KOMPILAČNÍ, TESTOVACÍ A DIAGNOSTICKÉ PROSTŘEDKY V INTERAKTIVNÍM PROSTŘEDÍ

Ing. Z. Rusín

1. Scénář interaktivní práce

Smyslem příspěvku je stanovit požadavky na softwareové prostředí interaktivní práce rutinního programátora a nalézt takovou koncepci manipulace s exekuovatelnými objekty v rámci hostitelského operačního systému, jež bude interaktivní programátorskou činnost patřičně stimulovat.

Vyjděme z obecně akceptovatelného scénáře zahrnujícího interaktivní pořízení zdrojových textů programů a testovacích dat, interaktivní kompilaci a interaktivní testování uživatelského kódu a stanovme programátorovy požadavky, konfrontovatelné s hledisky rozumné odezvy v interaktivní úloze, celkové výkonnosti systému, únosné míry tiskových výstupů a podobně.

Interaktivní pořizování a opravování textů musí umožňovat vstup podobných či zcela totožných vět s minimální pracností stejně jako manipulace s větším rozsahem textu v podobě stránky totožné s obsahem obrazovky koncového interaktivního zařízení. První požadavek bývá splněn, druhý zato předpokládá patřičné technické vybavení pro manipulaci se znakovým obsahem obrazovky. Po dobu programátorovy práce se stránkou textu nevyžaduje přísluŝný interaktivní proces přístup k procesoru řídícího kódu a je-li to nutné, může uvolnit i přidělenou část operační paměti jiným na procesoru aktivním úlohám. Může tedy být tato forma manipulace s texty preferována tam, kde je současně dosaženo patřičné kvality komunikací, dostatečné přenosové rychlosti a vysoké odolnosti výpočetního systému jako celku. Je-li programátor v důsledku špatného stavu komunikací či malé odolnosti systému nucen interaktivní práci drobit do malých časově nenáročných úseků, vzniká těžko odstranitelný návyk, snižující po změně v systému povětšině nejen výkonnost jednotlivce, ale výpočetního systému jako takového.

Diskuse na téma interaktivní kompilace nevede zdaleka k natolik jednoznačným závěrům jako předchozí odstavec. Nelze se vyhnout tématům z konstrukce kompilátorů jako interpretace zdrojového textu s okamžitým interaktivním zásahem programátora versus klasická lexikální a syntaktická analysa s následným výstupem chyb a opravou zdrojového textu, způsoby zotavení kompilace po chybách v obou případech atd. Vyslovíme proto svůj názor na rutinní interaktivní kompilace

profesionálního programátora, opírající se o víceleté dennodenní zkušenosti s rutinním programováním v interaktivním prostředí.

Prakticky všechny uživatelské projekty jsou realisovány modulární technikou. Pro testování celků je zde sotva myslitelná interpretace zdrojového textu v interaktivním prostředí. Ta se hodí snad pro ladění jednotlivého modulu a především pro jednorázové výpočty podle jednoznačných a praxí prověřených algoritmů, což zase málokdy tvoří náplň práce profesionálního programátora. Dáváme proto přednost klasické syntaktické analyse zdrojového textu v modulárním programování s rozsahově rozumným výstupem chybových zpráv na obrazovku koncového zařízení, s následným interaktivním editováním textu a opakovaným vstupem do kompilátoru. Je-li syntaxe v pořádku, nechť kompilace pokračuje podle programátorovy volby s výstupem buď stálého nebo dočasného cílového modulu, jenž může být okamžitě testován. Je výhodné, aby pro testovací účely nebylo nutné slinkovávat uživatelský program, ale aby zaváděcí program kaskádoval všechny požadované cílové moduly automaticky na základě externích referencí v modulech. Nechť má program-átor možnost při kompilaci určit, zda je modul vytvářen pro interaktivní testování, což v kladném případě cílový kód rozšíří o testovací sekvence. Požadujme ať i tento kód je exekuovatelný i v dávkovém prostředí.

Tím dáváme výrobci software možnost vytvořit jediný kompilátor daného vyššího jazyka, fungující stejně v dávkovém i interaktivním prostředí, programátorovi pak jediný způsob práce s programy v daném jazyce, v interaktivním prostředí rozšiřitelný o interaktivní testování. Bude-li převážná část prací skutečně provozována interaktivně, lze účinně snížit rozsah tiskových výstupů.

Než se věnujeme specifikaci požadavků na interaktivní testování a post-mortem diagnostiku programových chyb, zkoumejme obecné vlastnosti potřebného interaktivního prostředí.

2. Interaktivní programátorské prostředí

Uvažme případ procesu obsluhujícího několik desítek koncových interaktivních zařízení následujícím způsobem: řídící proces přijímá zprávu, analysuje ji a platný požadavek postoupí spolu s koncovým zařízením procesu výkonnému, jichž může být i pro tutéž činnost několik. Takovéto schema zpracování transakcí v reálném čase může být uspokojivé tehdy, jsou-li jednotlivé třídy požadavků v poměru součinu svých četností a spotřeby systémových zdrojů zhruba ekvivalentně obslouženy výkonnými procesy, které mají v celém systému zaručen dostatečný prostor /paměť, procesory řídícího kódu, diskové přenosy/,

a je-li úplný soubor vstupních zpráv natolik dobře formalizovaný, že analysa zpráv se nestává úzkým místem zpracování.

Konfrontujme toto schema s očekávanými požadavky interaktivního kompilačně - testovacího cyklu.

Nemá-li být programátor přinucen k trivialitě několika rutinních operací, které budou v podstatě opisem děrnoštítkového dávkového ladění, musí mít k disposici dostatečně mocný řídící jazyk, jímž bude nárokovat, přiřazovat, využívat a uvolňovat systémové zdroje v dynamickém toku interaktivní práce. Zůstaneme-li u reálné skutečnost, že převážná část interaktivních vstupů je realisována znakově, potřebujeme na této úrovni procedurální jazyk pascalovského typu, jehož blokova struktura bude využita právě k dynamické alokaci a uvolňování systémových zdrojů. Zdá se nám samozřejmé, že by pak koncové zařízení mělo být trvale spojeno s výkonným procesem, vněmž bude probíhat interaktivní interpretace příkazů řídícího jazyka, jejímž výsledkem bude ošetření parametrů a exekuce uživatelské či systémové procedury a signalisace výsledku uživateli - programátorovi. Uvědomíme-li si, že konečným výsledkem programátorovy práce není holý programový produkt neschopný uživatelské exekuce ani v dávkovém ani v interaktivním prostředí, ale naopak, že už samotný projekt by měl vycházet ze strukturovaného návrhu řídící procedury uživatelských programů, realisované v jazyce řízení systému a laděné a testované programátorem v rámci práce na daném projektu, je přiřazení jednoho procesu jednomu interaktivnímu programátorskému koncovému zařízení snad evidentní.

Zkoumejme důsledky tohoto závěru: především musí být interaktivní programátorská úloha traktována v systému týmž způsobem co procesy dávkové. Má-li ale současně existovat mnoho takovýchto úloh, je nezbytné minimalisovat jejich paměťové nároky tím, že veškerý obslužný kód, včetně kompilátorů a software vyšších jazyků, musí být globální a tudíž nutně reentrantní. Považujme jej tedy jednoduše za nedílnou součást supervisoru. Je zřejmé, že na totéž místo v konkrétním systému aspiruje každý globálně používaný produkt uživatelský. Tím jsou definovány dva požadavky pro manipulaci s exekuovatelnými objekty v rámci operačního systému:

1. Řídící jazyk musí stejným způsobem nakládat s volatelnými objekty uživatelskými i systémovými.
2. Cílový modul má mít odděleny oblasti s neměnným obsahem od oblastí přepisovatelných. Tyto oblasti mohou pak být globální.

Rozvinutím důsledků těchto dvou postulátů dojdeme k návrhu tvaru cílového modulu, jenž umožňuje jednak specifikovat jistou část pravi-

del řídícího jazyka systému, jednak jednotný systém kompilátorů, post mortem diagnostiky a interaktivního testování.

3. Cílový modul a exekuovatelné objekty

Cílový modul musí, jak známo, kromě inicialisačních oblastí kódu a dat obsahovat řídící informaci pro zaváděcí program. Uspořádejme nově cílový modul takto: je to seriově zpracovávaný diskový soubor se čtyřmi typy vět - větami popisujícími vlastnosti datových a kódových oblastí modulem zřizovaných a vlastností všech interních a externích pojmenovaných objektů zaváděcímu programu viditelných; větami adresní části modulu, modifikované zaváděcím programem; větami inicialisujícími kód a data a větami s informací pro post mortem diagnostiku a interaktivní testovací systém, jež jsou čteny a zpracovávány až těmito systémy mimo okamžik zavedení modulu do virtuální paměti.

Prvý typ vět, tzv. properties rekordy, popisují modul v pojmech kódových a datových oblastí virtuální paměti a jejich vlastností jako přístupnost ke psaní nebo k exekuci kódu, typ ochrany paměti, již zmíněná globálnost a další, dále obsahují popis externích referencí modulu a popis viditelných objektů v modulu jako common_data či entry points. Je-li modul zaváděn globálně, týká se to jen reentrantního kódu a pouze ke čtení přístupných datových oblastí; ostatní data a adresní část tvoří lokální kopie ve všech procesech, které modul užívají. Oddělení adresní části od kódu je zřejmou podmínkou globálnosti kódu. Detailní uspořádání modulu není pro naše účely podstatné.

Výše v textu jsme požadovali, aby řídící jazyk systému umožňoval volání kterékoli procedury systémové i uživatelské jednotným způsobem, patrně v podobě zápisu seznamu aktuálních parametrů procedury za jménem volaného objektu. Připustíme-li, že tento způsob odpovídá právě realisaci volací sekvence na úrovni primitivních instrukcí representovaných patrně instrukcemi assemblerového jazyka, vedou naše úvahy k eliminaci důvodů pro existenci primitivní instrukce SUPERVISOR_CALL / SVC /, která se stává nadbytečným archaismem tupě svazujícím reálný repertoár systémových funkcí do z hlediska uživatelského kódování naprosto neadekvátního způsobu volání. Připustíme-li dále, že veškeré kódování procedur operačního systému lze realisovat v systémově orientovaném vyšším jazyku, jehož přirozenou podmnožinou je právě jazyk řízení systému, a že totéž tím spíše můžeme tvrdit o programování uživatelském, eliminujeme i potřebu jakéhokoli assemblerového jazyka v rutinním programování vůbec.

Není smyslem tohoto článku specifikovat podrobněji předpokládaný

systémově orientovaný vyšší jazyk, nicméně využijeme dále v textu skutečnosti, že výpočetní systém hostí pouze software skupiny více či méně si podobných vyšších jazyků.

Od volání procedury se seznamem aktuálních parametrů je jen krůček k požadavku opatřit exekuovatelný objekt šablonou či maskou klíčových slov, parametrových typů a předvolených standardních hodnot parametrů, jež se stává součástí popisu vlastností objektu v properties rekordech cílového modulu. Interpretační systém řídícího jazyka pak má možnost provádět validaci aktuálních parametrů jakožto objektů řídícího jazyka, dává možnost volání objektu s neúplným seznamem parametrů a v interaktivním prostředí lze realisovat volání ve více krocích se zobrazením volitelné 'help' informace o volané proceduře nebo jejich jednotlivých parametrech v podobě nápovědy /screen_prompting/ se šablonou parametrů se standardními hodnotami, jejíž zaslání po modifikaci dosazením aktuálních hodnot parametrů tvoří aktuální volání daného objektu. Podmínkou je zde opět existence patřičných technických prostředků pro manipulaci se znakovým obsahem obrazovky stejně jako potřebné funkce interpretátoru řídícího jazyka a jeho svázanost se zaváděcím programem. Zmíněná 'help' informace může být součástí diagnostických vět cílového modulu, může ale též být obhospodařována databázově. Popsaná možnost nápovědy, jež má didaktický význam pro uživatele málo obeznámeného se systémem nebo pro případy méně frekventovaných či komplikovaných systémových nebo uživatelských prostředků, je samozřejmě jednou /a nepovinnou/ možností vyvolání exekuovatelného objektu v rámci řídícího jazyka. Závěrem povolme ještě existenci zkratek jmen či synonym všech volatelných objektů a vzhledem k frekventovanému užití některých procedur psaných v řídícím jazyce systému požadujme i kompilaci řídícího jazyka.

Šablona parametrů samozřejmě překračuje definiční rámec většiny vyšších jazyků. Lze ji tedy realisovat buďto jako extensi zdrojového jazyka na úrovni direktiv kompilátoru, nebo, patrně přirozeněji, samostatným prostředkem pro manipulaci s cílovými moduly.

4. Koncepce jednotného kompilačního okolí

Analysou funkcí kompilátoru obdržíme činnosti, které jsou pro všechny jazyky totožné, např. vstup zdrojového textu, hlášení syntaktických chyb, výstupy listáží a cílového modulu. Není obtížné stanovit schema umožňující maximálně využívat centrálního globálního kódu. Možná koncepce vypadá takto: pro každý jazyk existuje vstupní kompilační procedura, která validuje na jazyku závislé parametry kompilace a os-

tatní parametry postupuje globálním kompilačním okolí spolu s vhodnou referencí na vlastní kompilátor jazyka. Kompilační okolí validuje jazykům společné parametry, inicíalisuje subsystémy vstupu zdrojového textu, hlášení chyb a výstupu listáží, vytváří extensibilní pracovní datové oblasti a vyvolává kompilátor jazyka, který používá globální I/O systémy a datové oblasti. Kompilátor provádí lexikální a syntaktickou analysu a produkuje mezikód, jenž je dále globálním kompilačním okolím zpracováván do cílového tvaru jednotně pro všechny jazyky. Veškeré výstupy vlastního kompilátoru jsou drženy ve virtuální paměti ve formě tabulek. Součástí těchto tabulek jsou properties rekordy budoucího cílového modulu a diagnostická informace. Vstupní procedura i vlastní kompilátor jsou samozřejmě opět reentrantními součástmi globálního kódu. Pro úspěch celé koncepce je rozhodujícím prvkem projekt jednotného mezikódu, který musí vyhovět specifikům jednotlivých jazyků a měl by být schopen rozšíření pro nově implementované jazyky. Součástí popisovaného systému musí být i diagnostika výkonnosti kompilačního systému, nezbytná pro ladění programátorského prostředí, zejména z hlediska paměťových nároků. Detaily jako je způsob nastavení rozsahu uživatelské diagnostiky v cílovém modulu či generování kódových sekvencí pro interaktivní testování se zabývat nebudeme.

5. Jednotný diagnostický a interaktivně testovací systém

Generuje-li nám jednotný kompilační systém cílové moduly jednotného formátu, musí být jednotná diagnostika samozřejmostí. Řekli jsme, že parametry kompilátoru byla nastavena úroveň uživatelské diagnostiky v cílovém modulu. Stanovme jako minimální požadavek identifikaci řádku zdrojového textu na němž dochází k chybě při exekuci a identifikaci chyby samotné srozumitelným otevřeným textem. Pro vyšší úroveň diagnostiky požadujme znakový /nikoli binární/ výpis hodnot objektů z uživatelského programu v pojmech zdrojového jazyka v přehledném ale úsporném uspořádání a stanovení historie volání jednotlivých modulů uživatelského programu spolu s jednoznačnou identifikací cílových modulů /např. jménem modulu a časem kompilace, což lze ověřit v kompilačním protokolu/.

Nyní zkoumejme, jak tuto diagnostiku zajistit. Není obtížné odhalit, že jde o dvě separátní záležitosti - jednak o analysu posloupnosti volání, evidentně nezávislou na zdrojovém jazyce, a jednak o výpis objektů z uživatelského programu, patrně na jazyku závislý. Prvý problém lze elegantně rozřešit tím, že globální diagnostický systém, inicíalisovaný v rámci reusabilního virtuálního prostředí spolu se zaváděcím systémem a interpretátorem řídícího jazyka, získává informaci

o aktuální uživatelské rutině pomocí adresy návratu do volajícího kódu, kterou porovnává s obsahem tabulek zaváděcího programu, jež ostatně musí být přístupny i interpretačnímu systému řídícího jazyka pro zpracování šablon parametrů. Tím jsme stanovili další požadavek na manipulaci s volatelnými objekty v rámci hostitelského operačního systému. Globální diagnostika tedy provede analysu posloupnosti volání a předá řízení jazykově orientované diagnostické proceduře, která zjistí příčinu chyby a provede výpis uživatelských objektů. Tato procedura tak může být součástí software příslušného vyššího jazyka a její reference může být opět získána z tabulek zaváděcího systému, bude-li součástí properties informace v cílovém modulu. Je-li nutné, aby pro daný jazyk proběhla předem inicialisace této chybové procedury pro daný modul, obstará to pohodlně obvyklá standardní volací sekvence generovaná kompilátorem jako prolog modulu či procedury. Analysu chyby přenášíme do jazykově orientované diagnostické a vypisovací procedury proto, že většina chyb je jazykově závislá - třeba chyby formátu dat. Pro nezávislost diagnostiky na stavu standardního I/O systému daného jazyka stanovme jako další požadavek užití nezávislého globálního diagnostického I/O systému pro uživatelsky orientované post-mortem výpisy.

Chybová procedura jazyka užije tedy informaci obsaženou v diagnostických větách cílového modulu. Formátem této informace se opět zabývat nebudeme. Do diagnostického systému se vstupuje buď v důsledku hardwareového přerušení /chyby typu dělení nulou/ nebo přímo ze software daného jazyka, např. pro zmíněné chyby formátu dat. Lze připustit i u živatelské rozhraní pro statické monitorování stavu programu.

Interaktivní testovací systém definujme obdobně. Pouze inicialisace testovacího systému nechť není automatická, nýbrž ať je prováděna programátorem podle potřeby. Důvodem jsou zde zvýšené paměťové nároky v interaktivním prostředí. Jediným parametrem uživatelského rozhraní může být identifikace jazyka, pro nějž je interaktivní testovací prostředí aktivováno. Inicialisace může obsahovat zavedení testovacího kódu dotyčného jazyka a úpravu programátorského interaktivního prostředí, např. zvýšení reálné operační paměti úloze přidělené. Další přípravné činnosti, tj. především zpracování diagnostických vět cílového modulu, přenecháme interaktivnímu testovacímu systému jako počáteční činnost po vstupu do modulu připraveného kompilátorem pro interaktivní testování. V opačném případě by musel být seznam testovaných modulů zadáván explicitně předem jako parametr uživatelského rozhraní. V naší verzi kompilátor generuje potřebné testovací sekvence odvolávající se na obsah diagnostických vět cílového modulu a navíc generuje

vstupní sekvenci pro každý volatelný objekt v modulu. Testovací kód je obdobou výše popisované chybové procedury a bude jazykově závislý. Požadujme opět jeho reentrantnost a globálnost.

Je zřejmé, že jde o principy známé z traceování uživatelského kódu. Odlišnost je v tom, že traceování nyní po vstupu do kompilátorem připraveného uživatelského kódu přejde do dialogu s programátorem, jenž dynamicky řídí rozsah i obsah traceování a navíc může měnit obsahy objektů užívaných v traceovaném kódu a zasahovat do toku řízení při průchodu testovaným kódem. Postulujme též samozřejmost 'help' funkce při manipulacích v rámci testovacího systému, která může být opět dělena na část jazykům společnou a část specifickou konkrétnímu jazyku.

6. Autorův závěr

V předchozích odstavcích jsme představili jistou ucelenou koncepci manipulace s exekuovatelnými objekty, jež na jedné straně dává možnost slušné uživatelské diagnostiky a interaktivního testování uživatelského kódu a na straně druhé vůbec nerozlišuje mezi kódováním hostitelského operačního systému, nadstavbou vyšších jazyků a kódováním uživatelským, jejichž vyvolávání obstarává řídící jazyk systému za užití interaktivní nápovědy, 'help' informace a standardních hodnot parametrů jednotným způsobem.

Ponecháváme pozornému čtenáři na posouzení, zda realisace námi popisovaných principů by byla přínosem v u nás převládajících podmínkách instalací systémů JSEP. Sami jsme ovšem přesvědčeni, že odpověď musí být kladná, s tím, že realisace obdobných zásad by měla být spojena s přechodem k jiné architektuře procesorů řídícího kódu, kterou si především rozvoj mikroprocesorové techniky přímo vynucuje. Doufáme totiž, že se nemýlíme, považujeme-li současné špičkové výpočetní systémy za strukturované sítě účelových mikroprocesorů, sloužící především interakcím v reálném čase, v nichž nikde nenajdeme šestnáct uživatelských registrů důmyslně spravovaných assemblerovým kódováním programu dávkové agendy.

Literatura: obecná problematika výpočetního systému, jenž může být hostitelem zde popisovaných uživatelsky orientovaných interaktivních systémů je obsahem článku /1/, konkrétní jevová stránka jedné možné implementace takových systémů je popisovaná ve statí /2/.

/1/ Výpočetní systémy příštích let a jejich dopad na profesní sféru,
Z. Rusín, sborník Programování 82

/2/ Zkušenosti z tvorby a ladění programů VTV pod operačním systémem čtvrté generace,
J. Kučera, sborník Programování 83