Paralelní programování

přednášky

Jan Outrata

únor–duben 2011

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?