Programovací jazyky s podporou Real-time

ing. Dalibor Zacios

2003

Obsah

Abstrakt

Ačkoliv dnešní počítače dosahují obrovských výpočetních rychlostí, nejsou schopny díky běžně využívamu software zaručit odezvu v „reálném čase“. Proto vědci stále zkoumají různé přístupy, které by dokázaly zajistit chování programu s garantovanou odezvou. Článek představuje odlišností *real-time* aplikací od běžného přístupu. Také se zabývá vybranými programovacími jazyky či jejich nádstavbami.

1 Úvod

Real-time [1] programování je disciplína, která vznikla již v počátcích softwarového inženýrství. Snahou *real-time* programátorů je dosáhnout, aby se software *embedded* digitálních počítačů choval podobně jako analogová zařízení. Ideální *embedded* systém je skrytě hybridní, pracuje podle pravidel stanovených programátorem, ale chová se podle fyzikálních zákonů. Kritickým faktorem je odezva takového *embedded* systému v reálném čase.

Typickým příkladem může být softwarový proces, který interaguje s fyzikálním procesem. Softwarový proces dostává vstupy od fyzikálního procesu. Poté tyto vstupy zpracuje a vrátí zpět výsledek výpočtu. Od okamžiku převzetí vstupu do okamžiku předání výstupu fyzikálnímu procesu běží softwarový proces v softwarovém čase. Z pohledu fyzikálního procesu jedině *v* okamžiku předání vstupu a výstupu se *soft-time* může stát *real-time*-ovým. Mezi těmito okamžiky může být situace z pohledu *soft-time* velmi složitá a komplikovaná (např. kritická sekce – nedostupný semafor může zapříčinit delší a teoreticky neomezenou prodlevu). Problém je v tom, že programový čas je diskrétní.

Z toho plyne, že softwarové procesy nelze v případě časových kritérií neomezeně *skládat* – důvodem mohou být sdílené prostředky, které mohou vést i k *deadlocku*. Také jiný, nezávislý proces může zabránit dodržení časových požadavků (např. setrváním v kritické sekci).

Přesto *real-time* programování potřebuje kompoziční modely. Reálný svět je *paralelní*, proto i *embedded* software musí být paralelní. *Embedded* hardware je distribuovaný, tedy i *embedded* software musí být distribuovaný. V následujících kapitolách představím tři *real-time* programové modely – *synchronní*, *plánovaný* a *časový*.

[1] Některé anglické výrazy jsem se rozhodl nepřekládat z toho důvodu, že neexistují jejich zažité ekvivalenty v češtině. V textu jsou vysázeny většinou kurzívou.

1.1 Synchronní model

Princip tohoto modelu je založen na myšlence nulového výpočetního času. Programátor předpokládá, že jakákoliv výpočetní akce synchronního programu včetně komunikace trvá nulový čas.

Takový program běží v kontextu fyzikálního nebo výpočetního procesu, který generuje události jako podněty pro program. Synchronní program reaguje na vnější vstupy v nulovém čase okamžitým výpočtem výstupů (reakcí).

Synchronní program se chová *deterministicky*, pokud zpracovává nejvýše jednu reakci pro každou událost a řídící stav. Pokud počítá alespoň jednu reakci pro každou událost a řídící stav, je *reaktivní* (tedy výpočet je konečný). *Reaktivita* synchronního programu, tedy okamžitá a deterministická reakce na vstupy, je to, co programátora takového systému zajímá (a je samozřejmě i věcí kompilátoru zajistit reaktivitu programu).

Synchronní programování se často označuje jako *synchronní reaktivní programování*. Příkladem programovacích jazyků je např. Esterel [7], Signal [8] nebo Lustre [9].

1.2 Plánovaný model

Příkladem mohou být jazyky jako Ada nebo Real-Time Java. Typický plánovaný program se skládá z procesů, vláken nebo úloh. Plánovač rozhoduje, kdy která část programu poběží. *Soft-time* v plánovaném modelu je čas, který program spotřebuje pro své výpočetní aktivity. Příkladem může být doba odezvy – ta může záviset na výkonnosti počítače, využití procesoru nebo plánovacím schématu a není *a priori* určená plánovačem.

Plánování aktivit je podřízeno časovému limitu – *soft-time* musí být menší nebo rovno *real-time*. Kompilátor pro plánovaný model typicky řeší pouze funkčnost, nikoliv plánování, které probíhá za běhu programu.

Limity v plánovaném programu lze dělit na dva druhy – *hard* a *soft*. *Hard* limit musí být dodržen za každou cenu, zatímco *soft* limit může být o nějakou dobu opožděn, díky čemuž získáme snížený, ale stále přijatelný výkon aplikace. Příkladem *soft real-time* aplikací je audio a video zpracování. Naproti tomu aplikace jako řízení výroby či *embedded* řídící systémy vyžadují *hard* limity a bezpečnost zde bývá kritickým faktorem.

1.3 Časový model

Princip časového modelu je založen na myšlence pevně daného nenulového časového kvanta nezávislého na skutečné době výpočtu. *Soft-time* je roven *real-time*. Programátor specifikuje čas, který *timed* program potřebuje pro výpočet výstupu. Předpokládá přitom, že je k dispozici dostatek strojového času procesoru. Proto je program *time-safe*. *Time-safety* závisí na výkonnosti systému, využití procesoru a plánovacím schématu. Kompilátor pro *timed* program zajišťuje *time-safety*, tedy to, je-li k dispozici dostatek strojového času a pokud ne, program odmítne přeložit. Ověření *time-safety* je obtížné – vyžaduje testy plánovatelnosti a analýzu doby provádění programu. Navíc nemusí být tato kontrola dosažitelná v době překladu. Pokud se program „opozdí“, je vyvolána výjimka. Pokud program skončí včas, čeká s předáním výstupů do stanovené doby. *Timed* program může běžet v kontextu fyzikálního nebo jiného výpočetního procesu.

Timed model je vhodný pro *embedded* systémy, které vyžadují časovou předvídatelnost.

Příkladem *timed* programovacího jazyka je Giotto [6]. Klíčovým elementem je časová úloha, která je *periodická* a *deterministická* vůči vstupům a stavům systému.

2 Rozšířený jazyk C

Jedním z nejrozšířenějších přístupů k programování *real-time* aplikací je použití jazyka C. Využijeme tak většinu výhod rozšířeného a známého programovacího jazyka. Nicméně některá omezení vzniknou – například díky odlišnostem *real-time* operačním systémům je znesnadněna přenositelnost na jiné platformy. Další nevýhodou je díky paralelní struktuře a podmínkami pro dodržení *real-time* chování komplikovanější rozšiřování a údržba.

Jako jádro systému je použit HARTIK (The HArd Real TIme Kernel) [4].

2.1 Omezující faktory pro *real-time*

- **Kolize prostředků** – moderní *real-time* jádra často nabízí širokou škálu prostředků. Jejich kolizí vznikají chyby, které není snadné odhalit. Navíc pro některé omezující podmínky programu nelze garantovat v době kompilace jejich úspěšné splnění.

- **Komunikace** – mnoho *real-time* bezpečných protokolů pro komunikaci či paralelismus vyžadují jistý stupeň *a priori* znalosti chování dané úlohy (například množství prostředků využitých úlohou).
- **Sdílená paměť** – mnoho *real-time* jazyků komunikuje pomocí globálních proměnných. Ačkoliv je komunikace přes globální proměnnou pro *real-time* velmi výhodná, brání budoucímu využití modularity a paralelismu.
- **Dynamické vytváření úloh** – v běžném programu výborná vlastnost, v *real-time* možný zdroj mnoha problémů – dodržení časových limitů. Jedním z možných řešení je při vytvoření úlohy test plánovačem (žel praktická využitelnost je nízká). Vytváření úloh za běhu je obecně spojeno s náročnou administrativou a obzvláště v *hard real-time* úlohách je velmi problematické.

Proto autoři vyvinuli několik programových konstrukcí, které umožňují zlepšit statickou analýzu zdrojového kódu při zachování pružnosti jazyka C.

2.2 Čas

Podpora pro čas je první věcí, kterou jazyky pro *real-time* musí mít. Proto existuje datový typ *time*, který nabízí potřebnou míru abstrakce (včetně operací jako sčítání, odčítání, dělení, součin).

2.3 Podpora pro *real-time* paralelismus

Základní myšlenka vychází ze skutečnosti, že každá aplikace může být vyjádřena pomocí tří typů: **task**, ***resource*** a ***channel***. Programátor si může zvolit systém zasílání zpráv nebo využít sdílenou paměť.

Každá entita je deklarována. Díky tomuto přístupu lze snadno zabránit nekontrolovatelné replikaci úloh. Další výhodou je fakt, že všechny proměnné těchto tří typů lze omezit a jejich vytváření a inicializaci lze provést při startu. Protože známe již v době překladu *resources* a *channels* využívané každou úlohou, není problém během kompilace provádět potřebné kontroly.

Úlohy jsou nejdůležitější částí paralelní architektury. Musí splňovat jistá kritéria:

1. funkční a časová sémantika musí být zřetelně odděleny
2. lze definovat vícenásobné instance stejné úlohy (s různými parametry)
3. je třeba specifikovat použité *resources* a *channels* pro umožnění testu plánovače
4. kód typické úlohy má tři části – konstruktor, vlastní kód a destruktor
5. úlohy lze spouštět asynchronně dle potřeby; přesto se kód konstruktoru vykoná již během inicializace systému
6. úlohy lze seskupovat a společně vytvářet či rušit

U úloh lze specifikovat její vlastnosti a chování, jako *criticality* (*hard*, *soft*, *none*), *activation* (*timedriven* = periodická vs. *eventdriven* = aperiodická), *deadline*, *priority* a další.

Resources – příkladem mohou být senzory, které měří a předávají potřebné hodnoty. Také pro ně platí určitá pravidla:

1. možnost použít buď klasické semafory, nebo z pohledu *real-time* bezpečné protokoly (Priority Ceiling [10])
2. jednoduchý mechanismus podmíněné synchronizace k použití mimo kontext komunikace mezi *hard real-time* úlohami
3. možnost vytvoření nového *resource* za běhu s nízkou režií

Z tohoto důvodu existují dva typy konstruktorů – *resource* a *rtresource*. Rozdíl je v přístupu k vzájemně výlučným přístupům. Dále je pro *real-time* konstruktor zákaz použití podmínkových proměnných v synchronizujících úlohách.

***Channels* a porty** – slouží pro snadnou a pružnou komunikaci. Kanály jsou využívány pro výměnu zpráv a jsou implementovány jako jako komunikační struktury. Port je mechanismus, který slouží jako zásuvný modul pro *channel*. Každý port je vázán k *channel* se stejnou sémantikou a kompatibilními typy zpráv. Pro zasílání a příjem zpráv slouží metody *send* a *receive*.

Pro každý port lze nastavit jeho sémantiku pomocí jeho *properties* a tím lze dosáhnout požadovaného chování zasílaných zpráv.

2.4 Zpracování přerušení

Existují tři možnosti obsluhy přerušení:

- vykonání *handleru* s maximální prioritou (*fast handler*)
- naplánování (událostí řízené) úlohy s *deadline* dle potřeby (*safe handler*)
- obě akce zároveň

Pro zamezení provedení těchto akcí lze použít příkaz **ignore** `int_number`.

2.5 Omezení jazyka

Předkladač se snaží asistovat při psaní programu, aby byla co nejméně omezena svoboda programátora. Chybové hlášení se objeví pouze v případě nekorektních operací. Potenciálně nebezpečné situace jsou hlášeny jako varování, konečné rozhodnutí musí udělat programátor.

Nejdůležitější omezení daná jazykem:

- úlohy ani *resources* nelze zanořovat
- kód úlohy může využívat pouze *resources*, které jsou deklarovány v sekci **uses**; přímé použití globálních proměnných není doporučeno, ale není zakázáno
- *hard real-time* úlohy nesmí využívat *resources* ani *synch* portů
- *soft real-time* úlohy by naopak neměly používat *rtresources*
- použití rekurze není v *hard real-time* úlohách doporučeno

2.6 Implementace

Překladač je stále ve vývoji. Je založen na LCC kompilátoru. Lexikální analyzátor a syntaktický analyzátor založený na rekurzivním sestupu jsou psány ručně. Výhodou je efektivita, průzračnost a kompaktnost, nevýhodou je omezení vyjadřovací síly jazyka, složitější hledání chyb a další rozšiřování. Aby nebylo potřeba provádět backtracking, je tento jazyk typu LL(1).

Ačkoliv LCC provádí syntaktickou a sémantickou kontrolu během jednoho průchodu, vzhledem k sémantice použitého jazyka je třeba kvůli dodatečným kontrolám (nepřímé použití *resource*, volání rekurzivních funkcí) provést druhý průchod.

Resource je překladačem přeložen jako *struct*. Datové prvky se tím pádem stanou prvky struktury a na funkce se lze jednoduše odkazovat pomocí ukazatele na strukturu.

Vzájemné vyloučení je zajišťuje HARTIK a semafory. Jednotlivé úlohy se mapují do HARTIK-úloh po zapouzdření jejich parametrů do struktury, která umožní předání parametrů při vytváření úloh. *Channels* a porty jsou mapovány do portů HARTIKu. Každý výstupní soubor je asociován s *module initializer*, což je funkce zajišťující volání inicializačních procedur.

Tento systém je stále vyvíjen a jedním z cílů je vytvoření integrovaného vývojového prostředí pro tvorbu *real-time* programů. Toto prostředí zahrnuje grafický návrhový nástroj, statický nástroj k odhadu nejhorší doby vykonávání, simulátor plánovače, grafický trasovač pro *post-mortem* debug a plánovací analyzér.

3 Jiné programovací jazyky

3.1 Amulet

Amulet je *real-time* programovací prostředí založené na plánování událostí daným jejich *deadlines*. Je postaven na jazyce C a implementován jako množina rutin a maker. Program v Amuletu sestává z kolekce objektů, které spolu komunikují zasíláním zpráv. Je výrazně jednodušší než v minulé kapitole popisované rozšíření jazyka C.

3.2 Timber

Timber je imperativní objektově orientovaný jazyk, nabízející zapouzření stavů, objekty s identitou, rozšiřitelnou hierarchii rozhraní a obvyklé imperativní příkazy jako **loop** a přiřazení. Dědičnost ve stylu např. Smalltalku není v současnosti podporovaná, nicméně je ve fázi vývoje. Chybějící dědičnost je vyvážena bohatými možnostmi pro parametrizaci funkcí, metod a objektů. Dalšími rysy jazyka Timber, které nejsou v objektově orientovaných jazycích běžné, jsou parametrický polymorfismus, typová inference a přímočará paralelní sémantika.

Timber lze charakterizovat jako silně typovaný paralelní jazyk s implicitním vzájemným vyloučením a se systémem zasílání zpráv nabízející synchronní i asynchronní komunikaci. Nicméně na rozdíl od většiny paralelních modelů je proces v Timberu reprezentován jako

objekt se zapouzdřením svých stavů. Spuštění a vykonání procesu není bráno jako souvislé, nýbrž jako posloupnost reakcí na vnější události. Tyto reakce jsou neblokující a jsou vzájemně výlučné. Pořadí vykonávání je určeno jak jejich pořadím vzniku, tak i jejich deadline.

Timber je čistě funkcionální jazyk s rekurzivními definicemi, funkcemi vyšších řádů, algebraickými datovými typy, pattern-matching a polymorfismem dle Hindley/Milnera. Timber také podporuje typové konstruktory tříd a přetěžování stejně jako Haskell, na základě kterého vznikl.

4 Některé *real-time* aplikace

Pro lepší představu zde uvádím dva příklady *real-time* aplikací, jedna je „mission-ctitical“, druhá je z oblasti zpracování zvuku.

4.1 Výzkumní inteligentní roboti

Jednou z oblastí, kde jsou *real-time embedded* systémy nasazovány, je výzkum vesmíru a cizích planet. Díky „křemíkovým“ výzkumníkům je výzkum mnohem snažší, levnější a v neposlední řadě i bezpečnější. Jakákoliv chyba, která vyústí v havárii, nemá za následek ztrátu lidských životů.

Současní výzkumní roboti dosahují na jedné straně obrovských úspěchů (např. Mars Sojourner), na druhou stranu známe mnoho případů selhání (např. Mars Polar Lander).

Příčiny havárií mohou být různé, většinou jsou způsobeny softwarovou chybou. Například u Mars Polar Landeru se díky chybě v řídícím programu předčasně vypnul motor a následoval pád z 50-metrové výšky.

Řešením je vyvinout algoritmy, které jsou schopny dělat rozsáhlé rozhodnutí v reálném čase. Čas a nasazení těchto robotů prověří jejich skutečné kvality.

4.2 *Real-time* software pro zpracování hudby

Jinou oblastí, kde se *real-time* systémy masově používají, je oblast zpracování obrazu, zvuku, apod.

Open Sound World (OSW) je *data-flow* jazyk (objektově orientovaný) s možností dalšího rozšíření, který umožňuje zvukařům a muzikantům zpracovávat hudbu a zvuk v reálném čase. OSW umožňuje práci na různých úrovních zpracování zvuku, včetně vizuální kontroly a úpravy, XML nádstavbu a skriptování a v neposlední řadě nabízí možnost práce v jazyce C++ na vysoké úrovni abstrakce. Lze vytvářet komponenty, které lze kombinovat a modulárně skládat v programy.

OSW *real-time* plánovač podporuje jednotný časový model pro všechny komponenty a symetrický multiprocesing. Zajímavostí je nástroj zvaný *Externalizer*, který umožňuje i běžnému uživateli bez hlubších znalostí jazyka C++ a implementace komponent do jisté míry ovlivnit „vnitřnosti“ a funkčnost jednotlivých komponent.

Tanto jazyk je vyvíjen na platformě PC pro systémy Windows 98/2000 a Linux a pro stanice SGI se systémem Irix.

Aktuální verze je úspěšně využívána během živých vystoupení. Více informací lze nalézt v [13] a na webových stránkách programu `http://cnmat.cnmat.berkeley.edu/OSW/`

5 Závěr

Problematika *real-time* systémů je obecně velmi široká a existuje mnoho diametrálně odlišných aplikací a tím pádem i přístupů. Protože článek vznikl jako kompilát z několika odlišných zdrojů, neklade si za cíl hluboké proniknutí do podstaty *real-time* programování, spíše se snaží ukázat různé způsoby řešení takových problémů včetně možných aplikací.

Reference

[1] *Christoph M. Kirsch*: Principles of Real-Time Programming. Department of Electrical Engineering and Computer Sciences – University of California, Berkeley.

[2] *Ichiro Satoh, and Mario Tokoro*: Semantics for a Real-Time Object Oriented Programming Language. Department of Computer Science, Keio University, Yokohama.

[3] *Luigi Palopoli, Giorgio Buttazzo, and Paolo Ancilotti*: A C Language Extenson for Programming Real-Time Applications. RETIS Lab – Scuola Sup. S. Anna, Pisa.

[4] *G. Lamastra, G. Lipari, G. C. Butazzo, and A. Casile*: Hartik 3.0: A Portable Kernel for Real-Time Control Applications. In Proc. of 4th IEEE International Workshop on Real-Time Computing Systems and Applications (RTCSA), Taipei, Taiwan, October 1997.

[5] *Thomas A. Henzinger, and Christoph M. Kirsch*: The Embedded Machine: Predictable, Portable Real-Time Code. EECS, University of California, Berkeley.

[6] *T. A. Henzinger, B. Horowitz, and C. M. Kirsch*: Giotto: A Time-Triggered Language for Embedded Programming. In Proc. First International Workshop on Embedded Software (EMSOFT), LNCS 2211, pages 166-184. Springer Verlag, 2001.

[7] *G. Berry and L. Cosserat*: The Esterel Synchronous Programming Language and its Mathematical Semantics. Lecture Notes in Computer Science. Springer Verlag, 1985

[8] *P. le Guernic, T. Gautire, M. L. Borgne, and C. LeMaire*: Programming Real-Time Applications with Signal. In Proc. of the IEEE, 1991

[9] *N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud*: The Synchronous Data-Flow Programming Language Lustre. In Proc. of the IEEE, pages 1305-1320, 1991

[10] *G. Buttazzo*: Hard Real-Time Computing Systems. Predictable Scheduling Algorithms and Applications. Kluwer Academic Publishers, Boston, 1997

[11] *Shawn Laird and Douglas W. Jones*: Real Time Programming in Amulet. University of Iowa Department of Computer Science.

[12] *Brian C. Williams*: Model-Based Programming of Robotic Explorers and Intelligent Embedded Systems. Artificial Intelligence Laboratory, Massachusetts Institute of Technology, Cambrige.

[13] *Amar Chaudhary, Adrian Freed, and Matthew Wright*: Center for New Music and Audio Technologies, University of California, Berkeley.