- klasická priorita: $E < S < W$ (**požadavek okamžitého pokračování, signal and urgent wait**) – při signalizaci podmínka platí a uvolněný proces může pokračovat bez jejího opětovného testování, viz např. simulace semaforu
- při $W < S$ (např. v Javě) může pokračující signalizující proces podmínku před pokračováním uvolněného procesu znovu zneplatnit $\rightarrow$ uvolněný proces musí podmínku **znovu otestovat** a případně čekat, např. pro simulaci semaforu:

```
while s = 0
    waitC(notZero)
s ← s - 1
```

- $S < E$ nebo $W < E$ nevhodné (*proč?*)

Problém producenta a konzumenta

- podmíněné proměnné místo semaforů, operace monitoru

Producent-konzument

```
monitor PC
    dataType buffer[N] ← nil
    int i ← 0, j leftarrow 0
    condition notEmpty, notFull

    operation put(dataType item)
        if i + 1 mod N = j
            waitC(notFull)
        buffer[i] ← item
        i ← i + 1 mod N
        signalC(notEmpty)

    operation get()
        dataType item
        if i = j
            waitC(notEmpty)
        item = buffer[j]
        j ← j + 1 mod N
        signalC(notFull)
        return item
```

producent	konzument
dataType item	dataType item
loop forever	loop forever
1: item ← produce()	1: item ← PC.get()
2: PC.put(item)	2: consume(item)

Obrázek: Řešení problému producenta a konzumenta s omezeným bufferem

Problém čtenářů a písařů

Čtenáři-písaři

```
monitor RW
  int readers ← 0
  bool writing ← false
  condition OKtoRead, OKtoWrite

  operation StartRead()
    if writing or not emptyC(OKtoWrite)
      waitC(OKtoRead)
    readers ← readers + 1
    signalC(OKtoRead)

  operation EndRead()
    readers ← readers - 1
    if readers = 0
      signalC(OKtoWrite)

  operation StartWrite()
    if writing or readers ≠ 0
      waitC(OKtoWrite)
    writing ← true

  operation EndWrite()
    writing ← false
    if emptyC(OKtoRead)
      signalC(OKtoWrite)
    else
      signalC(OKtoRead)
```

čtenář		písař	
	loop forever		loop forever
1:	RW.StartRead()	1:	RW.StartWrite()
2:	čtení	2:	zápis
3:	RW.EndRead()	3:	RW.EndWrite()

Obrázek: Řešení problému čtenářů a písařů (monitor s podmíněnými proměnnými)

Problém čtenářů a písařů

- přednost čekajících písařů před novými čtenáři (*jak?*) a čekajících čtenářů před čekajícími písaři (*jak?*)
- čtenář při vstupu do krit. sekce uvolní (všechny) ostatní čekající čtenáře a umožní jim vstup (*jak?*) = kaskádové uvolnění, ale ne pro nové čtenáře (*proč?*)

$\Rightarrow$ absence vyhladovění čtenářů i písařů (čekajících na vstup do krit. sekce, ne na „vstup“ do monitoru)

Problém večeřících filozofů

- atomické testování a čekání na dostupnost obou hůlek
- podmíněné proměnné pro filozofy místo semaforů pro hůlky, pole počtů volných hůlek pro každého filozofa

Večeřící filozofové

```
monitor Phils
  int forks[N] ← [2,...,2]
  condition OKtoEat[N]

  operation takeForks(int i)
    if forks[i] ≠ 2
      waitC(OKtoEat[i])
    forks[i+1] ← forks[i+1] - 1
    forks[i-1] ← forks[i-1] - 1

  operation releaseForks(int i)
    forks[i+1] ← forks[i+1] + 1
    forks[i-1] ← forks[i-1] + 1
    if forks[i+1] = 2
      signalC(OKtoEat[i+1])
    if forks[i-1] = 2
      signalC(OKtoEat[i-1])
```

filozof *i*

```
    loop forever
1:  filozofování
2:  Phils.takeForks(i)
3:  jezení
4:  Phils.releaseForks(i)
```

Obrázek: Řešení problému večeřících filozofů

- může dojít k vyhladovění procesu: *scénář?*

Chráněný objekt (protected object)

- u (klasického) monitoru s podmíněnými proměnnými synchronizace (operace testování podmínky, waitC, signalC, emptyC) explicitní

= **monitor s implicitní synchronizací** – před a po operacích monitoru = **chráněných operací**

- před operací: **podmínka zahájení operace** (jen nad proměnnými objektu) = „bariéra“, při nesplnění čekání
- po operaci: otestování všech podmínek operací („bariér“) a při naplnění podmínky uvolnění jednoho čekajícího procesu

Chráněný objekt (protected object)

Čtenáři-písaři

```
protected object RW
   int readers ← 0
   bool writing ← false

   operation StartRead() when not writing
      readers ← readers + 1

   operation EndRead()
      readers ← readers - 1

   operation StartWrite()
         when not writing and
         readers = 0
      writing ← true

   operation EndWrite()
      writing ← false
```

čtenář		písař	
	loop forever		loop forever
1:	RW.StartRead()	1:	RW.StartWrite()
2:	čtení	2:	zápis
3:	RW.EndRead()	3:	RW.EndWrite()

Obrázek: Řešení problému čtenářů a písařů (chráněný objekt)

- může dojít k vyhladovění procesu: *scénář?*

Chráněný objekt (protected object)

- **efektivnější implementace** implicitní synchronizace snižující počet přepnutí procesů (ze signalizujícího na čekající a zpět)
  - procesy vykonávají operace monitoru i za jiné procesy (díky zapouzdření proměnných a vzájemnému vyloučení)!
  - podmínky operací („bariéry“) nesmí záviset na (lokálních) parametrech operace, jinak čekání na část podmínky bez parametru a pak v chráněné operaci otestování dat a případné další čekání

Monitory

- monitory, podmíněné proměnné a chráněné objekty dnes klasická **vysokoúrovňová sychronizační primitiva**

= strukturované dat. typy/třídy – vhodná pro použití v OOP

- na nich založené další synch. kontrukty v moderních (OO) prog. jazycích
- **centralizované** – jako všechna synch. primitiva založená na sdílení dat
- pro distribuované architektury vhodnější synchronizace založená na komunikaci

Simulátor konkurence

- abstrakce = libovolné proložení atom. akcí sekvenčních procesů
- konkurentní program **nelze klasicky ladit** – při každém spuštění může být jiné proložení = jiný scénář výpočtu

= **interpret konkurentního programu umožňující** uživateli **kontrolovat proložení atom. akcí** procesů

- po každé atom. akci zobrazí stav programu a volitelně program pozastaví a umožní výběr následující akce a z následujících akcí procesů, tzn. výběr procesu

= paralelní architektura emulující konkurentní prostředí – přepínáním samostatných kontextů procesů

- př. **Spin** – také pro ověření korektnosti programu, tzv. **model checker**
- př. **BACI (Ben-Ari Concurrency Interpreter)** – výukový, překladač a interpret dialektů jazyků C a Pascal, konstrukty **cobegin** (konkurentní vykonávání procedur, implicitní bariéra) a **delay**

Model checker

- dokazování korektnosti je možné pomocí stavového diagramu
- **stavový diagram** bývá **velký** i pro jednoduché konkurentní programy – náročná konstrukce a hledání nežádoucích stavů u vzájemného vyloučení, cyklů vedoucích k uváznutí a vyhladovění atd.

= počítačový program pro **konstrukci stavového diagramu a ověřování korektnosti programu**

- ověřuje, zda stavový diagram je modelem formule v temporální logice specifikující vlastnosti korektnosti
- př. **Spin** – také simulátor konkurence

API pro paralelní programování

- používající **(globální) sdílenou paměť**, ne posílání zpráv (= distribuované programování, např. PVM, MPI)
- realizace abstraktních paralelních procesů: procesy, **vlákna** (častěji) – spuštění nových z hlavního, po skončení „připojení“ (join)
- pro konkrétní prog. jazyk(y), platformu(y) a operační systém(y): C/C++/C#, POSIX, .NET

Nízkoúrovňová (multiprocessing, multithreading)

= **explicitní paralelizace** (správa procesů) **a synchronizace**, typicky **různé procesy**

- funkce/objekty pro **správu procesů** (vytvoření, ukončení), čekání na ukončení procesu
- dat. typy + funkce nebo objekty pro **synchronizační primitiva**: atomické akce (test-and-set, fetch-and-add atd.), semafory, kritické sekce, bariéry, monitory, podmíněné proměnné, ...

– větší kontrola za cenu potřeby znát paralelizační prostředí a „vědět, co dělám“

POSIX Threads (Pthreads)

= POSIX standard pro práci s vlákny

- knihovna kromě unixových OS (standardní součást) i pro MS Windows (v rámci SFU/SUA, Pthreads-win32 nad Win32 API)
- C funkce pthread_*: správa vláken, mutexy, bariéra, podmíněná proměnná, aktivní (spin) a R/W zámek
- plus POSIX API pro semafory: C funkce sem_*

Win32 Threading API (Windows Threads)

= součást Win32 API

- C funkce: správa vláken, kritická sekce, mutex, semafor, událost

Boost C++ Libraries

= knihovny C++ šablon, podobné jako STL (text, kontejnery, iterátory, algoritmy, matematika, správa paměti aj.), knihovna thread (**Boost Threads**)

- třídy a funkce: správa vláken, mutexy, bariéra, podmíněná proměnná

C++0x

= nový standard C++, t.č. (2011) ve fázi draftu, implementace Just Software

- kromě rozšíření prog. jazyka a std. knihovny další knihovny (z C++ TR1), podpora vláken
- třídy (jmenný prostor std): správa vláken, atomické operace, mutexy, podmíněné proměnné

.NET vlákna

= součást .NET 2.0 a 3.5

- středněúrovňové – **fond vláken** (viz dále)
- třídy (jmenný prostor System.Threading): správa vláken Thread, BackgroundWorker a ThreadPool, atomické akce Interlocked.*, zámky, mutex, semafor, událost, monitor (atribut Synchronization), podmíněná proměnná

API pro paralelní programování

Vysokoúrovňová (tzv. „logická paralelizace“, task parallelism)

= **implicitní paralelizace** (správa procesů) **a synchronizace**, typicky **stejné procesy**

– implicitní plánování procesů, tzv. **úloh (tasks)**, na vlákna z **fondu vláken (thread pool)**

- konstrukty prog. jazyka pro **paralelní vykonávání a synchronizaci úloh**: chráněné objekty a dat. struktury, paralelní cykly, algoritmy

– není nutné znát paralelizační prostředí za cenu menší kontroly

Open Multi-Processing (OpenMP)

= součást překladače (GCC, MS Visual C++, Intel Compiler/Parallel Studio aj.)

- vyznačení částí C/C++/Fortran programu, které mají být vykonávány paralelně, pomocí direktiv preprocesoru jazyka (v C/C++) #pragma omp *
  - paralelní blok (i podmíněně), cyklus (nastavení statického/dynamického plánování), sdílené/privátní proměnné (sumarizace/redukce privátních do sdílené), atomická akce, kritická sekce, bariéra (vedle implicitní)
  - při sekvenčním překladu ignorovány
- funkce a konstanty: zjištění a nastavení počtu vláken, test prezence v paralelním kontextu vykonávání, zjištění počtu procesorů, zámky aj.
- i rozšíření na platformy bez sdílené paměti (Cluster OpenMP)

Intel Threading Building Block (TBB)

= knihovna C++ šablon

- třídy a funkce: paralelní cykly, sumarizace/redukce a třídění parallel_*, atomické operace atomic<T>.*, mutex, dat. struktury (fronta, vektor, hashmap) concurrent_*, škálovatelná správa paměti scalable_*
- dynamické plánování vláken na jádra procesoru, umožněna i kontrola správy vláken (Task Scheduler)

Parallel Extensions

= součást .NET 4.0

- dvě části: Parallel LINQ (PLINQ) a **Task Parallel Library (TPL)**, plus datové struktury pro synchronizaci úloh
- ukládání úloh do front pomocí Task Manageru a implicitní plánování jako vláken na jádra procesoru (pomocí fondu vláken)
- úloha = lambda funkce
- třídy (jmenný prostor System.Threading.Tasks): správa úloh Task, paralelní vykonávání úloh a cykly Parallel.* (implicitní vytvoření úloh a bariéra), kolekce Concurrent*

Grand Central Dispatch (GCD)

= rozšíření prog. jazyka a knihovna pro Mac OS X, iOS, FreeBSD

- ukládání úloh do (dispatch) front nebo zdrojů (source) a implicitní plánování jako vláken na jádra procesoru (pomocí fondu vláken), po příp. (pro zdroj) vyvolání systémové události (časovač, síťový socket, souborový deksriptor aj.)
- úloha = funkce nebo tzv. **blok** = rozšíření C/C++/Objective-C zapouzdřující volání funkce a argumenty (podobné uzávěru)
- funkce/třídy: správa front (3 globální, privátní sériové) Dispatch Queues, zdrojů Dispatch Sources, skupin úloh Dispatch Groups (implicitní bariéra) a semaforů Dispatch Semaphores (jen několik úloh paralelně) dispatch_*

Frameworky pro heterogenní platformy

- výpočetní zařízení CPU, GPU (pro obecné negrafické výpočty, **GPGPU**) aj.
- nízkoúrovňové, součástí prog. jazyk (založený na C) + překladač a API – C nebo tzv. language binding z jiného jazyka

= programování tzv. **jader (kernels)** – v jazyce napsané a přeložené funkce **paralelně vykonávány zařízeními** nad daty z tzv. **streamu** (vektory, matice, atd.), řízení (překlad a spuštění funkcí, výměna dat, pomocí API) programem na hostitelském zařízení (CPU)

- synchronizace: bariéra, události (podmíněné proměnné)
- použití na výpočty všeho druhu
- **OpenCL**, ATI Stream, Nvidia CUDA, MS DirectCompute
- implementace v ovladačích grafických karet, matematických SW