

PŘÍRODOVĚDECKÁ FAKULTA UNIVERZITY PALACKÉHO
KATEDRA INFORMATIKY

BAKALÁŘSKÁ PRÁCE

Automatické dokazování správnosti paralelních programů:

Implementace systému zdrojů



2010

Vojtěch Píkal

Anotace

Práce popisuje systém zdrojů – nástroj pro paralelní programování, který pomáhá zajistit správnost vytvářeného programu. Systém zdrojů je programátorský nástroj, který slouží ke správě sdílených proměnných a zaručuje správný přístup k nim. Systém zajišťuje, že při nesprávném použití je při kompilaci kódu ohlášena chyba. V práci je proveden důkaz správnosti systému pomocí programové logiky a jsou diskutovány další možnosti rozšíření funkcionalit systému.

Tímto bych chtěl poděkovat svému vedoucímu práce RNDr. Michalu Krupkovi, Ph.D. za zajímavé téma a příkladné a hlavně trpělivé vedení a své přítelkyni Hance Čagánkové za psychickou podporu a jazykovou korekturu českých i anglických textů.

Obsah

1. Úvod	6
2. Teoretická část	7
2.1. Pojmenování problému	7
2.2. Teorie řešení	7
2.3. Jazyk implementace	9
2.3.1. Přenesení systému	9
3. Uživatelská část	10
3.1. LispWorks	10
3.1.1. Instalace	10
3.1.2. Prostředí LispWorks	10
3.2. Package MP-Secured	11
3.2.1. Zavedení do LispWorks	11
3.3. Systém zdrojů	12
3.3.1. Zdroje	12
3.3.2. Chráněné proměnné	12
3.3.3. Kritická sekce	12
3.4. Použití systému	13
3.4.1. Vytvoření zdroje	13
3.4.2. Vyžádání zdroje	14
3.4.3. Další funkce systému	14
3.5. Příklady použití	14
3.5.1. Jednoduchý praktický příklad	14
3.5.2. Teoretický příklad	15
4. Programátorská dokumentace	17
4.1. Popis kódu	17
4.1.1. Definice balíčku	17
4.1.2. Konstanty systému	18
4.1.3. Třída resource a její metody	18
4.1.4. Uživatelské informativní funkce	23
4.1.5. Kód pro vytvoření a odstranění zdroje	23
4.1.6. Aparát pro vyhodnocení kódu	27
4.1.7. Spojení předchozího	31
4.2. Důkazy správnosti	33
4.2.1. Jazyk a předpoklady	33
4.2.2. Vnitřní metody	33
4.2.3. Uzamykací metody	36

5. Další vývoj	40
5.1. Robustnost	40
5.2. Další kontrola programátora	40
5.2.1. Kontrola typu	40
5.2.2. Kontrola nevhodného přiřazení	41
5.2.3. Odstranění zbytečného zamykání zdrojů	41
5.3. Rozšíření ochrany na složená data	41
Závěr	43
Conclusions	44
Reference	45
F. Kód programu	46
F.1. Loader	46
F.2. Definice balíčku	46
F.3. Hlavní kód	46
G. Příklady	55
G.1. Paralelní format	55
G.2. Čtenáři a písář	55
H. Reference Manual	57
H.1. defres	57
H.2. delres	57
H.3. resourcep	58
H.4. with-resources	58
I. Obsah příloženého CD	59
I.1. Povinný obsah	59
I.2. Přidaný obsah	59

1. Úvod

V práci popisuji nástroj pro bezpečné paralelní programování – systém zdrojů. V první části se věnuji jeho teoretickým základům a popisuji smysl a účel takového nástroje. V druhé části předkládám uživatelskou příručku – návod k použití a pár praktických příkladů. Následující – třetí – část se věnuje vnitřní realizaci programu, popisuje kód a jednotlivé funkce, vysvětluje jejich fungování a použité výrazy jazyka. Zde je také proveden důkaz správnosti programu pomocí programové logiky. V průběhu celého textu se občas objevují nápady a postřehy jak systém vylepšit a jaké funkcionality by se daly ještě přidat. Tato možná vylepšení jsou shrnuta a diskutována v poslední části.

V dodatcích je kromě tištěné verze kódu programu a použitých příkladů také referenční příručka (reference manual) uživateli přístupných funkcí v angličtině. Smyslem tohoto dodatku je popsat funkce jasně a přesně ve výrazech a pojmech použitého jazyka.

2. Teoretická část

V této části je diskutován účel a smysl práce a její teoretické základy.

2.1. Pojmenování problému

Cílem této práce je vytvoření nástrojů pro programátora – uživatele, které pomohou kontrolovat správnost jeho programu, konkrétně paralelního programu.

Paralelní programování sebou totiž přináší spoustu nových možností udělat v programu chybu. A běžné způsoby hledání těchto chyb obecně selhávají. Nové chyby vznikají společnou prací různých procesů nad stejnými daty. Tyto problémy jsou řešeny různými nástroji synchronizace, pro jejichž použití existuje mnoho dobře zpracovaných teoretických modelů.

Při řešení problémů z praxe je však třeba většinou teoretická modelová řešení značně upravit, či vytvořit řešení přímo na míru danému problému. Zjistit správnost takového nového řešení je pak obecně obtížné.

Přitom mnohé chyby, kterých se lze při psaní paralelního programu dopustit jsou hloupé. Jedná se o chyby nevznikající špatnou analýzou problému či špatnou implementací, ale kvůli zbytečnému opomenutí. Příkladem takové chyby může být například uzamčení jiného zámku, než by příslušelo k dané sdílené proměnné, zápis či čtení ze sdílené proměnné mimo kritickou sekci, či třeba zápis do sdílené proměnné v kritické sekci určené pouze pro čtení. Jsou to chyby, které je možno vytvořit omylem a které se z důvodu autorské slepoty poté špatně odhalují. Program totiž dělá něco jiného, než si programátor myslí, že dělá. Chyba se také nemusí vždy projevit.

Existuje obdoba takovýchto hloupých chyb i v sekvenčních programech. Tam je to například použití neinicializované proměnné. Hloupé chyby v sekvenčních programech jsou dnes kompilátory schopny odhalit a upozornit na ně. V případě paralelních programů však nic takového neexistuje.

Cílem práce je tedy implementovat systém, který odhalí (například již při kompilaci) některé z takovýchto chyb a upozorní na ně programátora, či znemožní jejich vznik.

2.2. Teorie řešení

Teoretická část řešení je inspirována [2]. V tomto článku je teoreticky definován pojem zdroje, jako souboru logicky souvisejících proměnných a kritická sekce ve tvaru

with r when B do S ,

kde r je zdroj, B je podmínka a S libovolný podprogram pracující s proměnnými ze zdroje r . Pro kritickou sekci pak platí, že pokud se ji proces pokusí vykonat, je

pozastaven, dokud současně není splněno B a proces nemá výhradní kontrolu nad r . Poté je přikročeno k vykonání S . Po ukončení sekce je zdroj r opět uvolněn.

Článek dále definuje dvě omezení, která zaručují, že veškeré sdílené proměnné (tj. proměnné používané více procesy) jsou chráněné kritickou sekci.

1. Pokud proměnná x náleží zdroji r , nemůže se objevit v paralelním procesu mimo kritickou sekci pro r .
2. Pokud proměnná x je změněna v procesu S_i , nemůže se objevit v procesu S_j ($i \neq j$), pokud není součástí zdroje.

Článek pak dále dokazuje, že paralelní program používající takto definovanou kritickou sekci při dodržení daných omezení má jisté dobré vlastnosti (vzájemné vyloučení kódu pracujícího se stejnými proměnnými) a dobře se na takovémto programu dokazují vlastnosti další (například absence deadlocku či jistota ukončení).

Pro naše účely je důležité, že daná omezení jsou v přímém rozporu s výše uvedenými hloupými chybami, tedy je vylučují. Pokud tedy implementujeme systém, který dodržuje daná omezení, či je schopen jejich dodržování kontrolovat (vynucovat), zamezíme existenci takovýchto chyb.

Systém je plně implementován pro vlastní, nově zavedené, chráněné proměnné. V používání původních globálních proměnných je programátorovi ponechána plná volnost.

Nepoužívání (resp. střídme a opatrné používání) globálních proměnných patří k obecné programátorské kultuře. Zodpovědnost za následky použití globálních proměnných je ponechána na programátorovi.

V našem systému je poněkud pozměněna definice kritické sekce, konkrétně na

with r_1 to r_n do S .

Proces tedy může uzamknout více zdrojů najednou. Při původní definici toho lze dosáhnout vložením více kritických sekcí do sebe. Dále definujeme obdobnou kritickou sekci

with-read r_1 to r_n do S ,

která v S umožňuje proměnné z r_1 až r_n pouze číst, nikoliv zapisovat. Z kritické sekce byla také odstraněna podmínka B . Její absence na síle kritické sekce nic nemění, protože:

3. pokud se podmínka týkala proměnných, které nejsou součástí žádného zdroje, objevují se tyto pouze v rámci daného procesu a jejich stav je čistě v moci tohoto procesu.
4. pokud proměnné jsou součástí zdroje, aby k nim proces mohl přistoupit (a testovat je nějakou podmínkou), musel nejdříve tento zdroj získat, tedy opět má nad stavem podmínky výhradní moc.

Systém se podařilo v plném rozsahu implementovat pro jednoduchá data. O použití systému více v 3. kapitole o implementaci v 4. kapitole.

2.3. Jazyk implementace

Systém je implementován v jazyce Common Lisp. Konkrétně v jeho implementaci z vývojového prostředí LispWorks. Ten je jedním z mála programovacích jazyků, které implementaci systému umožňují.

Předně je Common Lisp jazyk robustní a přebírá za programátora spoustu úkonů, kterými se nebude muset implementace zabývat, jako například správa paměti.

Lisp je jednoduše rozšiřitelný, to znamená, že mnohé jeho části lze upravit či přepsat, včetně základních funkcí. Lze také jednoduše doplnit nové funkce, makra či převzít částečně či plně kontrolu nad jednotlivými fázemi kompilace a vykonání kódu.

Dále protože kód v Common Lispu je složen ze seznamů, které jsou zároveň výrazem jazyka, je možno s kódem jednoduše pracovat. Analyzovat jej, přetvářet, doplňovat, testovat. A to samotným programem. K takovéto práci s kódem slouží makra, která jsou takto daleko silnější, než v jiných jazycích.

Dále Common Lisp poskytuje systém balíčků (package) jednoduše umožňujícím ukrýt jeden kód před jiným.

A v neposlední řadě LispWorks poskytuje jednoduché rozhraní pro paralelní programování v podobě balíčku MP-package.

2.3.1. Přenesení systému

Přenesení systému do prostředí jiných programovacích jazyků je tak v principu možné, ale nepoměrně komplikovanější. Ve většině v současné době používaných jazyků by vyžadovalo doplnění specifikace jazyka.

3. Uživatelská část

I když je systém určen programátorům v jazyce Common Lisp, pro pochopení tohoto textu a vyzkoušení příkladů není znalost tohoto jazyka potřebná.

V této sekci bude popsáno vývojové prostředí použití implementovaného systému.

3.1. LispWorks

Systém je implementován a určen pro práci ve vývojovém prostředí LispWorks. Pro jeho vyzkoušení či případné použití je třeba si LispWorks nainstalovat.

3.1.1. Instalace

LispWorks lze zdarma stáhnout z adresy:

<http://www.lispworks.com/downloads/index.html>. Instalace samotná je standartní a není potřeba nic speciálně nastavovat a upravovat.

Tato verze prostředí nenabízí veškeré možnosti placené, jako je například sestavení plnohodnotné aplikace běžící bez prostředí, či některé knihovny. Doba sezení je také omezena na 5 hodin a program upozorňuje při spuštění na možnost pořízení plnohodnotné placené verze.

Pro účely vyzkoušení systému je však vše dostačující.







3.1.2. Prostor LispWorks

Vývojové prostředí LispWorks nabízí spoustu nástrojů, které nejsou pro prezentaci systému potřeba. Více o prostředí lze nalézt v [4], nebo v manuálu, který je součástí instalace.


Manuály a nápovědu lze nalézt buď v nabídce *Start* → *programy* → *LispWorks Personal* → *PDF Documentation* nebo *HTML Documentation*.


Další možností je položka *Help* přímo v menu aplikace LispWorks. Zde buď pro jednotlivé manuály položka *Manuals...*, nebo užitečněji pro přímé vyhledávání v manuálech položky *On Symbol...* a popřípadě *Search...* Položka *On symbol...* je obzvláště vhodná pro vyhledávání informací o libovolném prvku jazyka, ať už se jedná o funkci, symbol, speciální operátor či cokoliv jiného.


Nástroje prostředí se nacházejí pod menu v panelu nástrojů v podobě ikon. Všechny tyto nástroje lze použít i přes různé položky v různých podmenu, používání ikon v nástrojovém panelu je však rychlejší.


První sada (  ) ikon zleva zastupuje *Vytvoření*, *Otevření* a *Uložení souboru*. Druhá (  ) *Výjmutí*, *Kopírování* a *Vložení textu* (kódu).

Třetí sada je pro naše účely nezajímavá. Ze čtvrté sady nás zajímají pouze první tři nástroje a předposlední.

První nástroj s ikonkou vysílající červené a modré tečky ( *Listener*) je obdobou konzole. Lze zde běžným způsobem zadávat libovolný lispový výraz k vyhodnocení. Výsledky takového vyhodnocení jsou zobrazeny sem.


Druhý nástroj pod ikonkou popsané stránky () je *Editor*, a bude podrobněji popsán níže.



Třetí nástroj v řadě s ikonou modré pulzující tečky () je tzv. *Output Browser* a slouží ke zobrazení výstupu. Sem jste přepnuti pro výsledky kompilace či jiných vyhodnocení v editoru. Také výstupy (tisk) funkcí zkompileovaných v editoru je nasměrován sem.

Posledním nástrojem pro odzkoušení systému důležitým je *Process Browser* skrývajícím se pod předposlední ikonkou běžícího panáčka () . *Process Browser* zobrazuje v prostředí běžící procesy a informace o nich.

Vyhodnocování kódu je možno provádět buď přímo jeho zadáním do *Listeneru*, nebo pohodlněji pomocí ikon v *Editoru*. Tyto nástroje se nacházejí na druhé nástrojové liště, která se objeví po přepnutí do prostředí editoru. Šipky, které jsou na panelu nástrojů zleva první, slouží k přepínání mezi jednotlivými otevřenými soubory.

První ikona v další sadě - červené kolečko - slouží k umístění brakepointu.

Další ikona () (dvoje závorky s šipkou), slouží k vyvolání tzv. makroexpanze. O makrech a makroexpanzi lze více nalézt v [3], nebo v části 4.1. u [prvních maker](#).

Ikonka stránky s bleskem () (Compile Buffer) slouží ke zkompileování právě otevřené stránky, závorka s bleskem () (Compile Definition) slouží ke zkompileování označeného výrazu - např. jedné funkce.

Poslední ikona závorky se šnečkem () (Evaluate Definition) vyhodnotí vybraný výraz. To je obdobné, jako zadání výrazu do *Listeneru*.

3.2. Package MP-Secured

Systém je realizován pomocí balíčku funkcionalit MP-Secured, jehož zdrojové kódy jsou v textové podobě v přílohách a na přiloženém mediu. Více o implementaci, balíčcích a kódu v kapitole 4.

3.2.1. Zavedení do LispWorks

Pokud chceme používat systém zdrojů, tedy MPS package v našem projektu, nejjednodušší cestou je zkopírovat soubory MPS-package.lisp, MPS.lisp a load.lisp do adresáře, v němž je uložen soubor s naším s kódem a poté na začátek našeho kódu připojit výraz, který je umístěn v komentáři v souboru load.lisp.

3.3. Systém zdrojů

Systém zdrojů je rozšířením principu kritické sekce v paralelních programech. Slouží k jednoduché správě sdílených dat, jejich uspořádání do logických celků a hlavně ochraně před čtením či zápisem mimo kritickou sekci. To vše bez výrazného zvýšení nároků na programátora. Systém přináší pouze několik jednoduše pochopitelných pojmů a dvě jednoduchá makra k jejich používání.

3.3.1. Zdroje

Zdrojem (*resource*) rozumíme logicky ucelený soubor sdílených proměnných (*shared variables*). Zdroj je jednoznačně identifikován svým jménem a tyto pojmy lze zaměnit.

Zdroje se v mnohém podobají zámkům a terminologie používaná v souvislosti se zdroji je obdobná k zámkům. Zdroj lze žádat (*claim*), zamknout (*lock*) a odemknout (*unlock*). Žádáním rozumíme snahu procesu získat zdroj pro svou práci, po úspěšném vyžádání je zdroj procesem zamčen, tj. nemohou ho zamknout ani s ním pracovat ostatní zdroje. Odemčením zdroje umožníme ostatním žádajícím procesům zdroj zamknout.

Zdroj může být v jednu chvíli uzamčen a proměnné, které jsou jeho součástí, mohou být zapisovány a čteny pouze jedním procesem. Zdroje lze také vyžádat a uzamknout pouze pro čtení. V takovém případě mohou být jejich proměnné současně čteny více procesy.

3.3.2. Chráněné proměnné

Sdílenou proměnnou ve zdroji nazýváme chráněná proměnná (*protected variable*). Chráněná proměnná je součástí právě jednoho zdroje.

Chráněné proměnné tvoří vlastní jmenný prostor, je tedy možno mít chráněnou a běžnou proměnnou stejného názvu. To je však silně nedoporučeno kvůli přehlednosti kódu.

Chráněné proměnné jsou určeny pro ukládání jednoduchých dat (logických hodnot, čísel).

K chráněné proměnné není možno přistupovat mimo kritickou sekci.

Je vhodné pojmenovávat chráněné proměnné s tečkou na začátku, například `.foo` je vhodné jméno pro chráněnou proměnnou.

3.3.3. Kritická sekce

Systém dále implementuje vlastní kritickou sekci. Kritická sekce začíná vyžádáním zdroje či více zdrojů. Po úspěšném vyžádání a uzamčení zdrojů následuje provedení kódu uvnitř sekce. Uvnitř kódu sekce je možno přistupovat pouze k těm chráněným proměnným, které jsou součástí zdrojů pro tuto sekci vyžádaných. Po vykonání kódu jsou zdroje opět odemknuty.

Na počátku sekce je možno zadat, jedná-li se o sekci pro čtení či pro čtení a zápis.

Není-li specifikováno, jedná se o sekci pro čtení a zápis.

3.4. Použití systému

Zde je popsán praktický způsob používání systému přímo v kódu. Tedy jednotlivé funkce (makra), jejich syntaxe, vstupy a výstupy. Detailnější a přesnější popis lze nalézt v referenčním manuálu v dodatku [H.](#).

Pro použití jednotlivých funkcionalit systému je třeba je volat s předponou `mps:`. Tato předpona zajistí zavolání z package MP-Secured, v němž je systém implementován.

3.4.1. Vytvoření zdroje

Zdroje je vhodné vytvořit na počátku běhu programu. Zdroj musí být vytvořen dříve, než je vyžádán kritickou sekci.

Zdroje se vytvářejí a registrují pomocí makra `defres`.

mps:defres *Macro*

`defres name([var | (var value)]*)`

Definuje zdroj daného jména (*name*) a chráněné proměnné daného jména (*var*) a nastaví jejich počáteční hodnoty (*value*). Pokud není hodnota specifikována, hodnota chráněné proměnné je nastavena na **nil**. Dané proměnné jsou registrovány jako součást vytvořeného zdroje.

Zdroj nemusí obsahovat žádné proměnné. V takovém případě jej lze využít například pro kontrolu vzájemného vyloučení souběhu dvou kritických sekcí.

Redefinici zdroje lze provést za běhu programu. Není to však vhodné, protože pokud takto dojde k přidání či ubrání chráněných proměnných, které jsou součástí zdroje, může být potřeba přepsat (a znovu zkompilovat) části kódu daný zdroj používající.

Stejně tak může být nevhodné měnit typ hodnot, uložených v proměnných.

Odstranění zdroje lze provést také. V běžném programu by to nemělo být potřeba, může se to však hodit například v případě, kdy zjistíme, že nějaký zdroj je již porušený, a jakákoliv další práce s ním může vést pouze k šíření chyb ve výpočtu. V dynamickém prostředí, jakým je Lisp, je také potřeba mít možnost vše změnit či zrušit.

Odstranění zdroje se provede s pomocí makra `delres`.

mps:delres *Macro*

`delres name`

Delres odstraní zdroj daného jména (*name*) ze systému. Pokud daný zdroj neexistuje, je signalizována chyba.

3.4.2. Vyžádání zdroje

K proměnným obsaženým ve zdroji se lze dostat pouze po úspěšném vyžádání příslušného zdroje. Zdroj lze vyžádat pouze pomocí makra `with-resources` implementujícího kritickou sekci.

mps:with-resources *Macro*

`with-resources ((resource*) &optional read) &body body`

Provede tělo (*body*) při držení všech zadaných zdrojů (*resource*).

Vyžádání zdroje pouze pro čtení Pro vyžádání zdroje pouze pro čtení je třeba zadat argument *read* s hodnotou `t` (true - pravda). Pokud je zdroj vyžádán pouze pro čtení, je možno, aby si jej i jiný proces vyžádal také pouze pro čtení, a to stejným způsobem. Pokud je zdroj kritickou sekcí vyžádán pouze pro čtení a je v jejím těle pokus o zápis jeho chráněné proměnné, je při kompilaci signalizována chyba.

3.4.3. Další funkce systému

Zde je popis funkcí systému, které pro jeho ovládání nejsou nutné, ale mohou být užitečné při vytváření programu používajícího systém.

Existenci zdroje lze testovat s pomocí predikátu *resourcep*.

mps:resourcep *Function*

`resourcep name`

Resourcep ověří, jestli existuje zdroj daného jména (*name*). Pokud daný zdroj existuje, vrátí `t`, jinak `nil`.

3.5. Příklady použití

Zde bude předvedeno několik jednoduchých příkladů použití. Nejdříve jeden modelový demonstrující funkčnost systému a základní použití, poté jeden malý a praktický. Následuje spojení těchto dvou příkladů do jednoho.

Kódy příkladů jsou v textové podobě k dispozici v dodatcích [G](#). nebo na k práci přiloženému médiu.

3.5.1. Jednoduchý praktický příklad

Tento příklad demonstruje jednoduché praktické použití MPS, konkrétně pro tisk řetězců do sdíleného výstupu.

Jelikož **standard-output**, stejně jako jiné možné výstupy pro funkci *format*, je proudem, při současném tisku funkcí *format* více procesy do takového sdíleného média, může dojít kvůli současnému tisku do výstupu k promíchání jednotlivých

řetězců. Funkčnost samotného programu tím nemusí být poškozena, ale dojde k značnému znepráhlednění výstupu.

Problém je vyřešen jednoduchým, všeobecně použitelným způsobem.

Je vytvořen zdroj *format-output*, který obsahuje chráněnou proměnnou *.out*, jejíž hodnota je inicializována na **standard-output**.

Dále je vytvořena jednoduchá funkce *p-format*, která se chová jako běžný *format* s několika drobnými vyjímkami. Přesné chování funkce, závisí na jejím prvním argumentu.

Pokud je tento argument **nil**, což u funkce *format* odpovídá snaze o vytvoření běžného formátovaného řetězce a jeho vrácení, je voláno přesně stejné vyhodnocení běžné funkce *format*. V takovém případě není potřeba se zabývat žádnou synchronizací.

Pokud je však první argument **t**, což odpovídá vytisknutí výstupního řetězce do **standatd-output**, je volána funkce *format* s argumentem *.out* uvnitř kritické sekce pro zdroj *format-output*. Tím je zajištěno výhradní vlastnictví správného výstupu vůči jiným voláním *p-format*.

V ostatních případech je pak výstup také prováděn v kritické sekci, ovšem do původně zadaného zdroje.

Funkce tak zajišťuje synchronizaci nejen při tisku řetězce do **standatd-output**, ale i do jiných výstupů, za předpokladu, že je používán pouze *p-format*, místo běžného formátu a že není k těmto výstupům přistupováno i jinak.

Pokud by nějaký výstup trpěl stejným neduhem jako **standatd-output**, stačí pro jeho vyřešení předdefinovat zdroj *format-output* a přidat do něj další proměnnou. Pro bezproblémový tisk do tohoto vstupu pak stačí volat *p-format* s příslušnou proměnnou v prvním argumentu.

V případě většího používání *p-format* pro různé sdílené výstupy může docházet k situaci, kdy jeden proces čeká s tiskem do jednoho zdroje, až jiný proces dotiskne do zdroje úplně jiného. Protože současný tisk by výstup nijak nenarušil, jedná se vlastně o zbytečné čekání.

Řešením takového problému, vznikne-li jeho potřeba, je nadefinovat zdroj a chráněnou proměnnou pro každý výstup jednotlivě a předdefinovat *p-format* tak, aby se podle symbolu v prvním argumentu rozhodovalo, který zdroj uzamkne.

Efektivita použití nové funkce je předvedena v dalším příkladu.

3.5.2. Teoretický příklad

První příklad je variací na standartní problém čtenářů a písarů.

Na prvním řádku je kód pro nahrání MPS package.

Následuje definice zdroje *data* s pěti chráněnými proměnnými *.a* až *.e*. Hodnoty chráněných proměnných jsou inicializovány na 0.

Dále je definována funkce *data-reader*, simulující čtenáře.

Funkce má vstup *number*. Ten *number* slouží k očíslování jednotlivých čtenářů.

Čtenář nejdříve oznámí výpisem "Reader *number* came" počátek své činnosti. Následuje pětinasobný cyklus, jehož tělo vypadá takto:

```
(mps:with-resources ((data) t)
  (p-format t "R~S: data - ~S|~S|~S|~S|~S ~%"
    number .a .b .c .d .e)))
(sleep 1)
```

Tedy se zdrojem *data* uzamčeným pouze pro čtení je proveden výpis "R*number*: data - .a | .b | .c | .d | .e". Následně pomocí (*sleep*1) funkce 1 sekundu nedělá nic. Poté co tento cyklus proběhne 5-krát, funkce oznámí své ukončení výpisem "reader *number* done".

Funkce *data-writer* simuluje činnost písaře.

Funkce má pouze argument *output*, jehož účel je stejný, jako v předchozím případě. Funkce také oznamuje začátek své činnosti výpisem "Writer came", obdobně jako předchozí. Pak následuje cyklus.

V těle cyklu je nejdříve se zdrojem *data* uzavřeným pro čtení i zápis nastavena hodnota *.a* na *i*, kde *i* je pořadí cyklu. Poté je proveden výpis "A > .a" a funkce zastavena na 5 sekund.

Následuje zvýšení dalších čtyřech proměnných ze zdroje *data* o 1.


Poté, již mimo kritickou sekci, je činnost opět pozastavena na 1 sekundu. Toto pozastavení je tu kvůli tomu, aby čtenáři spíše vstoupili do kritické sekce. Následuje další kolo cyklu.



Po ukončení cyklu funkce výpisem "Writer done" oznámí ukončení činnosti.

Funkce *super-writer* slouží k demonstraci bezpečnosti přepisu (redefinice) zdroje pomocí *defres*.

Funkce přidá ke zdroji *data* chráněnou proměnnou *.f* a hodnoty všech proměnných ve zdroji obsažených nastaví na 100.

Následuje zakomentovaný kód, který pomocí funkce *process-run-function* spustí v samostatných vláknech pětkrát funkci *data-reader*, jednou *data-writer* a jednou *super-writer*. O *process-run-function* více v kapitole 15 v [4].

Kód lze spustit tak, že je kurzor nataven na jeho počáteční závorku a je zmáčknuto tlačítko *Evaluate definition*  na ovládacím panelu.

Poté je vhodné rychle přepnout na nástroj *Output Browser*  nebo *Process Browser*  pro sledování činnosti programu.

Program je možno takto spustit opakovaně.

Poznamenejme, že i když je definován pouze jeden zdroj, program při běhu nakonec využívá pohodlně dvou. Druhý – *format-output* – je začleněn pomocí připojení předchozího příkladu.

4. Programátorská dokumentace

V této kapitole je velmi detailně rozebrán způsob implementace systému. Nejdříve z hlediska kódu samotného, použitých mechanismů a prostředků jazyka.

Podrobnost rozboru slouží k tomu, aby i uživatel jiného jazyka, než je Common Lisp byl schopen implementaci systému porozumět, případně ji převést do svého jazyka.

V druhé sekci jsou pak provedeny důkazy správnosti použitých uzamykacích mechanismů pro kontrolu přístupu k datům.

Studium této kapitoly není potřeba k používání výsledného systému.

4.1. Popis kódu

Zde jsou popsány jednotlivé operátory systému, společně s mechanismy které jsou tímto vytvořeny. V nutných případech budou zběžně popsány také jednotlivé prostředky jazyka, použité při realizaci. Systém bude popsán od nejjednodušších, spodních vrstev abstrakce, které samostatně nedávají příliš velký smysl bez znalosti zamýšleného cíle. O výsledném cíli – uživatelském fungování systému – viz část 3.3. předchozí kapitoly.

4.1.1. Definice balíčku

```
(defpackage MP-secured
  (:nicknames "MPS" "mps")
  (:documentation "Package build over MP package.
                   Contains various tools for secured parallel programing.")
  (:use "MP")
  (:export
   defres delres with-resources
   resourcep)
  (:add-use-defaults t))
```

Zde je vytvářen package (balíček) našich funkcí. Balíčky jsou obdobné pojmu "knihovna" v jiných jazycích. Více o balíčcích viz [3].

Vytvářený balíček se jmenuje "MP-secured", ale je možno se na něj také odvolávat přezdívkami (argument :nicknames) "MPS" či "mps". Přezdívka se používá, pro zkrácení zápisu při volání funkcí z balíčku či při jiném odkazování na něj.

Balíček MPS používá balíček MP - což je zkratka pro multiprocessing. Tato knihovna je standardní součástí LispWorks a nabízí některé základní nástroje pro práci s procesy, jako zámky, časovače, zasílání zpráv a podobně. Více o MP package lze nalézt v kapitole 15 z [4]. MPS z MP využívá pouze zámky.

Dále jsou definovány interní a externí symboly MPS. K interním symbolům je možno bezproblémově přistupovat pouze v rámci jejich balíčku, externí jsou přístupné i odjinud a představují rozhraní balíčku. Symboly obsazené v použitém balíčku (v tomto případě MP) jsou interní.

(in-package MPS)

První výraz druhého souboru provede přepnutí do právě vytvořeného balíčku, aby následující definice funkcí, metod apod. byly provedeny uvnitř něj.

4.1.2. Konstanty systému

```
(defconstant *resources* (make-hash-table)
  "the hashtable of resources")
```

```
(defconstant *r-vars* (make-hash-table)
  "the hashtable of protected variables contained in resources")
```

Zde jsou definovány globální konstanty. (Globální v rámci balíčku MPS, mimo něj nejsou přístupné.)

Konstanty obsahují hashovací tabulku (každá jinou). Obě tabulky jsou na počátku prázdné.

Hashovací tabulky poskytují jednoznačné zobrazení (mapování) klíčů na hodnoty. Mapovanou hodnotu pro daný symbol lze zjistit pomocí funkce (*gethash* < symbol > < tabulka >). Pokud tabulka pro daný symbol nemá hodnotu, vrací **nil**. Záznam v tabulce lze zřídit pomocí příkazu (*setf* (*gethash* < symbol > < tabulka >) < hodnota >). Záznam lze odstranit příkazem (*remhash* < symbol > < tabulka >). Více o tabulkách lze nalézt zde [\[3\]](#).

První tabulka slouží k mapování názvů zdrojů na jejich vnitřní realizaci jednotlivé instance třídy *resource* (viz níže). Důvodem pro tento způsob uložení je odstínění uživatele programátora od reálné realizace systému zdrojů a také možnost jednoduše spravovat existující zdroje a jejich jména z jednoho místa.

Druhá tabulka obsahuje mapování jmen chráněných proměnných na jejich hodnoty. Důvody pro realizaci tímto způsobem jsou stejné jako v předchozím případě. Navíc je takto zajištěno vytvoření jednotného jmenného prostoru pro chráněné proměnné. (Tabulka obsahuje každý klíč pouze jedenkrát.) A znemožnění dostat se k hodnotám chráněných proměnných jinak, než pomocí předdefinovaných nástrojů systému (bez nabourání systému).

4.1.3. Třída *resource* a její metody

Třída *resource* je vnitřní implementací zdroje. Pomocí svých slotů a metod zajišťuje bezpečnost svého uzamykání a otevírání, tedy to, že v jednu chvíli může

být zdroj vlastněn nejvýše jedním procesem pro zápis, ale že může být vlastněn více procesy pro čtení. Také je umožněno vícenásobné uzamčení zdroje tímtož procesem, při dodržení výše uvedených požadavků.

Třída `resource`

```
(defclass resource ()
  ((lock :reader get-lock
        :initform (mp:make-lock :importantp t)
        :initarg :lock
        :type 'lock
        :documentation "the main lock of resource")
   (vars :accessor get-variables
        :initarg :variables
        :initform '()
        :type 'list
        :documentation "the list of variables
                        asociated with the resource")
   (check-lock :reader get-check
               :initform (mp:make-lock)
               :documentation "second lock
                               to support multiple reading")
   (counter :accessor get-count
            :initform 0
            :type 'number
            :documentation "the counter of processes
                            reading the resource")
   (readers :reader get-readers
            :initform (make-hash-table)
            :documentation "the hashtable from processes
                            currently reading resource to number of times
                            each process does it"))
  (:documentation "Basic resource class, that takes care
                  about the integrity of its data"))
```

Třída má pět slotů. Argumenty `:reader` či `:accessor` definují jména přístupových metod k jednotlivým slotům. Argument `:initform` počáteční hodnotu slotu, `:initarg` názvy argumentů pro nastavení hodnoty slotu při vytváření instance třídy, `:type` definuje typ, který je povoleno do slotu uložit.

Slot *lock* obsahuje hlavní zámek zdroje. Při žádání zdroje je vždy testován tento zámek.

Slot *check-lock* obsahuje druhý zámek zdroje, určený k vyloučení souběhu při evidenci počtu procesů čtoucích zdroj.

Zámky lze žádat pomocí funkce

```
(mp:process-lock < zámek >< další volitelné argumenty >)
```

a uvolnit pomocí

```
(mp:process-unlock < zámek >)
```

Předpona `mp:` značí, že se jedná o funkce z balíčku MP. Pokud je žádaný zámek již zamknut jiným procesem, žádající proces čeká, dokud daný zámek není uvolněn. Více o zámcích viz v kapitole 15 v [4].

Slot *vars* obsahuje seznam jmen chráněných proměnných, přiřazených k dané instanci zdroje. Neobsahuje hodnoty těchto proměnných. Tyto hodnoty ani nejsou na jména proměnných (symboly) navázána běžným způsobem. Jak a proč tomu tak viz [výše](#).

Slot *counter* slouží k evidenci počtu procesů čtoucích zdroj. Slot může nabývat pouze celočíselných kladných hodnot. Počáteční hodnota slotu je 0.

Slot *readers* obsahuje hashovací tabulku. Tato tabulka slouží k evidenci kolikrát jednotlivé procesy čtoucí daný zdroj vyžádali (a ještě nevrátili) tento zdroj. Více o tabulkách viz [níže](#).

Více o evidenci čtoucích procesů u příslušných [metod](#).

Vnitřní metody

```
(defmethod check-in ((res resource))
  "increments the internal counter
   of processes that reads the resource"
  (let ((check-lock (get-check res)))
    (mp:process-lock check-lock
                      "checking in")
    (let ((process mp:*current-process*)
          (readers (get-readers res)))
      (unless (gethash process readers)
        (setf (gethash process readers)
              0))
      (incf (gethash process readers))
      (incf (get-count res)))
    (mp:process-unlock check-lock)))

(defmethod check-out ((res resource))
  "decrements the internal counter
   of processes that reads the resource"
  (let ((check-lock (get-check res)))
    (mp:process-lock check-lock
                      "checking out")
```

```

(let ((process mp:*current-process*)
      (readers (get-readers res)))
  (when (zerop
        (decf (gethash process readers)))
    (remhash process readers))
  (decf (get-count res)))
(mp:process-unlock check-lock))

```

Metody *check-in* a *check-out* slouží ke zvyšování a snižování vnitřního počítadla *counter* a upravování tabulky *readers* bez vzájemného souběhu. V metodách se vždy nejdříve zamkne zámek *check-lock* daného zdroje, poté se provede úprava tabulky a počítadla, následně je zámek opět uvolněn. Globální proměnná *mp: *current-process** obsahuje aktuální proces. Více v [4][kap. 15].

V metodě *check-in* se po uzamčení nejdříve otestuje, jestli současný proces již má v tabulce záznam. Pokud ne, záznam je zřízen s počáteční hodnotou 0. Hodnota záznamu je poté v každém případě o 1 zvýšena. Následně je zvýšen *counter*. Poté je *check-lock* uvolněn.

V metodě *check-out* se postupuje velmi obdobně. Nejdříve je snížena hodnota záznamu pro aktuální proces. Je-li hodnota po snížení rovna 0 (predikát *zerop*), záznam je z tabulky *readers* odstraněn. Tím je zabráněno, aby se v tabulce hromadily záznamy pro již nečinné procesy. Poté je *counter* snižován.

Counter tak vždy udává právě počet procesů, které již na daném zdroji provedli *check-in* a ještě neprovedli *check-out*. Je také roven právě součtu všech hodnot v tabulce *readers*.

Metody představují abstrakční podvrstvu pro metody zamknutí a odemknutí zdroje pro čtení. Viz metody *lock-r* a *unlock-r*.

Důkazy správnosti kódu programovou logikou těchto i dalších metod jsou provedeny v části 4.2. následující kapitoly.

Uzamykací metody

```

(defmethod lock-rw ((res resource))
  "locks the resource for reading and writing
  and waits till all reading processes check-out"
  (let ((lock (get-lock res)))
    (mp:process-lock lock
                      (format nil "waiting for reading
                                and writing ~S"
                              (mp:lock-name lock))))
  (let ((number (gethash mp:*current-process*
                          (get-readers res))))
    (if number
        (mp:process-wait "waiting for readers to check-out"

```

```

                                (lambda ()
                                  (= (get-count res) number)))
      (mp:process-wait "waiting for readers to check-out"
        (lambda ()
          (zerop (get-count res))))))

(defmethod unlock-r ((res resource))
  "unlocks the resource for reading"
  (check-out res))

(defmethod unlock-rw ((res resource))
  "unlocks the resource for reading and writing"
  (mp:process-unlock (get-lock res)))

```

Metody pro zamčení a odemčení zdroje. Zamčení zdroje je primárně realizováno přes jeho hlavní zámek ve slotu *lock*. Metody pro čtení i zápis (s koncovkou *-rw*) a pouze pro čtení (koncovka *-r*) se od sebe zásadně liší.

Metoda *lock-r* nejdříve vyžádá a zamkne hlavní zámek *lock*, poté provede *check-in* a poté opět uvolní hlavní zámek. Tím je umožněno, aby další proces provedl to samé. Provedením *lock-r* se (skrze *check-in*) zvýší interní počítadlo *counter* a hodnota příslušného záznamu v tabulce *readers* u daného zdroje.

Metoda *unlock-r* provede *check-out* a tím počítadlo a hodnotu záznamu sníží.

Metoda *lock-rw* je určená k uzamčení zdroje pro čtení i zápis a musí tedy zabránit jakémukoliv dalšímu procesu, aby provedl uzamčení zdroje (*lock-r* nebo *lock-rw*) pro svou potřebu. Metoda nejdříve vyžádá a zamkne hlavní zámek. Pak je otestováno, jestli aktuální proces má záznam v tabulce *readers*. (Tento údaj se v průběhu testu a celé metody nemůže měnit, protože jediný proces, který k právě tomuto záznamu kdy přistupuje, je právě aktuální proces.)

Pokud tento záznam má, proces čeká dokud *counter* není roven hodnotě záznamu, tedy dokud veškeré ostatní čtoucí procesy neuvolní zdroj. V případě, že záznam chybí (tj. zdroj není tímto procesem uzamčen pro čtení), proces čeká dokud *counter* není roven 0, tedy dokud veškeré čtoucí procesy neuvolní zdroj. To s určitostí nastane, pokud je zaručeno, že každý proces, který provede *check-in*, provede i *check-out*.

V průběhu čekání na *check-out* čtoucích procesů nenastane další *check-in*, protože je uzamčen hlavní zámek. Stejně tak nemůže nastat, že by dva procesy čekaly na vynulování počítadla – jeden z nich musel předtím zamknout hlavní zámek, a druhý se tedy nemohl poté k tomuto čekání dostat.¹ Metoda *lock-rw*

¹Celá mašinérie kolem vnitřních počítadel vyplývá z možnosti jeden zdroj vícekrát uzamknout jedním procesem pro čtení (*-r*) i zápis (*-rw*), a to v libovolném pořadí.

Přitom tato možnost nijak nezvyšuje efektivitu, bezpečnost či čitelnost výsledného programu

nezvyšuje vnitřní počítadlo zdroje. Stejný proces ji může provést vícekrát po sobě bez uvolnění zdroje. (Vícenásobné zamčení zámku tímtož procesem je možné a je provedeno okamžitě. Zámek je poté pro plné uvolnění třeba odemknout stejným počtem volání *mp:process-unlock*.)

Metoda *unlock-rw* prostě uvolní hlavní zámek. Teprve poté je tedy možné provést jeho další uzamčení v *lock-r* nebo *lock-rw* jiným procesem.

4.1.4. Uživatelské informativní funkce

Následující sada funkcí slouží k zjišťování informací o stavu systému.

Predikát ověřující zdroj

```
(defun resourcep (res)
  "checks if there is a resource of the given name in the system"
  (second (multiple-value-list (gethash res *resources*))))
```

Funkce *resourcep* je predikátem, který určí zda daný symbol je nebo není zdrojem. To je určeno tak, že se ověří přítomnost name v tabulce **resources**.

Predikát je využíván i uvnitř systému pro ověřování správnosti argumentu.

4.1.5. Kód pro vytvoření a odstranění zdroje

Většina následujícího kódu slouží k ověřování parametrů při vytváření zdroje. Další umožňují dynamičtější přístup ke zdroji a jeho proměnným - tedy jeho redefinici a odstranění.

Vyčištění zdroje

```
(defmethod clear-res ((res resource))
  "deletes resource's variables from the system"
  (dolist (r (get-variables res))
    (remhash r *r-vars*))
  (setf (get-variables res) nil))
```

Jedná se o pomocnou metodu volanou při odstraňování zdroje, nebo jeho redefinici. Tato metoda odstraní ze systému veškeré proměnné daného zdroje. Nejdříve je vymaže z tabulky **r-vars**, poté nastaví seznam proměnných zdroje vars na *bf nil* (prázdný seznam).

a kódu. Jediné opakované uzamčení zdroje (tímtož procesem), které může mít smysl, je nejdříve pouze pro čtení, a poté i pro zápis.

Možnosti vylepšení systému jsou diskutovány v poslední kapitole.

Funkce pro ověření argumentů při tvorbě zdroje Následují pomocné funkce pro ověření vhodnosti proměnných.

```
(defun ensure-unique-var (var)
  "ensures if there is not another protected variable
  with the same name"
  (not (second (multiple-value-list (gethash var *r-vars*)))))
```

Funkce ověří, jestli se již v systému nenachází chráněná proměnná stejného názvu. To je ověřeno neexistencí záznamu pro daný symbol v tabulce **r-vars**.

```
(defun check-vars (vars name)
  "ensures if the variables' names are unique among the system"
  (dolist (var vars)
    (unless (symbolp var)
      (error "protected variable must be a symbol, got ~S
              as a variable name when creating a resource ~S"
              var name))
    (when (member var (cdr (member var vars)))
      (error "symbol ~S appears multiply among variables
              while defining resource ~S"
              var name))
    (unless (ensure-unique-var var)
      (error "Existing variable name ~S" var))))
```

Funkce pro každý prvek seznamu *vars* postupně ověří, jestli se jedná o symbol, že tento symbol je v seznamu *vars* pouze jednou a že symbol již není použit jako název chráněné proměnné. V případě, že je některé z těchto pravidel porušeno, je signalizována příslušná chyba.

```
(defun check-and-set-pairs (pairs name)
  "checks the suitability of given protected
  variables and, if they are acceptable
  adds them into the system"
  (check-vars (mapcar #'first pairs) name)
  (loop for (var val) in pairs do
    (setf (gethash var *r-vars*) val)))
```

Předpokládá se, že argument *pairs* je seznam ve tvaru (*< symbol > < hodnota > **). Mapováním funkce *first* na tento seznam je získán seznam prvních členů jednotlivých párů, tedy seznam ve tvaru (*< symbol > **), který je poté předán k otestování funkci *check-vars*.

Pokud toto ověření proběhne bez problémů, pro každý pár je vytvořen záznam v tabulce **r-vars**, čímž jsou chráněné proměnné a jejich hodnoty zaneseny do systému.

Funkce pro definici zdroje

```
(defun define-resource (name pairs)
  "Crates, verifies and stores the resource"
  (check-and-set-pairs pairs name)
  (setf (gethash name *resources*)
        (make-instance 'resource
                        :variables (mapcar #'first pairs)
                        :lock (mp:make-lock
                               :name (format nil "resource ~S" name)
                               :important-p t))))
  name)

(defun redefine-resource (name pairs)
  "redefines contents of the existing resource"
  (let ((res (gethash name *resources*)))
    (lock-rw res)
    (clear-res res)
    (unwind-protect
      (progn
        (check-and-set-pairs pairs name)
        (setf (get-variables res)
              (mapcar #'first pairs))
        (unlock-rw res)))
    name)
```

Funkce *define-resource* a *redefine-resource* slouží k definici, resp. redefinici zdroje. Při běžné definici jsou nejdříve ověřeny a zaneseny do systému proměnné.

Pokud toto proběhne úspěšně, je do tabulky **resources** vložen nový záznam s klíčem *name* (jméno zdroje) a hodnotou nové instance třídy *resource*. Této instanci je nastaven slot *vars* na seznam dodaných symbolů a do slotu *lock* je uložen zámek s názvem "resource *name*".

Při redefinici zdroje je nejdříve existující zdroj uzamčen pro čtení i zápis, aby nedošlo k přepsání zdroje, který někdo používá. Zdroj je poté vyčištěn a dále jsou provedeny stejné procedury jako při tvoření nového zdroje. Poté je zdroj odemknut.

Speciální operátor jazyka *unwind-protect* slouží k tomu, aby jeho druhý argument byl vždy vyhodnocen, nezávisle na tom, jestli nedojde k nějakému způsobu přerušení vyhodnocení jeho prvního argumentu. V tomto případě zajistí, aby *unlock-rw* bylo provedeno vždy, nezávisle na úspěchu *check-and-set-vars*. Takovýto neúspěch povede k uvolnění prázdného zdroje. To nejspíše povede k chybě při prvním pokusu o použití chráněné proměnné ze zdroje.

```
(defun trans-pairs (pairs)
```

```
(cons 'list
      (mapcar (lambda (pair)
                (if (consp pair)
                    '(list ',(first pair) ,(second pair))
                    '(list ',pair nil)))
              pairs)))
```

Funkce převede seznam ve tvaru (`< symbol > |(< symbol > < hodnota >)*`) na seznam ve tvaru (`((< symbol > nil)|(< symbol > < hodnota >)*)`) resp. (`((< symbol > {< hodnota > |nil})*)`)

Jak přesně se to děje není pro výklad důležité. Účelem této úpravy je převést uživatelský vstup do jednodušší zpracovatelného tvaru.

Uživatelská makra k definici a smazání zdroje

```
(defmacro defres (name &rest pairs)
  "defines or redefines the resource,
   depends if it already exists"
  (unless (symbolp name)
    (error "First argument of defres must be a symbol. Got ~S."
           name))
  (if (resourcep name)
      '(redefine-resource ',name ,(trans-pairs pairs))
      '(define-resource ',name ,(trans-pairs pairs))))

(defmacro delres (name)
  "removes the resource from the system"
  (unless (resourcep name)
    (error "First argument of delres must be
           a name of a resource. Got ~S."
           name))
  '(let ((res (gethash ',name *resources*)))
    (lock-rw res)
    (clear-res res)
    (remhash ',name *resources*)
    (unlock-rw res)))
```

Defres a *delres* jsou makra.

Makra patří ke speciálním výrazům jazyka Lisp. Makra lze používat úplně stejně jako běžné funkce, ale fungují poněkud jinak. Jejich vyhodnocení má dvě fáze; makroexpanzi, při níž je provedena definice makra na argumentech, ta většinou vrací nějaký kód; a evaluaci, při níž je vrácený kód běžně vyhodnocen.

Tento kód je vyhodnocen na místě kde bylo původní volání makra.

Na makro lze také pohlížet jednoduše jako na funkci vracející kód, makro totiž může v obou svých fázích běžně pracovat s voláním funkcí či maker. Je také důležité, že makroexpanze probíhá před kompilací, tedy nikoliv na straně uživatele programu. Toho lze využít například ke kontrole správnosti zadaných argumentů, tedy ke kontrole kódu a programátora.

Makra mají ještě mnoho důležitých vlastností, více v [3].

Makro *defres* slouží k definici nového zdroje. Makro nejdříve ověří, jestli argument *name* je symbol. Pokud není, je signalizována chyba. Následně se ověří, jestli *name* již není zdrojem. Pokud je, je vrácen a vyhodnocen kód pro redefinici, místo pro definici.

Makro *delres* slouží k úplnému smazání zdroje. Makro nejdříve ověří že *name* je skutečně jméno zdroje. Pokud ano, uzamkne jej a smaže.

4.1.6. Aparát pro vyhodnocení kódu

Následuje kód funkcí a maker sloužících k vytvoření správného prostředí uvnitř kritických sekcí. Tedy takového, kde jsou přístupné chráněné proměnné a podle typu kritické sekce je lze libovolně využívat (sekce pro čtení i zápis), ne z nich pouze číst (sekce pouze pro čtení).

Sekce pro čtení

```
(defmacro set-val (v var x)
  (error "Trying to change value ~S of the protected variable ~S
         to ~S in a read critical section"
         v var x))
(defun val (v var)
  (declare (ignore var))
  v)

(defsetf val set-val)
```

Funkce *set-val* a *val* společně s operátorem *defsetf* definují chování při volání funkce *val* a, což je důležitější chování při volání *setf* funkce se jménem *val*, tedy při pokusu o změnu hodnoty volání (*val* < objekt >).

Protože *val* a *set-val* jsou interní symboly MPS, tato změna se neprojeví mimo balíček a neovlivní tak cizí kódy, které by mohly obsahovat volání vlastní *val*.

Funkce *val* pouze vrací svůj první argument, funkce *set-val* vždy vyvolá chybu. Smysl této úpravy je vysvětlen u následujícího kódu.

```
(defmacro with-vars-r (vars &body body)
  "executes body with given protected
   variables accesible for reading"
  (let* ((g-syms (mapcar (lambda (var)
```

```

                                (make-symbol (format nil "~S"
                                                    var)))
                                vars))
(vals (mapcar (lambda (var)
                '(gethash ',var *r-vars*))
              vars))
(symbs (mapcar (lambda (var sym)
                '(',var (val ,sym ',var)))
              vars g-symbs))
(pairs (mapcar (lambda (sym val)
                '(',sym ,val))
          g-symbs vals)))
'(let ,pairs
    (symbol-macrolet ,symbs
      ,@body))))

```

Makro *with-variables-r* je poměrně složitým makrem. Makro slouží k vyhodnocení těla *body* (kódu) v prostředí, v němž jsou přístupné hodnoty chráněných proměnných *vars*, ale nelze je měnit. Navíc, při pokusu o takovou změnu je signalizována chyba.

Makro si nejdříve pro účely vygenerovaného kódu vytvoří několik pomocných seznamů.

Seznam *g-syms* se skládá z nově vytvořených symbolů. Ty jsou při každém volání makra vytvořeny nově a nezastíní tak žádné již existující symboly. Pro přehlednost jsou navíc tyto symboly vytvořeny v čitelnějším tvaru. Ke každému symbolu $\langle \text{symbol} \rangle$ z *vars* je vytvořen odpovídající symbol se jménem $\langle \text{symbol} \rangle$.

Seznam *vals* obsahuje hodnoty proměnných *vars*. Konkrétně, ke každému symbolu $\langle \text{symbol} \rangle$ z *vars* je vytvořen kousek kódu $(\text{gethash } \langle \text{symbol} \rangle *r\text{-vars}*)$.

Tento kód při svém vyhodnocení vrátí aktuální hodnotu chráněné proměnné $\langle \text{symbol} \rangle$. Tento kód je dále vložen na správné místo tak, aby se vyhodnotil právě jednou a to ve chvíli, až je kód skutečně volán.

Seznam *symbs* využívá ke své konstrukci již existujících seznamů *g-syms* a *vars*. Seznam obsahuje ke každému symbolu $\langle \text{symbol} \rangle$ z *vars* a korespondujícímu symbolu $\langle \#symbol \rangle$ z *g-syms* kousek kódu ve tvaru $(\langle \text{symbol} \rangle (\text{val } \langle \#symbol \rangle' \langle \text{symbol} \rangle))$. Smysl tohoto kusu kódu bude vysvětlen při jeho vložení na určené místo.

Posledním vytvořeným seznamem je seznam *pairs*. Tento seznam ke svému vytvoření používá seznamy *g-syms* a *vals* - oba vytvořené s pomocí *vars*. Ke každému symbolu $\langle \text{symbol} \rangle$ z *vars* tak seznam *pairs* obsahuje pár ve tvaru $(\langle \#symbol \rangle (\text{gethash } \langle \text{symbol} \rangle *r\text{-vars}*))$. Smysl těchto párů vyvstane na místě jejich vložení.

Nyní se konečně dostáváme k místu, kde makro-expanzní generuje výstupní kód.

Tento kód bude vypadat pro vstupní seznam *vars* ve tvaru (`< symbol-1 > < symbol-2 > ... < symbol-n >`) a tělo `< body >` takto

```
(let (((<#symbol-1> (gethash <symbol-1> *r-vars*))
      (<#symbol-2> (gethash <symbol-2> *r-vars*))
      ...
      (<#symbol-n> (gethash <symbol-n> *r-vars*))))
(symbol-macrolet (((<symbol-1> (val <#symbol-1> '<symbol-1>))
                  (<symbol-2> (val <#symbol-2> '<symbol-2>))
                  ...
                  (<symbol-n> (val <#symbol-n> '<symbol-n>))))
  <body>))
```

Volání makra tedy bude nahrazeno tímto kódem.

Speciální operátor *let* prostě vytvoří prostředí v němž budou na nové symboly navázány hodnoty korespondujících chráněných proměnných. Pro další pochpení je však nutno vědět, co dělá speciální operátor *symbol-macrolet*.

Symbol-macrolet pracuje v zásadě se dvěma argumenty; úvodním seznamem párů ve tvaru (`< symbol > < expanze >`) a dodaným tělem `< body >`.

Symbol-macrolet provede makroexpanzi každého symbolu `< symbol >` v těle `< body >` na výraz `< expanze >`.

V našem případě je tedy každý symbol `< symbol >` chráněné proměnné z *vars* nahrazen výrazem (`val < #symbol > '< symbol >`). Je zde tedy volána funkce *val*, která běžně prostě vrátí hodnotu prvního argumentu. Což je v našem případě `< #symbol >`, jehož hodnota je lokálně navázána na hodnotu chráněné proměnné `< symbol >`.

Avšak v případě, že se nahrazovaný symbol `< symbol >` chráněné proměnné nachází ve výrazu tvaru (`setf < symbol > < něco >`), který je změněn pomocí *symbol-macrolet* na (`setf(val < #symbol > '< symbol > < něco >`), dojde k volání funkce *set-val*, která signalizuje chybu. Chyba je signalizována již při kompilaci kódu.

Toto všechno se netýká chráněných proměnných, které nejsou obsaženy v argumentu *vars*. Pokud by se taková chráněná proměnná objevila, nebude její hodnota nijak nastavena a bude signalizována odpovídající chyba.

Tímto je tedy dosaženo následujícího:

5. není možno zapisovat do chráněných proměnných
6. není možno číst ani zapisovat do chráněných proměnných, které nebyly explicitně vyžádány
7. obě tyto chyby jsou detekovány již při kompilaci.

sekce pro čtení i zápis

```
(defun make-var-val-pairs (vars)
  (mapcar #'list
    vars
    (mapcar (lambda (v)
              '(gethash ',v *r-vars*))
            vars)))
```

Funkce *make-var-val-pairs* vytvoří seznam páru ve tvaru (`< symbol > (gethash' < symbol > *r-vars*)`) pro každý symbol `< symbol >` z argumentu *vars*. Účel tohoto seznamu bude osvětlen u následujícího makra.

```
(defmacro with-vars-rw (vars &body body)
  "executes the body with given protected variables
   accessible for reading and writing"
  (let ((result (gensym)))
    '(let ((,result nil)
          ,@(make-var-val-pairs vars))
      (setf ,result (progn ,@body))
      (loop for var in ',vars
            for val in ,(cons 'list vars) do
              (setf (gethash var *r-vars*) val))
      ,result))))
```

Makro *with-variables-r* je poněkud jednodušší, než makro předchozí. Slouží k vyhodnocení kódu `< body >` v prostředí, kde jsou navázány hodnoty chráněných proměnných ze seznamu *vars* a uložení případných změn.

Makro si nejdříve vytvoří nový symbol *result*, který slouží k uložení výsledku vyhodnocení těla. Poté je již přistoupeno ke generování kódu nahrazujícího volání makra.

Zde je nejdříve vytvořeno prostředí, v němž je symbol *result* navázán na *nil* a symboly vyžádaných chráněných proměnných na příslušné hodnoty. Následně je provedeno tělo, výsledek jehož posledního výrazu je navázán na *result*. Poté je provedeno zpětné uložení hodnot do tabulky **r-vars**. Nakonec je vrácen výsledek uložený v symbolu *result*.

Tímto je zajištěno následující:

8. lze měnit a číst chráněné proměnné, které byly explicitně vyžádány
9. nelze číst ani ukládat žádné jiné chráněné proměnné
10. pokus o čtení nevyžádané chráněné proměnné signalizuje chybu již při kompilaci

4.1.7. Spojení předchozího

Koncové uživatelské makro `with-resources`

```
(defmacro with-resources
  ((resources &optional (read nil)) &body body)
  "checks the resources, sorts them,
  and executes the body while holding them."
  (dolist (r resources)
    (unless (resourcep r)
      (error "~S is not registered resource" r))
    (when (member r (cdr (member r resources)))
      (error "resource ~S appears multiply among resources"
             r))))
  (let* ((result (gensym))
         (ress (mapcar
                  (lambda (r)
                    (gethash r *resources*))
                  (sort resources #'string<
                          :key #'string)))
         (vars (apply #'append
                      (mapcar #'get-variables ress))))
    (read-arg read))
    '(let ((,result nil))
      (dolist (r ',ress)
        (,(if read-arg 'lock-r 'lock-rw) r))
      (unwind-protect
        (setf ,result
              (,(if read-arg 'with-vars-r
                    'with-vars-rw
                    ,vars
                    ,@body)))
        (dolist (r ',ress)
          (,(if read-arg 'unlock-r 'unlock-rw) r)))
      ,result)))
```

Makro *with-resources* je koncovým uživatelským makrem většiny předchozího aparátu. Makro slouží k vyhodnocení kódu `< body >` při držení zdrojů `< resources >`.

Makro nejdříve otestuje vstupní parametr `< resources >`, jestli se jedná o seznam složený pouze ze zdrojů a jestli se mezi nimi nevyskytuje některý vícekrát. Pokud ano, je při kompilaci signalizována chyba.

Následně je vytvořen nový symbol *result*, dodaná jména zdrojů seřazena a do proměnné *ress* uložen stejně seřazený seznam odpovídajících instancí třídy

resource, do proměné *vars* seznam proměnných těchto zdrojů a do *read-arg* vyhodnocení argumentu *read*. Není-li *read* zadán, je jeho hodnota nastavena na **nil**.

Následuje generování kódu.

Ve vygenerovaném kódu jsou vždy nejdříve provedeno postupné uzamčení zdrojů ze seznamu *ress*. Poté vyhodnocení kódu `< body >` ve správném prostředí s chráněnými proměnnými ze seznamu *vars* – výsledek tohoto vyhodnocení je uložen v `< result >`. Nakonec jsou všechny zdroje z *ress* ve stejném pořadí odemknuty a uložený výsledek vrácen.

Uzamykací a odemykací funkce a vyhodnocující makro jsou vybrány na základě hodnoty volitelného argumentu *read*. Pokud je *read* **t** (true), jsou vybrány funkce *lock-r*, *unlock-r* a makro *with-variables-r*, v opačném případě jsou použity *lock-rw*, *unlock-rw* a *with-variables-rw*.

Výsledný kód pro `< read > = t` pak vypadá zhruba takto:

```
(let ((<result> nil))
  (dolist (r <resources>)
    (lock-r r))
  (unwind-protect
    (setf <result>
          (with-vars-r <variables> <body>))
    (dolist (r <resources>)
      (unlock-r r)))
  <result>)
```

pro `< read > = nil` takto

```
(let ((<result> nil))
  (dolist (r <resources>)
    (lock-rw r))
  (unwind-protect
    (setf <result>
          (with-vars-rw <variables> <body>))
    (dolist (r <resources>)
      (unlock-rw r)))
  <result>)
```

Tedy kromě zkontrolování vstupů makro zajistí uzamčení a odemčení zdrojů odpovídající funkcí a vyhodnocení `< body >` ve správném prostředí se správnými chráněnými proměnnými. Díky speciálnímu operátoru *unwind-protect* bude každý uzamčený zdroj opět odemčen, nezávisle na úspěšnosti vyhodnocení `< body >`.

Seřazení zdrojů slouží k naivním řešením deadlocku. V případě vnoření více kritických sekcí pro zápis do sebe může toto řešení selhat, protože může dojít k vytvoření cyklu čekání.

4.2. Důkazy správnosti

V této sekci budou pomocí programové logiky dokázány některé vlastnosti systému. Především vnitřní konzistence systému a správné chování kritických sekcí. Důkazy nebudou většinou úplné. Je předpokládána správnost zámků ze systémového balíčku MP.

4.2.1. Jazyk a předpoklady

Důkazy jsou prováděny nad původním kódem. Je použitý logický aparát obdobný tomu v [2]. Stejně jako tam, bude předpokládána intuitivní schopnost čtenáře pochopit a porozumět i ne zcela formální důkazy.

Bude používáno následující značení a předpoklady:

Zámek L vlastněný (uzamčený) procesem p bude značen $p(L)$, $not(L)$ pak značí nezamčený zámek L .

Předpokládáme, že zámek nemůže být v jednu chvíli uzamčen i odemčen zároveň a že může být uzamčen pouze jedním procesem. Možnost vícenásobného uzamčení zámku jedním procesem nebude pro jednoduchost uvažována.

Aktuální proces budeme značit symbolem $*cp*$.

V následujícím bude uvažován pouze jeden konkrétní zdroj instance třídy resource. Pro její součásti bude použito následující značení:

C pro hodnotu slotu counter; R pro tabulku readers; $R[p]$ pak pro hodnotu v R pro proces p ; $CH-L$ značí zámek check-lock, $M-L$ zámek lock.

Za konzistentní stav I konkrétního zdroje považujeme, pokud

$$C = \sum (R[p_i]; \forall i) \wedge R[p_i] > 0; \forall i.$$

Při inicializaci zdroje je $C = 0$ a R prázdné a oba zámky otevřené. I je tedy na počátku splněno.

4.2.2. Vnitřní metody

U metody *check-in* požadujeme, kromě zachování konzistence, aby se o 1 zvýšil counter C a hodnota $R[p]$ pro proces p , který provede *check-in*. To je možno zapsat takto:

$$\{I \wedge C = x \wedge R[*cp*] = y\} \text{check-in} \{I \wedge C = x + 1 \wedge R[*cp*] = y + 1\}$$

Začneme s platnou prekondicí.

$$\{I \wedge C = x \wedge R[*cp*] = y\} \Leftrightarrow \{I \wedge C = z + y \wedge R[*cp*] = y\}$$

```
(defmethod check-in ((res resource))
  "increments the internal counter of
  processes that reads the resource"
  (let ((check-lock (get-check res)))
```

;aby bylo možno pokračovat, musí být zámek odemknut, tedy:

$$\{I \wedge C = z + y \wedge R[*cp*] = y \wedge \text{not}(CH - L)\}$$

(mp:process-lock check-lock "checking in")

$$\{I \wedge C = z + y \wedge R[*cp*] = y \wedge *cp* (CH - L)\}$$

(let ((process mp:*current-process*)
(readers (get-readers res)))

;zde dochází k částečnému větvení programu, tedy

$$\{I \wedge C = z + y \wedge R[*cp*] = y \wedge *cp* (CH - L) \wedge (y = \text{nil} \vee y > 0)\}$$

(unless (gethash process readers)

$$\{I \wedge C = z + \text{nil} \wedge R[*cp*] = \text{nil} \wedge *cp* (CH - L)\}$$

(setf (gethash process readers) 0))

;V této větvi zde přestává platit I , protože

$$\{I \wedge R[*cp*] = 0\} \Rightarrow \text{false}.$$

Je tedy třeba rozepsat predikát:

$$\{z = \sum R[p_i \neq *cp*] > 0 \wedge C = z \wedge R[*cp*] = 0 \wedge *cp* (CH - L)\}$$

;stav programu pro $R[*cp*] \neq \text{nil}$:

$$\{I \wedge C = z + y \wedge R[*cp*] = y \wedge *cp* (CH - L) \wedge y > 0\}$$

;zobecnění obou větví:

$$\{z = \sum R[p_i \neq *cp*] > 0 \wedge C = z + y \wedge R[*cp*] = y \wedge *cp* (CH - L) \wedge y \geq 0\}$$

(incf (gethash process readers))

$$\{z = \sum R[p_i \neq *cp*] > 0 \wedge C = z + y \wedge R[*cp*] = y + 1 \wedge *cp* (CH - L) \wedge y \geq 0\}$$

(incf (get-count res)))

$$\{z = \sum R[p_i \neq *cp*] > 0 \wedge C = z + y + 1 \wedge R[*cp*] = y + 1$$

$$\wedge *cp* (CH - L) \wedge y \geq 0\} \Rightarrow$$

$$\{z + y + 1 = \sum R[p_i] > 0 \wedge C = z + y + 1 \wedge R[*cp*] = y + 1$$

$$\wedge *cp* (CH - L) \wedge y \geq 0\} \Rightarrow$$

$$\{I \wedge C = z + y + 1 \wedge R[*cp*] = y + 1 \wedge *cp* (CH - L)\} \Rightarrow$$

$$\{I \wedge C = x + 1 \wedge R[*cp*] = y + 1 \wedge *cp* (CH - L)\}$$

(mp:process-unlock check-lock)))

$$\{I \wedge C = x + 1 \wedge R[*cp*] = y + 1 \wedge \text{not}(CH - L)\}$$

Celkově tedy platí:

$$\{I \wedge C = x \wedge R[*cp*] = y\} \text{check-in} \{I \wedge C = x + 1 \wedge R[*cp*] = y + 1\}.$$

Navíc, v době, kdy je zdroj v nekonzistentním stavu, je uzavřen jeho $CH-L$ procesem $*cp*$ a po úpravě zdroje je zámek opět odemknut. Metoda *check-out* je v zásadě obdobná jako *check-in* a klademe na ni obdobné požadavky:

$$\{I \wedge C = x \wedge R[*cp*] = y\} \text{check-out} \{I \wedge C = x - 1 \wedge R[*cp*] = y - 1\}$$

Důkaz opět začneme ekvivalentně upravenou prekondicí

$$\{I \wedge C = y + z \wedge R[*cp*] = y\}$$

```
(defmethod check-out ((res resource))
  "decrements the internal counter of
   processes that reads the resource"
  (let ((check-lock (get-check res)))
```

$$\{I \wedge C = y + z \wedge R[*cp*] = y \wedge \text{not}(CH - L)\}$$

```
(mp:process-lock check-lock "checking out")
```

$$\{I \wedge C = y + z \wedge R[*cp*] = y \wedge *cp* (CH - L)\}$$

```
(let ((process mp:*current-process*)
      (readers (get-readers res)))
```

;V tomto je imperativní forma programové logiky nevhodná pro funkcionální program, rozepíšeme tedy příkazy postupně.

$$\{I \wedge C = y + z \wedge R[*cp*] = y \wedge *cp* (CH - L)\}$$

```
; (decf R[*cp*])
```

$$\{z = \sum R[p_i \neq *cp*] \wedge C = y + z \wedge R[*cp*] = y - 1 \wedge *cp* (CH - L)\}$$

;a pak dochází k větvení programu:

$$\{z = \sum R[p_i \neq *cp*] \wedge C = y + z \wedge R[*cp*] = y - 1 \wedge *cp* (CH - L) \wedge (y - 1 = 0 \vee y - 1 \neq 0)\}$$

```
(when (zerop
      (decf (gethash process readers)))
```

$$\{z = \sum R[p_i \neq *cp*] \wedge C = 1 + z \wedge R[*cp*] = 0 \wedge *cp* (CH - L)\}$$

```
(remhash process readers)
```

$$\{z = \sum R[p_i] \wedge C = 1 + z \wedge R[*cp*] = \text{nil} \wedge *cp* (CH - L)\}$$

;Druhá větev výpočtu:

$$\{z = \sum R[p_i \neq *cp*] \wedge C = y + z \wedge R[*cp*] = y - 1 \wedge *cp* (CH - L) \wedge y - 1 \neq 0\}$$

;zobecníme obě větve:

$$\{z = \sum R[p_i \neq *cp*] \wedge *cp* (CH - L) \wedge ((C = y + z \wedge R[*cp*] = y - 1) \vee (C = x \wedge R[*cp*] = \text{nil}))\}$$

```

(decf (get-count res)))

{z =  $\sum R[p_i \neq *cp*] \wedge ((C = y - 1 + z \wedge R[*cp*] = y - 1) \vee$ 
  ( $C = x - 1 \wedge R[*cp*] = nil$ ))  $\wedge *cp* (CH - L)$ }  $\Rightarrow$ 
{I  $\wedge ((C = y - 1 + z \wedge R[*cp*] = y - 1) \vee$ 
  ( $C = x - 1 \wedge R[*cp*] = nil$ ))  $\wedge *cp* (CH - L)$ }  $\Rightarrow$ 
{I  $\wedge C = x - 1 \wedge (R[*cp*] = y - 1 \vee R[*cp*] = nil) \wedge *cp* (CH - L)$ }

(mp:process-unlock check-lock))

```

```

{I  $\wedge C = x - 1 \wedge (R[*cp*] = y - 1 \vee R[*cp*] = nil)$ }

```

To odpovídá tomu, čeho jsme chtěli dosáhnout. Opět platí, že v době, kdy je zdroj v nekonzistentním stavu, je uzamčen zámek $CH-L$.

Protože *check-in* a *check-out* jsou jediné metody v jejichž těle jsou upravovány hodnoty C a R a protože to se vždy děje při uzamčeném zámku $CH-L$, který může v danou chvíli vlastnit pouze jediný proces, platí pro tyto úpravy vzájemné vyloučení (v rámci jednoho zdroje).

Z toho vyplývá, že tyto úpravy lze považovat za atomické a metody za nedělitelné.

Protože žádné jiné metody také nepoužívají $CH-L$ přímo, lze také dokázat, že obě metody skončí v konečném čase a mají jistotu provedení a absenci zbytečného čekání. (Poslední dvě vlastnosti značně závisí na plánovači MP a LispWorks.)

4.2.3. Uzamykací metody

Nyní budou dokázány některé vlastnosti pro metody pro uzamčení a odemčení zdroje.

Nejdříve metoda pro uzamčení zdroje pro čtení. Po té požadujeme aby s jistotou skončila a aby byla vícekrát proveditelná stejným, i jiným procesem. Dále musí samozřejmě zachovat výše uvedené I .

Tedy z hlediska výrazů požadujeme:

$$\{I\}_{\text{lock-r}}\{I\}$$

```

(defmethod lock-r ((res resource))
  "locks the resource for reading"
  (let ((lock (get-lock res)))

    {I  $\wedge not(M-L)$ }

    (mp:process-lock lock
      (format nil "waiting for read ~S"
        (mp:lock-name lock))))

    {I  $\wedge *cp* (M-L)$ }  $\Leftrightarrow$ 
    {I  $\wedge *cp* (M-L) \wedge C = x$ }

```

```

    (check-in res)

    {I ∧ *cp* (M-L) ∧ C = x + 1}

    (mp:process-unlock lock)))

    {I ∧ not(M-L) ∧ C = x + 1}

```

Při každém provedení *lock-r* se tedy zvýší o 1 příslušná vnitřní počítadla. Metoda dále potřebuje pro své provedení zámek *M-L*, avšak poté ho opět uvolní. Výše uvedené požadavky jsou také splněny.

Metoda uzamčení pro čtení i zápis musí svým vykonáním zaručit, že v současné chvíli žádné jiné procesy než **cp** neprovádějí nic se zdrojem (tzn. nezapisují ani nečtou). Vzhledem k tomu, jak byly definovány proměnné *C* a *R*, znamená to, že po provedení *lock-rw* musí platit $C = R[*cp*]$. To znamená, že žádné jiné procesy než **cp** nejsou počítány mezi čtenáře zdroje. Při zachování *I* celkově tedy:

$$\{I\}\text{lock-rw}\{I \wedge C = R[*cp*]\}.$$

```

(defmethod lock-rw ((res resource))
  "locks the resource for reading and writing
   and waits till all reading processes check out"
  (let ((lock (get-lock res)))

    {I ∧ not(M-L)}

    (mp:process-lock lock
      (format nil
        "waiting for read and write ~S"
        (mp:lock-name lock))))

    {I ∧ *cp* (M-L)}

    (let ((number (gethash mp:*current-process*
      (get-readers res))))

```

; Protože $R[*cp*]$ je upravováno pouze procesem **cp** v metodách *check-in* a *check-out* a proces **cp** je nyní zde, $R[*cp*]$ je v rámci této metody konstantní $\{I \wedge *cp* (M-L) \wedge R[*cp*] = \text{number} \wedge (\text{number} = y \vee \text{number} = \text{nil})\}$

```

    (if number

      {I ∧ *cp* (M-L) ∧ R[*cp*] = number ∧ number = y ∧ C = x} ⇔
      {I ∧ *cp* (M-L) ∧ R[*cp*] = y ∧ C = z + y}

      (mp:process-wait "waiting for readers to check-out"
        (lambda ()
          (= (get-count res) number))))

```

$$\{I \wedge *cp * (M-L) \wedge R[*cp*] = y \wedge C = y\} \Leftrightarrow$$

$$\{I \wedge *cp * (M-L) \wedge R[*cp*] = C = y\}$$

;větev výpočtu pro $R[*cp*] = \mathbf{nil}$

$$\{I \wedge *cp * (M-L) \wedge R[*cp*] = \mathbf{nil}\} \Leftrightarrow$$

$$\{I \wedge *cp * (M-L) \wedge R[*cp*] = \mathbf{nil} \wedge C = x\}$$

```
(mp:process-wait "waiting for readers to check-out"
  (lambda ()
    (zerop (get-count res))))))
```

$$\{I \wedge *cp * (M-L) \wedge R[*cp*] = \mathbf{nil} \wedge C = 0\}$$

; sjednocení obou větví:

$$\{I \wedge *cp * (M-L) \wedge ((R[*cp*] = \mathbf{nil} \wedge C = 0) \vee R[*cp*] = C = y)\} \Rightarrow$$

$$\{I \wedge *cp * (M-L) \wedge R[*cp*] = C\}$$

Metoda *lock-rw* tedy splňuje základní aserci, která pro ní byla stanovena. Dále za sebou vždy zavře zámek *M-L*. Metody *lock-r* a *lock-rw* mohou být provedeny stejným procesem v libovolném pořadí a množství za sebou.

(Tentýž proces může uzamknout jeden zámek vícekrát a pokud nejsou zúčastněny jiné procesy. $R[*cp*] = C$ vyplývá přímo z I .)

Metoda ve svém těle čeká, dokud není naplněna podmínka $C = R[*cp*]$. Jelikož $R[*cp*]$ je v rámci metody konstantní a obecně z I vyplývá $C \geq R[*cp*]$, jedná se o čekání na pokles C . Ten se děje skrze metodu *check-out* resp. *unlock-r* (viz 4.2.3. níže) prováděných jinými (čtoucími) procesy.

Protože je zajištěno, že každý proces, který provedl *lock-r*, provede i *unlock-r* (viz *with-resources* - Za předpokladu ukončení vyhodnocovaného kódu.), pokles C je zajištěn.

To, že v době čekání na splnění podmínky nebude C opětovně narůstat je zajištěno tím, že je již uzamčen zámek *M-P*, nemůže tedy proběhnout žádný *check-in* skrze *lock-r*. Tím je dokázána absence zbytečného čekání a částečně i jistota dokončení metody za předpokladu absence deadlocku jiných čtoucích procesů.

Protože tělo obou uzamykacích metod probíhá s držením zámku *M-L*, platí pro ně vzájemné vyloučení.

Je-li metoda *lock-r* vykonána jiným procesem p , proces $*cp*$ se při provádění metody *lock-rw* vždy zablokuje. (Pokud $*cp* \neq p$.) Respektive, pokud se tak nestane bez toho, aby se nejdříve opět snížilo $R[p]$, vede to k nepravdě.

$$\{I\}lock - r, lock - rw \Rightarrow$$

$$\{I \wedge not(M-L) \wedge C = y + z \wedge C \geq 1 \wedge R[p] = y \wedge y \geq 1\}lock - rw \Rightarrow$$

$$\{I \wedge *cp * (M-L) \wedge C = y + z$$

$$\wedge R[p] = y \wedge y \geq 1 \wedge C = R[*cp*]\} \Rightarrow$$

$$\{I \wedge C = R[p] + z \wedge R[p] \geq 1 \wedge C = R[*cp*]$$

$$\wedge C = Sum(R[p_i])\} \Rightarrow$$

$$\{I \wedge \sum R[p_i] = R[p] + z = R[*cp*] \wedge R[p] \geq 1\} \Rightarrow$$

$$\{I \wedge R[p] + R[*cp*] + q = R[p] + z = R[*cp*] \wedge R[p] \geq 1\} \Rightarrow$$

$\{R[p] + q = 0, R[p] > 0\}$
 $\Rightarrow false$

Po metodě *lock-rw* provedené procesem p_1 , se obecně jiný proces p_2 na metodě *lock-r* zablokuje, neboť k jejímu provedení potřebuje zámek $M-L$. Ze stejných důvodů by se po provedení *lock-rw* procesem p_1 zablokoval proces p_2 na metodě *lock-rw* k jejímu provedení potřebuje mít volný zámek $M-L$.

Metoda *unlock-r* je jenom zapouzdřená *check-out*.

```
(defmethod unlock-r ((res resource))
  "unlocks the resource for reading"
  (check-out res))
```

Platí pro ni tedy to samé, jako pro *check-out*.

$$\{I \wedge C = x \wedge R[*cp*] = y\} \text{unlock-r} \{I \wedge C = x - 1 \wedge R[*cp*] = y - 1\}$$

Metody *lock-r* a *unlock-r* musí dohromady fungovat tak, aby jejich postupné provedení jedním procesem nezměnilo stav programu.

$$\begin{aligned} &\{I \wedge \text{not}(M-L) \wedge C = x \wedge R[*cp*] = y\} \text{lock-r}, \text{unlock-r} \Rightarrow \\ &\{I \wedge \text{not}(M-L) \wedge C = x + 1 \wedge R[*cp*] = y + 1\} \text{unlock-r} \Rightarrow \\ &\{I \wedge \text{not}(M-L) \wedge C = x \wedge R[*cp*] = y\} \end{aligned}$$

Pokud je mezi vyhodnocení metod vložen podprogram P , neinterferující s I a $M-L$ pak pro vyhodnocení P platí:

$$\begin{aligned} &\{I \wedge \text{not}(M-L) R[*cp*] = y\} \text{lock-r}, P, \text{unlock-r} \{I \wedge \text{not}(M-L) \\ &\quad \wedge R[*cp*] = y\} \Rightarrow \\ &\{I \wedge \text{not}(M-L) R[*cp*] = y + 1\} P \{I \wedge \text{not}(M-L) \wedge R[*cp*] = y + 1\} \end{aligned}$$

To tedy umožňuje jakémukoliv procesu provést nad stejným zdrojem *lock-r* a *unlock-r*, avšak jakýkoliv jiný proces, který by se nad stejným zdrojem pokusil provést *lock-rw* je zablokován, dokud neskončí P a není proveden následný *unlock-r*.

Metoda *unlock-rw* provede prosté odemknutí zámku $M-L$. Musel být tedy příslušným procesem předtím zamknutý.

```
(defmethod unlock-rw ((res resource))
  "unlocks the resource for reading and writing"
  (mp:process-unlock (get-lock res)))
```

$$\{I \wedge *cp* (M-L)\} \text{unlock-rw} \{I \wedge \text{not}(M-L)\}$$

Metody *lock-rw* a *unlock-rw* vytvářejí dohromady kritickou sekci; Pokud mezi jejich volání vložíme podprogram P neinterferující s I či zámkem $M-L$, pak pro P platí:

$$\begin{aligned} &\{I \wedge \text{not}(M-L)\} \text{lock-rw}, P, \text{unlock-rw} \{I \wedge \text{not}(M-L)\} \Rightarrow \\ &\{I \wedge \text{not}(M-L) \wedge R[*cp*] = x\} \text{lock-rw}, P, \text{unlock-rw} \{I \wedge \text{not}(M-L)\} \Rightarrow \\ &\{I \wedge *cp* (M-L) \wedge R[*cp*] = x = C\} P \{I \wedge *cp* (M-L)\} \Rightarrow \end{aligned}$$

$$\{I \wedge *cp * (M-L) \wedge R[*cp*] = x = C\}$$

P

$$\{I \wedge *cp * (M-L) \wedge R[*cp*] = x = C\}$$

Tedy provedení *P* probíhá za držení zámku *M-L* a *P* provádějící proces **cp** je po celou dobu případným jediným čtoucím procesem. Během provádění *P* tedy není možné provést žádnou s uzamykacích či odemykacích metod na daném zdroji. *P* má tedy daný zdroj s jistotou k dispozici pouze pro sebe.

Veškerá tvrzení zde se týkala práce s jedním zdrojem. Jelikož jednotlivé zdroje jsou vzájemně zcela oddělené, vše platí i při práci s více zdroji.

5. Další vývoj

V této kapitole budou diskutovány jednotlivé možné směry dalšího rozšiřování a vylepšování systému.

5.1. Robustnost

Systém je možno vylepšit z hlediska běžných vlastností robustnosti.

Robustnost systému může být například zvýšena ošetřením případů paralelní definice nových zdrojů stejného jména.

Takovéto vylepšení systému však není primárním cílem práce, neboť takováto robustnost by podporovala způsob přístupu k chráněným proměnným, který příliš neodpovídá běžným globálním proměnným a může vést k dalším problémům jako zneplatnění části zkompilevaného kódu kvůli neexistenci tam použitých chráněných proměnných.

5.2. Další kontrola programátora

Velká část správnosti výsledného programu je zajištěna kontrolou programátora systémem. Tuto kontrolu by jistě šlo rozšířit.

5.2.1. Kontrola typu

Například celkem bez větších problémů by šlo přidat upozornění (varování), pokud by do chráněných proměnných měl být vložen jiný typ dat než pro jaké jsou doporučeny (čísla, znaky, konstantní symboly...). Takovouto kontrolu lze velmi jednoduše provést při tvorbě zdroje, hůře pak již v samotném kódu a v případě nedeterministického programu by ji bylo možno provádět až za běhu.

Taková kontrola příliš neodpovídá dynamicky typovanému jazyku, jímž Common Lisp je, ani jeho svobodné filozofii, která předkládá, že pokud programátor překračuje doporučení, ví, proč to dělá.

5.2.2. Kontrola nevhodného přiřazení

Jiným, vhodnějším příkladem rozšíření kontroly programátora je detekovat jako prohrěšek (signalizovat varování), pokud se uvnitř kritické sekce objeví přiřazení do proměnné, jejíž jméno odpovídá existující chráněné proměnné, která však náleží do zdroje pro tuto sekci nevyžádaného. V takovém případě lze oprávněně očekávat chybu programátora.

V případě přidání této funkcionality je také nutné dát programátorovy nástroj, kterým deklaruje, že ví, co dělá a zabrání signalizování varování.

5.2.3. Odstranění zbytečného zamykání zdrojů

Další cestou rozšíření kontroly programátora, která skutečně povede k zlepšení vlastností výsledného programu je detekovat v kódu zbytečná uzamčení zdrojů.

Jedním typem zbytečného uzamčení je takové, kdy je uzamčen zdroj, jehož proměnné nejsou uvnitř sekce využity. Takové zamčení je na první pohled zbytečné a mělo by být detekováno jako chyba. Odstranění takového uzamčení vede k snížení zbytečného čekání v programu.

I v tomto případě platí, že může nastat případ, kdy programátor chce uzamknout zdroj, aniž by pracoval s jeho proměnnými. Měl by tedy mít možnost deklarovat tuto skutečnost a případné varovné hlášení potlačit.

Druhý typ zbytečného uzamčení zdroje, který by bylo možno detekovat a vhodné odstranit z kódu, je jakékoliv vícenásobné uzamčení téhož zdroje jedním procesem. S výjimkou uzamčení zdroje pro zápis již uzamčeného pro čtení se tím nijak nezvýší zabezpečení ani fakticky nezmění stav programu. Takovéto vícenásobné zamčení vede jenom k zbytečnému zvýšení zátěže systému i výpočetního výkonu.

Posledním typem zbytečného uzamčení může být uzamčení zdroje pro zápis, i když z jeho proměnných je v kritické sekci jen čteno.

K detekování všech těchto typů zbytečného uzamykání zdrojů je třeba provádět analýzu kódu uvnitř kritické sekce, tedy od systému částečně převzít zodpovědnost a řízení procesu makroexpanze.

5.3. Rozšíření ochrany na složená data

Dalším možným směrem rozšíření systému je jeho adaptace na ochranu i nejjednodušších (složených) dat. Současný systém totiž není schopen ochránit proti zápisu uvnitř kritické sekce pro čtení. Například pokud by ve zdroji `a` v proměnné `.a` byl uložen jednoduchý seznam `(1 2)`, pak `(with-resources ((a) t) (setf (car .a) 3))` postupně vede na vyhodnocení výrazu `(setf (car (val #.a '.a)) 3)`, v prostředí, kde symbol `#.a` se vyhodnotí na seznam `(1 2)` (viz makra `with-resources` a `with-variables-r`).

Vyhodnocení výrazu pak nejen, že nevyvolá chybu, ale skutečně způsobí zápis do chráněné proměnné – změnu seznamu (1 2) v .a na (3 2). Tyto problémy se složenými daty lze teoreticky odstranit zrušením (nepoužíváním) kritické sekce pro zápis.

Další problém složených dat je však v tom, že je možno odkaz na takováto data či jejich část (např. podseznam) uložit do jiné proměnné. Změnou hodnoty této proměnné by pak bylo možno měnit hodnotu v chráněné proměnné mimo odpovídající kritickou sekci.

Řešení tohoto problému je zřejmě netriviální a vyžadovalo by mnoho nových funkcionalit a kontrol zápisu.

Závěr

Cílem práce bylo vytvořit programátorské nástroje zjednodušující uživateli práci při tvorbě paralelního programu. Vytvořený systém nabízí při použití dvou jednoduchých maker nejen kompletní řešení kritické sekce pro libovolný počet sdílených proměnných a vláken, ale zároveň i kontroluje správnost použití těchto nástrojů. Při správném použití systém znemožňuje jakoukoliv nežádoucí interferenci na chráněných proměnných.

Dále jsou diskutovány další možnosti automatizace ověřování správnosti paralelního programu s pomocí rozšíření tohoto systému.

Cíl práce byl tímto splněn.

Conclusions

The aim of this Bachelor thesis was to create tools for programmer that makes the work of its user easier. The usage of the created system offers not only complete solution of the critical section for any number shared variables and threat but also it controls if the tools are used correctly.

When the system is used in the right way it does not allow any unwanted interferences on the protected variables.

Furthermore are discussed other possibilities of an automatization of the verifying the correctness of parallel program with the help of an extension of this system. So the aim of the thesis was fulfilled.

Reference

- [1] GRAHAM, Paul. *On Lisp : Advanced Techniques for comon Lisp*.
Prentice Hall, 1993. 432 s.
Dostupné na: www.paulgraham.com/onlisp.html.
ISBN 0130305529.
- [2] OWICKY, Suzan; GRIES, David.
Verifying Properties of Parallel Programs : An Axiomatic Approach.
Communications of the ACM. 1976, 19, 5, s. 279-285.
- [3] Common Lisp HyperSpec [online].
Cambridge : LispWorks Ltd. 1996, 2005 [cit. 2010-05-25].
Dostupné na:
www.lispworks.com/documentation/hyperspec/front/index.htm
- [4] LispWorks User Guide and Reference Manual [online]. v 6.0.
Cambridge : LispWorks Ltd., December 2009 [cit. 2010-05-25].
Dostupné na: www.lispworks.com/documentation/index.html

F. Kód programu

Zde přidávám kód programu v tištěné podobě. První je text souboru load.lisp, který nejdříve nahraje a zkompileje soubor MPS-package a poté soubor MPS.lisp. V tomto souboru je v komentáři řádek kódu, který pokud přidáte na začátek libovolného souboru, budete v něm moci používat balíček MPS za předpokladu, že soubory load.lisp, MPS-package.lisp a MPS.lisp jsou umístěny ve stejném adresáři jako daný soubor.

F.1. Loader

soubor load.lisp

```
(let ((file (or *load-pathname* *compile-file-pathname*)))
  (load (merge-pathnames "MPS-package.lisp" file))
  (load (merge-pathnames "MPS" file)))

#|
; If you put next line on begining of your code and copy
files load.lisp, MPS-package.lisp and MPS.lisp into same
folder as your code is stored, you could use MPS package.
(load (merge-pathnames "load.lisp"
  (or *load-pathname* *compile-file-pathname*)))
|#
```

F.2. Definice balíčku

soubor MPS-package.lisp

```
(defpackage MP-secured
  (:nicknames "MPS" "mps")
  (:documentation "Package build over MP package.
                   Contains various tools for
                   secured parallel programing.")
  (:use "MP")
  (:export defres delres resourcep with-resources)
  (:add-use-defaults t))
```

F.3. Hlavní kód

soubor MPS.lisp

```
(in-package "MPS")
```

```

(defclass resource ()
  ((lock :reader get-lock
        :initform (mp:make-lock :importantp t)
        :initarg :lock
        :type 'lock
        :documentation "the main lock of resource")
   (vars :accessor get-variables
        :initarg :variables
        :initform '()
        :type 'list
        :documentation "the list of variables
                        asociated with the resource")
   (check-lock :reader get-check
               :initform (mp:make-lock)
               :documentation "second lock to support
                               multiple reading")
   (counter :accessor get-count
            :initform 0
            :type 'number
            :documentation "the counter of processes
                            reading the resource")
   (readers :reader get-readers
            :initform (make-hash-table)
            :documentation "the hashtable from processes
                            currently reading resource to number
                            of times each process does it"))
  (:documentation "Basic resource class, that takes care about
                  the integrity of its data"))

(defconstant *resources* (make-hash-table)
  "the hashtable of resources")

(defconstant *r-vars* (make-hash-table)
  "the hashtable of protected variables contained in resources")

(defmethod check-in ((res resource))
  "increments the internal counter of processes
   that reads the resource"
  (let ((check-lock (get-check res)))
    (mp:process-lock check-lock
                      "checking in")
    (let ((process mp:*current-process*)
          (readers (get-readers res)))

```

```

        (unless (gethash process readers)
          (setf (gethash process readers)
                0))
        (incf (gethash process readers))
        (incf (get-count res)))
      (mp:process-unlock check-lock)))

(defmethod check-out ((res resource))
  "decrements the internal counter of processes
   that reads the resource"
  (let ((check-lock (get-check res)))
    (mp:process-lock check-lock
                      "checking out")
    (let ((process mp:*current-process*)
          (readers (get-readers res)))
      (when (zerop
              (decf (gethash process readers)))
        (remhash process readers))
      (decf (get-count res)))
      (mp:process-unlock check-lock)))

(defmethod lock-r ((res resource))
  "locks the resource for reading"
  (let ((lock (get-lock res)))
    (mp:process-lock lock
                      (format nil "waiting for reading ~S"
                              (mp:lock-name lock)))

    (check-in res)
    (mp:process-unlock lock)))

(defmethod lock-rw ((res resource))
  "locks the resource for reading and writing
   and waits till all reading processes check-out"
  (let ((lock (get-lock res)))
    (mp:process-lock lock
                      (format nil "waiting for
                                reading and writing ~S"
                              (mp:lock-name lock))))

  (let ((number (gethash mp:*current-process*
                          (get-readers res))))

    (if number
        (mp:process-wait "waiting for readers to check-out"
                          (lambda ()

```



```

                                (= (get-count res) number)))
    (mp:process-wait "waiting for readers to check-out"
      (lambda ()
        (zerop (get-count res))))))

(defmethod unlock-r ((res resource))
  "unlocks the resource for reading"
  (check-out res))

(defmethod unlock-rw ((res resource))
  "unlocks the resource for reading and writing"
  (mp:process-unlock (get-lock res)))

(defmethod clear-res ((res resource))
  "deletes resource's variables from the system"
  (dolist (r (get-variables res))
    (remhash r *r-vars*))
  (setf (get-variables res) nil))

(defun ensure-unique-var (var)
  "ensures if there is not another
  protected variable with the same name"
  (not (second (multiple-value-list (gethash var *r-vars*)))))

(defun check-vars (vars name)
  "ensures if the variables' names are unique among the system"
  (dolist (var vars)
    (unless (symbolp var)
      (error "protected variable must be a symbol, got ~S
              as a variable name when creating a resource ~S"
              var name))
    (when (member var (cdr (member var vars)))
      (error "symbol ~S appears multiply among variables
              while defining resource ~S"
              var name))
    (unless (ensure-unique-var var)
      (error "Existing variable name ~S" var))))

(defun check-and-set-pairs (pairs name)
  "checks the suitability of given protected variables and,
  if they are acceptable adds them into the system"
  (check-vars (mapcar #'first pairs) name)
  (loop for (var val) in pairs do

```

```

        (setf (gethash var *r-vars*) val)))

(defun define-resource (name pairs)
  "Crates, verifies and stores the resource"
  (check-and-set-pairs pairs name)
  (setf (gethash name *resources*)
        (make-instance 'resource
                        :variables (mapcar #'first pairs)
                        :lock (mp:make-lock
                                :name (format nil "resource ~S" name)
                                :important-p t)))
  name)

(defun redefine-resource (name pairs)
  "redefines contents of the existing resource"
  (let ((res (gethash name *resources*)))
    (lock-rw res)
    (clear-res res)
    (unwind-protect
      (progn
        (check-and-set-pairs pairs name)
        (setf (get-variables res)
              (mapcar #'first pairs))
        (unlock-rw res)))
    name)

(defun trans-pairs (pairs)
  (cons 'list (mapcar (lambda (pair)
                        (if (consp pair)
                            '(list ',(first pair)
                                   ',(second pair))
                            '(list ',pair nil)))
                    pairs)))

#|
(trans-pairs '((a 1) b (c (list a b))))
=> ((LIST (QUOTE A) 1) (LIST (QUOTE B) NIL)
    (LIST (QUOTE C) (LIST A B)))
|#

(defun resourcep (res)
  "checks if there is a resource of
  the given name in the system"

```

```

(second (multiple-value-list (gethash res *resources*))))

(defmacro defres (name &rest pairs)
  "defines or redefines the resource,
   depends if it already exists"
  (unless (symbolp name)
    (error "First argument of defres must be a symbol.
           Got ~S."
           name))
  (if (resourcep name)
      `(redefine-resource ',name ,(trans-pairs pairs))
      `(define-resource ',name ,(trans-pairs pairs))))

(defmacro delres (name)
  "removes the resource from the system"
  (unless (resourcep name)
    (error "First argument of delres must be a name
           of a resource. Got ~S."
           name))
  `(let ((res (gethash ',name *resources*)))
    (lock-rw res)
    (clear-res res)
    (remhash ',name *resources*)
    (unlock-rw res)))

(defmacro set-val (v var x)
  (error "Trying to change value ~S of the protected variable ~S
         to ~S in a read critical section"
         v var x))

(defun val (v var)
  (declare (ignore var))
  v)

(defsetf val set-val)

(defmacro with-vars-r (vars &body body)
  "executes the body with given protected
   variables accesible for reading"
  (let* ((g-syms (mapcar (lambda (var)
                           (make-symbol
                            (format nil "~S" var)))
                          vars)))
    body))

```

```

        (vals (mapcar (lambda (var)
                        '(gethash ',var *r-vars*))
                      vars))
        (syms (mapcar (lambda (var sym)
                        '(,var (val ,sym ',var)))
                      vars g-syms))
        (pairs (mapcar (lambda (sym val)
                        '(,sym ,val))
                     g-syms vals)))
    '(let ,pairs
        (symbol-macrolet ,syms
          ,@body))))

#|
(with-vars-r (a b c) (print a) (setf a 1) (print a))
==
(LET ((#:A (GETHASH 'A *R-VARS*))
      (:B (GETHASH 'B *R-VARS*))
      (:C (GETHASH 'C *R-VARS*)))
      (SYMBOL-MACROLET ((A (VAL #:A 'A))
                        (B (VAL #:B 'B))
                        (C (VAL #:C 'C)))
        (PRINT A) (SETF A 1) (PRINT A)))

=>error
|#

(defun make-var-val-pairs (vars)
  (mapcar #'list
    vars
    (mapcar (lambda (v)
              '(gethash ',v *r-vars*))
            vars)))

#|
(make-var-val-pairs '(a b c))
((A (GETHASH (QUOTE A) *R-VARS*))
 (B (GETHASH (QUOTE B) *R-VARS*))
 (C (GETHASH (QUOTE C) *R-VARS*)))
|#

(defmacro with-vars-rw (vars &body body)
  "executes the body with given protected variables

```

```

    accessible for reading and writing"
(let ((result (gensym)))
  '(let ((,result nil)
        ,@(make-var-val-pairs vars))
    (setf ,result (progn ,@body))
    (loop for var in ',vars
          for val in ,(cons 'list vars) do
            (setf (gethash var *r-vars*) val))
    ,result)))
#|
(with-vars-rw (a b c) (incf a) (print a) (incf a))
==(LET ((#:G1071 NIL)
      (A (GETHASH 'A *R-VARS*))
      (B (GETHASH 'B *R-VARS*))
      (C (GETHASH 'C *R-VARS*)))
  (SETF #:G1071 (PROGN (INCF A) (PRINT A) (INCF A)))
  (LOOP FOR VAR IN '(A B C)
        FOR VAL IN (LIST A B C) DO
          (SETF (GETHASH VAR *R-VARS*) VAL))
  #:G1071)
|#

(defmacro with-resources
  ((resources &optional (read nil)) &body body)
  "checks the resources, sorts them,
  and executes the body while holding them."
  (dolist (r resources)
    (unless (resourcep r)
      (error "~S is not registered resource" r))
    (when (member r (cdr (member r resources)))
      (error "resource ~S appears multiply
              among resources" r)))
  (let* ((result (gensym))
        (ress (mapcar
                 (lambda (r)
                   (gethash r *resources*))
                 (sort resources #'string< :key #'string)))
        (vars (apply #'append
                      (mapcar #'get-variables ress))))
    (read-arg read))
  '(let ((,result nil))
    (dolist (r ',ress)
      (,(if read-arg 'lock-r 'lock-rw)

```

```

    r))
(unwind-protect (setf ,result
                      (,(if read-arg 'with-vars-r
                                'with-vars-rw)
                        ,vars
                        ,@body))
  (dolist (r ',ress)
    (,(if read-arg 'unlock-r 'unlock-rw)
      r)))
,result)))

```

G. Příklady

V této sekci jsou textové podoby kódů jednotlivých spustitelných příkladů. Nejdříve obyčejných čtenářů a písáře, poté praktické paralelní verze funkce `format` a nakonec elegantnějších čtenářů a písářů využívajících této funkce.

G.1. Paralelní format

soubor `simple-ex.lisp`

```
(load (merge-pathnames "load.lisp"
                        (or *load-pathname*
                            *compile-file-pathname*)))

(mps:defres format-output (.out *standard-output*))

(defun p-format (destination control-string &rest args)
  (case destination
    ((nil) (apply #'format
                  (cons nil (cons control-string args)))))
    ((t) (mps:with-resources ((format-output))
          (apply #'format
                  (cons .out (cons control-string args)))))
    (otherwise (mps:with-resources ((format-output))
          (apply #'format (cons destination
                                (cons control-string
                                      args)))))))
```

G.2. Čtenáři a písář

soubor `ctenari.lisp`

```
(load (merge-pathnames "simple-ex.lisp"
                        (or *load-pathname*
                            *compile-file-pathname*)))

(mps:defres data (.a 0)(.b 0)(.c 0)(.d 0)(.e 0))

(defun data-reader (number)
  (p-format t "reader ~S came~%" number)
  (loop for i below 5 do
    (mps:with-resources ((data) t)
      (p-format t "R~S: data - ~S|~S|~S|~S ~%"
                  number .a .b .c .d .e))
    (sleep (1+ (random number)))))
```

```

(p-format t "reader ~S done ~%" number))

(defun data-writer ()
  (p-format t "writer came~%")
  (loop for i from 1 to 5 do
    (mps:with-resources ((data))
      (setf .a i)
      (p-format t "A> ~S~%" .a)
      (sleep 5)
      (incf .b)
      (incf .c)
      (incf .d)
      (incf .e))
    (sleep 3))
  (p-format t "writer done ~%"))

(defun super-writer (output)
  (format output "SUPER")
  (mps:defres data (.a 100)(.b 100)(.c 100)
    (.d 100)(.e 100) (.f 100)))

#|
(progn
  (p-format t "~%START!~%")
  (loop for number from 5 downto 1 do
    (mp:process-run-function (format nil
                                     "Reader ~S" number)
                             '(:priority 500)
                             #'data-reader2
                             number))

    (mp:process-run-function "Writer"
                             '(:priority 500)
                             #'data-writer2)

    (mp:process-run-function "SUPER-Writer"
                             '(:priority 500)
                             #'super-writer
                             out)

  nil)
|#

```


H. Reference Manual

H.1. defres

Macro

Syntax:

`defres name[{variable | (variable value)}*] ⇒ name`

Arguments and Values:

name symbol.
variable A symbol.
value Any object, evaluated.

Description:

Defines *name* as a name of the resource and supplied *variables* (if any) as the names of its protected variables. If there is already a resource of the given *name*, its protected variables' names are freed and resource is rewritten without warning.

If there is a protected variable of a given name already in an different resource, error is signaled.

A value of the protected variable is set to a given value, if no value is given, the value of the protected variable is set to **nil**.

If error occurs during the evaluation, no resource or protected variables are created. But if an error occurs while redefining resource, it still exists and can be treated as normally (locked etc.), but it has no variables (nor old neither new).

H.2. delres

Macro

Syntax:

`delresname => result`

Arguments and Values:

name a symbol
result a boolean

Description:

Macro `delres` safely removes the resource of the given *name* and its variables from the system. The resource cannot be locked or claimed since now, except it would be defined again.

The function returns **t** if the resource is completely released this way, **nil** otherwise.

H.3. resourcep

Function

Syntax:

$\text{resourcep} \text{res} \Rightarrow \text{result}$

Arguments and Values:

res a symbol

result a boolean

Description:

This predicate tests if *res* is a name for currently defined resource.

H.4. with-resources

Macro

Syntax:

$\text{with-resources}((\text{resource}^*)[\text{read}])\&\text{body} \text{body} \Rightarrow \text{result}$

Arguments and Values:

resource A name of the resource.

read Generalized boolean, optional.

body The forms to execute.

result The result of executing *body*.

Description:

It executes a body of the code while holding the resources. Within the body, all protected variables of the hold resources are accessible. The value of the body is returned normally.

If any resource could not be claimed process is blocked until that resource is available. The order in which resources are written does not matter, they are always claimed in the same fixed order. If a name of a resource occurs twice among the resources, error is signaled.

If the argument *read* is specified for true, resources are claimed just for reading and could be parallel claimed with another process but also just for reading. In this case no changes could be done on protected variables during the execution of *body*. If there is any **setf** form on a protected variable in the code of a critical read section, an error is signaled during the compilation.

I. Obsah přiloženého CD

Zde je uveden stručný popis obsahu přiloženého CD.

I.1. Povinný obsah

`bin/`

Z důvodu povahy práce je tato složka prázdná. Výsledkem práce totiž není žádný samostatný spustitelný program, ale package MPS upravujících kompilátor. Ten lze nalézt ve složce `src`.

`doc/`

Dokumentace práce ve formátu PDF, vytvořená dle závazného stylu KI PřF pro diplomové práce, včetně všech příloh, a všechny soubory nutné pro bezproblémové vygenerování PDF souboru dokumentace (v ZIP archivu), tj. zdrojový text dokumentace, vložené obrázky, apod.

`src/`

Kompletní zdrojové kódy package MP-SECURED.

`readme.txt`

Instrukce pro instalaci a použití balíku MPS.

I.2. Přidaný obsah

Navíc CD/DVD obsahuje:

`data/`

Obsahuje kód ukázkových příkladů použití.

`install/`

Loader `load.lisp` slouží k jednoduchému zavedení MPS do programu. Více viz soubor `README.txt` na CD či sekce [3.2.1.](#) programu