O sierotce, co chciała się mózgiem elektronowym wyręczyć,

czyli kolejność znowu ma znaczenie

Tomasz IDZIASZEK

Wyobraźmy sobie biedną sierotkę, której macocha nakazała oddzielić groch od fasoli. Chcąc nie chcąc, dziewczę siada w kącie izby przed pokaźnym kopcem grochu pomieszanego z fasolą i zaczyna pracę. Praca jest niezwykle monotonna: sierotka bierze nasiono z górki i jeśli to fasola, odrzuca je na lewą stronę, a jeśli groch – na prawą; i tak w kółko. Sierotka jest całkiem porządna, więc aby nie rzucać nasionkami po całej izbie, ustaliła, że nasionka fasoli odkłada na lewą stronę lewą ręką, a nasionka grochu odkłada prawą ręką. Ponieważ na wybór ręki, którą sięgnie po kolejne nasionko, musi się zdecydować, zanim rozpozna jego rodzaj (w izbie jest dość ciemno), więc nierzadko będzie zmuszona do przełożenia nasionka z jednej ręki do drugiej, co, oczywiście, będzie spowalniać jej pracę.

Powiedzmy, że sierotka w większości przypadków wyciąga z górki nasionko fasoli (takie już ma szczęście). Zauważywszy to, może postąpić praktycznie i na początku zawsze wyciągać nasionka lewą ręką, tak by zminimalizować liczbę przełożeń nasionka z ręki do ręki, a gdy w kopczyku zostanie już w większości groch, to będzie wyciągać nasionka prawą ręką. Sierotka może mieć szczęście innego rodzaju, np. prawie zawsze co trzecie wyciągnięte nasionko to groch. W tej sytuacji mądre dziewczę stwierdza, że najbardziej opłacalny jest schemat: lewa ręka, lewa ręka, prawa ręka itd. Niestety, znając bajkowe realia, sierotka najpewniej będzie wyciągała nasionka w zupełnie losowej kolejności, zatem żadne wymyślne strategie nie pomogą jej w uniknięciu marnego losu i średnio co drugie nasionko będzie musiało zostać przełożone z ręki do ręki.

Czytelnik Postępowy od razu zauważy, że wszystkie kłopoty skończyłyby się, gdyby na miejscu sierotki postawić robota z chwytnym ramieniem, fotokomórką i mózgiem elektronowym. Nie dość, że wykonałby on zadanie sprawniej i na pewno bezbłędnie, to nie rozwodziłby się długo nad wyborem strategii postępowania, a praca zajęłaby mu tyle samo czasu, niezależnie od kolejności, w jakiej wyciągałby ziarenka z górki.

I moglibyśmy zakończyć ten artykuł powyższym morałem, gdyby nie to, że nie jest on do końca prawdziwy. Okazuje się, że w świecie maszyn nie wszystko jest takie jasne i poukładane, a w elektronowym mózgu może kryć się coś ze sprytnej sierotki. Kto ciekaw, tego zapraszam do eksperymentu.

* * *

Ponieważ nie wszyscy Czytelnicy dysponują odpowiednim sprzętem tudzież zapasem grochu i fasoli, więc eksperyment zasymulujemy na domowym komputerze, pisząc prosty program. Program będzie wczytywał z wejścia zero-jedynkowy ciąg, który kodować będzie rodzaje kolejnych nasionek. Zamiast sortowania, program będzie miał na celu policzenie nasionek, tzn. policzenie, ile zer i ile jedynek występuje w wejściowym ciągu. Jeśli te wartości będzie przechowywał na zmiennych x_0 i x_1, to po każdorazowym wczytaniu nowego bitu b należy wykonać taki kod:

if $b = 0$ **then** $x_0 := x_0 + 1$
else $x_1 := x_1 + 1$;

Skompilujmy nasz program i uruchommy go dla różnych ciągów zero-jedynkowych równej długości, w których dokładnie połowa bitów to zera. Wydawałoby się, że czas działania naszego programu musi być zawsze taki sam, w końcu wykonamy dokładnie tyle samo operacji porównań i dodawania, co więcej, wartości zmiennych x_0 i x_1 zwiększymy tyle samo razy. Okazuje się jednak, że na moim komputerze dla ciągu $B_1 = 0^{n/2}1^{n/2}$ długości $n = 10^7$, w którym pierwsza połowa ciągu to zera, a druga połowa to jedynki, program działa 0,11 s, natomiast dla ciągu B_2, w którym pozycje zer i jedynek są losowe, program działa 0,16 s, czyli prawie o połowę dłużej! Nie jest to bynajmniej błąd pomiaru czasu, gdyż kolejne uruchomienia programu potwierdzają pierwotną obserwację. Ponadto w obu

Rozwiązanie zadania F 817.
Położenie równowagi ciężarka odpowiada rozciągnięciu sprężyny o $x_0 = mg/k$, gdzie k jest stałą sprężyny ($F = -kx$).

Położenie równowagi spławika odpowiada zrównoważeniu siły ciężkości przez siłę wyporu:

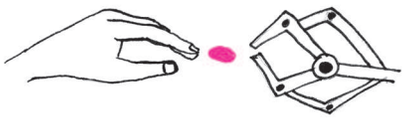
$$mg = V_0 \rho g,$$

czyli zanurzeniu jego objętości $V_0 = m/\rho$ na głębokość $h_0 = V_0/S = m/(\rho S)$, gdzie S jest polem przekroju poprzecznego spławika, a ρ gęstością cieczy w wiadrze.

Ponieważ w obu przypadkach siły generowane przez pionowe przesunięcie względem położenia równowagi są proporcjonalne do tego przesunięcia (i skierowane przeciwnie), więc ruch jest harmoniczny, a częstość jest pierwiastkiem z ilorazu odpowiedniego współczynnika proporcjonalności przez masę:

$$\sqrt{\frac{k}{m}} = \omega = \sqrt{\frac{S\rho g}{m}}.$$

Spadanie wiadra odpowiada zmianie efektywnego przyspieszenia ziemskiego. W związku z tym ciężarek będzie drgał z tą samą częstością, ale względem zmienionego położenia równowagi, oraz ze zmienioną amplitudą (czy można tak dobrać warunki początkowe, żeby amplituda nie zmieniła się?). Natomiast w przypadku spławika zmieni się częstość drgań (zmaleje) i amplituda (chyba że spadanie rozpoczęło się w momencie maksymalnego wychylenia z położenia równowagi, wtedy amplituda się nie zmieni, dlaczego?), ale położenie równowagi nie zmieni się.



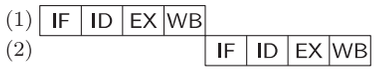
uruchomieniach programu wczytywanie danych zajmuje 0,03 s, więc dysproporcja czasu „właściwych obliczeń” jest w rzeczywistości jeszcze większa.

Jaka jest przyczyna takiego zachowania? Czyżby domowy komputer z jakichś powodów radził sobie lepiej z „przewidywalnymi” danymi, tak jak nasza sierotka? A jeśli tak jest w istocie, to jakie są tego powody?

Jak wiemy, kompilacja powoduje przetworzenie kodu programu na ciąg instrukcji niskiego poziomu. Na przykład, zwiększenie wartości zmiennej znajdującej się w rejestrze x procesora mogłoby zostać zapisane w wewnętrznym języku komputera jako

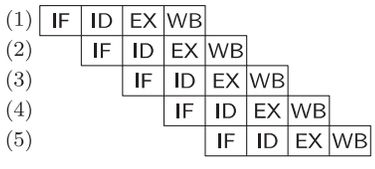
`INC` x

Na wykonanie powyższej instrukcji składa się kilka kroków. Po pierwsze, instrukcja musi być pobrana z pamięci. Następnie musi zostać zdekodowana: procesor odkrywa, że chodzi o zwiększenie zmiennej z rejestru. W kolejnym kroku należy wykonać instrukcję: przesłać wartość zmiennej z rejestru do jednostki arytmetycznej procesora i dokonać zwiększenia. I ostatecznie należy przesłać uaktualnioną wartość z powrotem do rejestru. (Jest to przykład dość uproszczony, w ogólnym przypadku wykonanie instrukcji może składać się z większej liczby kroków, tj. np. przekopiowanie danych z pamięci do rejestru procesora. We współczesnych procesorach liczba takich kroków może sięgać kilkudziesięciu.) Widać więc, że wykonanie pojedynczej instrukcji nie jest takie banalne. Jeśli założymy, że każdy krok wykonuje się w jednej jednostce czasu, to na wykonanie n instrukcji potrzebujemy $4n$ jednostek czasu (rys. 1).



Rys. 1. Na wykonanie instrukcji procesora składają się cztery kroki: IF (ang. *instruction fetch*), ID (*instruction decode*), EX (*execute*), WB (*write back*). Na wykonanie dwóch instrukcji potrzebujemy 8 jednostek czasu.

Zauważmy jednak, że skoro każdy z kroków jest wykonywany przez inną część procesora, to można przyspieszyć cały proces, umożliwiając tym częściom procesora pracę równoległą przez przetwarzanie kilku instrukcji naraz. (Tak jak na linii produkcyjnej w fabryce samochodów ekipa montująca podwozie pracuje równolegle z ekipą malującą karoserię – pracują po prostu na innych egzemplarzach.) Można postąpić tak: w pierwszym kroku pobieramy pierwszą instrukcję z pamięci. W drugim kroku dekodujemy tę instrukcję, ale w tym momencie część procesora odpowiedzialna za pobieranie instrukcji jest bezrobotna, wobec tego możemy jednocześnie pobrać z pamięci drugą instrukcję. W trzecim kroku wykonujemy pierwszą instrukcję, dekodujemy drugą i pobieramy z pamięci trzecią. Dzięki takiemu *potokowemu* przetwarzaniu instrukcji na wykonanie n z nich potrzebujemy tylko $n + 3$ jednostek czasu (rys. 2).

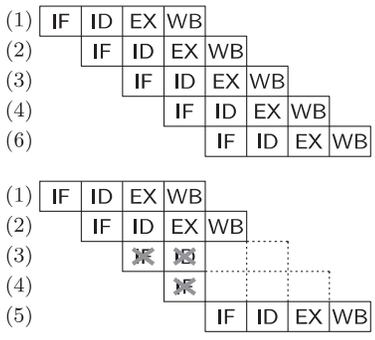


Rys. 2. Dzięki przetwarzaniu potokowemu (ang. *pipelining*) w 8 jednostkach czasu jesteśmy w stanie wykonać 5 instrukcji w przypadku pustego potoku i aż 8 instrukcji w przypadku wypełnionego potoku.

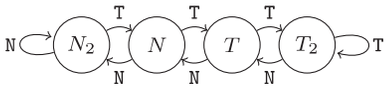
Jak każdy świetny pomysł, tak i ten ma pewne wady. Zauważmy, że przyjęliśmy tu milczące założenie, że zanim zakończymy wykonywanie danej instrukcji, musimy być w stanie rozpocząć wykonywanie następnej, a nawet trzeciej instrukcji z kolei. W szczególności musimy wiedzieć, co to będą za instrukcje. O tym, że nie zawsze posiadamy tę wiedzę, możemy się przekonać, kompilując nasz pierwszy program:

(1) `CMP` $b, 0$
(2) `IFEQJMP` (5)
(3) `INC` x_1
(4) `JMP` (6)
(5) `INC` x_0
(6) ...

W pierwszej instrukcji zapisanej pod adresem (1) wykonujemy porównanie zmiennej b z zerem. Następna instrukcja jest instrukcją skoku warunkowego: jeśli liczby porównywane ostatnio były równe, to instrukcja ta powoduje skok do instrukcji zapisanej pod adresem (5), w przeciwnym przypadku nie dzieje się nic. Odpowiada to wybraniu odpowiedniej gałęzi w instrukcji **if**. Jeśli wykonaliśmy skok (pierwsza gałąź), to w instrukcji (5) zwiększamy wartość x_0. Jeśli skoku nie wykonaliśmy (druga gałąź), to w instrukcji (3) zwiększamy wartość x_1, a następnie wykonujemy bezwarunkowy skok do instrukcji (6). Kluczowa jest w tym programie instrukcja skoku warunkowego `IFEQJMP`: mianowicie, dopóki jej nie wykonamy, nie wiemy, czy następną po niej będzie instrukcja (3), czy też instrukcja (5), nie wiemy więc, która z nich powinna być w potoku uwzględniona jako następna. Procesor jest zatem w sytuacji naszej sierotki, która,



Rys. 3. Sytuacja w potoku w przypadku, gdy skok warunkowy w instrukcji (2) nie nastąpił i gdy nastąpił. Liczby w nawiasach oznaczają numery instrukcji; w pierwszym przypadku w instrukcji (4) występuje bezwarunkowy skok do instrukcji (6).



Rys. 4. Czterostanowy wskaźnik zgadujący, czy skok zostanie wykonany.

dopóki nie dotknie nowego ziarenka, nie będzie wiedziała, czy należało użyć lewej czy też prawej ręki.

Można ten kłopot spróbować rozwiązać następująco: procesor zawsze będzie zakładał, że po instrukcji zapisanej pod adresem (i) następują instrukcje pod adresami $(i+1)$, $(i+2)$ itd. lub pod adresem, który wskazuje skok bezwarunkowy, i te właśnie instrukcje będzie uwzględniał w potoku. Jeśli jednak instrukcja (i) była rozkazem warunkowego skoku i po jej wykonaniu okazało się, że należy skoczyć pod adres (j), to częściowa praca związana z wykonaniem instrukcji $(i+1)$ i kolejnych jest unieważniana, a potok jest opróżniany i przetwarzanie zaczyna się na nowo od instrukcji (j) (rys. 3).

W przypadku programów, w których jest mało skoków warunkowych lub są rzadko wykonywane, taka strategia może się opłacać. Jednak w pesymistycznym przypadku, gdy każda kolejna instrukcja jest skokiem warunkowym, musimy opróżniać potok po wykonaniu każdej instrukcji, zatem czas wykonania n instrukcji będzie wynosił $3n+1$ jednostek – nasz potok nie będzie zatem wykorzystywany w pełni efektywnie.

Zauważmy jednak, że taka strategia nie może być stosowana w naszym procesorze: przecież program dla ciągów B_1 i B_2 powinien działać tak samo długo, gdyż w obu przypadkach potok byłby opróżniany w połowie instrukcji (2). Taka strategia odpowiada sytuacji, w której nasza sierotka postanowiłaby zawsze używać lewej ręki.

Możemy przyjąć inną metodę postępowania i próbować zgadnąć, czy dany skok warunkowy będzie wykonany, czy też nie. Okazuje się, że współczesne procesory tak właśnie postępują. Co więcej, ponieważ nie znają przyszłości, więc zachowują się jak nasza sierotka i zakładają, że będą miały szczęście i na podstawie zachowań instrukcji skoku warunkowego w przeszłości będą w stanie przewidzieć, jak będzie się ona zachowywała w przyszłości.

Zapiszmy historię wykonywania skoku warunkowego w postaci ciągu złożonego z liter `T` i `N`. Jedna z najprostszych strategii zgadywania jest następująca: przyjmujemy, że skok będzie wykonany, jeśli ostatnim razem był wykonany (czyli ostatnia litera w ciągu to `T`). Ta strategia ma sens w przypadku długich sekwencji analogicznych wyborów, jak to ma miejsce dla ciągu B_1. Można ją trochę ulepszyć, aby była niewrażliwa na sporadyczne przekłamania w historii, za pomocą wskaźnika mogącego przyjmować cztery stany: N_2, N, T, T_2; każde wykonanie skoku powoduje zmianę stanu wskaźnika o jeden w prawo zgodnie z rysunkiem 4, a niewykonanie skoku – w lewo. Zgadujemy, że skok zostanie wykonany, jeśli wskaźnik znajduje się w stanie T lub T_2.

Gdyby w naszym procesorze zastosowano powyższą strategię, wyjaśniałaby ona różnice w działaniu programu dla ciągów B_1 i B_2: w przypadku pierwszego z nich nasza strategia zgadywania nie zadziała tylko kilka razy (w środku ciągu i, być może, na początku), z kolei jest ona zupełnie bezużyteczna (jak w zasadzie jakakolwiek strategia zgadywania) w przypadku drugiego z nich.

Niestety, to nie wyjaśnia, dlaczego dla ciągu $B_3 = (001)^{n/3}$, w którym dokładnie co trzeci bit to 1, czas działania jest taki sam jak dla ciągu B_1. Wszak dla ciągu B_3 musielibyśmy pomylić się dla każdej jedynki, czyli co najmniej $n/3$ razy. Okazuje się, że strategia stosowana przez współczesne procesory próbuje uwzględniać możliwość wystąpienia krótkich cykli w historii wykonywania skoku i jest z tego powodu trochę bardziej skomplikowana. Mianowicie, przy podejmowaniu decyzji patrzymy na k ostatnich liter w historii i na podstawie tych liter wybieramy jeden z 2^k czterostanowych wskaźników, który uaktualniamy i podejmujemy decyzję zgodnie z jego wskazaniem. W ten sposób dostajemy mechanizm, który będzie zgadywał poprawnie, o ile w historii wykonywania skoku każde k kolejnych bitów jednoznacznie wyznacza bit $(k+1)$-szy. W moim komputerze jest $k=4$, zatem w ciągu B_3, którego historię można opisać regułami: po każdym `TT` jest `N`, po każdym `TN` jest `T`, a po każdym `NT` jest `T`, procesor nie pomyli się ani razu (z wyłączeniem kilku potencjalnych pomyłek na początku ciągu).

Rozwiązanie zadania M 1359.
Odpowiedź: Oto pokolorowanie, w którym taki odcinek nie istnieje.

Ustalmy punkt O płaszczyzny i pomalujmy go na czarno. Punkty na każdym okręgu o środku w O i promieniu niewymiernym pomalujmy na biało, zaś punkty na każdym okręgu o środku w O i promieniu wymiernym – na czarno. Przez dowolny odcinek dodatniej długości musi przechodzić pewien okrąg pierwszego typu, jak również drugiego. Zatem odcinek ten nie może być jednokolorowy.

Opisane strategie nie są jedynymi, które są stosowane we współczesnych procesorach. Istnieje cała gama rozwiązań, które pozwalają przewidywać typowe historie wykonywania skoku. W niektórych procesorach do problemu podchodzi się zupełnie inaczej. Zakłada się, na przykład, że w skompilowanym kodzie po każdej instrukcji skoku warunkowego musi wystąpić pewna liczba zwykłych instrukcji, które będą wykonane niezależnie od tego, czy skok nastąpi, czy nie, aby po ich wykonaniu adres docelowy był już obliczony.

* * *

Morał z tej bajki jest taki, że nawet kolejność, w jakiej podajemy dane programowi komputerowemu, ma znaczenie (mimo że na pierwszy rzut oka może wydawać się to niewiarygodne). Drugi morał jest ważną wskazówką dla programisty: w kodzie, w którym ważna jest efektywność wykonania, należy, o ile to możliwe, unikać skoków. Okazuje się, że nasz pierwszy program możemy przepisać tak, by nie występowała w nim instrukcja **if**:

$x_0 := x_0 + 1 - b;$
$x_1 := x_1 + b;$

Tak napisany program działa w czasie 0,11 s dla każdego z ciągów B_1, B_2 i B_3.

Artykuł powstał na podstawie następujących zadań:

198. Get Out!
z serwisu `acm.sgu.ru`,

Płotki
z Obozu Naukowo-Treningowego im. A. Kreczmara 2009,

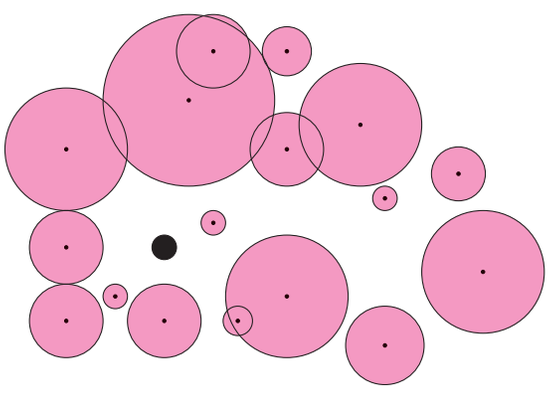
Ucieczka i *Budowanie płotu*
z Bałtyckiej Olimpiady Informatycznej 2007.

Ucieczka

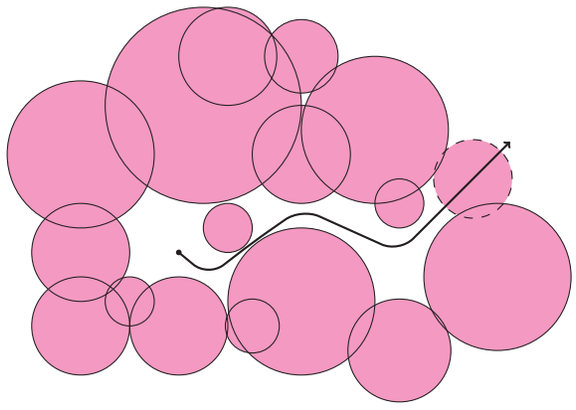
Jakub RADOSZEWSKI

Wyobraź sobie, Drogi Czytelniku, że jesteś kapitanem okrętu wojennego i w trakcie jednej z misji znalazłeś się na środku morza leżącego na terytorium wroga. Wiesz, że wróg rozmieścił w tej strefie pewną (skończoną) liczbę radarów. Każdy radar ma określony zasięg, być może różny w przypadku różnych radarów, i jest w stanie wykryć każdy podejrzany obiekt, który znajdzie się w jego zasięgu. Naszym siłom wywiadowczym udało się wykraść plan rozmieszczenia radarów. Na jego podstawie chcesz stwierdzić, czy możesz wydostać się z wrogich wód niezauważony przez radary.

Powyższa historia wojenna z lotu ptaka wygląda następująco: na płaszczyźnie zadana jest pewna liczba kół stanowiących obszary zabronione. Dla uproszczenia nasz statek również przedstawimy jako koło. Naszym zadaniem jest sprawdzić, czy możemy przemieścić się statkiem nieskończenie daleko od początkowej pozycji, nie dotykając przy tym żadnego z pozostałych kół (rys. 1).



Rys. 1



Rys. 2

Takie sformułowanie problemu nie jest jednak zbyt wygodne. Możemy je uprościć przez „odpompowanie” koła reprezentującego statek i „napompowanie” kół przedstawiających zasięgi radarów. Dokładniej, promienie wszystkich kół-radarów zwiększamy o promień statku, a sam statek zmniejszamy do jednego punktu – środka koła (rys. 2). Aby uzasadnić poprawność tego przekształcenia, wystarczy zauważyć, że bezpieczna trasa statku charakteryzuje się tym, iż jego środek nie zbliża się do żadnego radaru na odległość mniejszą niż suma promienia statku i zasięgu radaru. Po tej transformacji dużo łatwiej udzielić odpowiedzi na pytanie postawione w zadaniu; w sytuacji z rysunku 2 ucieczka