Paralelní programování

přednáška 3

Michal Krupka

1. března 2011

Ještě k atomickým proměnným

Další neatomické proměnné

- Mohou to být proměnné, jejichž velikost je *menší* než slovo: Příkaz

  $A[i] = 1;$

  nemusí být atomický, pokud je A bitové pole. (Proč?)

Vždy je potřeba konzultovat dokumentaci

Ještě k atomickým proměnným

Další neatomické proměnné

- Mohou to být proměnné, jejichž velikost je *menší* než slovo: Příkaz

  $A[i] = 1;$

  nemusí být atomický, pokud je A bitové pole. (Proč?)

Vždy je potřeba konzultovat dokumentaci

Sdílené proměnné, aktivní čekání

Synchronizace procesů

- obvykle nutná kvůli dosažení korektnosti programu
- nejjednodušší způsob: pomocí **sdílených proměnných** a **aktivního čekání**

Sdílené proměnné

- přístupné více procesům ke čtení i zápisu
- nejjednodušší příklad: globální proměnné, dále místo v paměti o známé adrese (např. předané procesu parametrem), lexikální uzávěr
- musí být atomické a volatilní

Aktivní čekání

- periodická kontrola hodnoty sdílené proměnné, kterou nastavuje jiný proces
- operace **await**, např:

$$\text{await } n = 0 \quad \Leftrightarrow \quad \text{loop while } n \neq 0$$

- v praxi se nepoužívá (pasivní čekání)

Sdílené proměnné, aktivní čekání

Synchronizace procesů

- obvykle nutná kvůli dosažení korektnosti programu
- nejjednodušší způsob: pomocí **sdílených proměnných** a **aktivního čekání**

Sdílené proměnné

- přístupné více procesům ke čtení i zápisu
- nejjednodušší příklad: globální proměnné, dále místo v paměti o známé adrese (např. předané procesu parametrem), lexikální uzávěr
- musí být atomické a volatilní

Aktivní čekání

- periodická kontrola hodnoty sdílené proměnné, kterou nastavuje jiný proces
- operace **await**, např:

$$\text{await } n = 0 \quad \Leftrightarrow \quad \text{loop while } n \neq 0$$

- v praxi se nepoužívá (pasivní čekání)

Sdílené proměnné, aktivní čekání

Synchronizace procesů

- obvykle nutná kvůli dosažení korektnosti programu
- nejjednodušší způsob: pomocí **sdílených proměnných** a **aktivního čekání**

Sdílené proměnné

- přístupné více procesům ke čtení i zápisu
- nejjednodušší příklad: globální proměnné, dále místo v paměti o známé adrese (např. předané procesu parametrem), lexikální uzávěr
- musí být atomické a volatilní

Aktivní čekání

- periodická kontrola hodnoty sdílené proměnné, kterou nastavuje jiný proces
- operace **await**, např:

$$\text{await } n = 0 \quad \Leftrightarrow \quad \text{loop while } n \neq 0$$

- v praxi se nepoužívá (pasivní čekání)

Dodavatel-odběratel

Dodavatel-odběratel (Producer-Consumer)

- jeden ze základních synchronizačních problémů
- dva procesy: **dodavatel** (**Producer**) periodicky získává data a ukládá je do sdíleného úložiště (**bufferu**), **odběratel** (**Consumer**) vyzvedává data z bufferu a zpracovává je
- dodavatel nesmí zapisovat do plného bufferu (příp. musí počkat)
- odběratel nesmí číst z prázdného bufferu (příp. musí počkat)

Jednoduchý případ: Pouze dva procesy, buffer pojme jednu položku.
Zobecnění: Buffer pojme více položek, více dodavatelů a odběratelů.

Pozn.: větší buffer má smysl kvůli vyrovnávání, ne pokud je rychlost dodavatele větší než odběratele.

Příklady

- dodavatel: komunikační linka, odběratel: webový prohlížeč
- dodavatel: klávesnice, odběratel: operační systém
- dodavatel: výpočetní proces, odběratel: GUI

Dodavatel-odběratel

Dodavatel-odběratel (Producer-Consumer)

- jeden ze základních synchronizačních problémů
- dva procesy: **dodavatel** (**Producer**) periodicky získává data a ukládá je do sdíleného úložiště (**bufferu**), **odběratel** (**Consumer**) vyzvedává data z bufferu a zpracovává je
- dodavatel nesmí zapisovat do plného bufferu (příp. musí počkat)
- odběratel nesmí číst z prázdného bufferu (příp. musí počkat)

Jednoduchý případ: Pouze dva procesy, buffer pojme jednu položku.
Zobecnění: Buffer pojme více položek, více dodavatelů a odběratelů.

Pozn.: větší buffer má smysl kvůli vyrovnávání, ne pokud je rychlost dodavatele větší než odběratele.

Příklady

- dodavatel: komunikační linka, odběratel: webový prohlížeč
- dodavatel: klávesnice, odběratel: operační systém
- dodavatel: výpočetní proces, odběratel: GUI

Dodavatel-odběratel

Dodavatel-odběratel (Producer-Consumer)

- jeden ze základních synchronizačních problémů
- dva procesy: **dodavatel** (**Producer**) periodicky získává data a ukládá je do sdíleného úložiště (**bufferu**), **odběratel** (**Consumer**) vyzvedává data z bufferu a zpracovává je
- dodavatel nesmí zapisovat do plného bufferu (příp. musí počkat)
- odběratel nesmí číst z prázdného bufferu (příp. musí počkat)

Jednoduchý případ: Pouze dva procesy, buffer pojme jednu položku.
Zobecnění: Buffer pojme více položek, více dodavatelů a odběratelů.

Pozn.: větší buffer má smysl kvůli vyrovnávání, ne pokud je rychlost dodavatele větší než odběratele.

Příklady

- dodavatel: komunikační linka, odběratel: webový prohlížeč
- dodavatel: klávesnice, odběratel: operační systém
- dodavatel: výpočetní proces, odběratel: GUI

Dodavatel-odběratel

Dodavatel-odběratel (Producer-Consumer)

- jeden ze základních synchronizačních problémů
- dva procesy: **dodavatel** (**Producer**) periodicky získává data a ukládá je do sdíleného úložiště (**bufferu**), **odběratel** (**Consumer**) vyzvedává data z bufferu a zpracovává je
- dodavatel nesmí zapisovat do plného bufferu (příp. musí počkat)
- odběratel nesmí číst z prázdného bufferu (příp. musí počkat)

Jednoduchý případ: Pouze dva procesy, buffer pojme jednu položku.
Zobecnění: Buffer pojme více položek, více dodavatelů a odběratelů.

Pozn.: větší buffer má smysl kvůli vyrovnávání, ne pokud je rychlost dodavatele větší než odběratele.

Příklady

- dodavatel: komunikační linka, odběratel: webový prohlížeč
- dodavatel: klávesnice, odběratel: operační systém
- dodavatel: výpočetní proces, odběratel: GUI

Dodavatel-odběratel

Zadání jednoduché verze

- Dodavatel v nekonečné smyčce volá funkci produce(), která vrací dodávaná data,
- data ukládá do sdílené proměnné buffer.
- Odběratel data vyzvedává z proměnné buffer,
- periodicky volá funkci consume() s těmito daty.
- Hodnota nil v proměnné buffer znamená žádná data.

Dodavatel-odběratel

Řešení jednoduché verze

Dodavatel-odběratel 1	
dataType buffer ← nil	
Producer	Consumer
dataType item	dataType item
p1: loop forever	c1: loop forever
p2: item ← produce()	c2: await buffer
p3: await not buffer	c3: item ← buffer
p4: buffer ← item	c4: buffer ← nil
	c5: consume(item)

Obrázek: Dodavatel-odběratel pro buffer délky 1

Dodavatel-odběratel

Korektnost

Dodavatel-odběratel 1

dataType buffer ← nil

Producer		Consumer	
	dataType item		dataType item
p1:	loop forever	c1:	loop forever
p2:	item ← produce()	c2:	await buffer
p3:	await not buffer	c3:	item ← buffer
p4:	buffer ← item	c4:	buffer ← nil
		c5:	consume(item)

Dodavatel-odběratel

Korektnost

Dodavatel-odběratel 1

dataType buffer ← nil

Producer		Consumer	
	dataType item		dataType item
p1:	loop forever	c1:	loop forever
p2:	item ← produce()	c2:	await buffer
p3:	await not buffer	c3:	item ← buffer
p4:	buffer ← item	c4:	buffer ← nil
		c5:	consume(item)

Podmínky **bezpečnosti** (neformálně):

Dodavatel-odběratel

Korektnost

Dodavatel-odběratel 1			
dataType buffer ← nil			
Producer		Consumer	
	dataType item		dataType item
p1:	loop forever	c1:	loop forever
p2:	item ← produce()	c2:	await buffer
p3:	await not buffer	c3:	item ← buffer
p4:	buffer ← item	c4:	buffer ← nil
		c5:	consume(item)

Podmínky **bezpečnosti** (neformálně):

- každá hodnota vrácená funkcí produce() se dostane do proměnné buffer,

Dodavatel-odběratel

Korektnost

Dodavatel-odběratel 1

dataType buffer ← nil

Producer		Consumer	
	dataType item		dataType item
p1:	loop forever	c1:	loop forever
p2:	item ← produce()	c2:	await buffer
p3:	await not buffer	c3:	item ← buffer
p4:	buffer ← item	c4:	buffer ← nil
		c5:	consume(item)

Podmínky **bezpečnosti** (neformálně):

- každá hodnota vrácená funkcí produce() se dostane do proměnné buffer,
- funkce consume() bude zavolána s každou hodnotou v proměnné buffer.

Dodavatel-odběratel

Korektnost

Dodavatel-odběratel 1

dataType buffer ← nil

Producer		Consumer	
	dataType item		dataType item
p1:	loop forever	c1:	loop forever
p2:	item ← produce()	c2:	await buffer
p3:	await not buffer	c3:	item ← buffer
p4:	buffer ← item	c4:	buffer ← nil
		c5:	consume(item)

Podmínky **bezpečnosti** (neformálně):

- každá hodnota vrácená funkcí produce() se dostane do buffer:
  - Producer.item je lokální proměnná,
  - podmínka živosti (a férovosti),

Dodavatel-odběratel

Korektnost

Dodavatel-odběratel 1

dataType buffer ← nil

Producer		Consumer	
	dataType item		dataType item
p1:	loop forever	c1:	loop forever
p2:	item ← produce()	c2:	await buffer
p3:	await not buffer	c3:	item ← buffer
p4:	buffer ← item	c4:	buffer ← nil
		c5:	consume(item)

Podmínky **bezpečnosti** (neformálně):

- každá hodnota vrácená funkcí produce() se dostane do buffer:
  - Producer.item je lokální proměnná,
  - podmínka živosti (a férovosti),
- funkce consume() bude zavolána s každou hodnotou v buffer:
  - p4 má splněnou prekondici buffer = nil,
  - je-li buffer ≠ nil, nevykonává se p4 a nutně dojde k c3 a c5 (živost).

Dodavatel-odběratel

Korektnost

Dodavatel-odběratel 1

dataType buffer ← nil

Producer		Consumer	
	dataType item		dataType item
p1:	loop forever	c1:	loop forever
p2:	item ← produce()	c2:	await buffer
p3:	await not buffer	c3:	item ← buffer
p4:	buffer ← item	c4:	buffer ← nil
		c5:	consume(item)

Podmínky **živosti**:

- p3: je-li buffer $\neq$ nil, v c2 se nečeká a dojde k c4 (férovost),
- c2: je-li buffer $=$ nil, v p3 se nečeká a dojde k p2 (rovněž férovost).

Dodavatel-odběratel

Korektnost

Dodavatel-odběratel 1

dataType buffer ← nil

Producer		Consumer	
	dataType item		dataType item
p1:	loop forever	c1:	loop forever
p2:	item ← produce()	c2:	await buffer
p3:	await not buffer	c3:	item ← buffer
p4:	buffer ← item	c4:	buffer ← nil
		c5:	consume(item)

Podmínky **živosti**:

- p3: je-li buffer $\neq$ nil, v c2 se nečeká a dojde k c4 (férovost),
- c2: je-li buffer $=$ nil, v p3 se nečeká a dojde k p2 (rovněž férovost).

Cvičení

1. Pozměňte řešení pro situaci, kdy nelze použít nil pro signalizaci prázdného bufferu (zavést novou proměnnou).
2. Přidejte do řešení možnost ukončení obou cyklů v případě, že funkce produce() vrátí nil.
3. Okomentujte zjednodušenou verzi bez proměnné item:

dataType buffer ← nil			
Producer		Consumer	
p1:	loop forever	c1:	loop forever
p2:	await not buffer	c2:	await buffer
p3:	buffer ← produce()	c3:	consume(buffer)
		c4:	buffer ← nil

Dodavatel-odběratel

Varianta: odběratel v hlavní smyčce GUI aplikace

- dodavatel provádí dlouhý výpočet, který nemá blokovat GUI,
- hlavní proces aplikace je odběratel, testuje data ve smyčce událostí (musí tedy dostávat pravidelné *Idle Events*; v praxi se moc nepoužívá).

Dodavatel-odběratel 2

dataType buffer ← nil

Producer	Consumer
dataType item	dataType item
p1: loop forever	c1: loop forever
p2: item ← produce()	// . . . zpracování události . . .
p3: await not buffer	c2: if buffer then
p4: buffer ← item	c3: item ← buffer
	c4: buffer ← nil
	c5: consume(item)

Obrázek: Dodavatel-odběratel, buffer délky 1, varianta pro smyčku událostí

Dodavatel-odběratel

Varianta: odběratel v hlavní smyčce GUI aplikace

- dodavatel provádí dlouhý výpočet, který nemá blokovat GUI,
- hlavní proces aplikace je odběratel, testuje data ve smyčce událostí (musí tedy dostávat pravidelné *Idle Events*; v praxi se moc nepoužívá).

Dodavatel-odběratel 2

dataType buffer ← nil

Producer		Consumer	
	dataType item		dataType item
p1:	loop forever	c1:	loop forever
p2:	item ← produce()		// . . . zpracování události . . .
p3:	await not buffer	c2:	if buffer then
p4:	buffer ← item	c3:	item ← buffer
		c4:	buffer ← nil
		c5:	consume(item)

Obrázek: Dodavatel-odběratel, buffer délky 1, varianta pro smyčku událostí

Dodavatel-odběratel

Zadání verze s omezeným bufferem

- buffer je jednorozměrné pole dané délky
- nil opět znamená prázdnou hodnotu
- varianta: předchozí bod neplatí (prázdná hodnota neexistuje)

Dodavatel-odběratel

Řešení problému omezeného bufferu s nil

Dodavatel-odběratel 3

dataType buffer[N] ← nil

Producer		Consumer	
	dataType item		dataType item
	integer i ← 0		integer i ← 0
p1:	loop forever	c1:	loop forever
p2:	item ← produce()	c2:	await buffer[i]
p3:	await not buffer[i]	c3:	item ← buffer[i]
p4:	buffer[i] ← item	c4:	buffer[i] ← nil
p5:	i ← i + 1 mod N	c5:	consume(item)
		c6:	i ← i + 1 mod N

Obrázek: Dodavatel-odběratel pro omezený buffer s nil

Dodavatel-odběratel

Řešení problému omezeného bufferu bez nil

- předpokládáme, že množina přirozených čísel je neomezená

Dodavatel-odběratel 4

dataType buffer[N]
integer bottom,top ← 0

Producer		Consumer	
	dataType item		dataType item
p1:	loop forever	c1:	loop forever
p2:	item ← produce()	c2:	await bottom < top
p3:	await top < bottom + N	c3:	item ← buffer[bottom mod N]
p4:	buffer[top mod N] ← item	c4:	bottom ← bottom + 1
p5:	top ← top + 1	c5:	consume(item)

Obrázek: Dodavatel-odběratel pro omezený buffer

Cvičení

4. Ověřte korektnost obou programů.
5. Přidejte do obou programů možnost ukončení (viz Cv. 2).
6. Upravte program Dodavatel-odběratel 4 tak, aby se proměnné bottom a top neinkrementovaly donekonečna.

Ještě ke kritickým sekcím

Použití v praxi

- obvykle pomocí **zámků** (**locks**)
- klasické objekty (někdy libovolné, někdy speciální třídy)
- mají metody na vstupní a výstupní protokol (enter, exit)
- kritická sekce se označí pomocí klíčového slova

Příklad v C#

```
Object thisLock = new Object();
lock (thisLock)
{
  // Kritická sekce
}
```

Cvičení

1. V globálním poli accounts jsou uloženy zůstatky bankovních účtů. Napište funkci transfer, která převede zadanou částku mezi účty: (transfer a1 a2 amount). Můžete použít pomocná globální data.

Ještě ke kritickým sekcím

Cvičení

2. Vylepšete funkci transfer tak, aby šlo současně (bez čekání) převádět částku mezi různými dvojicemi účtů.
3. Vylepšete funkci transfer tak, aby kromě požadavků z předchozího příkladu umožňovala provádět účetní uzávěrku, tj. test, zda součet zůstatků na všech účtech odpovídá očekávané hodnotě (jelikož provádíme pouze přesuny mezi našimi účty, musí být tento součet konstantní). Udělejte to tak, že napíšete funkci balance, která vrátí součet zůstatků na všech účtech. Tato funkce musí za všech okolností vracet stejnou hodnotu. Může dočasně zablokovat převody mezi účty (tj. způsobit čekání ve funkci transfer). Pozor na uváznutí!

Flynnova taxonomie

- klasifikace architektur počítačů podle možnosti
  - paralelního vykonávání instrukcí
  - paralelního zpracování dat
- Michael J. Flynn, 1966

	Single Instruction	**Multiple Instruction**
Single Data	SISD	MISD
Multiple Data	SIMD	MIMD

SISD

Klasická von Neumannova architektura (PC s jedním procesorem)

MISD

Neobvyklé, např. kvůli zabránění chybám (raketoplán)

SIMD

Paralelní zpracování dat stejným algoritmem (některé superpočítače, vektorové procesory, GPU)

MIMD

Více procesů zpracovává různá data (moderní PC, distribuované systémy)

Flynnova taxonomie

SISD

- jeden procesor provádí jeden proud instrukcí, pracuje s daty uloženými v jedné paměti

Příklad: klasická von Neumannova architektura

MISD

- více procesorů provádí současně více programů s týmiž daty

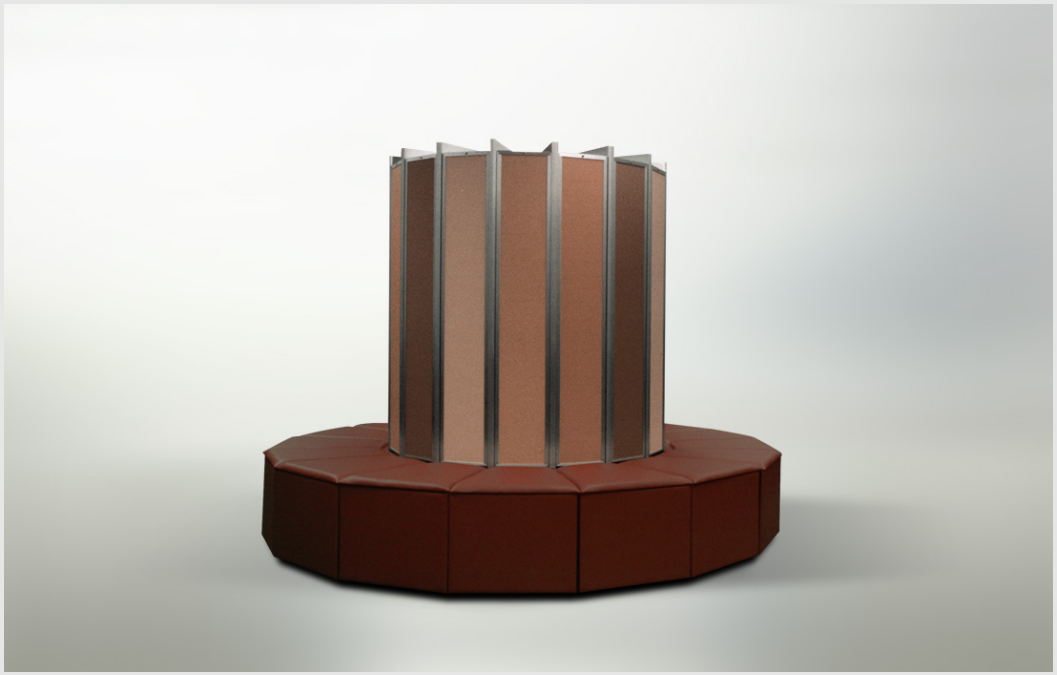
Příklady: nepříliš obvyklé, používá se kvůli předcházení chybám. Např. raketoplán (pět počítačů zpracovává stejná data).

Flynnova taxonomie

SIMD

- více procesorů provádí současně (ve stejném taktu) jeden proud instrukcí na různých datech
- mezi procesy není nutná synchronizace
- masivně paralelní výpočty

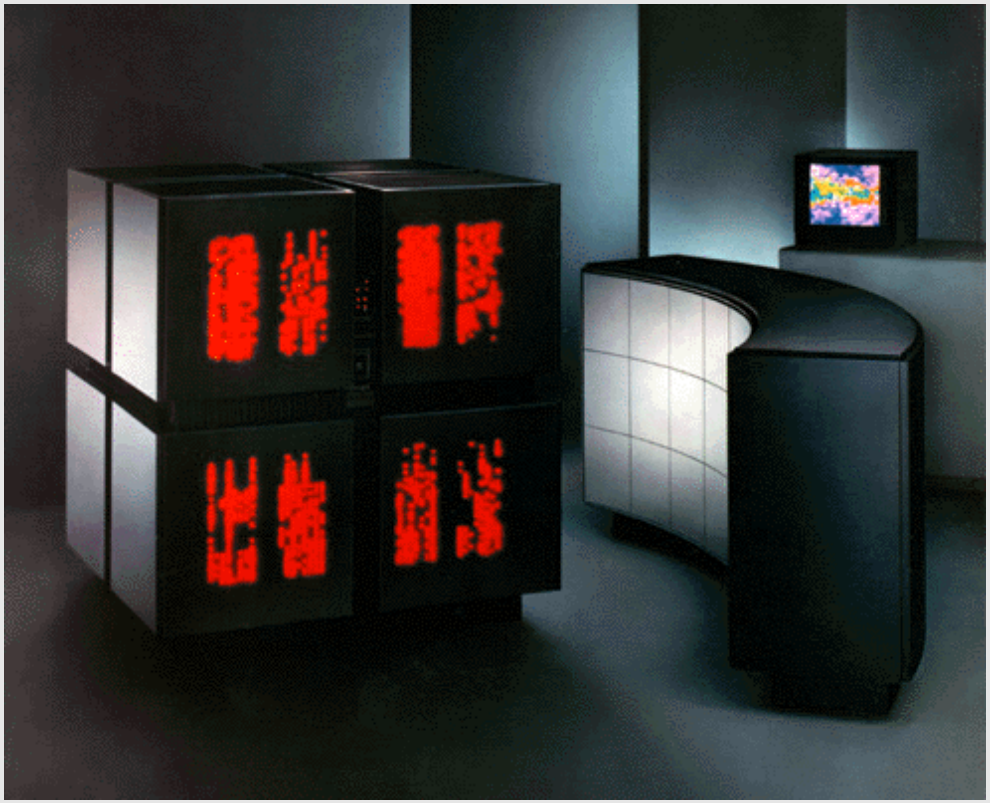
Příklady: Superpočítače v 70. a 80. letech, např. **Cray:**



- Cray-1, 1976
- 200 000 hradel
- 1 662 „procesorů“ 113 variant
- vektorové instrukce
- pohovka zdarma

Flynnova taxonomie

Connection Machine:



- CM-1, 2, konec 80. let
- 65 536 jednobitových procesorů,
- uspořádaných do 12-rozměrné hyperkrychle
- umělá inteligence

Další:

- vektorové instrukce v procesorech (UltraSPARC I: VIS, x86: MMX, POWER: AltiVec)

Flynnova taxonomie

MIMD

- více procesorů provádí současně více úloh na různých datech
- mezi procesy je nutná synchronizace

Příklad: moderní osobní počítače i superpočítače, distribuované systémy

Podskupinou je **SPMD: Single Program/Multiple Data**

- jednotlivé procesory vykonávají tentýž program (s různými daty), ale ne nutně ve stejném kroku,
- je nutná synchronizace pomocí **bariér** (viz dále)

Příklady

- nejběžnější způsob paralelního programování i na MIMD,
- moderní GPU: stovky jader, souběžně **tisíce procesů**
  - na některé úkoly mnohonásobně rychlejší než CPU
  - programové modely: CUDA (NVIDIA), OpenCL (Apple, otevřený standard, široce podporovaný)
  - aplikace: zpracování obrazu, modelování a simulace, výpočetní chemie, biologie, medicína...

Bariéry (Jordan 1978)

Synchronizační nástroj, používaný zejména při SPMD programování.

Bariéra

Místo v programu, na kterém procesy čekají, dokud jej všechny nedosáhnou.

Příklad: výpočet n-tého řádku Pascalova trojúhelníku pomocí n procesů.

0	1	0	0	0	0	0	0	0	0	0
0	1	1	0	0	0	0	0	0	0	0
0	1	2	1	0	0	0	0	0	0	0
0	1	3	3	1	0	0	0	0	0	0
0	1	4	6	4	1	0	0	0	0	0
0	1	5	10	10	5	1	0	0	0	0
0	1	6	15	20	15	6	1	0	0	0
0	1	7	21	35	35	21	7	1	0	0
0	1	8	28	56	70	56	28	8	1	0

Obrázek: Hodnoty pole A v jednotlivých krocích algoritmu

Bariéry

Algoritmus pro výpočet n-tého řádku Pascalova trojúhelníka

N-tý řádek Pascalova trojúhelníka	
integer A[N + 2] $\leftarrow$ 0, A[1] $\leftarrow$ 1	
i-tý proces pro i od 1 do N $-$ 1	
	integer sum
1:	loop N $-$ 1 times
2:	sum $\leftarrow$ A[i $-$ 1] + A[i]
3:	barrier
4:	A[i] $\leftarrow$ sum
5:	barrier

Obrázek: N-tý řádek Pascalova trojúhelníka

Bariéry

Implementace

Ukážeme jednu možnost implementace bariéry. Nejprve bariéra pro dva procesy, implementovaná symetricky:

Bariéra pro dva procesy	
Boolean arrive[poč. procesů] $\leftarrow$ [false, . . . , false]	
Proces i	Proces j
1: await not arrive[i]	1: await not arrive[j]
2: arrive[i] = true	2: arrive[j] = true
3: await arrive[j]	3: await arrive[i]
4: arrive[j] = false	4: arrive[i] = false

Obrázek: Symetrická bariéra pro dva procesy

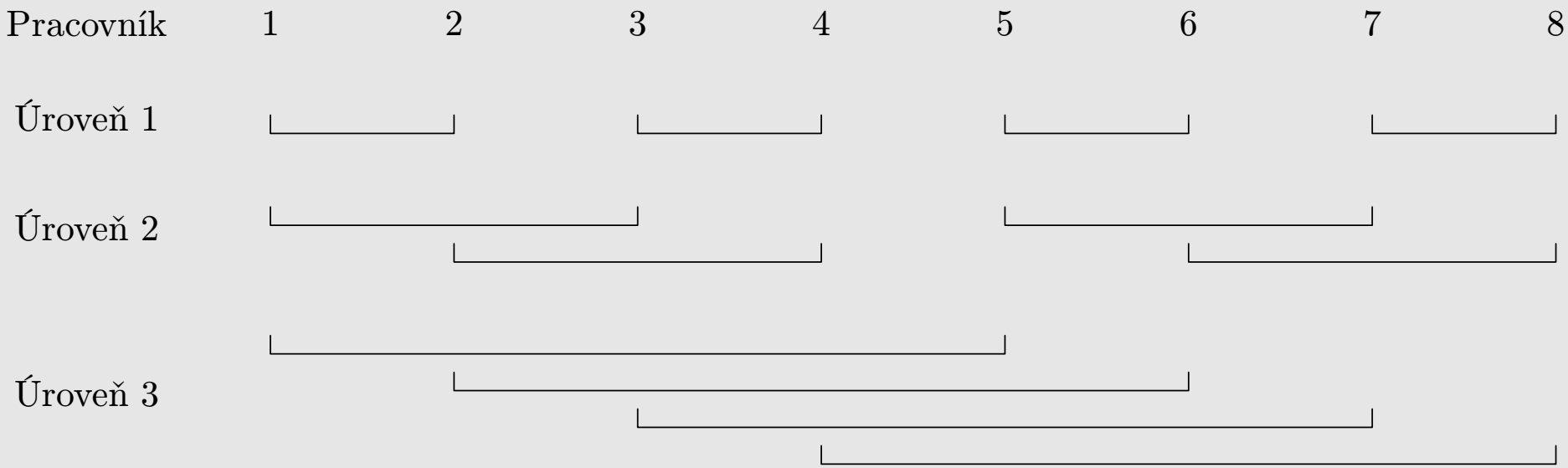
Cvičení

3. Stanovte podmínky bezpečnosti a živosti pro tento algoritmus a zdůvodněte jejich splnění.
4. Implementujte tento algoritmus.

Bariéry

Implementace pro více procesů: Butterfly (motýlkovitá) bariéra

- pro počet procesů $N = 2^X$, logaritmická složitost
- na úrovni s se synchronizují procesy ve vzdálenosti 2^s
- po projití všech úrovní se proces přímo nebo nepřímo synchronizoval s každým jiným procesem



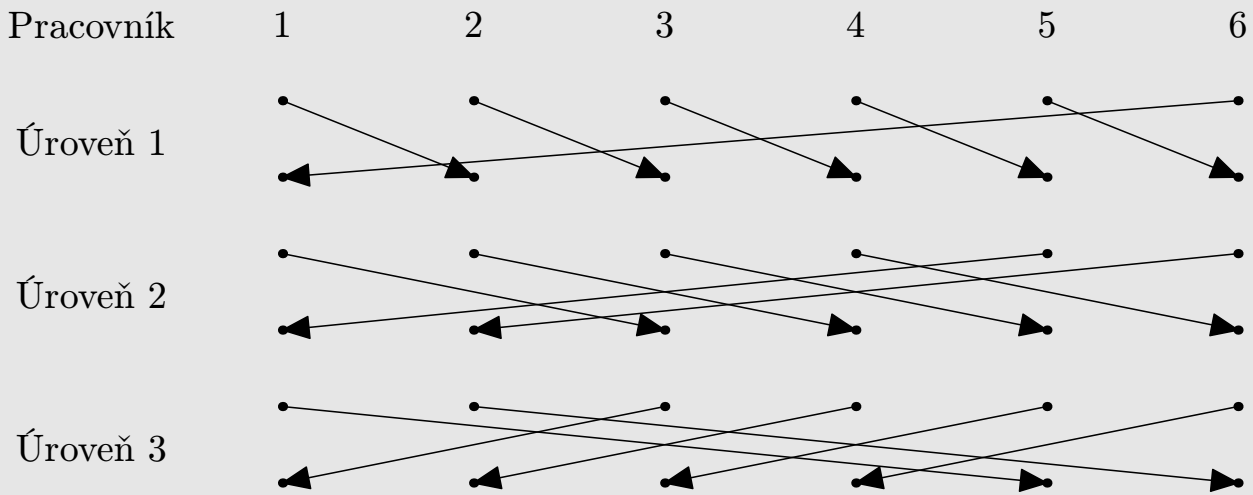
Cvičení

5. Implementujte motýlkovitou bariéru.

Bariéry

Implementace pro více procesů: Diseminační bariéra

- lepší pro počet procesů $N \neq 2^X$, logaritmická složitost
- na úrovni s se proces i synchronizuje s procesem ve vzdálenosti 2^s modulo N:
  - nastaví vlajku procesu $(i + 2^s)\ mod\ N$
  - čeká na a resetuje vlajku procesu $(i - 2^s)\ mod\ N$



Cvičení

6. Implementujte diseminační bariéru.

Čtenáři a písaři (Courtois, Heymans, Parnas 1971)

- klasický a zároveň praktický synchronizační problém

Problém: Dva druhy procesů, čtenáři a písaři, sdílejí data, čtenáři je čtou, písaři čtou i zapisují. Pro zabránění interferencí mezi zápisy a čteními vedoucím k nekonzistentnímu stavu dat musí mít písaři výlučný přístup k datům, pokud žádný písař s daty nepracuje, může číst libovolný počet čtenářů.

- třídy procesů soutěží o přístup k datům: písaři mezi sebou, čtenáři jako třída s písaři
- písaři se vzájemně vylučují
- čtenáři jako třída vylučují písaře

Čtenáři a písaři: první řešení pomocí semaforů

Čtenáři a písaři 1

Semaphore rw ← 1

Writer		Reader?
w1:	loop forever	
w2:	wait(rw)	
w3:	write()	
w4:	signal(rw)	

Čtenáři a písaři: první řešení pomocí semaforů

Čtenáři a písaři 1

Semaphore rw $\leftarrow$ 1

Integer nr $\leftarrow$ 0

	Writer		Reader?
w1:	loop forever	r1:	loop forever
w2:	wait(rw)	r2:	
w3:	write()	r3:	nr $\leftarrow$ nr + 1
w4:	signal(rw)	r4:	if nr = 1 then wait(rw)
		r5:	
		r6:	read()
		r7:	
		r8:	nr $\leftarrow$ nr - 1
		r9:	if nr = 0 then signal(rw)
		r10:	

Čtenáři a písaři: první řešení pomocí semaforů

Čtenáři a písaři 1

Semaphore rw $\leftarrow$ 1 , mutexR $\leftarrow$ 1

Integer nr $\leftarrow$ 0

	Writer		Reader
w1:	loop forever	r1:	loop forever
w2:	wait(rw)	r2:	wait(mutexR)
w3:	write()	r3:	nr $\leftarrow$ nr + 1
w4:	signal(rw)	r4:	if nr = 1 then wait(rw)
		r5:	signal(mutexR)
		r6:	read()
		r7:	wait(mutexR)
		r8:	nr $\leftarrow$ nr - 1
		r9:	if nr = 0 then signal(rw)
		r10:	signal(mutexR)

Čtenáři a písaři

Problém: Preference čtenářů. Jestliže čtenář čte data a jiný čtenář a písař chtějí pracovat s daty, přednost má čtenář. Čtenáři tak mohou neustále blokovat písaře.

Na následujícím slajdu je řešení spravedlivé ke čtenářům i písařům.

Čtenáři a písaři: druhé řešení pomocí semaforů

Čtenáři a písaři 2

Semaphore writers, readers, mutexR $\leftarrow 1$
Integer nr $\leftarrow 0$

Writer	Reader
	Integer prev, current
w1: loop forever	r1: loop forever
w2: wait(writers)	r2: wait(writers)
w3: wait(readers)	r3: wait(mutexR)
w4: signal(readers)	r4: prev $\leftarrow$ nr
w5: write()	r5: nr $\leftarrow$ nr $+ 1$
w6: signal(writers)	r6: signal(mutexR)
	r7: if prev $= 0$ then wait(readers)
	r8: signal(writers)
	r9: read()
	r10: wait(mutexR)
	r11: nr $\leftarrow$ nr $- 1$
	r12: current $\leftarrow$ nr
	r13: signal(mutexR)
	r10: if current $= 0$ then signal(readers)