Od programu obvykle chceme, aby zabíral co nejméně místa a přitom aby běžel co nejrychleji. Toho lze, jak známo, dosáhnout jeho optimalizací a snad každý, kdo alespoň trochu přičichl k programování, vysype toto obecné tvrzení z rukávu. Horší už to bude s odpovědí na otázku, co to ta optimalizace vlastně je.

Dobrý, lepší, nejlepší

Než se do úvah o optimalizaci pustíme, rád bych upozornil, že i když je následující povídání založeno především na zkušenostech s jazyky C a C++, mnohé z jeho závěrů platí i obecně.

Velikost versus rychlost

Na první pohled by se mohlo zdát, že ze dvou programů, které řeší týž problém, bude ten menší také rychlejší. Dále si ukážeme, že to obecně neplatí, ale určitá souvislost tu přece jenom je, zejména u opravdu rozsáhlých programů.

Dnešní operační systémy pracují s tzv. virtuální pamětí, což je mechanismus, který umožňuje využívat část diskového prostoru jako operační paměť, tedy RAM. (Mechanismy pro podporu virtuální paměti jsou dnes zpravidla zabudovány přímo v procesoru. Mají je např. i procesory Intel 80386 a pozdější.) Program se tedy nemusí celý vejít do operační paměti, některé části (tzv. stránky) kódu nebo dat mohou být uloženy na disku. Program pak při běhu “swapuje”: při výpadku stránky (page fault), tj. požadavku na data nebo kód, který není v operační paměti, přečte operační systém potřebné stránky z disku a případně odloží zpět ty, které nepotřebuje, ale které se předtím změnily.

Přitom je ovšem podstatné, že **přístup na disk je o několik řádů pomalejší než přístup do operační paměti.** Proto čím větší je program, tím větší je i pravděpodobnost výpadku stránky, a tak se může jinak kvalitní program stát prakticky nepoužitelným, jestliže zabírá příliš mnoho místa, nebo jestliže i jinak “rozumně” veliký program spustíme na počítači s příliš malou operační pamětí.

Tři oblasti optimalizace

Zamyslíme-li se nad problémem optimalizace obecněji, zjistíme, že můžeme hovořit o třech oblastech, ve kterých se překladač a programátor v různém poměru dělí o kompetence. První oblast se týká **volby algoritmů** při řešení problému a spadá výlučně do pravomoci programátora. V této oblasti lze získat nebo ztratit zdaleka nejvíce. Druhá oblast zahrnuje **spolupráci programátora s překladačem**; jde o využití registrových proměnných, voleb pro optimalizaci atd. Do třetí oblasti pak lze započítat optimalizace založené na specifických **vlastnostech strojního kódu cílového počítače** – ty už patří výlučně do kompetence překladače.

Význam třetí oblasti – stejně jako druhé – je menší než význam oblasti první, to ale neznamená, že je vždy zanedbatelný. Uplatní se hlavně u dlouhých programů, u programů obsahujících cykly s velkým počtem opakování apod. Povídání o druhé oblasti však bude nejrozsáhlejší, a proto si je necháme až na konec.

Optimalizace nejvyšší úrovně

Zeptáte-li se programátora, proč je jeho program tak pomalý, často uslyšíte nějaký povzdech na téma nedokonalého překladače, který vůbec neumí optimalizovat, nedokonalého operačního systému a bůhví čeho ještě. Málokdy uslyšíte o nedokonalém programátorovi, který nedokáže vybrat optimální algoritmus – a přece právě to může být příčinou pomalého běhu programu. Zisky nebo ztráty v této oblasti jsou totiž obvykle zdaleka nejvýraznější.

Typickým příkladem mohou být algoritmy pro **třídění polí**, tj. pro přerovnání jejich prvků podle velikosti. Ze základního kurzu programování víme, že složitost algoritmu třídění haldou *(heap sort)* nebo rychlého třídění *(quick sort)* je v případě pole o **n** prvcích rovna hodnotě **O(n*_ln_ n)**, tzn. je úměrná číslu **n*_ln_ n**, zatímco složitost jednodušších metod, jako je třídění vkládáním *(insertion sort)*, je $\mathbf{O(n^2)}$. Věc se tedy zdá jasná: zapomeneme na jednoduché metody, které se sice snadno učí a snadno programují, ale jinak za mnoho nestojí, a budeme používat jen metody efektivní, byť složité.

Zdání ovšem může klamat. Pro třídění polí s malým počtem prvků mohou totiž být jednoduché metody výhodnější. Postavme si vedle sebe několik podobných (a zdánlivě ekvivalentních) úloh:

– setřídit jedno jediné celočíselné pole o 1 000 000 prvků,

– setřídit 10 celočíselných polí o 100 000 prvků,

– setřídit 100 celočíselných polí o 10 000 prvků

atd.

I když ve všech případech třídíme milion celých čísel, výsledky různých třídicích metod se budou výrazně lišit. Připojená tabulka ukazuje průměrné časy v sekundách potřebné pro tyto úlohy při třídění vkládáním a při třídění haldou. Vidíme, že pro velmi malá pole může být výhodnější třídění vkládáním (které představuje více jednoduchých operací), než třídění haldou, které znamená sice méně operací, ovšem podstatně složitějších.

Optimalizace nejnižší úrovně

Tato oblast optimalizace využívá specifik strojního kódu cílového procesoru. Mnohé konstrukce z vyšších programovacích jazyků lze přeložit několika způsoby. Efekt je stejný, tedy program udělá totéž, ovšem výsledný kód je podle okolností menší nebo rychlejší.

Jako příklad vezmeme procesory jedné z nejrozšířenějších značek – Intel. Chceme-li např. do proměnné *X* typu *int* uložit hodnotu 0, můžeme použít buď instrukci

```
MOV dword ptr[X],0
```

nebo

AND dword ptr[X],0

Výsledek, tedy stav dat, je v obou případech stejný, avšak první instrukce zabere 6 bajtů a trvá 3 takty, zatímco druhá zabere 9 bajtů a trvá 1 takt. Zde tedy stojíme před volbou **rychlost nebo velikost**, nemůžeme však mít obojí.

V tomto případě můžeme ušetřit na jedné instrukci 2 takty. Abychom si uvědomili souvislosti, připomeňme si, že procesor s taktovací frekvencí 100 MHz – ale tak "zastaralé" procesory se už snad ani nedají sehnat – provede za sekundu právě sto milionů taktů. Abychom tedy zrychlili běh programu o jedinou sekundu, musíme ušetřit stovky milionů taktů. To je možné, pokud program obsahuje například velmi rozsáhlé cykly.

Je asi jasné, že optimalizace na této úrovni je výlučnou záležitostí překladače – samozřejmě pokud nechceme programovat v asembleru a neustále porovnávat časovou a prostorovou náročnost kódu.

Střední úroveň optimalizace

Do této oblasti spadají optimalizační techniky, na kterých spolupracuje překladač s programátorem. Většinou to znamená, že programátor překladači pomocí voleb nebo klíčových slov naznačí, kterou z možností má překladač použít a co může překladač o programu předpokládat. Obvykle ale může zasáhnout programátor a provést odpovídající úpravy na úrovni zdrojového kódu sám.

Tyto techniky mohou, ale nemusí vést k cíli. Některé lze použít jen za jistých předpokladů, jiné mohou někdy dokonce situaci zhoršit. Často je třeba zkusmo určit, které z nich mají a které nemají v daném případě žádaný účinek. Podíváme se na některé z nejčastěji používaných způsobů.

Slučování cyklů

Pokud se v programu vyskytují dva cykly se stejným rozsahem a s nezávislým obsahem, je rozumné je sloučit do jednoho cyklu (*loop jamming*). Chceme-li např. vynulovat dvě pole *a* a *b* o *N* prvcích, můžeme napsat

```
for(int i = 0; i < N; i++) a[i] = 0;

for(int i = 0; i < N; i++) b[i] = 0;
```

ale rozumnější je sloučit oba cykly do jednoho,

```
for(int i = 0; i < N; i++)

{

a[i] = 0;

b[i] = 0;

}
```

neboť se tím ušetří "administrativa", tj. instrukce, které řídí opakování cyklu.

Práce s registry

Používání klíčového slova *register* patří k nejstarším optimalizačním možnostem, zavedeným již v jazyce C podle Kernighana a Ritchieho. Jistě si vzpomenete, že toto klíčové slovo má za úkol naznačit překladači, že právě deklarovanou proměnnou nebo parametr budeme dále často používat a že by se měl pokusit uložit ji do některého z registrů procesoru; přístup k registrům je totiž zpravidla podstatně rychlejší než přístup do paměti.

Dnes už je však toto klíčové slovo téměř zbytečné, neboť většina překladačů používá rafinované algoritmy, pomocí nichž určí, které proměnné je vhodné uložit do registrů (a obvykle to odhadne lépe než programátor). Překladače často dávají programátorovi na vybranou, zda chce, aby se řídily jeho požadavky, aby registrové proměnné nepoužívaly vůbec (ani tam, kde je to v programu předepsáno – to se hodí při ladění), nebo aby je překladač používal dle vlastního uvážení. Poslední možnost zpravidla vede k nejlepším výsledkům.

Vložené funkce

Používání vložených funkcí *(inline)* není (zatím) součástí jazyka C, najdeme je až v jazyce C++, ale ve skutečnosti jde o jeden z nejstarších optimalizačních triků, které mohl programátor používat. S jejich obdobou jsme se mohli setkat např. už v dávných verzích Fortranu (jednopříkazové funkce – *statement function*).

Volání běžné funkce znamená vždy jistou režii. Při volání je to uložení parametrů a návratové adresy do zásobníku, vytvoření rámce zásobníku a lokálních proměnných atd., při návratu mj. předání návratové hodnoty, odstranění lokálních proměnných a skutečných parametrů atd.

Vložené funkce, tj. funkce s modifikátorem *inline*, se sice po syntaktické stránce chovají jako funkce, ale překladač je používá podobně jako makra. To znamená, že na místo volání vloží tělo této funkce. To zpravidla vede k rozsáhlejšímu, ale rychlejšímu programu.

Modifikátor *inline* ovšem – podobně jako modifikátor *register* – není pro překladač závazný. To znamená, že překladač může usoudit, že danou funkci není vhodné překládat jako vloženou, a bude s ní zacházet jako s obyčejnou funkcí. Zapíšeme-li takovouto funkci do hlavičkového souboru, může se stát, že překladač vytvoří její tělo v každém ze souborů, ve kterém takovýto hlavičkový soubor použijeme. Pak použitím vložených funkcí nezískáme žádnou výhodu, pouze zvětšíme výsledný program. (To ovšem není typická situace.)

Překladače obvykle dávají na vybranou možnost klíčové slovo *inline* ignorovat; to se hodí opět zejména při ladění. Vedle toho nabízejí některé překladače také možnost překládat jako vložené některé z knihovních funkcí, které jsou jinak překládány “normálně” (obvykle se tato volba jmenuje *inline intrinsic functions*).

Poznamenejme ještě, že překladač může v případě potřeby použít funkci obojím způsobem – jako vloženou i jako normální.

Šíření konstant a kopií

Tato optimalizační technika je založena na skutečnosti, že použití konstant v instrukcích je rychlejší než práce s registry a práce s registry je rychlejší než práce s operační pamětí. Šíření konstant (*constant propagation*) lze použít v situaci, kdy do jedné proměnné přiřadíme konstantu a tuto hodnotu pak "pošleme dál", aniž bychom ji mezitím použili. To znamená, že pokud v programu napíšeme např.

a = 10; // (1)

b = a; // (2)

kde *a, b* jsou typu *int*, přepíše to překladač do tvaru

a = 10;

b = 10;

Přitom mezi (1) a (2) mohou být další příkazy, nesmějí ovšem pracovat s hodnotu uloženou do *a*. (Zde se tedy "rozšířila" konstanta 10.)

Šíření kopií (*copy propagation*) je podobné: Jestliže přiřadíme do jedné proměnné hodnotu a tuto hodnotu pak "pošleme dál", aniž bychom ji mezitím použili, může si překladač ponechat v registru její hodnotu a tak urychlit běh programu tím, že ušetří přístup do paměti. Jestliže tedy v programu napíšeme

a = x;

F(a);

kde *a* i *x* jsou stejného typu, přepíše to překladač do tvaru

a = x;

f(x);

Nepoužitý kód a nepoužitá paměť

Při optimalizaci (ale také vinou nepozornosti programátora, při rozvoji maker nebo šablon atd.) může nastat situace, že do proměnné uložíme hodnotu, kterou program nikdy nepoužije. Takové přiřazení lze z programu vypustit *(store elimination)*. Podívejme se na učebnicový příklad. Jestliže ve zdrojovém textu napíšeme

a = x;

f(a);

a = y;

a překladač použije techniku šíření kopií, vznikne

a = x; // (1)

f(x);

```
a = y;
```

Přiřazení hodnoty proměnné *a* je naprosto zbytečné, samozřejmě za předpokladu, že *a* není globální proměnná a funkce *f()* ji nepoužívá. To ale znamená, že překladač může příkaz (1) prostě odstranit a přepsat tento úsek programu do tvaru

```
f(x);

a = y;
```

Podobně se v programu může objevit nepoužitý (mrtvý) kód *(dead code)* – úsek programu, který nemůže být nikdy prováděn. Překladač ho může odstranit. Tím se sice program nezrychlí, ale zmenší, a to může za jistých okolností také vést ke zrychlení.

Společné podvýrazy

Tato optimalizace je podobná šíření kopií. Jestliže se na několika místech v programu objeví týž výraz nebo dílčí výraz, může překladač přepsat program tak, že se tento podvýraz vypočte jen jednou a jeho hodnota se uloží do pomocné proměnné. To znamená, že např. následující úsek programu

```
x = (a+b)*c;

y = sin(a+b);
```

přepíše do tvaru

```
pom = a+b;

x = pom*c;

y = sin(pom);
```

Je určitě zřejmé, že eliminace společných podvýrazů *(common subexpression elimination)* povede téměř vždy ke zrychlení programu. Za jistých okolností – jestliže se podvýrazy objevují dostatečně často – může způsobit i zmenšení programu.

Příště

Tolik pro dnešek – příště si povíme o některých dalších optimalizačních technikách a připojíme i několik obecných úvah o významu optimalizace.

Miroslav Virius