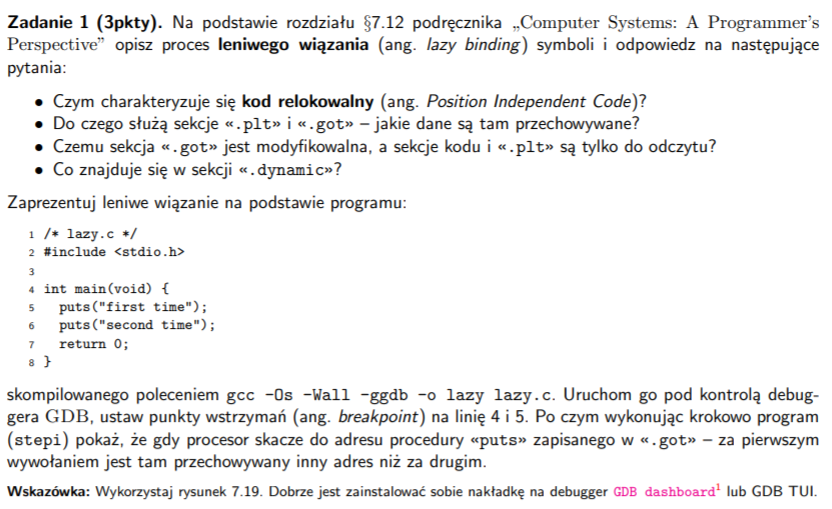
**Systemy Komputerowe**

Lista 7



**Według podanego źródła:** Leniwe wiązanie to sposób na oszczędzenie procesorowi instrukcji, które musi wykonać w przypadku wywołania procedury. Adres nie jest wiązany z samą procedurą dopóki nie będziemy próbować się do niego odwołać. W momencie wołania wykonujemy co prawda tę samą pracę co bez „leniwego wiązania”, ale każde kolejne wołanie tego symbolu będzie już kosztowało tylko jedną instrukcję plus odwołane do pamięci.

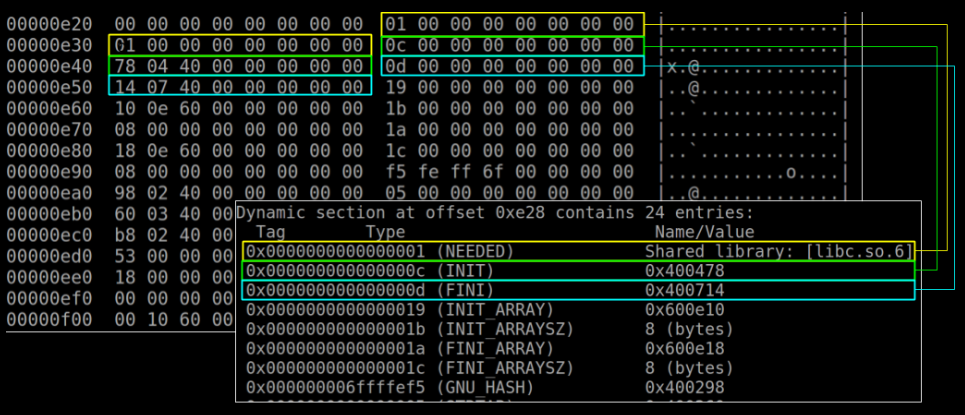
Jest to możliwe dzięki interakcji dwóch struktur ze sobą: Global offset table (GOT) oraz Procedure linkage table (PLT). GOT zawiera offsety dla każdego globalnego symbolu w programie. Jako, że odległości pomiędzy dowolną instrukcją w kodzie a dowolną zmienną są zawsze takie same (zmienne są ułożone w pamięci za kodem) możemy spisać na jakich pozycjach względem kodu są umieszczone odpowiednie symbole. Właśnie takie informacje znajdziemy w tej tabeli. Jest to część sekcji .data.

Jeśli symbol jest zdefiniowany w jakiejś bibliotece, ma swój wpis w GOT oraz PLT. PLT jest częścią sekcji .text. Jeśli jakaś procedura jest zawołana, jest dla niej tworzony wpis w PLT.

* **Kod relokowalny (PIC)** jest to kod, którego adresy pamięci są niezależne od ułożenia innych programów/bibliotek/kodów w maszynie. Wszystkie potrzebne adresy w takim pliku są wyliczane na podstawie przesunięć względem odpowiednich sekcji w tym samym pliku \*.o. Dopiero potem linker wylicza bezwzględne adresy, pod którymi zostaną umieszczone odpowiednie symbole w pamięci maszyny.
* **.plt** – Jest to sekcja „procedure linkage table” i przechowuje w sobie informacje o zewnętrznych procedurach i dostępie do nich. Każda zawołana procedura ma wpis w takiej sekcji. Dla danej procedury zawiera m.in. informacje gdzie jej adres znajduje się w GOT oraz pomaga przy realizacji leniwego wiązania. Dopiero w trakcie uruchomienia dynamiczny linker znajduje odpowiednie adresy w zewnętrznych plikach.

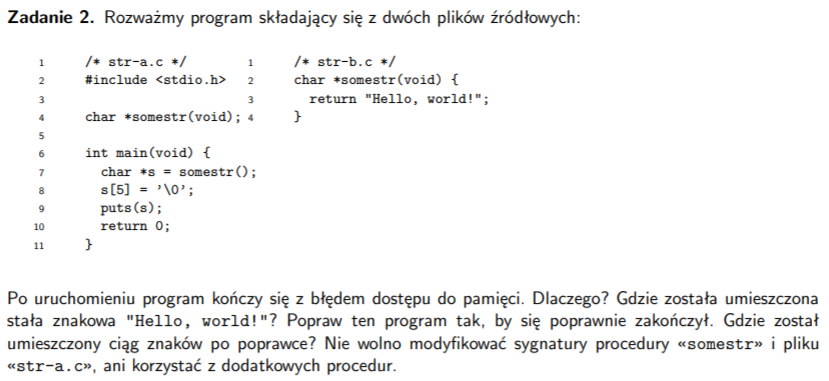
**.got –** Jest to sekcja „global offset table” i przechowuje przesunięcia dla odpowiednich symboli względem innych sekcji programu \*.o. Symbole są umieszczane w pamięci za kodem programu i ich pozycja nie zmienia się względem tego kodu. Dlatego też możemy obliczyć ich względną pozycję i umieścić ją w tabeli.

* Sekcja **.got** jest modyfikowalna, ponieważ maszyna zmienia ją według tablicy **.plt** kiedy zawołamy procedurę po raz pierwszy. Wpisuje tam odpowiednie adresy procedur, które w kolejnych wywołaniach posłużą szybszej obsłudze. Z racji na to, że **.got** jest w sekcji **.data**, a **.plt** w sekcji **.text**, tej drugiej sekcji nie wolno nam modyfikować, tak samo jak nie możemy modyfikować samego kodu.
* W sekcji .dynamic znajdują się informacje dla linkera. Ponieważ linker wykonuje dużo różnych rzeczy oprócz relokacji, sekcja .dynamic jest taką „ściągą”, która zawiera informacje o potrzebnych procedurach, bibliotekach i miejscu ich położenia.



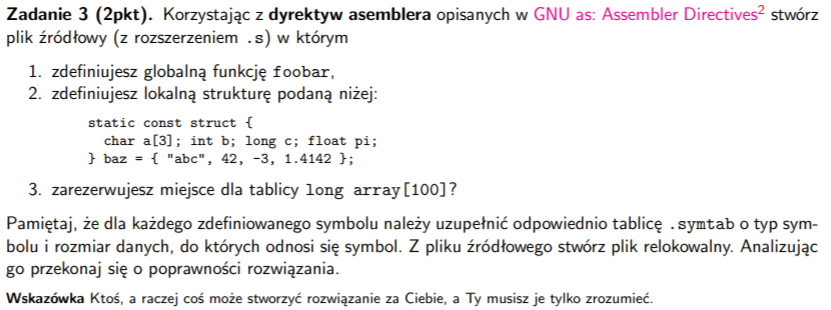
Po lewej stronie jest wartość, która identyfikuje co to jest, po prawej jej typ. I tak na przykład dla biblioteki libc.so mamy wartości: 0x0 oraz 0x1 (oznaczenie libc.c oraz typ: „needed”).

* Na początku: 0x7FFFF7FFE190, 0x(5…)43e3
* Później: 0x00007FFFF7E17130, 0x(5…)600f

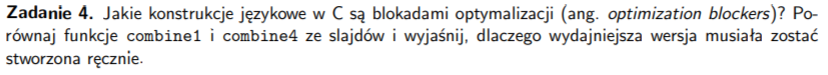


Zmianą jest deklaracja zmiennej **char[13] = „Hello, world!”** i zwrócenie tej zmiennej w procedurze.

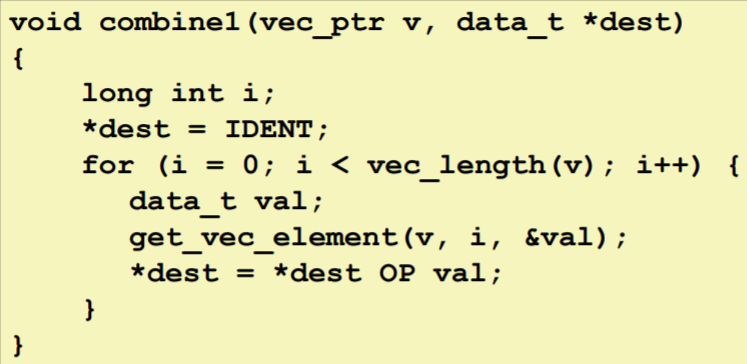
Przed poprawką ciąg znakowy został ustalony jako read-only. Umieszczony w sekcji „.rodata”. Po poprawce jest to już coś, co możemy modyfikować, gdyż zmieniliśmy to na tablicę, więc z założenia powinno dać się to modyfikować.

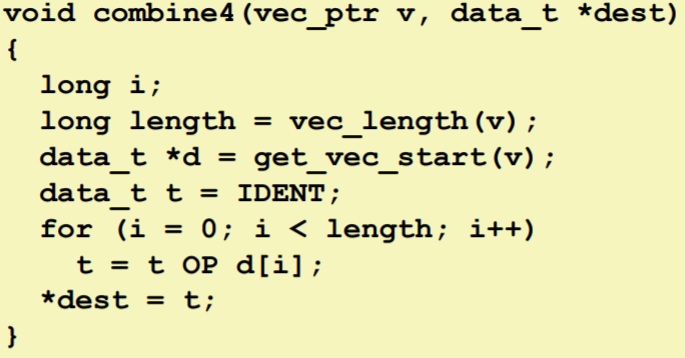


Kod z maszyny wirtualnej, Documents -> 3.o

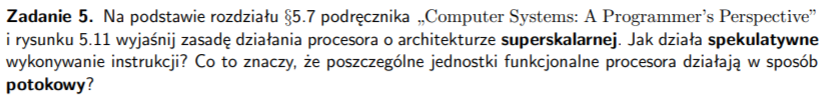


Spójrzmy na obie funkcje:

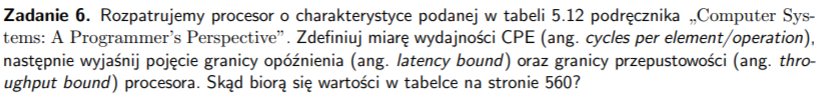




1. Get\_vec\_element mogła mieć jakieś side-efekty (typu zmiana v)
2. Vec\_length mogła mieć jakieś side-efekty (typu zmiana v)
3. Mógł mieć miejsce memory aliasing w przypadku v (?)



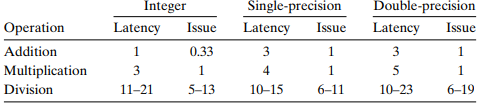
* Procesory o architekturze **superskalarnej** są w stanie wykonywać więcej niż jedną instrukcję na raz jeśli wykryją, że dane dwie instrukcje są od siebie niezależne. Np. wykonują równolegle 3 instrukcje arytmetyczne typu: a\*b, a\*c, a\*d, dzięki czemu zamiast standardowych 9 cykli procesora mamy tylko 5. Może także wykonywać instrukcje poza kolejnością. Czyli wykonać najpierw późniejszą instrukcję tak długo jak nie zmieni ona sensu programu.
* **Spekulatywne** wykonywanie instrukcji to przewidywanie drogi, którą podąży program na rozgałęzieniu. Procesor dekoduje instrukcje i przewiduje czy program pójdzie daną drogą oraz zaczyna wykonywać instrukcje w tej gałęzi, nawet jeśli nie jest jeszcze potwierdzone czy przewidywanie jest poprawne. Jeśli wyjdzie, że dana droga nie jest poprawna, wraca do rozgałęzienia i zaczyna wykonywać instrukcje z tej drugiej drogi.
* Działają w sposób **potokowy**, czyli mogą funkcjonować jako seria etapów następujących po sobie. Jednostka wykonuje część pracy, którą dostała od poprzedniej jednostki, która wykonała jakąś inną pracę, aby dostać ten wynik. Np. jednostki dodające liczby zmiennoprzecinkowe. Dodawanie obywa się w 3 cyklach, więc 3 jednostki mogą wykonać po kolei każdy z 3 etapów: (1) przetworzenie wykładników, (2) dodanie liczb do siebie, (3) zaokrąglenie wyniku.



**CPE** (cycles per element/operation) to według wykładu ilość cykli, które procesor spędzi na wykonaniu danej operacji. Na wykładzie mieliśmy dobry przykład tego jak optymalizacje wpływają na ilość cykli potrzebnych do wykonania funkcji.  
  
**Granica opóźnienia** (latency bound) to graniczny czas, który jest definiowany przez czas wykonania danej operacji przez procesor. Jest to techniczna granica, szybciej nie możemy wykonać jakiejś operacji niż procesor jest ją w stanie fizycznie zrobić.

**Granica przepustowości** (throughput bound) to czas jaki musi upłynąć, abyśmy mogli wykonać dwie instrukcje po sobie (wrzucić je w potok). Dla dodawania jest to na przykład 0.33, czyli możemy wykonać jakąś kolejną operację zaraz po skończonej poprzedniej operacji.

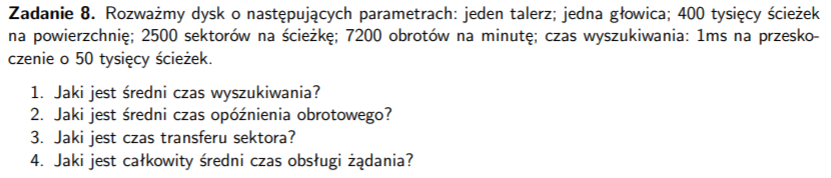
Wartości z tabelki na stronie 501 biorą się z:



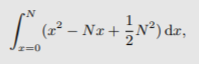
Najciekawsze przypadki:

Każde dodawanie zajmuje cykl procesora, więc możemy zrobić 3 równoległe dodawania. Stąd 0.33 issue. Dodawanie floatów zajmuje 3 cykle, więc każdy etap dodawania możemy rozłożyć po jednostkach i każda z nich będzie przeprowadzać jeden etap. Stąd co cykl możemy rozpoczynać nowy pierwszy etap innej operacji. Wraz ze zwiększeniem ilości bitów do przetworzenia (32 lub 64) zwiększa się czas operacji. Dzielenie jest operacją najbardziej kosztowną. Także nie możemy zbyt dobrze tej operacji upakować w etapach po jednostkach, ponieważ różne dzielne i dzielniki dają różne ilości cykli procesora. Tym bardziej, że ich czas issue jest tylko trochę mniejszy od czasu wykonania, więc być może czasami musimy wykonać najpierw 2 etapy i zacząć wykonywać trzeci, aby móc zacząć wykonywać równolegle drugie dzielenie.





Policzmy na początek jak to wygląda (zakładam, że dane są po jednej stronie).

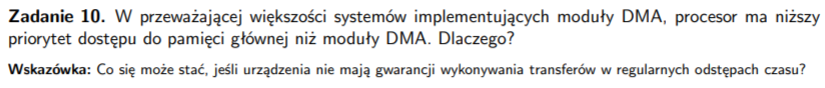
1. Tseek to czas wyszukiwania. Przeliczmy jak to wygląda:
   1. Średni czas = (Suma wszystkich możliwości)/Ilość tych możliwości
   2. Głowica może w danym momencie być w pozycjach od 0 do N włącznie (pozycje na talerzu).
   3. Powiedzmy, że pozycja początkowa głowicy to x, a końcowa to y.
   4. X może się wahać od 0 do N, y tak samo.
   5. Liczymy daną odległość jako |x-y|
   6. Powstaje z tego całka:
   7. 
   8. Robijamy tą w środku na:
   9. 
   10. Powstaje:
   11. 
   12. 
   13. Na zewnątrz otrzymujemy więc:
   14. 
   15. Czyli wynik:
   16. 
   17. Dzielimy to jeszcze przez wszystkie możliwości, czyli N^2 i wychodzi N/3

Więc odpowiedź do pytania to: 400000/3 = 133333,(3) a jeśli 50tys ścieżek robimy w 1ms to jest to: 2.(6) ms.

1. Opóźnienie obrotowe: 7200 obrotów na minutę, czyli 120 na sekundę. Czyli jeden obrót zajmuje 1/120 sekundy. Średnio musimy przejść połowę ścieżki żeby znaleźć interesującą nas wartość, czyli mamy: (1/120) \* ½ = 0.00416 s = 4.16ms

3. Czas transferu sektora: 1/120 (tyle zajmuje jeden obrót w sekundach). Mamy 2500 sektorów na ścieżkę, czyli 1/2500 ścieżki to jeden sektor. Więc końcowo: 1/120 \* 1/2500 = 3.(3) \* 10^(-6). Na milisekundy to będzie \*10^3, czyli 3.(3) \* 10^(-9)

4. Suma: 2.(6) + 4.16 + 10^(-9). Czyli w przybliżeniu jakieś 6.76ms.



Kiedy DMA nie używa pamięci, pamięci używa procesor. DMA tylko „kradnie czas” dostępu do pamięci procesorowi. DMA musi mieć priorytet dostępu nad procesorem, żeby zapobiec dwóm zjawiskom:

* Załóżmy, że wejście dostajemy od urządzenia, które nie ma buffera na swój input. Jeśli nie przyjmiemy inputu w czasie, ten przepadnie. Jeśli nie chcemy żeby przepadł, musielibyśmy mieć jakiś system, który ponownie poprosi tamtą maszynę o powtórzenie informacji, a to może nastręczyć kolejne problemy z przywracaniem stanu, robieniem obliczeń jeszcze raz itd. itd.
* Jeśli na wyjściu mamy urządzenie, które nie bufforuje outputu, może się zdarzyć, że komputer nie zdąży na czas dostarczyć informacji i będą dziury w transmisji. Przez to mogą powstać jakieś błędy w komunikacji.
* Jeśli mamy do przesłania duży blok danych, CPU woła DMA to tego transferu. Dzięki temu w tym czasie CPU może wykonywać jakieś inne operacje, za to DMA wtedy przesyła odpowiednie informacje po szynie. Gdyby to procesor miał przesyłać dane z priorytetem, DMA mogłoby czekać naprawdę długo i transfer pamięci by zajął. Albo inaczej, procesor zająłby się transferem zamiast robić coś pożytecznego. Wszystko by spowolniło.