Politechnika Opolska

Wydział Elektrotechniki, Automatyki i Informatyki

Instytut Automatyki i Informatyki

PRACA DYPLOMOWA

inżynierska

PROJEKT I REALIZACJA LABORATORIUM PROGRAMOWANIA NISKOPOZIOMOWEGO DLA PROCESORÓW INTEL W TRYBIE CHRONIONYM

Promotor:

dr inż. Grzegorz Bialic

Pracę wykonał:

Grzegorz Wyszyński

Opole, styczeń 2010

PROJEKT I REALIZACJA LABORATORIUM PROGRAMOWANIA NISKOPOZIOMOWEGO DLA PROCESORÓW INTEL W TRYBIE CHRONIONYM

S t r e s z c z e n i e

Niniejsza praca zawiera opis architektury x86 oraz tryby pracy ze szczególnym uwzględnieniem trybu chronionego procesora. Architekturę procesora 8086 zaprezentowano z punktu widzenia programisty. Wprowadzone i wyjaśnione zostały podstawowe instrukcje języka asembler. Opisane zostały wszystkie wykorzystane narzędzia z których korzystano podczas realizacji pracy, pokazano sposób korzystania z interfejsu API systemu WindowsNT, wytłumaczono zasadność użycia i sposób przygotowania plików wsadowych bat. Przedstawiony został również sposób przygotowania środowiska do pracy z pakietem NASM10, oraz opisane najbardziej popularne mechanizmy programowania. Wykorzystano program DEV C++ aby pokazać sposoby łączenia języka C z językiem asemblera w składniach INTEL oraz AT&T.

PROJECT AND REALIZATION OF LOW LEVEL PROGRAMMING LABORATORY FOR INTEL PROCESSORS WORKING IN PROTECTED MODE

S u m m a r y

The thesis shows fundamental description of x86 architecture. Processor working modes are characterised. Architecture of 8086 processor is shown at angle of programming. Basic instructions of assembler language are introduced. All tools and direction of use are described. The way of use of API interface in WindowsNT system is shown. The project shows how to prepare the working environment of the NASM10 package. The popular techniques of programming are described. DEV C++ was used to show how to combine C language with the low level language in INTEL and AT&T syntaxes.

SPIS TREŚCI

1. CEL I ZAKRES PRACY . . . . . str. 4
2. WSTĘP . . . . . str. 5
3. ARCHITEKTURA x86 . . . . . str. 6
3.1. PROCESOR INTEL 8086 . . . . . str. 6
3.2. PRACA W TRYBIE RZECZYWISTYM . . . . . str. 8
3.3. PRACA W TRYBIE CHRONIONYM . . . . . str. 10
3.3.1. ADRESOWANIE PAMIĘCI . . . . . str. 11
3.3.2. STRONICOWANIE PAMIĘCI . . . . . str. 13
3.3.3. TRYB WIRTUALNY . . . . . str. 14
4. JĘZYK ASEMBLER WPROWADZENIE . . . . . str. 15
4.1. INSTRUKCJE JĘZYKA ASEMBLER . . . . . str. 15
4.2 ADRESACJA . . . . . str. 17
4.3. PROCEDURY . . . . . str. 17
5. NARZĘDZIA . . . . . str. 19
5.1. MASM 10 . . . . . str. 19
5.2. WINDOWS API . . . . . str. 20
5.3. EDYTOR ZASOBÓW . . . . . str. 20
5.4. DEV C++ . . . . . str. 20
6. ĆWICZENIA LABORATORYJNE . . . . . str. 21
6.1. OBSŁUGA KONSOLI . . . . . str. 21
6.2. WPROWADZENIE DO OKIEN . . . . . str. 24
6.3. STRUKTURA OPENFILENAME . . . . . str. 26
6.4. APLIKACJA WIELOWĄTKOWA . . . . . str. 28
6.5. APLIKACJA SIECIOWA . . . . . str. 30
6.6. ŁĄCZENIE Z JĘZYKAMI WYŻSZEGO POZIOMU . . . . . str. 32
7. PODSUMOWANIE . . . . . str. 35
8. LITERATURA . . . . . str. 36
9. SPIS RYSUNKÓW . . . . . str. 37
10. SPIS TABEL . . . . . str. 38
11. DODATEK . . . . . str. 39

1. CEL I ZAKRES PRACY

Podstawowym celem pracy jest przygotowanie materiałów w postaci opisu teoretycznego przykładowych zadań z rozwiązaniami oraz środowiska programistycznego laboratorium programowania niskopoziomowego, obejmującego zagadnienia związane z pracą procesora Intel w trybie chronionym oraz programowaniem pod Windows w języku assembler.

Część teoretyczna pracy przedstawia i charakteryzuje tryby pracy procesorów zgodnych z INTEL x86. Tryb rzeczywisty został omówiony na przykładzie procesora 8086, natomiast fragment pracy dotyczy trybu chronionego prezentując funkcjonalności dla rodziny procesorów INTEL zgodnych z 386.

Kolejne rozdziały zawierają wstęp do języka asemblera zgodnego z asemblerem NASM10, opis narzędzi i sposobów asemblacji oraz konsolidacji do plików wykonywalnych.

Przygotowane laboratorium obejmuje szeroki zakres tematyczny – programowanie konsoli, okien, podstawowe funkcje bibliotek systemu WindowsNT, programowanie wielowątkowe, sieciowe oraz łączenie z językami wyższego poziomu.

Osiągnięcie podstawowego celu pracy wymaga realizacji następujących celów pośrednich:

- określenia tematyki projektu
- wyboru narzędzi
- instalacji i konfiguracji oprogramowania
- przygotowania środowiska do kompilacji i likowania
- dostarczenie gotowych programów przykładowych oraz rozwiązań
- stworzenie materiałów pomocniczych do ćwiczeń

2. WSTĘP

Kompilatory języków wysokiego poziomu oferują obecnie wysoką efektywność programowania. Powstające kody źródłowe są szybkie i zajmują niewiele miejsca. Lecz podczas rozwiązywania pewnych problemów np. wektorowych problemów numerycznych gdzie wymagana jest ponadprzeciętna szybkość działania niezbędnym jest użycie języka najniższego poziomu jakim jest asembler.

Programowanie w języku asembler posiada trzy główne zalety: szybkość wykonania, pełny dostęp do sprzętu oraz niskie wymagania względem wielkości pamięci operacyjnej. Wiele języków programowania umożliwia wstawianie kodu asemblera aby zoptymalizować szybkość wykonania kluczowych operacji. Używany jest też przy rozwiązywaniu aplikacji czasu rzeczywistego o ostrych ograniczeniach czasowych. Pełny dostęp do sprzętu sprawa, że język ten jest używany do tworzenia sterowników różnych urządzeń. Wraz ze spadkiem cen pamięci operacyjnych dla systemów zgodnych z x86 coraz mniejsze znaczenie ma zajętość pamięci operacyjnej. W tym przypadku asembler wykorzystywany jest częściej w urządzeniach typu embedded, mikrokontrolerach czy urządzeniach przenośnych.

Często spotykane jest programowanie w języku assembler w trybie rzeczywistym – ostatnim systemem korzystającym z tego trybu jest DOS. Głównym celem takich programów są wartości edukacyjne. Istnieje wiele źródeł, dzięki którym można nauczyć się programować wykorzystując funkcje BIOS i DOS. Jednak należy uznać tą technologię za przestarzałą i skupić się na programowaniu aplikacji WIN32. Dzięki temu można korzystać z bibliotek WinAPI, tworzyć programy posiadające elementarne interfejsy graficzne przez co język asembler nabiera nowego znaczenia.

3. ARCHITEKTURA X86

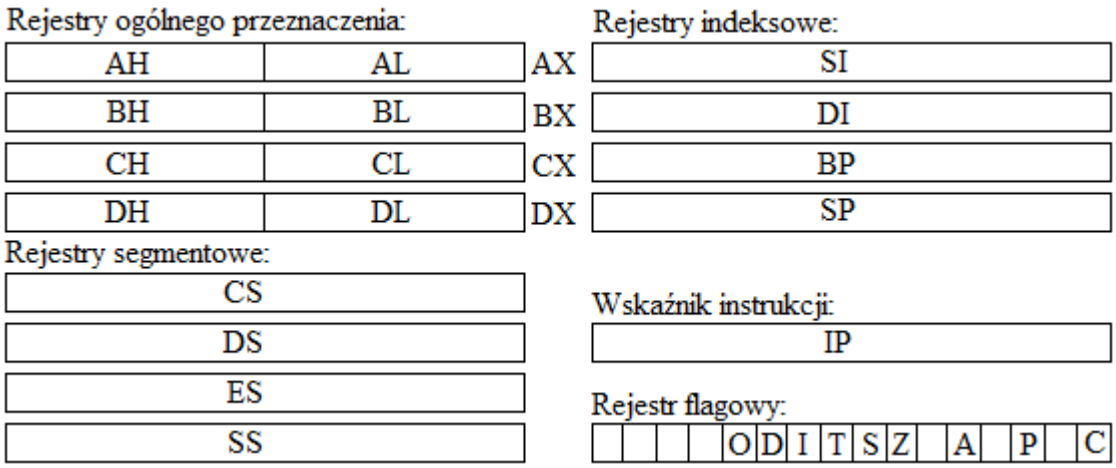
Architektura x86 bazuje na zestawie instrukcji procesora Intel 8086. Nazwa wywodzi się od ostatnich dwóch cyfr następców procesora 8086, czyli 286, 386, 486. W kolejnych seriach nowych procesorów zaszły duże zmiany. Dodano wiele rozszerzeń i zestawów instrukcji, lecz wszystkie kolejne rodziny procesorów są zgodne wstecz do podstawowej wersji swojego praprzodka, procesora Intel 8086.

Obecnie nazwa architektury x86 jest często mylona. Popularnie oznacza kompatybilność do 32 bitowego trybu adresowania. Jeżeli oznacza się kompilację programu x86, program powinien być zgodny z procesorem 8086. Aby poprawnie podkreślić zgodność z 32 bitowym trybem adresowania, można użyć symbolu x86-32 lub .386. Zgodność z procesorami Pentium lub wyższymi powinno się oznaczać .586. 64 bitowy tryb adresowania jest często oznaczany terminem x64 lub x86-64.

Istnieje także termin x87, który oznacza nie zawsze dostępny podzbiór instrukcji zmiennoprzecinkowych x86. Nazwa ta wywodzi się od pierwszych odrębnych koprocesorów zmiennoprzecinkowych, których nazwy kończyły się na „87”. Zestaw tych instrukcji nie był niezbędny do wykonywania programu, lecz znacznie przyspieszał wykonywanie operacji numerycznych, takich jak np. sinus, cosinus lub pierwiastkowanie. Obecnie koprocesory numeryczne są zintegrowane z głównym procesorem.

3.1. PROCESOR INTEL 8086

Procesor ten został wprowadzony na rynek w 1978 roku, jako pierwszy w pełni szesnastobitowy procesor. Jest on następcą procesora 8080 oraz protoplastą wszystkich dzisiejszych procesorów zgodnych z IBM PC. Instrukcje tego procesora stanowią podzbiór instrukcji współczesnych procesorów Intel Core.



Rys. 3.1. Rejestry procesora 8086.

Procesor posiada cztery szesnastobitowe rejestry ogólnego przeznaczenia do których mona się odwoływać jak do ośmiu ośmiobitowych rejestrów. Rejestry te poza ogólnym wykorzystaniem posiadają dodatkowe właściwości. AX jest głównym rejestrem stałoprzecinkowym, BX to rejestr bazowy wykorzystywany jako wskaźnik do danych, CX jest rejestrem licznikowym w programowych operacjach pętli lub operacjach na łańcuchach. DX poza dopełnieniem rejestru AX adresuje układy wejścia – wyjścia. Cztery szesnastobitowe rejestry indeksowe służące do adresowania offsetu. SI oraz DI są wskaźnikami do danych odpowiednio – źródła oraz przeznaczenia w operacji na łańcuchach. Mogą być także wskaźnikami do danych. BP jest wskaźnikiem danych w segmencie stosu, SP jest wskaźnikiem szczytu stosu. Cztery rejestry segmentowe – danych (DS), stosu (SS), kodu programu (CS) i ekstra segment (ES). Rejestr IP wskazuje na aktualnie wykonywaną instrukcję. Wszystkie rejestry są pokazane na rys. 3.1.

Tabela. 3.1. Zestawienie znaczników rejestru flagowego.

symbol znacznika	**polska nazwa znacznika**	**angielska nazwa znacznika**	**typ znacznika**	**opis znacznika**
CF	znacznik przeniesienia	carry flag	stanu	jeden gdy wystąpiło przeniesienie przy dodawaniu lub pożyczka przy odejmowaniu
FF	znacznik parzystości	parity flag	stanu	jeden gdy po zakończeniu operacji liczba bitów w młodszym bajcie jest parzysta

AF	znacznik przeniesienia pomocniczego	auxiliary flag	stanu	jeden gdy wystąpiło przeniesienie przy dodawaniu lub pożyczka przy odejmowaniu między bitami trzy i cztery
ZF	znacznik zera	zero flag	stanu	jeden gdy wynik operacji wynosi zero
SF	znacznik znaku	sign falg	stanu	jeden gdy wynik w najbardziej znaczącym bicie wynosi jeden
OF	znacznik przepełnienia	overflow flag	stanu	jeden gdy wystąpiło przeniesienie lub pożyczka z bitu znaku
TF	znacznik pracy krokowej	trap flag	sterujący	jeden umożliwia tryb pracy krokowej (wykorzystywany przy debuggerach)
IF	znacznik przerwania	interrupt flag	sterujący	jeden odblokowuje system przerwań
DF	znacznik kierunku	direction flag	sterujący	jeden gdy podczas operacji na łańcuchach adresy rosną

W procesorze 8086 występuje też rejestr flagowy, w którym każdy bit jest nazywany znacznikiem i oznacza stan procesora. Rejestr ten jest szesnastobitowy, lecz wykorzystane zostało jedynie dziewięć bitów. Można wyróżnić dwa typy znaczników. Typ stanu jest ustawiany po wykonaniu instrukcji w zależności od wyniku. Od typu sterującego zależy sposób pracy procesora. Znaczniki sterujące wpływają na sposób wykonania instrukcji (np. znacznik kierunku) oraz na zachowanie procesora (znaczniki przerwania i pracy krokowej). Znaczenie wszystkich flag zostało opisane w tabeli 3.1.

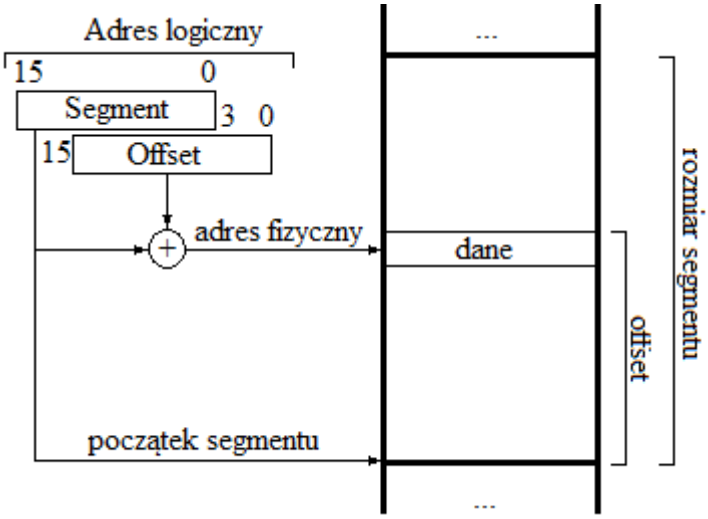
3.2. PRACA W TRYBIE RZECZYWISTYM

Tryb rzeczywisty (ang. real mode) jest jedynym trybem w jakim pracuje procesor Intel 8086. Ten tryb wyróżnia 20 bitowa szyna pamięci, nieograniczony dostęp do całej pamięci, wejścia-wyjścia oraz urządzeń zewnętrznych. Nie wspiera ochrony pamięci oraz wielozadaniowości.

Przestrzeń adresowa pamięci jest segmentowana. Wyróżnia się cztery segmenty: danych, kodu programu, stosu programu oraz ekstra segment danych. Te segmenty adresowane są przez odpowiednie rejestry: DS, CS, SS, ES. Wielkość jednego segmentu wynosi 64kB, co wynika z maksymalnego możliwego offsetu (przesunięcia względem początku segmentu). Kolejne adresy segmentu rozpoczynają się co 16 bajtów, przez co

segmenty mogą nakładać się na siebie oraz dwa różne adresy logiczne mogą być tym samym adresem fizycznym.

Dwudziestobitowy adres fizyczny wyliczany jest poprzez sumę dwóch 16 bitowych adresów segmentu i offsetu. Proces ten przedstawiony jest na rys. 3.2. Aby uzyskać 20 bitowy adres bezpośredni, przy należy przesunąć bitowo adres segmentu w lewo o cztery pozycje (zwolnione bity przyjmują wartość zero) i następnie dodać adres offsetu. Dzięki takiej metodzie dostępu do pamięci można zaadresować 220 bajtów pamięci, co jest równe 1MB.



Rys. 3.2. Tworzenie adresu fizycznego w trybie rzeczywistym.

Adres fizyczny wyliczony z sumy segmentu i offsetu może przekroczyć adres FFFFFh. W takim przypadku obliczane z adresu jest modulo 220. Próba zaadresowania 4 między pierwszym a drugim megabajtem pamięci kończy się zaadresowaniem pamięci między zerowym a pierwszym megabajtem – logiczny adres F800:8000 to fizyczny adres 0000:0000.

Procesory 286 posiadały 24 bitową szynę pamięci i w trybie chronionym mogły poprawnie działać z adresami powyżej FFFFFh. W rezultacie niektóre programy mogły przestać działać. Wprowadzono logiczną bramę na linii adresowej A20, która umożliwiała programowe włączenie lub wyłączenie dodatkowego bitu adresacji. Obecnie po włączeniu zasilania BIOS podczas testowania pamięci włącza adresację linii A20, następnie wyłącza. System operacyjny podczas uruchamiania powrotnie włącza adresację linii A20.

Późniejszy tryb rzeczywisty począwszy od procesorów zgodnych z 386 umożliwia korzystanie z „wielkiego trybu rzeczywistego” (ang. huge real mode). Jest to tryb rzeczywisty z możliwością adresowania 232 pamięci. Polega to na przełączeniu procesora

w tryb chroniony, załadowaniu rejestru GDTR oraz wybranych rejestrów segmentowych selektorami a następnie powrocie do trybu rzeczywistego. Po powrocie procesor korzysta z zapisanych deskryptorów co umożliwia zaadresowanie większego obszaru pamięci.

3.3. PRACA W TRYBIE CHRONIONYM

Tryb chroniony (ang. protected mode) wprowadzony został w architekturze 80286. Posiada on wiele udogodnień, takich jak ochrona pamięci, wielozadaniowość, stronicowanie pamięci, pamięć wirtualna. Funkcje trybu chronionego zwiększają możliwości kontroli aplikacji poprzez system operacyjny.

Zgodnie ze specyfikacją projektową firmy Intel, wszystkie kolejne procesory rodziny x86 miały być ze sobą zgodne wstecz. Począwszy od rodziny 80286, wszystkie procesory, po włączeniu zasilania pracują w trybie rzeczywistym. Mogą być przełączone programowo w tryb chroniony. Obecnie czynność tą w trakcie włączania wykonuje system operacyjny, uniemożliwiając użytkownikowi przełączenie procesora w tryb rzeczywisty.

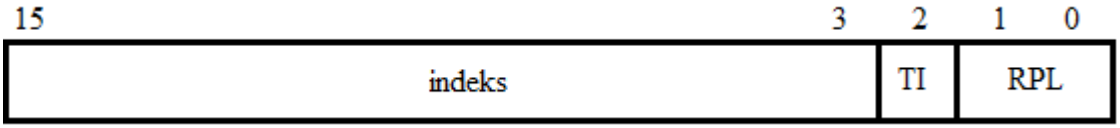
Trybu chronionego w architekturze 80286 nie używano na szeroką skalę. Spowodowane było to głównie brakiem możliwości przełączenia pracy procesora z trybu chronionego w tryb rzeczywisty bez ponownego uruchomienia komputera. Używanie trybu chronionego wymagało również nowych systemów operacyjnych.

Aby nie zniechęcać do używania trybu chronionego, Intel w procesorach 80386 wprowadził kilka zmian. Wydłużono słowo adresowe offsetu do 32 bitów, co odpowiada możliwości zaadresowania 4GB pamięci. Również dzięki tej zmianie cała pamięć jest dostępna bez konieczności przełączania segmentów. Wprowadzono stronicowanie pamięci. Rozwiązano problem przełączenia procesora z powrotem do trybu rzeczywistego bez konieczności ponownego uruchomienia. Dodano tryb wirtualny.

Ochrona pamięci zadania jest podstawowym czynnikiem umożliwiającym wprowadzenie współbieżności. Bez niej, dwa współbieżnie wykonujące się zadania, mogłyby wzajemnie nadpisywać sobie dane. Jest to trudne do wykrycia, ponieważ w nowoczesnych systemach operacyjnych wiele zadań wykonuje się współbieżnie i wskazanie zadania które wykonuje się niepoprawnie byłoby niemożliwe. System ochrony pamięci przydziela każdemu procesowi pewną jej część, do której ma wyłączny dostęp. Próba naruszenia innego obszaru pamięci kończy się błędem ochrony.

3.3.1. ADRESOWANIE PAMIĘCI

W trybie chronionym rejestry segmentowe procesora nie przechowują fizycznych adresów segmentów, lecz selektory - logiczne struktury które pozwalają na wyznaczenie adresu liniowego a następnie fizycznego. Selektor, przedstawiony na rys. 3.3, to 16 bitowa struktura składająca się z trzech pól. Trzynastobitowe pole INDEX określa numer pozycji w tablicy deskryptorów. Pole TI określa której tablicy deskryptorów dotyczy dany selektor (zero dla GDT, jeden dla LDT). RPL określa poziom uprzywilejowania. Tryb użytkownika będzie miał zawsze niższy przywilej od trybu jądra.



Rys. 3.3. Format selektora.

Deskryptor to struktura która opisuje charakterystykę różnych obszarów pamięci używanych podczas wykonywania zadania. Są poukładane w tablice. Można wyróżnić trzy typy tablic deskryptorów:

GDT (ang. Global Descriptor Table) – globalna tablica deskryptorów wykorzystywana przes system operacyjny do realizacji swoich zadań,

LDT (ang. Local Descriptor Table) – lokalna tablica deskryptorów tworzona dla poszczególnego zadań,

IDT (ang. Interrupt Descriptor Table) – tablica deskryptorów przerwań wykorzystywana przez procedury obsługi przerwań.

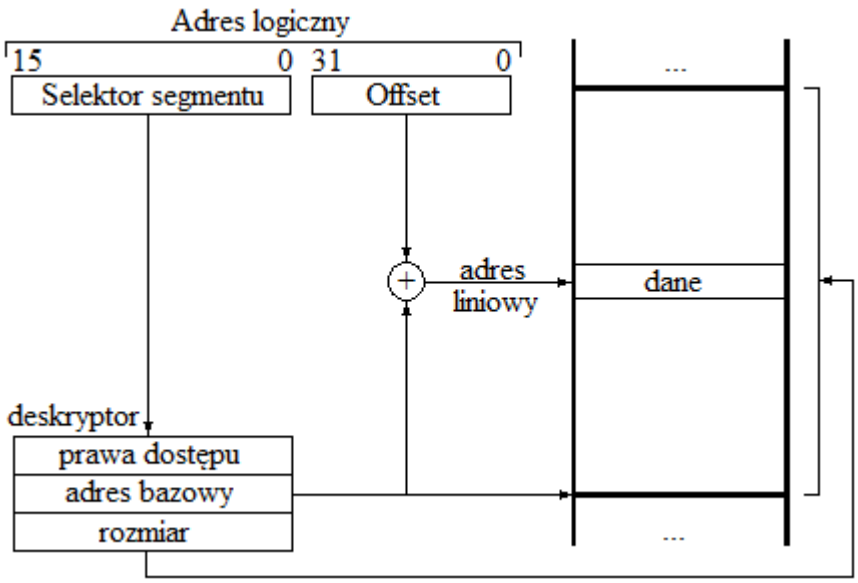
Tablica GDT poza deskryptorami segmentów pamięci wykorzystywanych przez system operacyjny, posiada deskryptory tablic LDT oraz TSS (ang. Task State Segment). Położenie tablicy GDT jest zdefiniowane poprzez liniowy adres w pamięci. Każda tablica LDT posiada swój deskryptor w tablicy GDT określający jej położenie.

Rozmiar deskryptora wynosi 48 bitów. Jego struktura może się różnić w zależności od rodzaju deskryptora. Wyróżnia się deskryptory segmentów programu i danych, które przechowują odpowiedni adres bazowy segmentu, oraz deskryptory systemowe, które przechowują adres bazowy deskryptorów TSS, tablic LTD oraz różnego rodzaju furtek.

Procesory począwszy od 286 posiadają dodatkowe instrukcje oraz rejestry, pozwalające na obsługę pamięci w trybie chronionym. Dwa 48 bitowe rejestry GDTR oraz IDTR zawierają 32 bitowe adresy odpowiednich tablic oraz 16 bitowy ich rozmiar. Rejestr

LDTR zawiera selektor wskazujący w głównej tablicy deskryptorów miejsce deskryptora który zawiera informacje o położeniu oraz rozmiarze lokalnej tablicy deskryptorów aktualnie wykonującego się zadania.

Podczas odwołania się do pamięci w pierwszej kolejności następuje sprawdzenie pola TI w selektorze zapisanym w rejestrze segmentowym. Jeżeli pole to zawiera zero następuje przesunięcie pola INDEX o trzy w lewo (operacja ta jest jednoznaczna z operacją mnożenia przez osiem) i sumowanie z adresem bazowym GDT przechowywanym w rejestrze GDTR. Jeżeli wynik jest mniejszy od rozmiaru tablicy GDT następuje pobranie deskryptora segmentu a następnie wyliczenie adresu liniowego. Gdy pole TI wynosi jeden, adres bazowy tablicy GDT jest sumowany z przesuniętym o trzy w lewo polem INDEX z ukrytej tablicy LDTR. Następnie z tablicy GDT pobierany jest deskryptor tablicy LDT. Przesunięte o trzy w lewo pole INDEX z selektora zapisanego w rejestrze segmentowym jest sumowane z adresem bazowym zapisanym w pobranym deskryptorze tablicy LTD. Wynik jest porównywany z rozmiarem tablicy LDT i jeżeli się wszystko zgadza, pobierany jest deskryptor segmentu i wyliczany zostaje adres liniowy. W obu przypadkach kiedy adres deskryptora przekracza rozmiar tablicy GDT lub LDT generowany jest błąd ochrony pamięci.



Rys. 3.4. Uproszczony proces translacji adresu wirtualnego na adres liniowy.

Adres liniowy jest wyliczany poprzez sumę adresu bazowego segmentu, pobranego z odpowiedniego deskryptora, z adresem offsetu który jest zapisany w odpowiednim

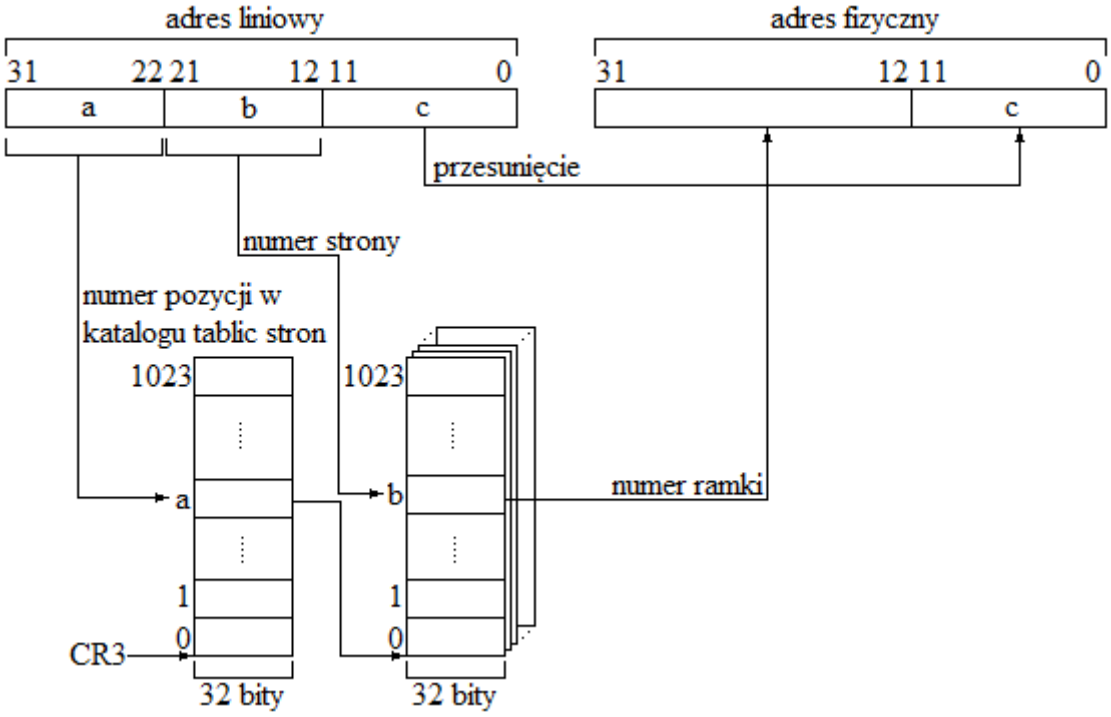
rejestrze adresowym procesora. Uproszczony proces translacji adresu logicznego (wirtualnego) na adres liniowy przedstawiony został na rys. 3.4.

3.3.2. STRONNICOWANIE PAMIĘCI

Jedną z cech współczesnych systemów operacyjnych jest wielozadaniowość. W pamięci przechowywane jednocześnie wiele współbieżnie wykonywanych zadań. Wraz z rozwojem procesorów wielordzeniowych dla komputerów osobistych zadania nie tylko wykonywane są współbieżnie ale też równolegle. Stronicowanie pamięci pozwala na pracę zadania z pamięcią wirtualną. Proces ma wrażenie że pracuje na jednym dużym ciągłym obszarze pamięci, a w rzeczywistości jest ona pofragmentowana, nieciągła i może być częściowo przechowywana w pamięci masowej.

Stronicowanie polega na podzieleniu wirtualnej przestrzeni pamięci na strony i fizycznej przestrzeni pamięci na ramki. Strony i ramki są tego samego rozmiaru i zazwyczaj zajmują 4kB. W pamięci umieszczona jest tablica stron która przypisuje każdemu numerowi strony numer ramki pamięci. Przy rozmiarze strony 4kB tablica stron zawiera 2^{20} pozycji co daje rozmiar 4MB całej tablicy. Tablica stron będąc w pamięci też jest poddana procesowi stronicowania i podzielona na bloki 4kB, więc aby uzyskać dostęp do poszczególnych bloków tablicy stron przewidziano dodatkową strukturę katalogu stron. Katalog stron jest przechowywany pod stałym adresem fizycznym zapisanym w rejestrze CR3, ma rozmiar jednej strony i zawiera adresy fizyczne wszystkich 1024 części tablicy stron.

32 bitowy adres liniowy składa się dziesięciobitowego numeru pozycji w katalogu tablicy stron, dziesięciobitowego numeru strony oraz dwunastobitowego offsetu (przesunięcia) względem początku strony. Zamiana adresu liniowego w adres fizyczny polega na odczytaniu adresu fizycznego tablicy z numeru pozycji w katalogu stron, następnie z tablicy stron pod pozycją numeru strony znajduje się numer ramki. Offset względem numeru strony jest taki sam jak offset wobec numeru ramki więc zostaje przeniesiony do adresu fizycznego bez zmian. Proces stronicowania pamięci został przedstawiony na rysunku 3.5.



Rys. 3.5. Proces stonicowania.

W sytuacji gdy rozmiar wirtualnej przestrzeni adresowej przekracza rozmiar zainstalowanej pamięci fizycznej, system operacyjny wymienia strony między pamięcią operacyjną a pamięcią dyskową. Realizuje to dzięki zapisanym w tablicy stron znacznikom: obecność strony w pamięci, modyfikacja strony oraz odwołanie do strony. W każdej pozycji tablicy stron znajdują się również informacje o tym czy ramka jest zajętaoraz do której strony i jakiego procesu lub procesów należy.

3.3.3. TRYB WIRTUALNY

Tryb wirtualny, zaimplementowany w procesorach 386 i nowszych, pozwala na uruchamianie aplikacji przeznaczonych dla trybu rzeczywistego w trybie chronionym. Tryb ten emuluje szesnastobitowe segmentowe adresowanie pamięci. Przerwania z trybu wirtualnego mogą być obsługiwane przez emulację w systemie operacyjnym lub przez przełączanie na czas wykonania przerwania w tryb rzeczywisty. Tryb wirtualny nie jest w pełni zgodny z trybem rzeczywistym. Nie można w tym trybie przełączyć się do trybu chronionego.

4. JĘZYK ASEMBLER WPROWADZENIE

Język asemblera jest językiem najniższego poziomu tłumaczonym w procesie asemblacji (składania) na kod maszynowy. Pierwsze asemblery (programy asemblujące, składające) powstały w latach pięćdziesiątych dwudziestego wieku. Związany jest on ściśle z architekturą sprzętową z którą jest zgodny. Jest używany do pisania jąder systemów operacyjnych, tworzenia sterowników urządzeń oraz mikrokontrolerów, wykorzystywany przy odpluskwianiu w językach wysokiego poziomu, używany w językach wysokiego poziomu do optymalizacji, wykorzystywany podczas deasemblacji. Wiele zastosowań tego języka jest powodem dla którego powinno się znać jego podstawy. Znajomość tego języka powoduje lepsze zrozumienie działania architektury komputera. Jeden rozkaz w języku asemblera odpowiada dokładnie jednej instrukcji w kodzie maszynowym. Zamiana języka asemblera w kod maszynowy jest odwzorowaniem jeden do jednego. Powstał on poprzez zamianę kodu maszynowego na trzy lub czteroliterowe mnemoniki.

4.1. INSTRUKCJE JĘZYKA ASEMBLER

Wszystkie instrukcje asemblera można podzielić na klika różnych kategorii. Podstawowym podziałem jest grupowanie według rodzaju przeznaczenia instrukcji. Można wyróżnić: rozkazy transmisji danych, instrukcje arytmetyczno-logiczne, pętle i skoki oraz operacje trybu chronionego.

Głównym zadaniem rozkazów transmisji danych są operacje na rejestrach i pamięci. Podstawowym rozkazem z tej grupy jest MOV. Odpowiada on za przesyłanie danych. Inne rozkazy odpowiadają za operacje ładowania adresu efektywnego do rejestru,

operacje na stosie, operacje na łańcuchach danych, ustawianie wartości bajtów. Do grupy tej należą także rozkazy wejścia-wyjścia, które odpowiadają za dostęp do portów.

Instrukcje arytmetyczno-logiczne to grupa instrukcji umożliwiających arytmetykę stałoprzecinkową oraz operacje logiczne. Instrukcje arytmetyczne mogą wyznaczać wartości wyrażeń takich jak dodawanie, odejmowanie, mnożenie i dzielenie. Operują na różnych kodach – binarnym, U2, BCD. Operacje logiczne wyznaczają wartości wyrażeń logicznych takich jak suma i alternatywa logiczna, alternatywa wykluczająca oraz negacja. W grupie wyrażeń logicznych znajdują się także wyrażenia przesunięć bitowych oraz testowania bitów.

Pętle i skoki umożliwiają sterowanie wykonaniem programu. Skoki warunkowe są wykonywane na podstawie wyniku porównania lub po sprawdzeniu stanu rejestru flagowego. Specyficznymi rodzajami skoków są CALL – skok do procedury oraz INT – skok do procedury obsługi przerwań programowych. Instrukcja LOOP umożliwia tworzenie pętli.

Tabela. 4.1. Instrukcje trybu chronionego.

nazwa instrukcji	**opis instrukcji**
ARPL	Zmiana uprzywilejowania selektora.
CLTS	Sprawdzenie znacznika przełączenia zadania.
LAR	Pobranie praw dostępu selektora.
LGTD	Załadowanie rejestru globalnej tablicy deskryptorów.
LIDT	Załadowanie rejestru tablicy deskryptorów przerwań.
LLDT	Załadowanie rejestru lokalnej tablicy deskryptorów.
LMSW	Załadowanie nowej wartości do rejestru stanu słowa procesora.
LSL	Pobranie do rejestru długości segmentu.
LTR	Załadowanie rejestru zadania.
SGTD	Pobranie do pamięci rejestru globalnej tablicy deskryptorów.
SIDT	Pobranie do pamięci rejestru tablicy deskryptorów przerwań.
SLDT	Pobranie do pamięci rejestru lokalnej tablicy deskryptorów.
SMSW	Pobranie rejestru stanu słowa procesora.
STR	Pobranie rejestru wartości zadania.
VERR	Weryfikacja segmentu do odczytu.
VERW	Weryfikacja segmentu do zapisu.

Operacje trybu chronionego pozwalają zarządzać tablicami GDT oraz LDT, sprawdzać stany selektorów oraz sterować znacznikami słowa procesora. Wszystkie instrukcje dostępne po wprowadzeniu trybu chronionego zawarte są w tabeli 4.1.

Innym podziałem jest rozróżnienie instrukcji według podawanych argumentów. Istnieją instrukcje bezargumentowe (np. cli), jednoargumentowe (np. mul) lub dwuargumentowe (np. mov).

4.2. ADRESACJA

Adresowanie argumentów jest przydatne, kiedy należy przesłać dane między rejestrami lub między rejestrem a pamięcią. Istnieje kilka podstawowych sposobów na adresowanie, mogą być one łączone lub rozszerzane.

Adresowanie natychmiastowe powoduje załadowanie rejestru stałą. Bezpośrednie adresowanie ładuje rejestr daną z pamięci, której adres zawarty jest bezpośrednio w rozkazie i nie trzeba go wyliczać. Adresowanie indeksowe wykorzystuje rejestry SI oraz DI do wskazania adresu danej. Ładowana dana znajduje się pod adresem wskazywanym przez rejestr. Adresowanie bazowe wykorzystuje rejestry BX oraz BP i działa podobnie do adresowania indeksowego.

Adresowanie bazowe i indeksowe można rozszerzyć o przemieszczenie. Polega to na dodaniu do adresu przechowywanego w rejestrze stałej. W takim przypadku adres wyliczany jest poprzez sumę adresu zawartego w rejestrze i stałej. Analogiczne do przemieszczenia jest skalowanie – suma stałej zamieniana jest na iloczyn. Skalowanie i przemieszczenie można łączyć. Tryb adresowania bazowy indeksowany wykorzystuje sumę lub różnicę rejestrów bazowych i indeksowanych do wyliczenia adresu. Tryb ten można łączyć z przemieszczeniem oraz skalowaniem.

4.3. PROCEDURY

Procedury to podprogramy które mogą być wielokrotnie wykonywane z różnych fragmentów programu. Główną ich zaletą jest przenoszenie między programami oraz oszczędność pamięci.

W trybie rzeczywistym procedury systemowe są wywoływane poprzez przerwanie programowe. Parametry są przekazywane w rejestrach w zależności od spełnianej funkcji. Numer procedury przerwania 21h przekazuje się poprzez rejestr AH. Parametry zwracane

są przez rejestry. Procedury użytkownika mogą przyjmować parametry – przez dane globalne, stos lub rejestry. Jednak większość podprogramów w trybie rzeczywistym pobiera parametry przez rejestry i dane globalne zachowując podobieństwo obsługi do przerwań programowych.

W trybie chronionym można przekazywać zmienne do procedury przez rejestry, lecz się tego nie stosuje, ponieważ ilość rejestrów jest ograniczona. Wszystkie zmienne przekazywane są przez stos – w podprogramie pobiera się je za pomocą adresu dna stosu przechowywanego w rejestrze EBP. Dodatkowo często spotyka się deklarowanie dodatkowych zmiennych lokalnych. Umieszcza się je również na stosie, a dostęp do nich jest podobny jak do parametrów. Wszystkie funkcje systemowe zwracają wynik w rejestrze EAX lub zapisują pod adresy wskaźników, które przekazuje się poprzez parametry funkcji.

5. NARZĘDZIA

W tym rozdziale przedstawiono wykorzystywane narzędzia których użyto do zrealizowania celu pracy.

5.1 MASM 10

MASM 10 jest kompletnym darmowym środowiskiem programistycznym stworzonym do pisania programów w 32 bitowym asemblerze. Instalacja jest w pełni automatyczna i dostarcza komponenty takie jak:

- kompilator asemblera, konsolidator oraz kompilator zasobów
- edytor
- pliki nagłówkowe
- przykładowe kody źródłowe
- pliki pomocy

Środowisko to zawiera rozbudowany zestaw makroinstrukcji i dyrektyw. Umożliwiają one tworzenie takich obiektów jak struktury i unie które są często wymagane jako parametry funkcji systemowych. Dzięki dyrektywie INVOKE przekazanie parametrów do funkcji systemowych jest bardziej czytelne – nie trzeba umieszczać ich ręcznie na stosie. Zaletą tego pakietu jest możliwość użycia podstawowych konstrukcji języków wysokiego poziomu takich jak warunki oraz pętle.

MASM 10 wymaga systemu operacyjnego Windows 2000 lub nowszego. Pakiet instalacyjny można pobrać z sieci Internet. Znajduje się on pod adresem [9].

5.2 WINDOWS API

Winows API (ang. Application Programming Interface) jest interfejsem programowania aplikacji zawierającym zbiór bibliotek systemu operacyjnego Windows. Biblioteki te pozwalają tworzyć programy odpowiednio dla systemu operacyjnego.

Interfejs ten jest niezbędny dla programów działających w trybie chronionym pod kontrolą systemu operacyjnego Windows. Za pomocą swoich mechanizmów umożliwia tworzenie elementów specyficznych dla tego systemu.

Cała dokumentacja jest darmowa dostępna w sieci Internet na platformie MSDN (ang. Microsoft Developers Network Library) pod adresem [8].

5.3 EDYTOR ZASOBÓW

Zasoby programu to pliki programów systemu Windows opisujące elementy okna np. menu, przyciski, czcionki, kursory, bitmapy. Nowoczesne programy powinny używać plików zasobów ponieważ umożliwiają one prostą zmianę wyglądu okna, zmianę języka programu jedynie przez edycję pliku zasobów. Wprowadzają też modułowość i podział programu na część typowo funkcjonalną oraz część graficzną.

Do prostego tworzenia plików zasobów wykorzystano program ResEdit. Umożliwia on utworzenie okna jak w typowych narzędziach programowania RAD. Plik można zapisać w formacie res. Po zmianie bibliotek i stałych plik zasobów jest gotowy do kompilacji i konsolidacji i programem głównym. Program jest darmowy i dostępny w sieci Internet pod adresem [10].

5.4 DEV C++

Dev C++ jest to darmowe środowisko zintegrowane z kompilatorem MinGW. MinGW jest darmowym portem zestawu kompilatorów GCC. Projekt nie jest już rozwijany, lecz rozwijana jest wersja wxDev C++ która umożliwia programowanie RAD.

Podstawowym językiem programowania w tym środowisku jest C++. Jest możliwość wstawiania kodu asemblera. Dodatkowym atutem jest możliwość przełączenia wstawek asemblerowym między składniami AT&T oraz Intel.

6. ĆWICZENIA LABOTATORYJNE

W tym rozdziale przedstawiono przygotowane ćwiczenia laboratoryjne, ich zadania oraz sposób wykonania.

Głównym założeniem wszystkich ćwiczeń jest podział na przykład oraz zadanie. Przykład ma zawierać podstawowe elementy zadania i być możliwie prosty. Przykładów może być kilka jeśli zadanie ma większy stopień złożoności. Zadanie jest oparte o przykład i pewną dodatkową funkcjonalność lub możliwość która jest celem zadania i należy ją zaimplementować. Kolejność ćwiczeń wynika ze stopnia trudności oraz odwołań do zadań poprzednich.

Wszystkie programy używają biblioteki masm32rt.inc która dołącza do programu w odpowiedniej kolejności biblioteki WinAPI. Jest to ułatwienie pozwalające uniknąć błędów konsolidacji, pozwalające bezpośrednio przejść do kodu programu.

6.1. OBSŁUGA KONSOLI

Każdy proces systemowy może wykonywać się w tle, utworzyć konsolę lub okno graficzne. Proces może posiadać wyłącznie jedną konsolę – nawet jeśli jest złożony z kilku wątków. Utworzenie konsoli wykonuje bezargumentowa funkcja AllocConsole. Aby prawidłowo zakończyć proces, konsolę należy zwolnić funkcją FreeConsole.

Podstawową jednostką do identyfikacji różnego rodzaju obiektów w systemie Windows jest uchwyt. Jest to czterobajtowa struktura zawierająca wskaźnik do obiektu. Wskaźnik ten jest wykorzystywany podczas odwołań do obiektu który wskazuje.

Proces posiadając przygotowaną konsolę powinien pobrać uchwyt wejścia-wyjścia, który jest wymagany do wyświetlania informacji w konsoli. Funkcja GetStdHandle

pobiera uchwyt i zwraca w rejestrze eax. Aby funkcja działała poprawnie parametrem jest STD_OUTPUT_HANDLE czyli standardowe wyjście.

Z programami konsolowymi ściśle związana jest linia poleceń. Dzięki niej możemy przekazywać do programu dowolne parametry. Każdy parametr powinien być oddzielony spacją. Jeżeli parametrem jest tekst zawierający spację wymagany jest cudzysłów. Aby je odczytać można skorzystać z funkcji systemowych GetCommandLineW oraz CommandLineToArgvW. Pierwsza z nich zwraca wskaźnik na łańcuch linii poleceń. Druga z nich zamienia ten łańcuch na dwie zmienne: ilość parametrów i wskaźnik do tablicy zawierającej wskaźniki do kolejnych parametrów. Otrzymujemy dwie zmienne znane z języka C++ gdzie dane te są parametrami funkcji main() argc oraz argv[]. Przykład odczytania parametrów przedstawia rys. 6.1.

```
INVOKE GetCommandLineW
INVOKE CommandLineToArgvW,eax,OFFSET argc
mov argv,eax
mov edi, argv
xor ecx, ecx

@p1:
   mov eax,[edi]
   mov ebx, eax
   add edi, 4

   push ecx
   INVOKE lstrlenW, eax
   INVOKE WriteConsoleW, hInstance, ebx, eax,OFFSET tmpV, NULL
   INVOKE WriteConsole, hInstance, OFFSET cr, 2,OFFSET tmpV, NULL
   pop ecx
   inc ecx

cmp ecx, argc
jne @p1
```

Rys. 6.1. Kod źródłowy odczytujący parametry i wyświetlający je w konsoli.

W programach korzystających z konsoli często używa się operacji na plikach. Po otwarciu pliku chcąc odczytać jego zawartość potrzebujemy wolnej pamięci o rozmiarze identycznym z ilością odczytywanych danych.

Najprostszym rozwiązaniem jest utworzenie pustej tablicy o stałym rozmiarze. Rozwiązanie to jest nieoptymalne – plik lub ilość danych może być mniejsza od bufora i część pamięci zostanie niewykorzystana. Możliwe jest też przekroczenie bufora kiedy rozmiar odczytywanych danych jest większy od bufora.

Rozwiązanie optymalne to dynamiczna alokacja dodatkowej pamięci w zależności od rozmiaru odczytywanych danych. Do tego celu wykorzystano funkcję VirtualAlloc. Funkcja ta rezerwuje fizyczną przestrzeń w pamięci operacyjnej lub pliku wymiany jeżeli rodzaj alokacji ustawiony jest na MEM_COMMIT. Zwracany jest adres bazowy zarezerwowanego regionu pamięci. Należy pamiętać o zwolnieniu pamięci funkcją VirtualFree.

Otwarcie pliku jest możliwe dzięki funkcji CreateFile. Funkcja zwraca uchwyt który jest używany przy dostępie do pliku lub urządzenia wejścia-wyjścia w zależności od parametrów. Najważniejsze parametry to nazwa pliku oraz tryb otwarcia który określa czy ma zostać utworzony nowy plik czy otworzyć istniejący.

Po otwarciu pliku sprawdzono rozmiar poprzez funkcję GetFileSize której głównym parametrem jest uchwyt otwartego pliku. Następnie odczytano zawartość pliku do bufora dynamicznego. Przykład przedstawiono na rys. 6.2.

```
INVOKE CreateFile,ADDR fName, GENERIC_READ, 0, NULL,
OPEN_EXISTING,FILE_ATTRIBUTE_ARCHIVE,0
mov hFile, eax

.IF eax==INVALID_HANDLE_VALUE
    INVOKE lstrlen, OFFSET txt2
    INVOKE WriteConsole, hInstance, OFFSET txt2, eax,ADDR tmpV, NULL
    jmp @end
.ENDIF

INVOKE GetFileSize,hFile,0
mov fSize, eax

INVOKE VirtualAlloc,0,eax,MEM_COMMIT,PAGE_READWRITE
.IF eax == NULL
    invoke lstrlen, OFFSET txt4
    invoke WriteConsole, hInstance,OFFSET txt4, eax,OFFSET tmpV, NULL
.ELSE
    mov hBuff, eax
    invoke ReadFile, hFile, eax, fSize, OFFSET readBytes, 0
     . IF eax == -1
         invoke lstrlen, OFFSET txt3
         invoke WriteConsole, hInstance,OFFSET txt3, eax,ADDR tmpV, NULL
     . ELSE
         invoke WriteConsoleW, hInstance, hBuff, fSize,OFFSET tmpV, NULL
      . ENDIF
.ENDIF
INVOKE VirtualFree,eax,OFFSET fSize,MEM_RELEASE
INVOKE CloseHandle, File
```

Rys. 6.2. Przykład odczytania pliku do bufora.

Ćwiczeniem do wykonania jest napisanie programu, który w zależności od podanego parametru wyświetlałby plik w oknie konsoli, lub zapisywał do pliku. Wymagana jest obsługa błędów przy nieprawidłowej liczbie parametrów lub przy nieprawidłowym formacie parametrów.

Celem ćwiczenia jest wprowadzenie do programowania w trybie chronionym na przykładzie programu konsolowego. Nauczenie obsługi i wywołań funkcji systemowych oraz używania odpowiednich dyrektyw. Operacje obsługi parametrów linii poleceń, obsługa plików i alokacja pamięci to podstawowe zagadnienia programowania systemowego które są uniwersalne i mogą zostać wykorzystane w językach wysokiego poziomu.

6.2. WPROWADZENIE DO OKIEN

Tworzenie programu z podstawowym interfejsem graficznym wymaga zupełnie innego podejścia. Wszystkie zdarzenia dziejące się w oknie programu obsługiwane są przez komunikaty. Każdy komunikat wywołuje specjalną procedurę obsługi okna. W zależności od rodzaju komunikatu procedura obsługi okna może przedsięwziąć różne działania. Procedura przyjmuje zawsze cztery parametry: hDlg, uMsg, wParam, lParam. Przykładową konstrukcję procedury obsługi zdarzeń zawiera rys. 6.3. Pierwszy jest uchwytem do obsługiwanego okna, drugi zawiera wiadomość. Dwa ostatnie przenoszą dodatkowe informacje o wiadomości. Przykładowo dla wartości uMsg=WM_COMMAND wiemy że komunikat dotyczy zdarzenia pewnej kontrolki. Identyfikator kontrolki przenosi parametr wParam.

```
DlgProc proc hDlg,uMsg,wParam,lParam:DWORD
     pushad

     popad
     xor eax,eax
   ret
DlgProc endp
```

Rys. 6.3. Konstrukcja procedury obsługi zdarzeń.

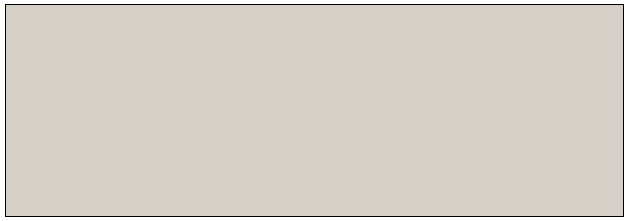
Okno programu możemy wywołać na dwa sposoby. Funkcją CreateWindowEx ma nieokreśloną liczbę parametrów oraz wymaga rejestracji klas standardowego okna. Oprócz standardowego wywołania okna może także tworzyć elementy potomne np. przyciski, pola

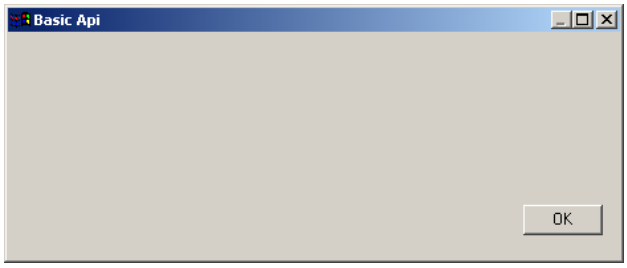
edycji, etykiety. Alternatywą jest DialogBoxParam tworząca modalne okno dialogowe. Posiada stałą liczbę parametrów. Zaletą jest tworzenie okna na podstawie pliku zasobów określającego wszelkie jego ustawienia. Jedyną wadą jest modalność – brak możliwości tworzenia wielu okien dla jednego programu. Aby uprościć strukturę kodu oraz podzielić program na część widoku i kontrolera użyto DialogBoxParam.

Plik zasobów jest plikiem tekstowym .rc zawierającym ustawienia oraz elementy okna. Podstawowe okno zawiera tytuł, definicję czcionki oraz parametrów takich jak np. przyciski minimalizacji, maksymalizacji, paski przewijania. Przykład zależności wyglądu okna od zawartości pliku .rc przedstawia tabela 6.1.

Tabela. 6.1. Przykład wyglądu okna w zależności od określonych zasobów.

zawartość pliku .rc	wygląd okna

`ID_DLG_MAIN DIALOG 108, 70, 308, 105`
`{`
 	
`}`
	

`ID_DLG_MAIN DIALOG 108, 70, 308, 105`
`STYLE WS_VISIBLE | WS_CAPTION |`
`WS_MAXIMIZEBOX | WS_SYSMENU`
`CAPTION "Basic Api"`
`FONT 8, "Helv"`
`{`
`  PUSHBUTTON "OK", ID_PB_OK, 259, 80,`
`40, 14`
`}`
	

Opis okna DIALOG rozpoczyna się od ID_DLG_MAIN. Jest to stała umieszczona na początku pliku zawierająca unikalny numer identyfikacyjny. Numer ten jest używany podczas obsługi komunikatów. Następnie podaje się pozycję i rozmiar okna. Element STYLE definiuje podstawowe zasoby okna jak pasek tytułu, widzialność, menu systemowe oraz przycisk maksymalizacji. CAPTION zawiera tytuł okna a FONT rozmiar i styl czcionki.

Plik zasobów zawiera także opis i umieszczenie wszystkich elementów okna. W tabeli 6.1 przedstawiono sposób definicji przycisku. Pierwszy parametr to etykieta przycisku, następnie ID_PB_OK - identyfikator elementu. Ostatnie cztery zmienne odpowiadają położeniu w oknie oraz rozmiarom przycisku.

Przykładowy program zawiera okno z trzema elementami. Po naciśnięciu przycisku zawartość pola edycyjnego jest kopiowana do etykiety. Aby uzyskać taki rezultat należy sprawdzić w procedurze obsługi okna czy otrzymano komunikat dotyczący zdarzenia kontrolki i czy jest nią przycisk OK co przedstawia rys. 6.4.

```
.IF uMsg==WM_COMMAND
   .IF wParam==ID_PB_OK
       INVOKE GetDlgItemText, hDlg, ID_EDIT, OFFSET buff1, SIZEOF buff1
       INVOKE SetDlgItemText, hDlg, ID_LTEXT, OFFSET buff1
   .ENDIF
.ENDIF
```

Rys. 6.4. Obsługa komunikatu naciśnięcia przycisku.

Funkcja GetDlgItemText pobiera tekst z pola edycyjnego poprzez odwołanie do numeru identyfikującego kontrolkę. Tekst zapisywany jest do bufora i następnie w analogiczny sposób zapisywany w etykiecie za pomocą funkcji SetDlgItemText.

Ćwiczenie polega na modyfikacji przykładu. Należy dodać trzy kontrolki – pole edycyjne, przycisk i pole wyboru. Po naciśnięciu przycisku program ma sprawdzać stan pola wyboru. Jeżeli fałsz należy pobrać z pola edycyjnego nazwę pliku, otworzyć go i wyświetlić w innym polu edycyjnym. Jeżeli prawda należy pobrać z pola edycyjnego nazwę pliku, utworzyć go i zapisać do niego zawartość innego pola edycyjnego.

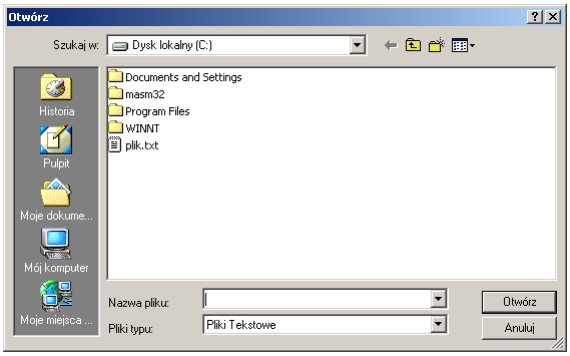
Jedynym elementem nie zawartym w przykładzie jest pole wyboru. Wstawienie pola wyboru do pliku zasobów należy wykonać za pomocą edytora zasobów ResEdit. Stan przycisku można sprawdzić funkcją IsDlgButtonChecked która zwraca BST_CHECKED w przypadku prawdy i BST_UNCHECKED w przypadku fałszu. Dodatkowo należy zaprogramować przycisk zamykający program

Celem ćwiczenia jest zrozumienie obsługi komunikatów które mówią o zdarzeniach dotyczących okna. Zorientowanie się w programie ResEdit oraz podstawowej obsłudze plików zasobów (kompilacja, konsolidacja). Należy umieć uruchamiać okno programu.

6.3. STRUKTURA OPENFILENAME

Struktura OPENFILENAME jest parametrem funkcji GetOpenFileName i GetSaveFileName służących do wyświetlenia dialogu z użytkownikiem wskazującego

ścieżki dostępu do pliku. Struktura ta umożliwia ustawienie opcji dialogu takich jak np. filtry do plików, ścieżka domyślna, etykieta. Przykład dialogu zawiera rysunek 6.5.



Rys. 6.5. Przykład dialogu z wykorzystaniem funkcji GetOpenFileName.

Aby wykorzystać funkcję inicjacji dialogu należy wcześniej odpowiednio zainicjalizować strukturę. Pola hwndOwner oraz hInstance to uchwyty do okna zawierającego dialog oraz do pamięci zawierające szablon okna. Kolejne cztery pola odpowiadają ustawieniom filtrów plików oraz niestandardowym filtrom użytkownika. Następnie należy ustawić rozmiary i adresy buforów ścieżki i edycji nazwy pliku. lpstrInitialDir odpowiada za katalog domyślny po otwarciu dialogu a lpstrTitle to tytuł okna dialogu. Następnie pole z flagami i adresy do zmiennych wskaźnikowych zawierających odpowiednie przesunięcia w ścieżce do pliku. Dalsze zmienne to bufor rozszerzenia pliku, ustawienia procedury przeciągania pliku, adres pliku zasobów i pola zarezerwowane. Na rysunku 6.6 zaprezentowano przykład inicjalizacji struktury OPENFILENAME.

```
mov oofn.lStructSize,SIZEOF oofn
push hDlg
pop oofn.hwndOwner
push hInstance
pop oofn.hInstance
mov oofn.lpstrFilter,OFFSET ofilter
mov oofn.lpstrCustomFilter,OFFSET ocustFilter
mov oofn.nMaxCustFilter,SIZEOF ocustFilter
mov oofn.nFilterIndex,0
mov oofn.lpstrFile,OFFSET ofname
mov oofn.nMaxFile,256
mov oofn.lpstrFileTitle,OFFSET oftitle
mov oofn.nMaxFileTitle,SIZEOF oftitle
mov oofn.lpstrInitialDir,OFFSET oinitDir
mov oofn.lpstrTitle,OFFSET oDlgTitle
mov oofn.Flags,OFN_ENABLESIZING OR \
                        OFN_EXPLORER OR \
```

```
                        OFN_FORCESHOWHIDDEN OR \
                        OFN_PATHMUSTEXIST OR \
                        OFN_OVERWRITEPROMPT OR \
                        OFN_HIDEREADONLY OR \
                        OFN_FILEMUSTEXIST OR \
                        OFN_NODEREFERENCELINKS


mov oofn.nFileOffset,0
mov oofn.lpfnHook,NULL
mov oofn.lpTemplateName,NULL
```

Rys. 6.6. Inicjalizacja struktury OPENFILENAME.

Dostęp do odpowiedniego pola struktury uzyskuje się po podaniu zmiennej z nazwą struktury, następnie kropka i nazwa pola struktury.

Program przykładowy po wskazaniu pliku poprzez dialog otwiera go i zapisuje zawartość do pola edycji. Zadaniem do zrealizowania jest zmiana przykładu w taki sposób, aby poprzez dialog była możliwość zapisu do pliku zawartości pola edycji. Dodatkowo należy ustawić filtry pliku na rozszerzenia txt, asm oraz c.

Celem ćwiczenia jest poznanie obsługi struktur pakietu NASM 10. Dialog otwierający i zapisujący pliki jest często używany i należy znać jego obsługę oraz podstawowe parametry struktury OPENFILENAME.

6.4. APLIKACJA WIELOWĄTKOWA

Wątek jest szczególną odmianą procesu, istniejącą w obrębie wykonywanego zadania. Podstawową różnicą między procesem i wątkiem jest współdzielenie wszystkich przestrzeni adresowych oraz struktur systemowych. Wątek może posiadać dane lokalne niedostępne dla innych wątków. Wątek uruchamia procedurę która wykonuje się współbieżnie względem innych wątków. Stwarza to problem kolejności wykonywania programu oraz dostępu do wspólnych danych czy potoków. Rozwiązaniem tego problemu jest synchronizacja wątków poprzez semafory, muteksy czy sekcje krytyczne.

Po wprowadzeniu na rynek komputerów osobistych procesorów wielordzeniowych programowanie wielowątkowe nabiera coraz większego znaczenia. Proces wykonujący obliczenia który jest jednowątkowy nie wykorzysta możliwości procesora wielordzeniowego. Najlepszym rozwiązaniem dla procesorów n-rdzeniowych jest wykorzystanie n wątków w procesie, choć nie zawsze jest to możliwe ze względu na wykorzystywany algorytm.

Aby utworzyć wątek należy stworzyć procedurę. Następnie adres procedury przekazywany jest do funkcji CreateThread. Najlepszym przykładem pokazującym współbieżne wykonywanie się wątków jest uruchomienie dwóch wątków w obrębie procesu, które wyświetlają napis z różnym opóźnieniem. Przykład procedury takiego wątku zawiera rys. 6.7.

```
Thread1 proc hIns:DWORD
mov ecx, 5
    p1:
       push ecx
       INVOKE WriteConsole, hInstance, OFFSET txt2, 3,ADDR tmpV, NULL
       INVOKE Sleep, 500
       pop ecx
    loop p1

    INVOKE SetEvent, hEvent1
ret
Thread1 endp
```

Rys. 6.7. Przykład procedury uruchamianej jako wątek.

W przykładzie uruchomione zostają dwa wątki. Wyświetlają one liczby jeden oraz dwa z różnym opóźnieniem. Wątek główny jest zsynchronizowany z wątkami potomnymi poprzez zdarzenia i czeka na ich zakończenie. Synchronizację osiągnięto funkcją WaitForMultipleObjects.

Zadaniem jest podzielone na dwie części. W pierwszej z nich należy zapoznać się dokładnie z obsługą i synchronizacją wątków poprzez dodanie kolejnych. W drugiej części każdy wątek oblicza kilka pierwiastków kwadratowych wykorzystując metodę Newtona-Raphsona.

$$\begin{cases} x_1 = \dfrac{a}{b} \\ x_n = \dfrac{1}{b}\left(x_{n-1} + \dfrac{a}{x_{n-1}}\right) \end{cases}$$

gdzie: a – liczba pierwiastkowana

b – stopień pierwiastka

Każdy pierwiastek powinien być wyliczony jak najdokładniej wykorzystując maksymalną liczbę iteracji, równocześnie nie przekraczając czasu około dwóch sekund. Obliczenia należy wykonać używając koprocesora, lecz zgodnie ze wzorem pomijając instrukcję fsqrt. Wątki należy podzielić tak aby jeden wykonywał pierwiastki parzyste a

drugi nieparzyste. Prawidłowo napisany program wykona się dwukrotnie szybciej na procesorze wielordzeniowym. Wyniki pierwiastkowania należy przedstawić w konsoli z dokładnością do 0.0001.

Celem ćwiczenia jest zapoznanie się z tworzeniem i synchronizacją wątków oraz pokazaniem korzyści wynikających z ich użycia. Zadanie rozwija umiejętności implementacji algorytmów matematycznych oraz wymusza stworzenie algorytmu wyświetlającego liczby w postaci zmiennoprzecinkowej. Program wymusza korzystanie z koprocesora jednocześnie przypominając jego obsługę.

6.5. APLIKACJA SIECIOWA

Interfejs sieci TCP/IP definiuje standard gniazd systemu Windows. Aby dwie aplikacje mogły się komunikować, muszą być podłączone do wspólnego gniazda. Aplikacje wykorzystują protokół IPv4 oraz model komunikacji P2P (ang. peer to peer – każdy z każdym).

Gniazdo sieciowe (ang. socjet) jest abstrakcyjnym pojęciem reprezentujący dwukierunkowy punkt końcowy urządzenia. Gniazdo składa się typu protokołu wykorzystywanego do wymiany danych, adresu sieciowego oraz numeru portu identyfikującego aplikację końcową, która wymienia dane.

Korzystanie z funkcji gniazd sieciowych wymaga dołączenia biblioteki ws2_32lib.dll. Niezbędna jest również inicjalizacja poprzez funkcję WSAStartup. Funkcja przyjmuje dwa parametry. Pierwszy z nich to specjalny numer wersji biblioteki. Drugim parametrem jest struktura WSADATA w której zapisywane są informacje o obsługiwanych gniazdach.

Gniazdo sieciowe tworzy funkcja socket. Funkcja ta pobiera trzy parametry – zestaw protokołów, tryb interakcji oraz protokół warstwy transportowej. Zestaw protokołów TCP/IP określa stała AF_INET. Tryb interakcji określa połączeniowy lub bezpołączeniowy sposób komunikacji.

Połączenie TCP ustanawia funkcja connect. Parametrami są wcześniej utworzony deskryptor gniazda oraz adres i rozmiar struktury sockaddr_in, zawierającej docelowy adres IP oraz port. Przed ustanowieniem połączenia należy odpowiednio wypełnić tą strukturę. Szesnastobitowy całkowitoliczbowy numer portu jest konwertowany na format WSAAPI poprzez funkcję htons. Adres IP hosta źródłowego można uzyskać poprzez funkcję gethostbyname, która jako parametr przyjmuje nazwę sieciową docelowego

komputera. Jeśli uda się odnaleźć podany komputer zwracany jest wskaźnik na strukturę hostent w której pole h_list jest podwójnym wskaźnikiem na tablicę adresów komputera. Podobnie działa funkcja gethostbyaddr. Należy podać trzy parametry – wskaźnik na bufor zawierający adres IP, długość adresu oraz określić rodzaj adresu stałą AF_INET.

Za odbiór i wysyłanie danych odpowiadają funkcje recv oraz send. Obie pobierają takie same parametry – deskryptor gniazda, adres bufora danych, długość bufora oraz znacznik odbioru lub wysyłania któremu najczęściej nadawana jest wartość zerowa. Send wysyła dane z bufora poprzez gniazdo sieciowe a recv odbiera dane z gniazda zapisując je w buforze. Cały proces przedstawia rysunek 6.8.

```
INVOKE GetDlgItemText,hDlgEx,ID_IP,ADDR szIP,SIZEOF szIP

mov Port,55555
INVOKE gethostbyname,ADDR szIP
cmp eax, 0
jne AddrName

   INVOKE inet_addr,ADDR szIP
   mov IP,eax

   INVOKE gethostbyaddr,ADDR IP,4,AF_INET
AddrName:

cld
lea esi,dword ptr [eax+hostent.h_len]
lea edi,SockAddr.sin_addr
lodsw
movzx ecx,ax
lodsd
mov esi,dword ptr [eax]
rep movsb

INVOKE    inet_ntoa,DWORD PTR SockAddr.sin_addr
INVOKE    lstrcpy,ADDR szIP,eax

INVOKE htons,Port
mov [SockAddr.sin_port],ax
mov [SockAddr.sin_family],AF_INET

INVOKE socket,AF_INET,SOCK_STREAM,0
mov hSocketTx,eax

INVOKE connect,hSocketTx,ADDR SockAddr,SIZEOF SockAddr
```

Rys. 6.8. Przykład tworzenia gniazda i połączenia sieciowego.

Aby przełączyć gniazdo w tryb nasłuchiwania należy przyłączyć je do medium komunikacyjnego funkcją bind, która wymaga deskryptora gniazda oraz tablicy

sockaddr_in. Następnie można użyć funkcji listen podając deskryptor gniazda oraz maksymalną liczbę oczekujących połączeń.

Zadanie polega na stworzeniu aplikacji sieciowej – komunikatora działającego w oparciu architekturę P2P. Należy w ten sposób połączyć ze sobą trzy aplikacje. Dodatkowo jedna aplikacja po otrzymaniu specjalnej komendy powinna uruchamiać kalkulator.

Celem ćwiczenia jest zapoznanie się ze sposobami komunikacji sieciowej. Gniada wykorzystywane są niezależnie od systemu operacyjnego i uczestniczą w większości aplikacji sieciowych. Ćwiczenie rozszerza uruchomienie zdalne kalkulatora poprzez funkcję ShellExecute oraz zapoznanie się z biblioteką masm32.lib pomocną w wykonywaniu podstawowych operacji jak np. porównywanie łańcuchów tesktowych.

6.6. ŁĄCZENIE Z JĘZYKAMI WYŻSZEGO POZIOMU

Tworzenie dużych projektów w języku asemblera jest złym rozwiązaniem. Większość kompilatorów wysokiego poziomu umożliwia wstawianie kodu napisanego w języku asemblera. Można to wykorzystać aby przyspieszyć działanie krytycznych części programu. To ćwiczenie pokazuje jak połączyć kody asemblera i C/C++ używając darmowego środowiska DevC++ wykorzystując składnię Intel oraz AT&T.

Kod języka asemblera należy wpisywać w cudzysłowu jako parametr funkcji asm(). Po skompilowaniu i uruchomieniu wykonuje się jedna instrukcja asemblera. Jeżeli używa się więcej instrukcji należy w cudzysłowu zmieniać linię na nową kodem \n. Rysunek 6.9 zawiera przykład użycia języka asemblera.

```
asm(".intel_syntax  \n"
    "mov ah, [ebp-1] \n"
    "shr ah, 1 \n"
    "mov [ebp-1], ah \n"
    ".att_syntax \n");
```

Rys. 6.9. Przykład języka asemblera w środowisku DevC++ z włączoną składnią Intel.

Dostęp do zmiennych globalnych zadeklarowanych w C/C++ jest taki sam jak w NASM 10. Zmienne funkcji deklarowane są na stosie. Dostęp uzyskuje się poprzez operacje na rejestrze ebp. Wykorzystując dodatnie przesunięcie względem rejestru ebp uzyskuje się dostęp do parametrów funkcji. Ujemne przesunięcie pozwala na dostęp do zmiennych lokalnych funkcji. Przesunięcie jest równoważne z rozmiarem zmiennej. W przypadku większej ilości zmiennych przesunięcia sumują się. Przy dostępie do

parametrów funkcji należy uwzględnić adres powrotu z funkcji. Dodatkowo każda zmienna na stosie jest czterobajtowa – jeżeli przekazany parametr jest jednobajtowy znajduje się w pierwszym bajcie. Przykład zawiera tabela 6.2.

Tabela. 6.2. Przykład dostępu do zmiennych.

zawartość stosu	**sposób Dostępu**
parametr N	`[ebp+4+N*4]`
parametr 1	`[ebp+8]`
adres powrotu z funkcji	`[ebp+4]`
stara wartość rejestru ebp	`[ebp]`
Zmienna lokalna 1	`[ebp-4]`
Zmienna lokalna N	`[ebp-N]`

Domyślną składnią języka asemblera w środowisku DevC++ jest AT&T. Aby przełączyć ją na składnię Intel należy użyć przełącznika .intel_syntax. Aby kod napisany w C/C++ mógł się kompilować poprawnie, po ostatniej instrukcji należy ją wyłączyć przełącznikiem .att_syntax. W składni AT&T należy pamiętać o odwrotnej kolejności podawania argumentów. Przy instrukcji przeniesienia mov pierwszy argument jest miejscem źródłowym a drugi miejscem docelowym. Nazwy rejestrów poprzedza się znakiem procenta a stałe całkowite przy instrukcjach porównania oraz logicznych znakiem dolara. Dużą różnicą jest adresowanie pamięci. Przemieszczenie podaje się przed nawiasem okrągłym w którym umieszczamy rejestr wskazujący. Adresowanie bazowe indeksowane wymaga podania w nawiasie okrągłym kolejno rejestru bazowego i następnie indeksowego. Chcąc skalować rejestr indeksowy należy jeszcze dodać stałą. Przykłady różnic składni zawiera tabela 6.3.

Tabela. 6.3. Przykłady różnic w składni Intel oraz AT&T.

kod Intel	**kod AT&T**
`mov eax, 1`	`mov $1, %eax`
`mov ebx, 0ffh`	`mov $0xff, %ebx`
`mov ebx, eax`	`mov %eax, %ebx`
`mov eax, [ecx]`	`mov (%ecx), %eax`
`mov eax, [ebx+20h]`	`mov 0x20(%ebx), %eax`
`add eax, [ebx+ecx*2h]`	`add (%ebx, %ecx, 0x2), %eax`

`lea eax, [ebx+ecx]`	`lea (%ebx, %ecx), %eax`
`sub eax, [ebx+ecx+4h-20]`	`sub -0x20(%ebx, %ecx, 0x4), %eax`
`shl eax, 4`	`sall $4, %eax`
`mov [si], byte ptr 2`	`movb $2, (%si)`
`mov [si], word ptr 2`	`movw $2, (%si)`
`mov [si], dword ptr 2`	`movl $2, (%si)`

Ćwiczenie polega na zamianie fragmentu kodu napisanego w języku C/C++ na kod napisany w języku asemblera w obu składniach.

Celem ćwiczenia jest poznanie sposobów łączenia języka asemblera z językami wysokiego poziomu. Należy zrozumieć sposób przekazywania zmiennych do funkcji oraz tworzenie zmiennych lokalnych aby uzyskać do nich dostęp. Należy się zapoznać z dwoma konkurencyjnymi składniami – Intela używaną w systemach Windows oraz AT&T tożsamą z systemami Linux/UNIX.

7. PODSUMOWANIE

Podstawowym celem pracy było stworzenie kompletnych ćwiczeń laboratoryjnych z zakresu programowania niskopoziomowego w trybie chronionym, wykorzystując pakiet NASM10.

Przygotowano sześć obszernych ćwiczeń z możliwością podziału na mniejsze. Trudność ćwiczeń jest rosnąca. Każde ćwiczenia zawiera programy przykładowe oraz materiały pomocnicze z opisami niezbędnych funkcji oraz wskazówkami odnośnie wykonania ćwiczenia. Wykorzystano darmowe narzędzia, które z łatwością można wykorzystać w domu lub w uczelni.

Ćwiczenia zawierają wszystkie zaplanowane zagadnienia, wykorzystując nowoczesne możliwości systemów operacyjnych. Wykorzystanie funkcji WinAPI jest uniwersalne – w językach programowania metodyka pewnych zagadnień jest taka sama.

Ćwiczenia można rozszerzyć o kolejne zagadnienia takie jak tworzenie dynamicznych bibliotek DLL (ang. dynamic-link library), programowanie usług czy wprowadzenie do tworzenia sterowników.

8. LITERATURA

[1] WRÓBEL E.: Praktyczny kurs asemblera. Helion, Gliwice 2004.

[2] SYCK G.: Turbo Assembler biblia użytkownika. LT&P, Warszawa 1994

[3] BŁASZCZYK A.: Asembler w Windows. Helion, Gliwice 2004.

[4] PIROGOV V.: Asembler podręcznik programisty. Helion, Gliwice 2005.

[5] STANISŁAWSKI W.: Wprowadzenie do sieciowych systemów operacyjnych. Oficyna wydawnicza PWSZ w Nysie, Nysa 2006.

[6] SILBERSCHATZ A., GALVIN P., GAGNE G.: Podstawy systemów operacyjnych. WNT, Warszawa 2006.

[7] ICZELION: Win32 Assembly Homepage. http://win32assembly.online.fr/, 12 listopada 2009.

[8] Microsoft: MSDN Library. http://msdn.microsoft.com/en-us/library/default.aspx, 12 listopada 2009.

[9] MASM32: SDK Version 10. http://www.masm32.com/, 12 listopada 2009.

9. SPIS RYSUNKÓW

3.1. Rejestry procesora 8086. . . . . str. 7
3.2. Tworzenie adresu fizycznego w trybie rzeczywistym. . . . . str. 9
3.3. Format selektora. . . . . str. 11
3.4. Uproszczony proces translacji adresu wirtualnego na adres liniowy . . . . . str. 12
3.5. Proces stonicowania. . . . . str. 14
6.1. Kod źródłowy odczytujący parametry i wyświetlający je w konsoli. . . . . str. 22
6.2. Przykład odczytania pliku do bufora. . . . . str. 23
6.3. Konstrukcja procedury obsługi zdarzeń. . . . . str. 24
6.4. Obsługa komunikatu naciśnięcia przycisku. . . . . str. 26
6.5. Przykład dialogu z wykorzystaniem funkcji GetOpenFileName. . . . . str. 27
6.6. Inicjalizacja struktury OPENFILENAME. . . . . str. 28
6.7. Przykład procedury uruchamianej jako wątek. . . . . str. 29
6.8. Przykład tworzenia gniazda i połączenia sieciowego. . . . . str. 31
6.9. Przykład języka asemblera w środowisku DevC++ z włączoną składnią Intel. . . . . str. 32

10. SPIS TABEL

3.1 Zestawienie znaczników rejestru flagowego. . . . . str. 7
4.1 Instrukcje trybu chronionego. . . . . str. 16
6.1. Przykład wyglądu okna w zależności od określonych zasobów. . . . . str. 25
6.2 Przykład dostępu do zmiennych . . . . . str. 33
6.3 Przykłady różnic w składni Intel oraz AT&T. . . . . str. 33

11. DODATEK

1.1. Konsola przykład pierwszy.

```
include \masm32\include\masm32rt.inc

.data?
hInstance dd ?
hFile dd ?
tmpV dd ?
fSize dd ?
hBuff dd ?
readBytes dd ?

argc dd ?
argv dd ?
.data

txt2 db "Blad otwarcia pliku.", NULL
txt3 db "Blad odczytu pliku.", NULL
txt4 db "Blad alokacji pamieci.", NULL
fName db "plik.txt"

.code
Start:
INVOKE AllocConsole
INVOKE GetStdHandle, STD_OUTPUT_HANDLE
mov hInstance, eax


INVOKE CreateFile,ADDR fName, GENERIC_READ, 0, NULL,
OPEN_EXISTING,FILE_ATTRIBUTE_ARCHIVE,0
mov hFile, eax

    .IF eax==INVALID_HANDLE_VALUE
        INVOKE lstrlen, OFFSET txt2
        INVOKE WriteConsole, hInstance, OFFSET txt2, eax,ADDR tmpV, NULL
        jmp @end
    .ENDIF

    INVOKE GetFileSize,hFile,0
    mov fSize, eax
```

```
    INVOKE VirtualAlloc,0,eax,MEM_COMMIT,PAGE_READWRITE
    .IF eax == NULL
        INVOKE lstrlen, OFFSET txt4
        INVOKE WriteConsole, hInstance,OFFSET txt4, eax,OFFSET tmpV, NULL
    .ELSE
        mov hBuff, eax
        INVOKE ReadFile, hFile, eax, fSize, OFFSET readBytes, 0

        .IF eax == -1
            INVOKE lstrlen, OFFSET txt3
            INVOKE WriteConsole, hInstance,OFFSET txt3, eax,ADDR tmpV, NULL
        .ELSE
            INVOKE WriteConsoleW, hInstance, hBuff, fSize,OFFSET tmpV, NULL
        .ENDIF
    .ENDIF

    INVOKE VirtualFree,eax,OFFSET fSize,MEM_RELEASE
    INVOKE CloseHandle, hFile

@end:

INVOKE FreeConsole
INVOKE ExitProcess,0

END Start
```

1.2. Konsola przykład drugi.

```
include \masm32\include\masm32rt.inc

.data?
hInstance dd ?
tmpV dd ?
argc dd ?
argv dd ?
.data
cr db 10,13,NULL

.code
Start:
INVOKE AllocConsole
INVOKE GetStdHandle, STD_OUTPUT_HANDLE
mov hInstance, eax

INVOKE GetCommandLineW
INVOKE CommandLineToArgvW,eax,OFFSET argc
mov argv,eax
mov edi, argv
xor ecx, ecx

@p1:
   mov eax,[edi]
   mov ebx, eax
   add edi, 4

   push ecx
   INVOKE lstrlenW, eax
   INVOKE WriteConsoleW, hInstance, ebx, eax,OFFSET tmpV, NULL
```

```
   INVOKE WriteConsole, hInstance, OFFSET cr, 2,OFFSET tmpV, NULL
   pop ecx
   inc ecx

cmp ecx, argc
jne @p1

INVOKE FreeConsole
INVOKE ExitProcess,0

END Start
```

1.3. Rozwiązanie.

```
include \masm32\include\masm32rt.inc

.data?
hInstance dd ?
hFile dd ?
tmpV dd ?
fSize dd ?
hBuff dd ?
readBytes dd ?
writeBytes dd ?

argc dd ?
argv dd ?

.data

txt1 db "Brak argumentow.", NULL
txt2 db "Blad otwarcia pliku.", NULL
txt3 db "Blad odczytu pliku.", NULL
txt4 db "Blad alokacji pamieci.", NULL
txt5 db "Blad zapisu pliku.", NULL

.code
Start:
INVOKE AllocConsole
INVOKE GetStdHandle, STD_OUTPUT_HANDLE
mov hInstance, eax


INVOKE GetCommandLineW
INVOKE CommandLineToArgvW,eax,OFFSET argc
mov argv,eax
mov ebx, argc

.IF ebx == 2
    mov edi, argv
    add edi, 4
    mov eax,[edi]
    mov ebx, eax

    INVOKE CreateFileW, ebx, GENERIC_READ, 0, NULL,
OPEN_EXISTING,FILE_ATTRIBUTE_ARCHIVE,0
```

```
    mov hFile, eax

    .IF eax==INVALID_HANDLE_VALUE
        INVOKE lstrlen, OFFSET txt2
        INVOKE WriteConsole, hInstance, OFFSET txt2, eax,ADDR tmpV, NULL
        jmp @end
    .ENDIF

    INVOKE GetFileSize,hFile,0
    mov fSize, eax

    INVOKE VirtualAlloc,0,eax,MEM_COMMIT,PAGE_READWRITE
    .IF eax == NULL
        INVOKE lstrlen, OFFSET txt4
        INVOKE WriteConsole, hInstance,OFFSET txt4, eax,OFFSET tmpV, NULL
    .ELSE
        mov hBuff, eax
        INVOKE ReadFile, hFile, eax, fSize, OFFSET readBytes, 0

        .IF eax == -1
            INVOKE lstrlen, OFFSET txt3
            INVOKE WriteConsole, hInstance,OFFSET txt3, eax,ADDR tmpV, NULL
        .ELSE
            INVOKE WriteConsoleW, hInstance, hBuff, fSize,OFFSET tmpV, NULL
        .ENDIF
    .ENDIF

    INVOKE VirtualFree,eax,OFFSET fSize,MEM_RELEASE
    INVOKE CloseHandle, hFile

.ELSEIF ebx == 3
    mov edi, argv
    add edi, 4
    mov eax,[edi]
    mov ebx, eax

    INVOKE CreateFileW, ebx, GENERIC_WRITE, 0, NULL,
OPEN_ALWAYS,FILE_ATTRIBUTE_ARCHIVE,0
    mov hFile, eax
    .IF eax==INVALID_HANDLE_VALUE
        INVOKE lstrlen, OFFSET txt2
        INVOKE WriteConsole, hInstance, OFFSET txt2, eax,ADDR tmpV, NULL
        jmp @end
    .ELSE
        mov edi, argv
        add edi, 8
        mov eax,[edi]
        mov ebx, eax
        INVOKE lstrlenW, ebx
        mov ecx, 2
        mul ecx

        INVOKE WriteFile, hFile, ebx, eax, OFFSET writeBytes, NULL

        .IF eax == 0
            INVOKE lstrlen, OFFSET txt5
            INVOKE WriteConsole, hInstance, OFFSET txt5, eax,ADDR tmpV, NULL
        .ENDIF
```

```
        INVOKE CloseHandle, hFile
    .ENDIF
.ELSE
    INVOKE lstrlen, OFFSET txt1
    INVOKE WriteConsole, hInstance, OFFSET txt1, eax,OFFSET tmpV, NULL
.ENDIF

@end:

INVOKE FreeConsole
INVOKE ExitProcess,0

END Start
```

2.1a. Przykład plik zasobów.

```
#include "..\include\resource.h"

#define ID_DLG_MAIN   100
#define ID_LTEXT      101
#define ID_PB_OK      102
#define ID_EDIT       103
#define ID_PB_EXIT    104
#define ID_CHBOX      105
#define ID_ACTION     106
#define ID_PATH       107


ID_DLG_MAIN DIALOG 108, 70, 308, 105
STYLE DS_MODALFRAME | DS_SETFONT | WS_VISIBLE | WS_BORDER | WS_CAPTION |
WS_DLGFRAME | WS_GROUP | WS_MAXIMIZEBOX | WS_POPUP | WS_THICKFRAME | WS_SYSMENU
| WS_TABSTOP
CAPTION "Basic Api"
FONT 8, "Helv"
{
    LTEXT           "", ID_LTEXT, 8, 8, 223, 25, SS_LEFT
    PUSHBUTTON      "OK", ID_PB_OK, 259, 80, 40, 14
    EDITTEXT        ID_EDIT, 6, 34, 293, 41, ES_AUTOHSCROLL
    PUSHBUTTON      "Exit", ID_PB_EXIT, 215, 81, 40, 14
}
```

2.1b. Przykład plik źródłowy.

```
include \masm32\include\masm32rt.inc

ID_DLG_MAIN = 100
ID_LTEXT    = 101
ID_PB_OK    = 102
ID_EDIT     = 103
ID_PB_EXIT  = 104

.data?
hInstance dd ?
fSize dd ?

.data
buff1  db 65 dup(NULL)
```

```
.code
   DlgProc proc hDlg,uMsg,wParam,lParam:DWORD
     pushad

     .IF  uMsg==WM_CLOSE
        INVOKE EndDialog,hDlg,0
     .ELSEIF uMsg==WM_COMMAND
        .IF wParam==ID_PB_EXIT
            INVOKE EndDialog,hDlg,0
        .ELSEIF wParam==ID_PB_OK
            INVOKE GetDlgItemText, hDlg, ID_EDIT, OFFSET buff1, SIZEOF buff1
            INVOKE SetDlgItemText, hDlg, ID_LTEXT, OFFSET buff1
        .ENDIF
     .ENDIF

     popad
     xor eax,eax
     ret
   DlgProc endp

   Start:

     INVOKE GetModuleHandle,NULL
     mov hInstance,eax
     INVOKE DialogBoxParam,hInstance,ID_DLG_MAIN,0,ADDR DlgProc,0
     INVOKE ExitProcess,0

END Start
```

2.2a. Rozwiązanie plik zasobów.

```
#include "..\include\resource.h"

#define ID_DLG_MAIN   100
#define ID_LTEXT      101
#define ID_PB_OK      102
#define ID_EDIT       103
#define ID_PB_EXIT    104
#define ID_CHBOX      105
#define ID_ACTION     106
#define ID_PATH       107


ID_DLG_MAIN DIALOG 108, 70, 308, 105
STYLE DS_3DLOOK | DS_CENTER | DS_MODALFRAME | DS_FIXEDSYS | WS_VISIBLE |
WS_BORDER | WS_CAPTION | WS_DLGFRAME | WS_POPUP | WS_SYSMENU
CAPTION "Basic Api"
FONT 8, "Ms Shell Dlg"
{
    PUSHBUTTON      "OK", ID_PB_OK, 259, 80, 40, 14
    PUSHBUTTON      "Exit", ID_PB_EXIT, 215, 80, 40, 14
    AUTOCHECKBOX    "Zapis", ID_CHBOX, 60, 80, 33, 10
    PUSHBUTTON      "Akcja", ID_ACTION, 11, 80, 41, 14
    EDITTEXT        ID_PATH, 103, 80, 96, 16, ES_AUTOHSCROLL
    EDITTEXT        ID_EDIT, 6, 34, 293, 41, ES_AUTOHSCROLL
    LTEXT           "", ID_LTEXT, 8, 8, 223, 25, SS_LEFT
}
```

2.2b. Rozwiazanie plik źródłowy.

```
include \masm32\include\masm32rt.inc

ID_DLG_MAIN = 100
ID_LTEXT    = 101
ID_PB_OK    = 102
ID_EDIT     = 103
ID_PB_EXIT  = 104
ID_CHBOX    = 105
ID_ACTION   = 106
ID_PATH     = 107
.data?
hInstance dd ?
fSize dd ?

.data
buff1  db 65 dup(NULL)
hFile dd ?
xRead dd ?
xWrite dd ?
.code
   DlgProc proc hDlg,uMsg,wParam,lParam:DWORD
     pushad

     .IF  uMsg==WM_CLOSE
        INVOKE EndDialog,hDlg,0
     .ELSEIF uMsg==WM_COMMAND
        .IF wParam==ID_PB_EXIT
            INVOKE EndDialog,hDlg,0
        .ELSEIF wParam==ID_PB_OK
            INVOKE GetDlgItemText, hDlg, ID_EDIT, OFFSET buff1, SIZEOF buff1
            INVOKE SetDlgItemText, hDlg, ID_LTEXT, OFFSET buff1
        .ELSEIF wParam==ID_ACTION
            INVOKE IsDlgButtonChecked, hDlg, ID_CHBOX
            .IF eax==BST_CHECKED
               INVOKE GetDlgItemText, hDlg, ID_PATH, OFFSET buff1, SIZEOF buff1
               INVOKE CreateFile,OFFSET
buff1,GENERIC_WRITE,NULL,NULL,CREATE_ALWAYS,NULL,NULL
                mov hFile,eax
                .IF hFile!=NULL
                    INVOKE GetDlgItemText,hDlg,ID_EDIT,OFFSET buff1,SIZEOF
buff1
                    mov xWrite,eax
                    INVOKE WriteFile,hFile,OFFSET buff1,xWrite, ADDR
xWrite,NULL
                    INVOKE CloseHandle,hFile
                .ENDIF
            .ELSE
               INVOKE GetDlgItemText, hDlg, ID_PATH, OFFSET buff1, SIZEOF buff1
               INVOKE CreateFile,OFFSET buff1,GENERIC_READ,NULL,NULL,\
                    OPEN_EXISTING,NULL,NULL
               mov hFile,eax
                  .IF hFile!=NULL
                     INVOKE ReadFile,hFile,OFFSET buff1,SIZEOF buff1, ADDR
xRead,NULL
                     .IF eax!=0
                        INVOKE SetDlgItemText,hDlg, 101,OFFSET buff1
```

```
                    .ENDIF
                    INVOKE CloseHandle,hFile
                .ENDIF
            .ENDIF
        .ENDIF
     .ENDIF

     popad
     xor eax,eax
     ret
   DlgProc endp

   Start:

     INVOKE GetModuleHandle,NULL
     mov hInstance,eax
     INVOKE DialogBoxParam,hInstance,ID_DLG_MAIN,0,ADDR DlgProc,0
     INVOKE ExitProcess,0

END Start
```

3.1a. Przykład plik zasobów.

```
#include "..\include\resource.h"

#define ID_DLG_MAIN   100
#define ID_GROUP      1001
#define ID_CHANGE     1002

ID_DLG_MAIN DIALOG 179, 74, 249, 238
       STYLE DS_MODALFRAME | WS_POPUP | WS_VISIBLE | WS_CAPTION | WS_SYSMENU
CAPTION "Asm Notepad"
FONT 8, "Helv"
MENU 1000

BEGIN
    EDITTEXT        101, 6, 6, 237, 226, ES_MULTILINE | ES_AUTOVSCROLL |
                    ES_AUTOHSCROLL | ES_NOHIDESEL | ES_WANTRETURN |
                    WS_VSCROLL | WS_HSCROLL
END
1000 MENU
BEGIN
    POPUP "&Plik"
        BEGIN
        MENUITEM "&Otwórz", 1020
        MENUITEM "&Zakorz", 1030
        END
END
```

3.1b. Przykład plik źródłowy.

```
include \masm32\include\masm32rt.inc

ID_DLG_MAIN   =  100
ID_OTWORZ     = 1020
ID_KONIEC     = 1030
```

```
OFN_ENABLESIZING    = 00800000h
OFN_FORCESHOWHIDDEN = 10000000h

.data?
hInstance dd ?
hFile dd ?
oofn OPENFILENAME <>
bufor db 30000 dup(?)
xRead dd ?

.data
ofilter     db 'Pliki Tekstowe',NULL,'*.txt',NULL
ocustFilter db 256 dup(NULL)
ofname      db 256 dup(NULL)
oftitle     db 256 dup(NULL)
oinitDir    db 'C:\',NULL
oDlgTitle   db 'Otwórz',NULL

.code
   DlgProc proc hDlg,uMsg,wParam,lParam:DWORD
     pushad

     .IF  uMsg==WM_CLOSE
       INVOKE EndDialog,hDlg,0
     .ELSEIF uMsg==WM_COMMAND
        .IF wParam==ID_KONIEC
             INVOKE EndDialog,hDlg,0
        .ELSEIF wParam==ID_OTWORZ
             mov oofn.lStructSize,SIZEOF oofn

             push hDlg
             pop oofn.hwndOwner

             push hInstance
             pop oofn.hInstance

             mov oofn.lpstrFilter,OFFSET ofilter

             mov oofn.lpstrCustomFilter,OFFSET ocustFilter
             mov oofn.nMaxCustFilter,SIZEOF ocustFilter ;256
             mov oofn.nFilterIndex,0

             mov oofn.lpstrFile,OFFSET ofname
             mov oofn.nMaxFile,256

             mov oofn.lpstrFileTitle,OFFSET oftitle
             mov oofn.nMaxFileTitle,SIZEOF oftitle

             mov oofn.lpstrInitialDir,OFFSET oinitDir

             mov oofn.lpstrTitle,OFFSET oDlgTitle

             mov oofn.Flags,OFN_ENABLESIZING OR \
                        OFN_EXPLORER OR \
                        OFN_FORCESHOWHIDDEN OR \
                        OFN_PATHMUSTEXIST OR \
                        OFN_OVERWRITEPROMPT OR \
                        OFN_HIDEREADONLY OR \
```

```
                        OFN_FILEMUSTEXIST OR \
                        OFN_NODEREFERENCELINKS
             mov oofn.nFileOffset,0
             mov oofn.lpfnHook,NULL
             mov oofn.lpTemplateName,NULL

             INVOKE GetOpenFileName,ADDR oofn
             .IF eax!=NULL
                INVOKE CreateFile,OFFSET ofname,GENERIC_READ,NULL,NULL,\
                    OPEN_EXISTING,NULL,NULL
                mov hFile,eax
                .IF hFile!=NULL
                    INVOKE ReadFile,hFile,OFFSET bufor,SIZEOF bufor, ADDR
xRead,NULL
                    .IF eax!=0
                        INVOKE SetDlgItemText,hDlg, 101,OFFSET bufor
                        INVOKE SendMessage,hDlg,WM_SETTEXT,0,OFFSET oftitle
                    .ENDIF
                INVOKE CloseHandle,hFile
             .ENDIF
         .ENDIF
        .ENDIF
     .ENDIF

     popad
     xor eax,eax
     ret
   DlgProc endp

   Start:

     INVOKE GetModuleHandle,NULL
     mov hInstance,eax

     INVOKE DialogBoxParam,hInstance,ID_DLG_MAIN,0,ADDR DlgProc,0

     INVOKE ExitProcess,0

END Start
```

3.2a. Rozwiązanie plik zasobów.

```
#include "..\include\resource.h"

#define ID_DLG_MAIN   100
#define ID_GROUP      1001
#define ID_CHANGE     1002

ID_DLG_MAIN DIALOG 179, 74, 249, 238
       STYLE DS_MODALFRAME | WS_POPUP | WS_VISIBLE | WS_CAPTION | WS_SYSMENU
CAPTION "Asm Notepad"
FONT 8, "Helv"
MENU 1000

BEGIN
    EDITTEXT        101, 6, 6, 237, 226, ES_MULTILINE | ES_AUTOVSCROLL |
                    ES_AUTOHSCROLL | ES_NOHIDESEL | ES_WANTRETURN |
```

```
                    WS_VSCROLL | WS_HSCROLL
END
1000 MENU
BEGIN
    POPUP "&Plik"
        BEGIN
        MENUITEM "&Otwórz", 1020
             MENUITEM "&Zapisz", 1040
        MENUITEM "&Zakorz", 1030
        END
END
```

3.2b. Rozwiązanie plik źródłowy.

```
include \masm32\include\masm32rt.inc

ID_DLG_MAIN   =  100
ID_OTWORZ     = 1020
ID_KONIEC     = 1030

ID_ZAPISZ     = 1040
ID_EDIT       =  101

OFN_ENABLESIZING    = 00800000h
OFN_FORCESHOWHIDDEN = 10000000h

.data?
hInstance dd ?
hFile dd ?
oofn OPENFILENAME <>
sofn OPENFILENAME <>

bufor db 30000 dup(?)
xRead dd ?
xWrite dd ?
.data
filter     db 'Pliki Tekstowe',NULL,'*.txt',NULL, 'Pliki
asemblera',NULL,'*.asm',NULL
custFilter db 256 dup(NULL)
fname      db 256 dup(NULL)
ftitle     db 256 dup(NULL)
initDir    db 'C:\',NULL
oDlgTitle   db 'Otwórz',NULL

sdlgTitle   db 'Zapisz',NULL

.code
   DlgProc proc hDlg,uMsg,wParam,lParam:DWORD
     pushad

     .IF  uMsg==WM_CLOSE
       INVOKE EndDialog,hDlg,0
     .ELSEIF uMsg==WM_COMMAND
        .IF wParam==ID_KONIEC
             INVOKE EndDialog,hDlg,0
        .ELSEIF wParam==ID_OTWORZ
             mov oofn.lStructSize,SIZEOF oofn
```

```
            push hDlg
            pop oofn.hwndOwner

            push hInstance
            pop oofn.hInstance

            mov oofn.lpstrFilter,OFFSET filter

            mov oofn.lpstrCustomFilter,OFFSET custFilter
            mov oofn.nMaxCustFilter,SIZEOF custFilter ;256
            mov oofn.nFilterIndex,0

            mov oofn.lpstrFile,OFFSET fname
            mov oofn.nMaxFile,256

            mov oofn.lpstrFileTitle,OFFSET ftitle
            mov oofn.nMaxFileTitle,SIZEOF ftitle

            mov oofn.lpstrInitialDir,OFFSET initDir

            mov oofn.lpstrTitle,OFFSET oDlgTitle

            mov oofn.Flags,OFN_ENABLESIZING OR \
                       OFN_EXPLORER OR \
                       OFN_FORCESHOWHIDDEN OR \
                       OFN_PATHMUSTEXIST OR \
                       OFN_OVERWRITEPROMPT OR \
                       OFN_HIDEREADONLY OR \
                       OFN_FILEMUSTEXIST OR \
                       OFN_NODEREFERENCELINKS
            mov oofn.nFileOffset,0
            mov oofn.lpfnHook,NULL
            mov oofn.lpTemplateName,NULL

            INVOKE GetOpenFileName,ADDR oofn
            .IF eax!=NULL
               INVOKE CreateFile,OFFSET fname,GENERIC_READ,NULL,NULL,\
                   OPEN_EXISTING,NULL,NULL
               mov hFile,eax
               .IF hFile!=NULL
                   INVOKE ReadFile,hFile,OFFSET bufor,SIZEOF bufor, ADDR
xRead,NULL
                   .IF eax!=0
                       INVOKE SetDlgItemText,hDlg, ID_EDIT,OFFSET bufor
                       INVOKE SendMessage,hDlg,WM_SETTEXT,0,OFFSET ftitle
                   .ENDIF
               INVOKE CloseHandle,hFile
               .ENDIF
            .ENDIF
        .ELSEIF wParam==ID_ZAPISZ
            mov sofn.lStructSize,SIZEOF sofn

            push hDlg
            pop sofn.hwndOwner

            push hInstance
            pop sofn.hInstance
```

```
            mov sofn.lpstrFilter,OFFSET filter

            mov sofn.lpstrCustomFilter,OFFSET custFilter
            mov sofn.nMaxCustFilter,SIZEOF custFilter
            mov sofn.nFilterIndex,0

            mov sofn.lpstrFile,OFFSET fname
            mov sofn.nMaxFile,256

            mov sofn.lpstrFileTitle,OFFSET ftitle
            mov sofn.nMaxFileTitle,SIZEOF ftitle

            mov sofn.lpstrInitialDir,OFFSET initDir

            mov sofn.lpstrTitle,OFFSET sdlgTitle

            mov sofn.Flags,OFN_ENABLESIZING OR \
                        OFN_EXPLORER OR \
                        OFN_FORCESHOWHIDDEN OR \
                        OFN_PATHMUSTEXIST OR \
                        OFN_OVERWRITEPROMPT OR \
                        OFN_HIDEREADONLY OR \
                        OFN_NODEREFERENCELINKS
            mov sofn.nFileOffset,0
            mov sofn.lpfnHook,NULL
            mov sofn.lpTemplateName,NULL

            INVOKE GetSaveFileName,OFFSET sofn
            .IF eax!=NULL
                INVOKE CreateFile,OFFSET
fname,GENERIC_WRITE,NULL,NULL,CREATE_ALWAYS,NULL,NULL
                mov hFile,eax
                .IF hFile!=NULL
                    INVOKE GetDlgItemText,hDlg,ID_EDIT,OFFSET bufor,SIZEOF
bufor
                    mov xWrite,eax
                    INVOKE WriteFile,hFile,OFFSET bufor,xWrite, ADDR
xWrite,NULL
                    INVOKE CloseHandle,hFile
                .ENDIF
            .ENDIF
        .ENDIF
     .ENDIF

     popad
     xor eax,eax
     ret
   DlgProc endp

   Start:

     INVOKE GetModuleHandle,NULL
     mov hInstance,eax

     INVOKE DialogBoxParam,hInstance,ID_DLG_MAIN,0,ADDR DlgProc,0

     INVOKE ExitProcess,0
```

```
END Start
```

4.1. Przykład.

```
include \masm32\include\masm32rt.inc

.data?
hInstance dd ?
hEvent1 dd ?
hEvent2 dd ?


Thread1ID dd ?
Thread2ID dd ?
tmpV dd ?


String db 20 dup(?)
.data

txt1 db " 1 ", NULL
txt2 db " 2 ", NULL

.code

Thread1 proc hIns:DWORD

mov ecx, 5

    p1:
             push ecx
             INVOKE WriteConsole, hInstance, OFFSET txt2, 3,ADDR tmpV, NULL
             INVOKE Sleep, 500
             pop ecx
    loop p1

     INVOKE SetEvent, hEvent1
ret
Thread1 endp

Thread2 proc hIns:DWORD

mov ecx, 5

    p1:
             push ecx
             INVOKE WriteConsole, hInstance, OFFSET txt1, 3,ADDR tmpV, NULL
             INVOKE Sleep, 800
             pop ecx
    loop p1

    INVOKE SetEvent, hEvent2

ret
Thread2 endp

Start:
```

```
INVOKE AllocConsole
INVOKE GetStdHandle, STD_OUTPUT_HANDLE
mov hInstance, eax


INVOKE CreateEvent, NULL, TRUE, FALSE, NULL
mov hEvent1, eax

INVOKE CreateEvent, NULL, TRUE, FALSE, NULL
mov hEvent2, eax

INVOKE CreateThread, NULL, 0, OFFSET Thread1, 0,0,OFFSET Thread1ID
INVOKE CreateThread, NULL, 0, OFFSET Thread2, 0,0,OFFSET Thread2ID

INVOKE WaitForMultipleObjects, 2, OFFSET hEvent1, TRUE, INFINITE

INVOKE FreeConsole
INVOKE ExitProcess,0

END Start
```

4.2. Rozwiązanie.

```
include \masm32\include\masm32rt.inc

.data?
hInstance dd ?
hEvent1 dd ?
hEvent2 dd ?


Thread1ID dd ?
Thread2ID dd ?
tmpV dd ?


String db 20 dup(?)
.data

txt1 db " 1 ", NULL
txt2 db " 2 ", NULL

dot db ".", NULL

ent db 10, 13, NULL
round dd 0.5
multi dw 10000

.code

sqrt proc liczba:DWORD
LOCAL const:WORD
LOCAL buff:DWORD
mov const, 2

      finit
       fild liczba
       fild const
```

```
        fdiv


        mov ecx, 06FFFFFFh
        p1:
                fst buff
                fild liczba
                fxch
                fdiv

                fld buff
                fadd

                fild const
                fdiv
        loop p1
       fstp buff
       mov eax, buff
ret
sqrt endp

print1 proc

        mov di, 10
        xor ecx, ecx
        p3:
                xor edx, edx
                div di
                push dx
                inc cx
                or eax, eax
                jnz p3
                cld
                mov edi, OFFSET String
        p2:
                pop ax
                add al, '0'

                stosb
                loop p2

                mov al, NULL
                stosb
       INVOKE lstrlen, OFFSET String
        INVOKE WriteConsole, hInstance, OFFSET String, eax,ADDR tmpV, NULL
ret
print1 endp

float proc liczba:dword
LOCAL buff:DWORD

    finit
    fld liczba
        fld round
        fsub
        fist buff
        fld round
        fadd
```

```
      mov eax, buff
      call print1

    INVOKE WriteConsole, hInstance, OFFSET dot, 1,ADDR tmpV, NULL

      fild buff
      fsub
      fild multi
      fmul
      fld round
      fsub
      fistp buff
      mov eax, buff
      call print1

    INVOKE WriteConsole, hInstance, OFFSET ent, 2,ADDR tmpV, NULL

    ret
float endp

Thread1 proc hIns:DWORD

mov ecx, 5
    p1:
        push ecx

        shl ecx, 1
        INVOKE sqrt, ecx
        INVOKE float, eax

        pop ecx
    loop p1
     INVOKE SetEvent, hEvent1
ret
Thread1 endp

Thread2 proc hIns:DWORD
mov ecx, 5
    p1:

        push ecx

        shl ecx, 1
        inc ecx
        INVOKE sqrt, ecx
        INVOKE float, eax

        pop ecx
    loop p1
    INVOKE SetEvent, hEvent2

ret
Thread2 endp

Start:

INVOKE AllocConsole
INVOKE GetStdHandle, STD_OUTPUT_HANDLE
```

```
mov hInstance, eax


INVOKE CreateEvent, NULL, TRUE, FALSE, NULL
mov hEvent1, eax

INVOKE CreateEvent, NULL, TRUE, FALSE, NULL
mov hEvent2, eax

INVOKE CreateThread, NULL, 0, OFFSET Thread1, 0,0,OFFSET Thread1ID
INVOKE CreateThread, NULL, 0, OFFSET Thread2, 0,0,OFFSET Thread2ID

INVOKE WaitForMultipleObjects, 2, OFFSET hEvent1, TRUE, INFINITE

INVOKE FreeConsole
INVOKE ExitProcess,0

END Start
```

5.1a. Plik zasobów wspólny.

```
#include "..\include\resource.h"

#define ID_MAIN    100
#define ID_LISTBOX 101
#define ID_WYSLIJ  102
#define ID_TEXT    103
#define ID_IP      104
#define ID_NAPIS   105


ID_MAIN DIALOG 0, 0, 214, 202
STYLE DS_MODALFRAME | WS_POPUP | WS_VISIBLE | WS_CAPTION | WS_SYSMENU
CAPTION "Klient"
FONT 8, "Ms Shell Dlg"
{

    LISTBOX         ID_LISTBOX, 4, 32, 205, 137, WS_VSCROLL | WS_TABSTOP
    PUSHBUTTON      "Wyslij", ID_WYSLIJ, 157, 177, 47, 14
    EDITTEXT        ID_TEXT, 4, 178, 148, 13, ES_AUTOHSCROLL
    EDITTEXT        ID_IP, 24, 12, 53, 12, ES_AUTOHSCROLL
    LTEXT           "IP", ID_NAPIS, 9, 15, 8, 8, SS_LEFT
}
```

5.1b. Przykład klient pierwszy kod źródłowy.

```
include \masm32\include\masm32rt.inc
include \masm32\include\ws2_32.inc
includelib \masm32\lib\ws2_32.lib

ID_MAIN    = 100
ID_LISTBOX = 101
ID_WYSLIJ  = 102
ID_TEXT    = 103
ID_IP      = 104
ID_NAPIS   = 105
```

```
.data?
hInstance dd ?
hLog      dd ?
hThread   dd ?
hDlgEx    dd ?
hKlientRx dd ?
hSocketRx dd ?
hSocketTx dd ?
wsadata WSADATA <>

.data
buffRx      db 64 dup(NULL)
buffTx      db 64 dup(NULL)
txt1 db ' :: listen',NULL
txt2 db ' :: rx',NULL
txt3 db ' :: tx',NULL
.code

   Rx proc
      LOCAL Port:DWORD
      LOCAL PortOdp:DWORD
      LOCAL szIP[32]:BYTE
      LOCAL IP:DWORD
      LOCAL SockAddr:sockaddr_in
      LOCAL SockAddrLen:DWORD

      mov Port, 55556

      INVOKE gethostname, ADDR szIP, SIZEOF szIP
      cmp eax, 0
      jne @e1

      INVOKE gethostbyname,ADDR szIP
      cmp eax, 0
      je @e1

      mov eax, dword ptr [eax+hostent.h_list]
      mov eax, [eax]
      mov eax, [eax]
      mov IP, eax

      INVOKE inet_ntoa, IP
      cmp eax, INADDR_NONE
      je @e1

      INVOKE lstrcpy, ADDR szIP, eax

      INVOKE socket, AF_INET, SOCK_STREAM, 0
      mov hSocketRx, eax
      cmp eax, INVALID_SOCKET
      je @e1

      INVOKE htons, Port

      mov [SockAddr.sin_port],ax
      mov [SockAddr.sin_family],AF_INET
      mov [SockAddr.sin_addr],INADDR_ANY
```

```
    INVOKE bind, hSocketRx, ADDR SockAddr, SIZEOF SockAddr
    INVOKE listen, hSocketRx, 1
    cmp eax, 0
    jne @e1

    INVOKE SendMessage, hLog, LB_ADDSTRING, 0, OFFSET txt1
    @p1:
            mov SockAddrLen, SIZEOF SockAddr
            INVOKE accept, hSocketRx, ADDR SockAddr, ADDR SockAddrLen
            mov hKlientRx, eax
            INVOKE recv, eax, OFFSET buffRx, 64, 0
            INVOKE SendMessage, hLog, LB_ADDSTRING, 0, OFFSET txt2
            INVOKE SendMessage, hLog, LB_ADDSTRING, 0, OFFSET buffRx
    jmp @p1

    @e1:
    ret
 Rx endp

tx proc parametr:DWORD
   LOCAL SockAddr:sockaddr_in
   LOCAL IP:DWORD
   LOCAL szIP[128]:BYTE
   LOCAL Port:DWORD
   LOCAL PortOdp:DWORD

   INVOKE    GetDlgItemText,hDlgEx,ID_IP,ADDR szIP,SIZEOF szIP

   mov Port,55555
   INVOKE gethostbyname,ADDR szIP
   cmp eax, 0
   jne AddrName

   INVOKE    inet_addr,ADDR szIP
   mov IP,eax

   INVOKE    gethostbyaddr,ADDR IP,4,AF_INET

   AddrName:

   cld
   lea esi,dword ptr [eax+hostent.h_len]
   lea edi,SockAddr.sin_addr
   lodsw
   movzx ecx,ax
   lodsd
   mov esi,dword ptr [eax]
   rep movsb

   INVOKE    inet_ntoa,DWORD PTR SockAddr.sin_addr

   INVOKE    lstrcpy,ADDR szIP,eax

   INVOKE    SetDlgItemText,hDlgEx,ID_IP,ADDR szIP


   INVOKE    htons,Port
   mov    [SockAddr.sin_port],ax
```

```
  mov    [SockAddr.sin_family],AF_INET

  INVOKE    socket,AF_INET,SOCK_STREAM,0
  mov    hSocketTx,eax

  INVOKE    connect,hSocketTx,ADDR SockAddr,SIZEOF SockAddr


  INVOKE    GetDlgItemText,hDlgEx,ID_TEXT,ADDR buffTx,SIZEOF buffTx
  INVOKE lstrlen, OFFSET buffTx
  INVOKE send, hSocketTx, OFFSET buffTx, 64, 0
    INVOKE SendMessage, hLog, LB_ADDSTRING, 0, OFFSET txt3
    INVOKE SendMessage, hLog, LB_ADDSTRING, 0, OFFSET buffTx
  .IF hSocketTx != 0
          INVOKE closesocket, hSocketTx
          mov hSocketTx, 0
  .ENDIF
  ret
tx endp

DlgProc proc hDlg,uMsg,wParam,lParam:DWORD
  pushad
  .IF uMsg==WM_INITDIALOG
          INVOKE CreateThread, NULL, 0, OFFSET Rx, 0,0,OFFSET hThread
          INVOKE GetDlgItem,hDlg,ID_LISTBOX
          mov hLog,eax

          mov eax, hDlg
          mov hDlgEx, eax

  .ELSEIF  uMsg==WM_CLOSE
          .IF hSocketRx != 0
                INVOKE closesocket, hSocketRx
          .ENDIF

          .IF hKlientRx != 0
                INVOKE closesocket, hKlientRx
          .ENDIF

         INVOKE TerminateThread, hThread, 0
     INVOKE EndDialog,hDlg,0
  .ELSEIF uMsg==WM_COMMAND
      .IF wParam == ID_WYSLIJ
         INVOKE CreateThread, NULL, 0, OFFSET tx, 0, 0, OFFSET hThread
      .ENDIF
  .ENDIF

  popad
  xor eax,eax
  ret
DlgProc endp

Start:

  INVOKE GetModuleHandle,NULL
  mov hInstance,eax

  INVOKE WSAStartup, 00000101h, OFFSET wsadata
```

```
     INVOKE DialogBoxParam,hInstance,ID_MAIN,0,ADDR DlgProc,0

     INVOKE ExitProcess,0

END Start
```

5.2. Przykład klient drugi kod źródłowy.

```
include \masm32\include\masm32rt.inc
include \masm32\include\ws2_32.inc
includelib \masm32\lib\ws2_32.lib

ID_MAIN    = 100
ID_LISTBOX = 101
ID_WYSLIJ  = 102
ID_TEXT    = 103
ID_IP      = 104
ID_NAPIS   = 105


.data?
hInstance dd ?
hLog      dd ?
hThread   dd ?
hDlgEx    dd ?
hKlientRx dd ?
hSocketRx dd ?
hSocketTx dd ?
wsadata WSADATA <>

.data
buffRx      db 64 dup(NULL)
buffTx      db 64 dup(NULL)
txt1 db ' :: listen',NULL
txt2 db ' :: rx',NULL
txt3 db ' :: tx',NULL
.code

   Rx proc
      LOCAL Port:DWORD
      LOCAL PortOdp:DWORD
      LOCAL szIP[32]:BYTE
      LOCAL IP:DWORD
      LOCAL SockAddr:sockaddr_in
      LOCAL SockAddrLen:DWORD

      mov Port, 55555

      INVOKE gethostname, ADDR szIP, SIZEOF szIP
      cmp eax, 0
      jne @e1

      INVOKE gethostbyname,ADDR szIP
      cmp eax, 0
      je @e1

      mov eax, dword ptr [eax+hostent.h_list]
      mov eax, [eax]
```

```
    mov eax, [eax]
    mov IP, eax

    INVOKE inet_ntoa, IP
    cmp eax, INADDR_NONE
    je @e1

    INVOKE lstrcpy, ADDR szIP, eax

    INVOKE socket, AF_INET, SOCK_STREAM, 0
    mov hSocketRx, eax
    cmp eax, INVALID_SOCKET
    je @e1

    INVOKE htons, Port

    mov [SockAddr.sin_port],ax
    mov [SockAddr.sin_family],AF_INET
    mov [SockAddr.sin_addr],INADDR_ANY

    INVOKE bind, hSocketRx, ADDR SockAddr, SIZEOF SockAddr
    INVOKE listen, hSocketRx, 1
    cmp eax, 0
    jne @e1

    INVOKE SendMessage, hLog, LB_ADDSTRING, 0, OFFSET txt1
    @p1:
            mov SockAddrLen, SIZEOF SockAddr
            INVOKE accept, hSocketRx, ADDR SockAddr, ADDR SockAddrLen
            mov hKlientRx, eax
            INVOKE recv, eax, OFFSET buffRx, 64, 0
            INVOKE SendMessage, hLog, LB_ADDSTRING, 0, OFFSET txt2
            INVOKE SendMessage, hLog, LB_ADDSTRING, 0, OFFSET buffRx
    jmp @p1

    @e1:
    ret
 Rx endp

tx proc parametr:DWORD
   LOCAL SockAddr:sockaddr_in
   LOCAL IP:DWORD
   LOCAL szIP[128]:BYTE
   LOCAL Port:DWORD
   LOCAL PortOdp:DWORD

   INVOKE    GetDlgItemText,hDlgEx,ID_IP,ADDR szIP,SIZEOF szIP

   mov Port,55556
   INVOKE gethostbyname,ADDR szIP
   cmp eax, 0
   jne AddrName

   INVOKE    inet_addr,ADDR szIP
   mov IP,eax

   INVOKE    gethostbyaddr,ADDR IP,4,AF_INET
```

```
  AddrName:

  cld
  lea esi,dword ptr [eax+hostent.h_len]
  lea edi,SockAddr.sin_addr
  lodsw
  movzx ecx,ax
  lodsd
  mov esi,dword ptr [eax]
  rep movsb

  INVOKE    inet_ntoa,DWORD PTR SockAddr.sin_addr

  INVOKE    lstrcpy,ADDR szIP,eax

  INVOKE    SetDlgItemText,hDlgEx,ID_IP,ADDR szIP


  INVOKE    htons,Port
  mov    [SockAddr.sin_port],ax
  mov    [SockAddr.sin_family],AF_INET

  INVOKE    socket,AF_INET,SOCK_STREAM,0
  mov    hSocketTx,eax

  INVOKE    connect,hSocketTx,ADDR SockAddr,SIZEOF SockAddr


  INVOKE    GetDlgItemText,hDlgEx,ID_TEXT,ADDR buffTx,SIZEOF buffTx
  INVOKE lstrlen, OFFSET buffTx
  INVOKE send, hSocketTx, OFFSET buffTx, 64, 0
    INVOKE SendMessage, hLog, LB_ADDSTRING, 0, OFFSET txt3
    INVOKE SendMessage, hLog, LB_ADDSTRING, 0, OFFSET buffTx
  .IF hSocketTx != 0
          INVOKE closesocket, hSocketTx
          mov hSocketTx, 0
  .ENDIF
  ret
tx endp

DlgProc proc hDlg,uMsg,wParam,lParam:DWORD
  pushad
  .IF uMsg==WM_INITDIALOG
          INVOKE CreateThread, NULL, 0, OFFSET Rx, 0,0,OFFSET hThread
          INVOKE GetDlgItem,hDlg,ID_LISTBOX
          mov hLog,eax

          mov eax, hDlg
          mov hDlgEx, eax

  .ELSEIF  uMsg==WM_CLOSE
          .IF hSocketRx != 0
                INVOKE closesocket, hSocketRx
          .ENDIF

          .IF hKlientRx != 0
                INVOKE closesocket, hKlientRx
          .ENDIF
```

```
            INVOKE TerminateThread, hThread, 0
        INVOKE EndDialog,hDlg,0
    .ELSEIF uMsg==WM_COMMAND
        .IF wParam == ID_WYSLIJ
            INVOKE CreateThread, NULL, 0, OFFSET tx, 0, 0, OFFSET hThread
        .ENDIF
    .ENDIF

    popad
    xor eax,eax
    ret
  DlgProc endp

  Start:

    INVOKE GetModuleHandle,NULL
    mov hInstance,eax

    INVOKE WSAStartup, 00000101h, OFFSET wsadata
    INVOKE DialogBoxParam,hInstance,ID_MAIN,0,ADDR DlgProc,0

    INVOKE ExitProcess,0

END Start
```

5.2. Rozwiązanie klient drugi kod źródłowy.

```
include \masm32\include\masm32rt.inc

include \masm32\include\ws2_32.inc
includelib \masm32\lib\ws2_32.lib

include \masm32\include\Winmm.inc
includelib \masm32\lib\Winmm.lib


ID_MAIN    = 100
ID_LISTBOX = 101
ID_WYSLIJ  = 102
ID_TEXT    = 103
ID_IP      = 104
ID_NAPIS   = 105


.data?
hInstance dd ?
hLog      dd ?
hThread   dd ?
hDlgEx    dd ?
hKlientRx dd ?
hSocketRx dd ?
hSocketTx dd ?
wsadata WSADATA <>

.data
buffRx      db 64 dup(NULL)
buffTx      db 64 dup(NULL)
```

```
txt1 db ' :: listen',NULL
txt2 db ' :: rx',NULL
txt3 db ' :: tx',NULL

command1 db 'run_calc',NULL
command2 db 'run_wordpad', NULL
command3 db 'open_cdrom', NULL



calc    db 'calc.exe',NULL
wordpad db 'wordpad.exe',NULL
cdrom   db 'set cdaudio door open wait',NULL
.code

   Rx proc
      LOCAL Port:DWORD
      LOCAL PortOdp:DWORD
      LOCAL szIP[32]:BYTE
      LOCAL IP:DWORD
      LOCAL SockAddr:sockaddr_in
      LOCAL SockAddrLen:DWORD

      mov Port, 55555

      INVOKE gethostname, ADDR szIP, SIZEOF szIP
      cmp eax, 0
      jne @e1

      INVOKE gethostbyname,ADDR szIP
      cmp eax, 0
      je @e1

      mov eax, dword ptr [eax+hostent.h_list]
      mov eax, [eax]
      mov eax, [eax]
      mov IP, eax

      INVOKE inet_ntoa, IP
      cmp eax, INADDR_NONE
      je @e1

      INVOKE lstrcpy, ADDR szIP, eax

      INVOKE socket, AF_INET, SOCK_STREAM, 0
      mov hSocketRx, eax
      cmp eax, INVALID_SOCKET
      je @e1

      INVOKE htons, Port

      mov [SockAddr.sin_port],ax
      mov [SockAddr.sin_family],AF_INET
      mov [SockAddr.sin_addr],INADDR_ANY

      INVOKE bind, hSocketRx, ADDR SockAddr, SIZEOF SockAddr
      INVOKE listen, hSocketRx, 1
      cmp eax, 0
```

```
      jne @e1

      INVOKE SendMessage, hLog, LB_ADDSTRING, 0, OFFSET txt1
      @p1:
              mov SockAddrLen, SIZEOF SockAddr
              INVOKE accept, hSocketRx, ADDR SockAddr, ADDR SockAddrLen
              mov hKlientRx, eax
              INVOKE recv, eax, OFFSET buffRx, 64, 0

             INVOKE szCmp, OFFSET buffRx, OFFSET command1
             .IF eax==8
                INVOKE ShellExecute, 0, NULL, OFFSET calc, NULL, 0,
SW_SHOWNORMAL
             .ENDIF

             INVOKE szCmp, OFFSET buffRx, OFFSET command2
             .IF eax==11
                INVOKE ShellExecute, 0, NULL, OFFSET wordpad, NULL, 0,
SW_SHOWNORMAL
             .ENDIF

             INVOKE szCmp, OFFSET buffRx, OFFSET command3
             .IF eax==10
                INVOKE mciSendString, OFFSET cdrom, NULL, NULL, NULL
             .ENDIF

              INVOKE SendMessage, hLog, LB_ADDSTRING, 0, OFFSET txt2
              INVOKE SendMessage, hLog, LB_ADDSTRING, 0, OFFSET buffRx
      jmp @p1

      @e1:
      ret
   Rx endp

  tx proc parametr:DWORD
     LOCAL SockAddr:sockaddr_in
     LOCAL IP:DWORD
     LOCAL szIP[128]:BYTE
     LOCAL Port:DWORD
     LOCAL PortOdp:DWORD

     INVOKE    GetDlgItemText,hDlgEx,ID_IP,ADDR szIP,SIZEOF szIP

     mov Port,55556
     INVOKE gethostbyname,ADDR szIP
     cmp eax, 0
     jne AddrName

     INVOKE    inet_addr,ADDR szIP
     mov IP,eax

     INVOKE    gethostbyaddr,ADDR IP,4,AF_INET

     AddrName:

     cld
     lea esi,dword ptr [eax+hostent.h_len]
     lea edi,SockAddr.sin_addr
```

```
  lodsw
  movzx ecx,ax
  lodsd
  mov esi,dword ptr [eax]
  rep movsb

  INVOKE    inet_ntoa,DWORD PTR SockAddr.sin_addr

  INVOKE    lstrcpy,ADDR szIP,eax

  INVOKE    SetDlgItemText,hDlgEx,ID_IP,ADDR szIP


  INVOKE    htons,Port
  mov    [SockAddr.sin_port],ax
  mov    [SockAddr.sin_family],AF_INET

  INVOKE    socket,AF_INET,SOCK_STREAM,0
  mov    hSocketTx,eax

  INVOKE    connect,hSocketTx,ADDR SockAddr,SIZEOF SockAddr


  INVOKE    GetDlgItemText,hDlgEx,ID_TEXT,ADDR buffTx,SIZEOF buffTx
  INVOKE lstrlen, OFFSET buffTx
  INVOKE send, hSocketTx, OFFSET buffTx, 64, 0
    INVOKE SendMessage, hLog, LB_ADDSTRING, 0, OFFSET txt3
    INVOKE SendMessage, hLog, LB_ADDSTRING, 0, OFFSET buffTx
  .IF hSocketTx != 0
          INVOKE closesocket, hSocketTx
          mov hSocketTx, 0
  .ENDIF
  ret
tx endp

DlgProc proc hDlg,uMsg,wParam,lParam:DWORD
  pushad
  .IF uMsg==WM_INITDIALOG
          INVOKE CreateThread, NULL, 0, OFFSET Rx, 0,0,OFFSET hThread
          INVOKE GetDlgItem,hDlg,ID_LISTBOX
          mov hLog,eax

          mov eax, hDlg
          mov hDlgEx, eax

  .ELSEIF  uMsg==WM_CLOSE
          .IF hSocketRx != 0
                INVOKE closesocket, hSocketRx
          .ENDIF

          .IF hKlientRx != 0
                INVOKE closesocket, hKlientRx
          .ENDIF

         INVOKE TerminateThread, hThread, 0
    INVOKE EndDialog,hDlg,0
  .ELSEIF uMsg==WM_COMMAND
      .IF wParam == ID_WYSLIJ
```

```
            INVOKE CreateThread, NULL, 0, OFFSET tx, 0, 0, OFFSET hThread
        .ENDIF
    .ENDIF

    popad
    xor eax,eax
    ret
  DlgProc endp

  Start:

    INVOKE GetModuleHandle,NULL
    mov hInstance,eax

    INVOKE WSAStartup, 00000101h, OFFSET wsadata
    INVOKE DialogBoxParam,hInstance,ID_MAIN,0,ADDR DlgProc,0

    INVOKE ExitProcess,0

END Start
```

6.1. Program źródłowy.

```
#include <cstdlib>
#include <iostream>
#include <conio.h>
using namespace std;

void encrypt(char *buff, char *text){
    char sign;

    for (int i=0; i<strlen(text); i++){

        sign = text[i];


       for (int j=0; j<8; j++){

           if ( ((sign & 0x01) == 0x01) && (( *(buff+i*8+j) & 0x01) == 0x00))
*(buff+i*8+j) = *(buff+i*8+j) + 1;
           if ( ((sign & 0x01) == 0x00) && (( *(buff+i*8+j) & 0x01) == 0x01))
*(buff+i*8+j) = *(buff+i*8+j) - 1;
           sign = sign>>1;

           }

       int NullEnd = strlen(text);
       for (int j=0; j<8; j++)
            if (( *(buff+NullEnd*8+j) & 0x01) == 0x01) *(buff+NullEnd*8+j) =
*(buff+NullEnd*8+j) - 1;

       }
}

void decrypt(char *buff){
     char sign;
     int i=0;
     while(1){
```

```
        sign = 0x00;
        for (int j=0; j<8; j++)
        sign = sign | ((*(buff+i*8+j) & 0x01)<<j);
        i++;
        if (sign == 0x00) break;
        else cout << sign;
     }
}

int main(int argc, char *argv[])
{
  FILE * pFile;
  pFile = fopen ("Untitled.bmp","r+b");
  if (pFile == NULL){
   puts("File open error.");
   system("PAUSE");
   return EXIT_SUCCESS;

  }

  short bitCount;
  fseek(pFile, 28, SEEK_SET);
  fread( &bitCount, 2, 1, pFile);

  if (bitCount == 24){
        fseek(pFile, 34, SEEK_SET);
        int imageSize;
        fseek (pFile, 0, SEEK_END);
        imageSize=ftell (pFile) -54;
        char text[]  = "Zaraz wracam";
        cout << "File size:" << imageSize << endl;
        cout << "Text size:" << strlen(text)+1 << endl;

        char *buff = (char*) malloc(imageSize);
        fseek(pFile, 0, SEEK_SET);
        char head[54];
        fread(head, 54, 1, pFile);
        fseek(pFile, 54, SEEK_SET);
        fread( buff, imageSize, 1, pFile);
        fclose (pFile);

        if (imageSize > strlen(text) +1){
           encrypt(buff, text);
           decrypt(buff);
        }

        pFile = fopen ("Untitled.bmp","w+b");
        fwrite(head, 1, 54, pFile);
        fwrite(buff, 1, imageSize, pFile);

        free(buff);
  }else{
        cout << "Please use 24bit bmp file.";
  }

  fclose (pFile);
  cout << endl;
  system("PAUSE");
```

```
  return EXIT_SUCCESS;
}
```

6.1. Program po zamianie części na składnię AT&T.

```
#include <cstdlib>
#include <iostream>
#include <conio.h>
using namespace std;
int debug;

int test;
void encrypt(char *buff, char *text){

    char sign;

    for (unsigned int i=0; i<strlen(text); i++){

        sign = text[i];

       for (unsigned int j=0; j<8; j++){



     asm("mov -8(%ebp), %eax \n"
         "sall $3, %eax \n"

         "mov -12(%ebp), %ebx \n"
         "add %ebx, %eax \n"
         "mov 8(%ebp), %ebx \n"
         "add %ebx, %eax \n"
         "mov %eax, %esi \n"
         "mov (%esi), %ah \n"

         "and $1, %ah \n"

         "cmp $0, %ah \n"

         "jne end \n"

         "mov -1(%ebp), %ah \n"
         "and $1, %ah \n"
         "cmp $1, %ah \n"
         "jne end2 \n"

         "mov (%esi), %ah \n"
         "inc %ah \n"
         "mov %ah, (%esi) \n"
         "jmp end2 \n"

         "end: \n"

         "mov -1(%ebp), %ah \n"
         "and $1, %ah \n"
         "cmp $0, %ah \n"
         "jne end2 \n"

         "mov (%esi), %ah \n"
```

```
         "dec %ah \n"
         "mov %ah, (%esi) \n"

         "end2: \n"

         "mov -1(%ebp), %ah \n"
         "shr %ah \n"
         "mov %ah, -1(%ebp) \n"

        );



          // if ( ((sign & 0x01) == 0x01) && (( *(buff+i*8+j) & 0x01) == 0x00))
*(buff+i*8+j) = *(buff+i*8+j) + 1;
          // if ( ((sign & 0x01) == 0x00) && (( *(buff+i*8+j) & 0x01) == 0x01))
*(buff+i*8+j) = *(buff+i*8+j) - 1;
          // sign = sign>>1;

           }

       int NullEnd = strlen(text);
       for (int j=0; j<8; j++)
            if (( *(buff+NullEnd*8+j) & 0x01) == 0x01) *(buff+NullEnd*8+j) =
*(buff+NullEnd*8+j) - 1;

       }
}

void decrypt(char *buff){
     char sign;
     int i=0;
     while(1){
        sign = 0x00;
        for (int j=0; j<8; j++)
        sign = sign | ((*(buff+i*8+j) & 0x01)<<j);
        i++;
        if (sign == 0x00) break;
        else cout << sign;
     }
}

int main(int argc, char *argv[])
{
  FILE * pFile;
  pFile = fopen ("Untitled.bmp","r+b");
  if (pFile == NULL){
   puts("File open error.");
   system("PAUSE");
   return EXIT_SUCCESS;

  }



  short bitCount;
  fseek(pFile, 28, SEEK_SET);
```

```
  fread( &bitCount, 2, 1, pFile);

  if (bitCount == 24){
        fseek(pFile, 34, SEEK_SET);
        int imageSize;
        fseek (pFile, 0, SEEK_END);
        imageSize=ftell (pFile) -54;
        char text[]  = "kalehjh oasd";
        cout << "File size:" << imageSize << endl;
        cout << "Text size:" << strlen(text)+1 << endl;

        char *buff = (char*) malloc(imageSize);
        fseek(pFile, 0, SEEK_SET);
        char head[54];
        fread(head, 54, 1, pFile);
        fseek(pFile, 54, SEEK_SET);
        fread( buff, imageSize, 1, pFile);
        fclose (pFile);

        if (imageSize > strlen(text) +1){
           encrypt(buff, text);
           decrypt(buff);
        }

        pFile = fopen ("Untitled.bmp","w+b");
        fwrite(head, 1, 54, pFile);
        fwrite(buff, 1, imageSize, pFile);

        free(buff);
  }else{
        cout << "Please use 24bit bmp file.";
  }

  fclose (pFile);
  cout << endl;
  system("PAUSE");
  return EXIT_SUCCESS;
}
```

6.1. Program po zamianie części na składnię Intel.

```
#include <cstdlib>
#include <iostream>
#include <conio.h>
using namespace std;
int debug;

int test;
void encrypt(char *buff, char *text){

    char sign;

    for (unsigned int i=0; i<strlen(text); i++){

        sign = text[i];

       for (unsigned int j=0; j<8; j++){
```

```
         asm(".intel_syntax  \n"

             "mov eax, [ebp-8] \n"
             "shl eax, 3 \n"
             "mov ebx, [ebp-12] \n"
             "add eax, ebx \n"
             "mov ebx, [ebp+8] \n"
             "add eax, ebx \n"
             "mov esi, eax \n"
             "mov ah, [esi] \n"
             "and ah, 0x01 \n"
             "cmp ah, 0x00 \n"
             "jne end \n"

             "mov ah, [ebp-1] \n"
             "and ah, 0x01 \n"
             "cmp ah, 0x01 \n"
             "jne end2 \n"

             "mov ah, [esi] \n"
             "inc ah \n"
             "mov [esi], ah \n"
             "jmp end2 \n"

             "end: \n"

             "mov ah, [ebp-1] \n"
             "and ah, 0x01 \n"
             "cmp ah, 0x00 \n"
             "jne end2 \n"

             "mov ah, [esi] \n"
             "dec ah \n"
             "mov [esi], ah \n"

             "end2: \n"

             "mov ah, [ebp-1] \n"
             "shr ah, 1 \n"
             "mov [ebp-1], ah \n"

             ".att_syntax \n");

           // if ( ((sign & 0x01) == 0x01) && (( *(buff+i*8+j) & 0x01) == 0x00))
*(buff+i*8+j) = *(buff+i*8+j) + 1;
           // if ( ((sign & 0x01) == 0x00) && (( *(buff+i*8+j) & 0x01) == 0x01))
*(buff+i*8+j) = *(buff+i*8+j) - 1;
           // sign = sign>>1;

            }

       int NullEnd = strlen(text);
       for (int j=0; j<8; j++)
            if (( *(buff+NullEnd*8+j) & 0x01) == 0x01) *(buff+NullEnd*8+j) =
*(buff+NullEnd*8+j) - 1;

       }
```

```
}

void decrypt(char *buff){
     char sign;
     int i=0;
     while(1){
        sign = 0x00;
        for (int j=0; j<8; j++)
        sign = sign | ((*(buff+i*8+j) & 0x01)<<j);
        i++;
        if (sign == 0x00) break;
        else cout << sign;
     }
}

int main(int argc, char *argv[])
{
  FILE * pFile;
  pFile = fopen ("Untitled.bmp","r+b");
  if (pFile == NULL){
   puts("File open error.");
   system("PAUSE");
   return EXIT_SUCCESS;

  }



  short bitCount;
  fseek(pFile, 28, SEEK_SET);
  fread( &bitCount, 2, 1, pFile);

  if (bitCount == 24){
        fseek(pFile, 34, SEEK_SET);
        int imageSize;
        fseek (pFile, 0, SEEK_END);
        imageSize=ftell (pFile) -54;
        char text[]  = "kalendarze asfijh oasd";
        cout << "File size:" << imageSize << endl;
        cout << "Text size:" << strlen(text)+1 << endl;

        char *buff = (char*) malloc(imageSize);
        fseek(pFile, 0, SEEK_SET);
        char head[54];
        fread(head, 54, 1, pFile);
        fseek(pFile, 54, SEEK_SET);
        fread( buff, imageSize, 1, pFile);
        fclose (pFile);

        if (imageSize > strlen(text) +1){
           encrypt(buff, text);
           decrypt(buff);
        }

        pFile = fopen ("Untitled.bmp","w+b");
        fwrite(head, 1, 54, pFile);
        fwrite(buff, 1, imageSize, pFile);
```

```
        free(buff);
  }else{
        cout << "Please use 24bit bmp file.";
  }

  fclose (pFile);
  cout << endl;
  system("PAUSE");
  return EXIT_SUCCESS;
}
```