Programming from the Ground Up

Jonathan Bartlett

Książka ta może być kupiona pod adresem **http://www.bartlettpublishing.com/**
Nie jest to książka referencyjna, tylko wprowadzająca. Dlatego nie jest przydatna sama w sobie do nauki jak profesjonalnie programować w języku asemblera x86, jako że kilka szczegółów zostało pominiętych aby uczynić proces nauki gładszym. Celem książki jest pomóc studentom zrozumieć jak działa język asemblera i programowanie komputerowe, a nie być referencją do tematu. Informacje referencyjne o konkretnym procesorze mogą być dostępne poprzez skontaktowanie się z firmą go produkującą.
Aby otrzymać kopię tej książki w formie elektronicznej, proszę odwiedzić **http://savannah.nongnu.org/projects/pgubook/**
Miejsce to zawiera instrukcje dla ściągania kopii tej książki jak zdefiniowano przez **GNU Free Documentation License**.

Spis Treści

Rozdział 1. Wprowadzenie

Witamy w Programowaniu

Kocham programowanie. Nie tylko lubię tworzyć działające programy, ale lubię robić to stylowo. Programowanie jest jak poezja. Przekazuje wiadomość, nie tylko do komputera, ale do tych co modyfikują i używają twojego programu. Wraz z programem budujesz swój własny świat z własnymi regułami. Kreujesz swój świat stosownie do swojej koncepcji obydwu, problemu i rozwiązania. Mistrzowscy programiści tworzą swoje światy z programami które są jasne i zwięzłe, jak poezja lub esej.
Jeden z największych programistów, Donald Knuth, opisuje programowanie nie jako mówienie komputerowi jak coś zrobić, ale mówienie komuś jak mógłby poinstruować komputer żeby coś zrobić. Celem jest aby programy były pojmowane jako czytane przez ludzi, nie tylko przez komputery. Twoje programy będą modyfikowane i aktualizowane przez innych długo po tym jak ty przejdziesz do innych projektów. Dlatego programowanie nie jest aż tak komunikacją z komputerem jak komunikacją z tymi co przyjdą po tobie. Programista jest rozwiązującym problem, poetą, i instruktorem, wszystko w jednym. Twoim celem jest rozwiązać problem jak najlepiej, robiąc to z umiarkowaniem i wyczuciem, i nauczenie twojego rozwiązania przyszłych programistów. Mam nadzieję, że ta książka może nauczyć przynajmniej jakiejś poezji i magii które czynią programowanie ekscytującym.
Większość książek instruktażowych do programowania frustruje mnie bez końca. Kończąc je możesz ciągle zapytać "jak komputer rzeczywiście działa?" i nie masz dobrej odpowiedzi. Starają się one przejść ponad tematami które są trudne nawet pomimo że są ważne. Zabiorę cię w te trudne tematy ponieważ to jedyna droga aby posunąć się dalej ku mistrzowskiemu programowaniu. Moim celem jest przeprowadzić cię od niewiedzy niczego o programowaniu do zrozumienia jak myśleć, pisać, i uczyć się jak programista. Nie będziesz wiedział wszystkiego ale będziesz miał podstawy jak wszystko razem pasuje.
Kończąc tę książkę powinieneś być w stanie:
- rozumieć jak działa program i jak współdziała z innymi programami
- czytać programy innych ludzi i uczyć się jak one działają

- uczyć się szybko nowych języków programowania
- uczyć się szybko zaawansowanych koncepcji informatycznych

Nie nauczę cię wszystkiego. Nauki komputerowe to ogromne pole, zwłaszcza jeśli połączysz teorię i praktykę programowania komputerowego. Jednakże, dostarczę podstaw z których później łatwo możesz pójść dokądkolwiek będziesz chciał

Jest coś z problemu jajka i kury w nauczaniu programowania, zwłaszcza języka asemblerowego. Jest wiele rzeczy do nauczenia się - prawie zbyt wiele do nauczenia się za jednym razem, ale każda część zależy od wszystkich innych. Dlatego musisz być cierpliwy kiedy uczysz się programowania. Jeśli nie rozumiesz czegoś za pierwszym razem, przeczytaj powtórnie. Jeśli nadal tego nie rozumiesz, czasami najlepiej jest przyjąć to na wiarę i powrócić do tego później. Często po wielu próbach programowania idee te będą nabierać sensu. Nie zniechęcaj się. To długa wspinaczka, ale bardzo wartościowa.

Na końcu każdego rozdziału są trzy zestawy ćwiczeń przeglądowych. Pierwszy zestaw jest mniej więcej przeżuwaniem - sprawdza czy potrafisz powtórzyć to czego się nauczyłeś w tym rozdziale. Drugi zestaw zawiera pytania aplikacyjne - sprawdza czy potrafisz zastosować to czego się nauczyłeś do rozwiązywania problemów. Ostatni zestaw sprawdza czy jesteś zdolny rozszerzyć swoje horyzonty. Niektóre z tych pytań mogą nie być do odpowiedzenia aż do czasu późniejszego w książce, ale dają ci kilka rzeczy do przemyślenia. Inne pytania wymagają trochę poszukiwań w zewnętrznych źródłach aby odkryć odpowiedź. Jeszcze inne wymagają prostej analizy swoich opcji i wytłumaczenia najlepszego rozwiązania. Wiele z tych pytań nie ma dobrych lub złych odpowiedzi ale to nie znaczy, że są nieważne. Uczenie się z wyników zawartych w programowaniu, uczenie się jak osiągnąć odpowiedź i uczenie się jak spojrzeć wprzód są wszystkie główną częścią pracy programisty.

Jeśli masz problemy których po prostu nie możesz przejść, jest lista mailingowa dla tej książki gdzie czytelnicy mogą dyskutować i otrzymywać pomoc w tym o czym czytają. Ten adres to **pgubook-readers@nongnu.org**. Lista mailingowa jest otwarta na każdy rodzaj pytań lub dyskusji nad wierszami tej książki. Możesz subskrybować do tej listy poprzez http://mail.nongnu.org/mailman/listinfo/pgubook-readers.

Twoje Narzędzia

Książka ta uczy języka asemblerowego dla procesora x86 i systemu operacyjnego GNU/Linux. Dlatego podamy wszystkie przykłady używając zestawu narzędzi GCC standardu GNU/Linux. Jeśli nie jesteś obeznany z GNU/Linux i zestawem narzędzi GCC, będą one krótko opisane. Jeśli jesteś nowy w Linuksie, powinieneś sprawdzić przewodnik dostępny w http://rute.sourceforge.net/. To co zamierzam pokazać jest programowaniem w ogóle raczej niż używanie szczególnego zestawu narzędzi na specyficznej platformie, ale standaryzacja na jednej platformie znacznie ułatwia to zadanie.

Nowi w Linuksie powinni także spróbować się zaangażować w ich lokalnych Grupach Użytkowników GNU/Linux. Członkowie Grup Użytkowników są zwykle bardzo pomocni dla nowych ludzi, i będą pomagać we wszystkim od instalacji Linuksa po uczenie używania go najbardziej efektywnie. Wykaz Grup Użytkowników GNU/Linux jest dostępny na http://www.linux.org/groups/

Wszystkie programy były przetestowane używając Linuksa Red Hat 8.0, i powinny działać także na jakiejkolwiek innej dystrybucji GNU/Linux. Nie będą działać na nielinuksowych systemach operacyjnych takich jak BSD lub innych. Jednakże, wszystkie umiejętności poznane w tej książce powinny być łatwe do przeniesienia do jakiegokolwiek innego systemu.

Więc co to jest GNU/Linux? GNU/Linux jest systemem operacyjnym modelowanym na UNIX-ie. Słowo Gnu pochodzi od Projektu GNU (http://www.gnu.org/), który zawiera większość programów które będziesz uruchamiał, włączając w to zestaw narzędzi GCC których będziesz używał do programowania. Zestaw narzędzi GCC zawiera wszystkie programy niezbędne do tworzenia programów w różnych językach komputerowych.

Linux jest nazwą kernela. Kernel jest jądrem systemu operacyjnego które zarządza wszystkim. Kernel jest zarówno barierą i bramą. Jako brama pozwala programom korzystać ze sprzętu w sposób zunifikowany. Bez kernela, musiałbyś pisać programy do współpracy z każdym modelem urządzenia kiedyś zrobionego. Kernel wykonuje wszystkie sprzętowo-

specyficzne interakcje więc ty już nie musisz. Zarządza także dostępem do plików i interakcjami pomiędzy procesami. Na przykład, kiedy piszesz, twoje pisanie przechodzi przez kilka programów zanim osiągnie twój edytor. Po pierwsze, kernel jest tym co zawiaduje twoim sprzętem, więc jest pierwszym odnotowującym naciśnięcie klawisza. Klawiatura przesyła skankody do kernela, które ten konwertuje do rzeczywistych liter, liczb i symboli które one reprezentują. Jeśli używasz systemu okienkowego (jak Microsoft Windows lub X Window System) to system okienkowy czyta naciśnięty klawisz z kernela i dostarcza to do programu jaki bieżąco jest na wyświetlaczu użytkownika.

Przykład 1-1. Jak komputer przetwarza sygnały z klawiatury

Klawiatura -> Kernel -> System Okienkowy -> Programy Aplikacyjne

Kernel kontroluje także przepływ informacji pomiędzy programami. Kernel jest dla programów bramą do świata otaczającego. Za każdym razem kiedy dane wędrują między procesami, kernel kontroluje te wiadomości. W naszym przykładzie klawiaturowym powyżej, kernel mógł być zaangażowany w system okienkowy do komunikacji naciśnięcia klawisza do programu aplikacyjnego.
Jako bariera, kernel powstrzymuje programy od przypadkowego nadpisania cudzych danych i od używania plików i urządzeń do których nie mają uprawnień. Ogranicza zniszczenia jakie źle napisane programy mogą spowodować innym uruchomionym programom.
W naszym przypadku kernelem jest Linux. Teraz, kernel sam z sobie nie zrobi niczego. Nie możesz nawet bootować komputera mając tylko kernel. Pomyśl o kernelu jako o rurkach na wodę w domu. Bez tych rurek umywalki nie będą działać, ale rurki są bezużyteczne gdy nie ma umywalek. Razem, aplikacje użytkownika (z projektu GNU lub innych miejsc) i kernel (Linux) tworzą system operacyjny, GNU/Linux.
Najczęściej książka ta będzie używała niskopoziomowego języka asemblera. Generalnie są trzy rodzaje języków:

Język Maszynowy
To jest to co rzeczywiście komputer widzi i na czym pracuje. Każda komenda którą komputer widzi jest podawana jako liczba lub sekwencja liczb.

Język Asemblerowy
Jest taki sam jak język maszynowy oprócz tego, że komendy liczbowe zostały zastąpione sekwencjami liter które są łatwiejsze do zapamiętania. Inne drobne rzeczy zostały zrobione także dla ułatwienia.

Język Wysoko-Poziomowy
Języki wysoko-poziomowe są aby sprawić programowanie łatwiejszym. Język asemblerowy wymaga pracy z samą maszyną. Języki wysoko-poziomowe pozwalają opisać program w bardziej naturalnym języku. Pojedyncza komenda w języku wysoko-poziomowym zwykle jest równoważnikiem kilkunastu komend w języku asemblerowym.

W tej książce będziemy się uczyć języka asemblerowego, chociaż dotkniemy nieznacznie języków wysoko-poziomowych. Mamy nadzieję, że ucząc się języka asemblerowego, twoje zrozumienie jak programować i jak działają komputery pójdzie o krok dalej.

Rozdział 2. Architektura Komputera

Przed uczeniem się jak programować, powinieneś najpierw zrozumieć jak komputer interpretuje programy. Nie potrzebujesz stopnia naukowego z inżynierii elektrycznej ale powinieneś rozumieć niektóre podstawy.

Współczesna architektura komputera jest oparta na architekturze zwanej architekturą Von Neumanna, od nazwiska jej twórcy. Architektura Von Neumanna dzieli komputer na dwie główne części - CPU (od Centralnej Jednostki Procesującej) i pamięć. Ta architektura jest używana we wszystkich współczesnych komputerach, włączając w to komputery osobiste, superkomputery, "mainframes", i nawet telefony komórkowe.

Struktura Pamięci Komputera

Aby zrozumieć jak komputer widzi pamięć wyobraź sobie lokalny urząd pocztowy. Zwykle mają tam pomieszczenie wypełnione skrzynkami pocztowymi. Te skrzynki są podobne do pamięci komputera w tym, że każda jest numerowanym stałych rozmiarów miejscem przechowywania. Na przykład, jeśli masz 256 megabajtów pamięci, znaczy to, że twój komputer posiada z grubsza 256 milionów stało-rozmiarowych miejsc przechowywania. Lub, używając naszej analogii, 256 milionów skrzynek pocztowych. Każde lokum ma numer i taki sam stałej długości rozmiar. Różnica między skrzynką pocztową a pamięcią komputera jest taka, że w skrzynce możesz umieścić różnego rodzaju rzeczy a w lokum pamięci komputerowej możesz tylko przechowywać pojedynczą liczbę.
Pewnie jesteś ciekaw dlaczego komputer jest zorganizowany w taki sposób. Jest tak ponieważ jest to proste w implementacji. Jeśli komputer byłby skomponowany z wielu różno rozmiarowych lokum lub jeśli mógłbyś w nich przechowywać różnego rodzaju dane byłby trudny i drogi w implementacji.
Pamięć komputera jest używana do wielu rzeczy. Wszystkie wyniki jakichkolwiek obliczeń są przechowywane w pamięci. Faktycznie, wszystko co jest "przechowywane" jest przechowywane w pamięci. Pomyśl o swoim komputerze w domu, i wyobraź sobie co jest przechowywane w pamięci twojego komputera.
- Pozycja twojego kursora na ekranie
- Rozmiar każdego okna na ekranie
- Kształt każdej litery każdej używanej czcionki
- Położenie wszystkich kontrolek w każdym oknie
- Grafika dla wszystkich ikon toolbara
- Tekst dla każdej wiadomości o błędzie i okienka dialogowego
- Lista się wydłuża i wydłuża

Na dodatek do tego wszystkiego, architektura Von Neumanna wyszczególnia, że nie tylko dane komputerowe powinny być w pamięci, ale programy które kontrolują operacje komputerowe także powinny być tam. Faktycznie, w komputerze nie ma różnicy pomiędzy programem a daną programu oprócz tego jak jest ona użyta przez komputer. Obydwie są przechowywane i udostępniane w ten sam sposób.

CPU

Więc jak funkcjonuje komputer? Oczywiście, proste przechowywanie danej niewiele pomaga - potrzebujesz być zdolnym do dostępu, manipulowania i przemieszczania jej. To jest miejsce gdzie wkracza CPU.
CPU wczytuje instrukcje z pamięci pojedynczo i wykonuje je. Jest to znane jako cykl pobierz-wykonaj. CPU zawiera następujące elementy do przeprowadzenia tego:
- licznik programu
- dekoder instrukcji
- magistrala danych
- rejestry ogólnego przeznaczenia
- jednostka arytmetyczno-logiczna

Licznik programu jest używany aby powiadomić komputer skąd pobrać następną instrukcję. Wspomnieliśmy wcześniej, że nie ma różnicy w sposobie przechowywania danych i programów, są one tylko różnie interpretowane przez CPU. Licznik

programu przechowuje adres pamięci następnej instrukcji do wykonania. CPU rozpoczyna od przejrzenia licznika programu i pobrania liczby przechowywanej w pamięci spod wyznaczonej lokalizacji. Wtedy jest ona przesłana do *dekodera instrukcji* który wylicza co ta instrukcja znaczy. Obejmuje to jaki proces powinien nastąpić (dodawanie, odejmowanie, mnożenie, przenoszenie danych, itd.) i jakie miejsca pamięci zamierza się włączyć w ten proces. Instrukcje komputerowe zwykle posiadają obydwa aktualną instrukcję i listę miejsc pamięci które są używane do ich przeprowadzenia.
Teraz komputer używa *magistrali danych* do pobrania miejsc pamięci użytych w tym obliczeniu. Magistrala danych jest połączeniem między CPU i pamięcią. To jest rzeczywisty obwód który je łączy. Jeśli spojrzysz na płytę główną komputera, obwody które odchodzą od pamięci są twoją magistralą danych.
W dodatku do pamięci na zewnątrz procesora, procesor sam posiada specjalne, bardzo szybkie miejsca pamięci zwane rejestrami. Są dwa rodzaje rejestrów - rejestry ogólnego i rejestry specjalnego przeznaczenia. Rejestry ogólnego przeznaczenia są tam gdzie główna akcja zachodzi. Dodawanie, odejmowanie, mnożenie, porównywanie, i inne operacje generalnie używają rejestrów ogólnego przeznaczenia do procesowania. Jednakże, komputery posiadają bardzo niewiele rejestrów ogólnego przeznaczenia. Większość informacji jest przechowywana w pamięci głównej, pobierana do rejestrów dla procesowania i potem odkładana z powrotem do pamięci kiedy proces jest ukończony. Rejestry specjalnego przeznaczenia są rejestrami które mają specyficzne przeznaczenie. Będziemy je omawiać gdy dojdziemy do nich.
Teraz kiedy CPU uzyskał wszystkie dane których potrzebuje, przesyła je i zdekodowaną instrukcję do jednostki arytmetyczno-logicznej dla dalszego procesowania. Tutaj instrukcja jest rzeczywiście wykonywana. Po obliczeniu wyniki są umieszczane na magistrali danych i przesyłane do odpowiedniego miejsca w pamięci lub do rejestru, według wskazań instrukcji.
To bardzo uproszczony opis. Procesory trochę się rozwinęły w ostatnich latach i są teraz wiele bardziej wszechstronne. Chociaż podstawowa operacja jest ciągle ta sama, jest ona komplikowana przez użycie hierarchii "cache", procesorów superskalarnych, "pipeliningu", wykonywania "out-of-order", tłumaczenia mikrokodu, koprocesorów, i innych optymalizacji. Nie martw się jeżeli nie wiesz co te słowa znaczą, możesz poszukać ich w Internecie jeśli chcesz wiedzieć więcej o CPU.

Kilka Zasad

Pamięć komputera jest zestawem ponumerowanych, stałorozmiarowych lokalizacji. Liczba przypisana do każdej lokalizacji jest zwana jej *adresem*. Rozmiar pojedynczej lokalizacji jest zwany *bajtem*. W procesorach x86, bajt jest liczbą pomiędzy 0 i 255.
Możesz być ciekaw jak komputery mogą wyświetlać i używać tekstu, grafiki, i nawet dużych liczb kiedy wszystko co mogą zrobić jest przechowywaniem liczb pomiędzy 0 i 255. Przede wszystkim, wyspecjalizowany sprzęt jak karty graficzne posiadają specjalne interpretacje każdej z liczb. Wyświetlając na ekranie, komputer używa tabel kodów ASCII do tłumaczenia liczb które przesyłasz w litery do wyświetlenia na ekranie, każdą liczbę tłumacząc w dokładnie jedną literę lub cyfrę. Dla przykładu, duża litera A jest reprezentowana przez liczbę 65. Cyfra 1 jest reprezentowana przez liczbę 49. Więc, aby napisać "HELLO", mógłbyś podać komputerowi sekwencję liczb 72, 69, 76, 76, 79. Żeby napisać liczbę 100, mógłbyś podać komputerowi sekwencję 49, 48, 48. Lista znaków ASCII i ich numerycznych kodów znajduje się w dodatku D.
Na dodatek używając liczb do reprezentowania znaków ASCII, ty jako programista ustalasz także znaczenie liczb. Na przykład, gdybym prowadził sklep, używałbym liczb do reprezentowania każdego towaru który sprzedaję. Każda liczba byłaby powiązana z serią innych liczb które byłyby kodami ASCII dla tego co chcę wyświetlić kiedy towar jest skanowany. Mógłbym mieć więcej liczb dla ceny, w zależności jak wiele mam na liście, itd.
A co jeśli potrzebujemy liczb większych od 255? Możemy po prostu użyć kombinacji bajtów do reprezentacji większych liczb. Dwa bajty mogą być użyte do reprezentowania każdej liczby pomiędzy 0 i 65536. Cztery bajty mogą być użyte do reprezentowania każdej liczby pomiędzy 0 i 4294967295. Obecnie, jest całkiem trudno napisać programy do połączenia bajtów razem aby zwiększyć rozmiar twoich liczb, i wymaga to trochę matematyki. Szczęśliwie, komputer będzie robił to za nas dla liczb do długości 4 bajtów. W rzeczywistości, czterobajtowe liczby są tym z czym będziemy pracować z założenia.

Wspominaliśmy wcześniej że w dodatku do pamięci regularnej którą posiada komputer, posiada on również specjalnego przeznaczenia lokalizacje przechowywania zwane rejestrami. Rejestry są tym czego komputer używa do obliczeń. Myśl o rejestrze jako o miejscu na twoim biurku - trzyma on rzeczy nad którymi aktualnie pracujesz. Możesz mieć wiele informacji poupychanych w szufladkach, ale rzeczy nad którymi pracujesz teraz są na biurku. Rejestry trzymają liczby którymi obecnie manipulujesz.
W komputerach których używamy rejestry są po cztery bajty długie, każdy. Rozmiar typowego rejestru jest zwany rozmiarem słowa komputerowego. Procesor x86 ma czterobajtowe słowa. To oznacza, że najbardziej naturalnym na tych komputerach jest wykonywanie obliczeń czterech bajtów na raz. To daje nam z grubsza 4 miliardy wartości.
Adresy są także czterobajtowej (1 słowo) długości, i dlatego także pasują do rejestru. Procesor x86 może mieć dostęp do 4294967296 bajtów jeśli jest zainstalowana wystarczająca ilość pamięci. Zauważ, to znaczy, że możemy przechowywać adresy w taki sam sposób jak każdą inną liczbę. Rzeczywiście, komputer nie może podać różnicy pomiędzy wartością która jest adresem, wartością która jest liczbą, wartością która jest kodem ASCII, lub wartością którą zdecydowałeś się użyć do innego celu. Liczba staje się kodem ASCII kiedy spróbujesz ją wyświetlić. Liczba staje się adresem kiedy spróbujesz poznać bajt na który ona wskazuje. Znajdź chwilę aby to przemyśleć ponieważ jest to kluczowe do zrozumienia jak programy komputerowe działają.
Adresy przechowywane w pamięci są zwane także *wskaźnikami*, ponieważ oprócz posiadania wartości regularnej, pokazują inną lokalizację w pamięci.
Jak wspomnieliśmy, instrukcje komputerowe są także przechowywane w pamięci. Faktycznie, są one przechowywane dokładnie w taki sam sposób jak inne dane. Jedyny sposób w jaki komputer wie, że lokalizacja pamięci jest instrukcją jest taki, że rejestr specjalnego przeznaczenia zwany wskaźnikiem instrukcji wskazuje na nią w tym lub innym punkcie. Jeśli wskaźnik instrukcji wskazuje na słowo, jest ono załadowane jako instrukcja. W innym przypadku komputer nie ma sposobu aby znać różnicę między programami a innymi typami danych.

Interpretacja Pamięci

Komputery są bardzo dokładne. Ponieważ są dokładne, programiści muszą być równie dokładni. Komputer nie ma żadnej koncepcji co twój program zamierza zrobić. Dlatego, będzie on tylko wykonywał dokładnie to co mu powiesz aby zrobił. Jeśli przez przypadek wpiszesz liczbę regularną zamiast kodów ASCII które wywołują liczbowe cyfry, komputer pozwoli na to - i skończysz z krzaczkami na ekranie (komputer będzie poszukiwał co twoja liczba oznacza w ASCII i wypisze to). Jeżeli powiesz komputerowi żeby zaczął wykonywać instrukcje od lokalizacji zawierającej dane zamiast instrukcji programu, kto wie jak on to zinterpretuje - ale na pewno będzie próbował. Komputer będzie wykonywał twoje instrukcje w kolejności przez ciebie określonej, nawet jeśli jest to bez sensu.
Prawda jest taka, że komputer zrobi dokładnie to co mu powiesz, bez znaczenia jak mało sensu jest w tym. Dlatego, jako programista, powinieneś wiedzieć dokładnie jak zaaranżowałeś swoje dane w pamięci. Pamiętaj, komputery mogą przechowywać tylko liczby, więc litery, muzyka, strony sieci, dokumenty, i cokolwiek jeszcze są właśnie długimi sekwencjami liczb w komputerze, które poszczególne programy wiedzą jak interpretować.
Na przykład, powiedzmy, że chcesz przechowywać informacje o klientach w pamięci. Jednym sposobem aby to zrobić mogłoby być ustawienie maksymalnego rozmiaru dla nazwy i adresu klienta - powiedzmy 50 znaków ASCII dla każdego, co mogłoby być 50 bajtów dla każdego. Potem, następnie mamy liczbę dla wieku klienta i jego "id" klienta. W ten sposób, mógłbyś mieć blok pamięci wyglądający tak:

Początek Rekordu:
Nazwa klienta (50 bajtów) - początek rekordu
Adres klienta (50 bajtów) - początek rekordu + 50 bajtów
Wiek klienta (1 słowo - 4 bajty) - początek rekordu + 100 bajtów
Numer 'id' klienta (1 słowo - 4 bajty) - początek rekordu + 104 bajtów

W ten sposób, podając adres rekordu klienta, wiesz gdzie reszta danych się znajduje. Jednakże, to ogranicza nazwę i adres klienta tylko po 50 znaków ASCII każdy.
Co jeśli nie chcemy precyzować ograniczeń? Następnym sposobem aby to zrobić mogłoby być posiadanie wskaźników do tych informacji. Na przykład, zamiast nazwy klienta, moglibyśmy mieć wskaźnik do jego nazwy. W ten sposób, pamięć mogłaby wyglądać tak:

Początek Rekordu:
Wskaźnik nazwy klienta (1 słowo) - początek rekordu
Wskaźnik adresu klienta (1 słowo) - początek rekordu + 4
Wiek klienta (1 słowo) - początek rekordu + 8
Numer 'id' klienta (1 słowo) - początek rekordu + 12

Rzeczywista nazwa i adres mogłyby być przechowywany gdzie indziej w pamięci. W ten sposób, łatwo powiedzieć gdzie każda część danych jest względem początku rekordu, bez dokładnego limitowania rozmiaru nazwy i adresu. Jeśli długości pól wewnątrz naszych rekordów zmieniałyby się, nie moglibyśmy wiedzieć gdzie zaczyna się następne pole. Ponieważ rekordy mogłyby być różnych rozmiarów, trudno byłoby znaleźć gdzie zaczyna się następny rekord. Dlatego, prawie wszystkie rekordy są stałych długości. Dane zmiennej długości są zwykle przechowywane oddzielnie od reszty rekordu.

Metody Dostępu do Danych

Procesory mają wiele różnych sposobów dostępu do danych, zwanych trybami adresowania. Najprostszy tryb to *tryb natychmiastowy* w którym udostępniana dana jest wbudowana w samą instrukcję. Na przykład, jeśli chcemy zainicjalizować rejestr na 0, zamiast dawać komputerowi adres skąd przeczyta 0, możemy wywołać tryb natychmiastowy i dać mu liczę 0.
W *trybie adresowania rejestrowego*, instrukcja posiada raczej rejestr do dostępu niż lokalizację pamięci. Reszta trybów będzie się zajmować adresami.
W *trybie adresowania bezpośredniego*, instrukcja zawiera adres pamięci do dostępu. Na przykład, mógłbym powiedzieć, proszę załadować ten rejestr daną spod adresu 2002. Komputer mógłby pójść bezpośrednio do bajtu numer 2002 i skopiować zawartość do naszego rejestru.
W *trybie adresowania indeksowanego*, instrukcja zawiera adres pamięci do dostępu, i także podaje indeks rejestru do przesunięcia tego adresu. Na przykład, moglibyśmy wybrać adres 2002 i indeks rejestru. Jeśli indeks rejestru zawiera liczbę 4, aktualny adres spod którego dana jest załadowana mógłby być 2006. W ten sposób, jeśli masz zestaw liczb od lokalizacji 2002, możesz przebiegać je używając indeksu rejestru. W procesorach x86, możesz także podać *mnożnik* do indeksu. To pozwala ci na dostęp do pamięci po bajcie na raz lub po słowie na raz (4 bajty). Jeśli udostępniasz całe słowo, twój indeks rejestru musi być pomnożony przez 4 aby otrzymać dokładną lokalizację czwartego elementu z twojego adresu. Na przykład, jeśli chciałbyś udostępnić czwarty bajt z lokalizacji 2002, mógłbyś załadować indeks rejestru na 3 (pamiętaj, zaczynamy liczenie od 0) i nastawić mnożnik na 1 skoro przechodzisz bajt na raz. To mogłoby dać ci lokalizację 2005. Jednakże, jeśli chciałbyś udostępnić czwarte słowo od lokalizacji 2002, mógłbyś załadować indeks rejestru na 3 i ustawić mnożnik na 4. To mogłoby dać lokalizację od 2014 - czwarte słowo. Znajdź czas aby to sobie obliczyć żeby mieć pewność, że rozumiesz jak to działa.
W *trybie adresowania pośredniego* instrukcja zawiera rejestr który zawiera wskaźnik gdzie dana powinna być dostępna. Na przykład, jeśli użyliśmy trybu adresowania pośredniego i wybraliśmy rejestr **%eax**, i ten rejestr **%eax** zawiera wartość 4, jakakolwiek wartość była zlokalizowana w pamięci, będzie użyte 4. W adresowaniu bezpośrednim, moglibyśmy załadować wartość 4, ale w adresowaniu pośrednim, używamy 4 jako adresu do znalezienia danej której chcemy.
Ostatecznie, jest jeszcze *tryb adresowania wskaźnika bazowego*. Jest on podobny do trybu adresowania pośredniego, ale zawiera także liczbę zwaną przesunięciem dla dodania do wartości rejestru przed użyciem go do szukania. Będziemy używać tego trybu całkiem często w tej książce.

W sekcji nazwanej *Interpretacja Pamięci* dyskutowaliśmy nad strukturą w pamięci przechowującą informacje o kliencie. Powiedzmy, że chcemy mieć dostęp do wieku klienta, który jest ósmym bajtem danych, i mamy adres początku struktury w rejestrze. Moglibyśmy użyć adresowania wskaźnika bazowego i określamy rejestr jako wskaźnik bazowy, a 8 jako nasze przesunięcie. To jest podobnie jak adresowanie indeksowe, z tą różnicą, że przesunięcie jest stałe i wskaźnik jest trzymany w rejestrze, a w adresowaniu indeksowym przesunięcie jest w rejestrze a wskaźnik jest stały.
Są inne formy adresowania, ale te są najważniejsze.

Przegląd

Znajomość Koncepcji
- Opisz cykl pobierz-wykonaj.
- Co to jest rejestr? Jak byłyby trudniejsze obliczenia bez rejestrów?
- Jak duże są rejestry na maszynach których będziemy używać?
- W jaki sposób komputer wie jak interpretować podany bajt lub ustawienie bajtów w pamięci?
- Co to są tryby adresowania i po co są używane?
- Co robi wskaźnik instrukcji?

Użycie Koncepcji
- Jakiej danej użyłbyś w rekordzie pracownika? Jak mógłbyś umieścić ją w pamięci?
- Jeżeli mam wskaźnik na początek rekordu pracownika, i chciałbym mieć dostęp do danej wewnątrz niego, jaki tryb adresowania mógłbym użyć?
- W trybie adresowania wskaźnika bazowego, jeśli masz rejestr trzymający wartość 3122, i przesunięcie 20, do jakiego adresu próbujesz mieć dostęp?
- W trybie adresowania indeksowego, jeśli adres bazowy wynosi 6512, indeks rejestru ma 5, i mnożnik jest 4, do jakiego adresu próbujesz mieć dostęp?
- W trybie adresowania indeksowego, jeśli adres bazowy jest 123472, indeks rejestru ma 0, i mnożnik jest 4, do jakiego adresu próbujesz mieć dostęp?
- W trybie adresowania indeksowego, jeśli adres bazowy jest 9123478, indeks rejestru ma 20, i mnożnik jest 1, do jakiego adresu próbujesz mieć dostęp?

Idąc Dalej
- Jaka jest minimalna liczba trybów adresowania potrzebnych do obliczeń?
- Dlaczego włączanie trybów adresowania nie jest koniecznie potrzebne?
- Zbadaj i potem opisz jak "pipelining" (lub jeden z innych skomplikowanych wskaźników) wpływa na cykl pobierz-wykonaj.
- Zbadaj i potem opisz kompromisy pomiędzy instrukcjami stałej długości i zmiennej długości.

Rozdział 3. Twoje Pierwsze Programy

W tym rozdziale będziesz się uczyć procesu pisania i budowania programów języka asemblerowego Linuksa. Dodatkowo, będziesz się uczyć struktury programów języka asemblerowego i kilku komend tego języka. Po przejściu tego rozdziału, możesz chcieć także zapoznać się z Dodatkiem B i Dodatkiem F.
Te programy mogą cię przerażać na początku. Jednakże, przejdź przez nie z uwagą, czytaj je i ich objaśnienia tak wiele razy jak potrzeba, a będziesz miał zbudowaną solidną podstawę wiedzy. Proszę wypróbuj te programy na tyle sposobów ile możesz. Nawet, jeśli twoje próby nie zadziałają, każda porażka pomoże w nauce.

Rozpoczynanie Programu

Prawda, ten pierwszy program jest prosty. Rzeczywiście nie robi nic tylko wychodzi! Jest krótki, ale ukazuje trochę podstaw języka asemblerowego i programowania Linuksowego. Powinieneś otworzyć ten program w edytorze dokładnie tak jak jest napisany, z nazwą pliku **exit.s**. Program następuje. Nie martw się, że go nie rozumiesz. Ta sekcja tylko wymaga wpisania go i uruchomienia. W sekcji zwanej "Zarys Programu Języka Asemblerowego" opiszemy jak działa.

#CEL: Prosty program który wychodzi i zwraca kod statusu z powrotem do kernela Linuksa

#WEJŚCIE: nic

#WYJŚCIE: zwraca kod statusu. Można go obejrzeć wpisując
#echo $?
#po uruchomieniu programu

#ZMIENNE:
#%eax przechowuje numer wywołania systemowego
#%ebx przechowuje status powrotu

.section .data
.section .text
.globl _start
_start:
movl $1, %eax #to jest numer komendy kernela Linuksa (wywołanie systemowe) dla wyjścia z programu
movl $0, %ebx #to jest numer statusu który będzie zwracać system operacyjny. Zmieniając go będą zwracane inne rzeczy przez echo $?
int $0x80 #to budzi kernel do uruchomienia komendy wyjścia

To co wpisałeś jest zwane *kodem źródłowym*. Kod źródłowy jest czytelną dla ludzi formą programu. W celu przekształcenia go w program który komputer może uruchomić, potrzebujemy *zasemblować* i *zlinkować* go.
Pierwszy krok to *zasemblować* go. Asemblacja jest to proces który przekształca to co napisałeś w instrukcje dla maszyny. Maszyna sama tylko czyta zestawy liczb, ale ludzie preferują słowa. *Język asemblerowy* jest czytelniejszą formą instrukcji które komputer rozumie. Asemblacja przekształca czytelny dla ludzi plik w czytelny dla maszyny. Aby zasemblować program wpisz komendę

as exit.s -o exit.o #Dla procesorów 64 bitowych Intel i AMD należy użyć komendy **as --32** - przyp. tłum.

as jest komendą która uruchamia asembler, **exit.s** to plik źródłowy, a **-o exit.o** mówi asemlerowi żeby wyjście umieścił w pliku **exit.o. exit.o** jest *plikiem obiektowym*. Plik obiektowy jest kodem który jest w języku maszynowym, ale nie został połączony kompletnie. W większości dużych programów, będziesz miał kilkanaście plików źródłowych, i będziesz każdy konwertował do pliku obiektowego. *Linker* jest programem który odpowiada za złożenie plików obiektowych razem i dodanie informacji, tak że kernel wie jak go załadować i uruchomić. W naszym przypadku, mamy tylko jeden plik obiektowy, więc linker tylko dodaje informację umożliwiającą jego uruchomienie. Aby zlinkować plik, wpisz komendę

ld exit.o -o exit #Dla procesorów 64 bitowych Intel i AMD należy użyć komendy **ld -m elf_i386** - przyp. tłum.

ld jest komendą do uruchomienia linkera, **exit.o** jest plikiem obiektowym który chcemy zlinkować, i **-o exit** instruuje linker

aby wyjście nowego programu umieścił w pliku nazwanym **exit**. Jeśli któraś z komend zgłasza błędy, mogłeś popełnić błąd zarówno przy wpisywaniu programu jak i komendy. Po skorygowaniu programu, musisz powtórnie uruchomić wszystkie komendy. *Musisz zawsze zreasemblować i zrelinkować programy po modyfikacji pliku źródłowego dla uaktywnienia zmian w programie*. Możesz uruchomić **exit** przez wpisanie komendy

./exit

./ jest użyty dla powiadomienia komputera, że ten program nie znajduje się w jednym z zwykłych katalogów z programami, ale jest zamiast tego w bieżącym katalogu. Zauważysz kiedy wpiszesz tę komendę, że jedyna rzecz która się wydaży to przejście do następnej linii. Jest tak dlatego, że ten program nie robi nic oprócz wyjścia. Jednakże, natychmiast po uruchomieniu tego programu, jeśli wpiszesz

echo $?

Będzie odpowiedź **0**. To co się dzieje jest to, że każdy program przy wychodzeniu daje Linuksowi *kod statusu wyjścia*, który mówi mu czy wszystko przebiegło prawidłowo. Jeśli wszystko było dobrze, zwraca 0. Programy UNIX-owe zwracają liczby inne niż zero dla zaznaczenia porażki lub innych błędów, ostrzeżeń lub statusów. Programista determinuje co każda liczba oznacza. Możesz obejrzeć ten kod przez wpisanie **echo $?**. W następnym podrozdziale zobaczymy co każda część kodu robi.

Zarys Programu Języka Asemblerowego

Spójrz na program który właśnie zrobiliśmy. Na początku jest wiele wierszy zaczynających się od haszy (#). To są *komentarze*. Komentarze nie są tłumaczone przez asembler. Są one używane tylko dla programistów aby coś powiedzieć komuś kto będzie czytał ten kod w przyszłości. Większość programów które napiszesz będzie zmieniana przez innych. Nabądź przyzwyczajenia pisania komentarzy w swoim kodzie które ułatwią zrozumienie zarówno dlaczego napisano ten program i jak on działa. Zawsze włączaj do swoich komentarzy:
- Cel tego kodu
- Przegląd procesu zaangażowanego
- Cokolwiek dziwnego co twój program robi i dlaczego

Po komentarzach, następny wiersz mówi

.section .data

Cokolwiek zaczynające się od kropki nie jest bezpośrednio tłumaczone na instrukcję maszynową. Zamiast tego, jest to instrukcja dla samego asemblera. Są one zwane *dyrektywami asemblerowymi* lub *pseudo-operacjami* ponieważ są one zarządzane przez asembler i nie są w rzeczywistości uruchamiane przez komputer. Komenda **.section** dzieli twój program na sekcje. Ta komenda rozpoczyna sekcję danych, gdzie wypisujesz miejsca pamięci których będziesz potrzebował na dane. Nasz program nie używa żadnego, więc nie potrzebujemy tej sekcji. Jest tutaj dla kompletności. Prawie każdy program który napiszesz w przyszłości będzie miał dane.
Zaraz za tym mamy

.section .text

które rozpoczyna sekcję tekstu. Sekcja tekstu programu jest miejscem gdzie znajdują się instrukcje programu.
Następną instrukcją jest

.globl _start

To instruuje asembler, że **_start** jest ważne do zapamiętania. **_start** jest *symbolem*, który oznacza, że będzie zmieniony na coś innego podczas asemblacji lub linkowania. Symbole są generalnie używane do zaznaczania położenia programów lub danych, więc możesz odsyłać do nich poprzez nazwę zamiast poprzez ich liczbę lokalizacyjną. Wyobraź sobie, że musisz odsyłać do każdej lokalizacji pamięci poprzez jej adres. Po pierwsze, to mogłoby być bardzo stresujące ponieważ musiałbyś zapamiętać lub zaglądać na adres numeryczny pamięci każdego kawałka kodu lub danych. W dodatku, za każdym razem gdy musisz umieścić kawałek danych lub kodu musiałbyś zmienić wszystkie adresy w swoim programie! Symbole są używane więc asembler i linker mogą się opiekować utrzymywaniem ścieżki do adresów, a ty możesz się skoncentrować na pisaniu twojego programu.
.globl oznacza, że asembler nie powinien wymazywać tego symbolu po asemblacji ponieważ linker będzie go potrzebował. **_start** jest specjalnym symbolem który zawsze potrzebuje być zaznaczony z **.globl** ponieważ zaznacza on lokalizację początku programu. *Bez zaznaczenia lokalizacji w ten sposób, kiedy komputer ładuje twój program nie będzie wiedział gdzie zacząć uruchamiać twój program*.
Następny wiersz

_start:

definiuje wartość etykiety **_start**. *Etykieta* jest symbolem za którym znajduje się dwukropek. Etykiety definiują wartość symboliczną. Kiedy asembler asembluje program, musi przypisać każdej danej wartość i każdej instrukcji adres. Etykiety wskazują asemblerowi poprzez wartość symboliczną gdzie będzie następna instrukcja lub element danych. W ten sposób, jeśli rzeczywista lokalizacja fizyczna danej lub instrukcji zmienia się, nie musisz przepisywać odwołań do niej - symbol automatycznie wskazuje tę nową wartość.
Teraz wchodzimy w rzeczywiste instrukcje komputera. Pierwsza taka instrukcja to:

movl $1, %eax

Gdy program się wykonuje, instrukcja ta przesyła liczbę **1** do rejestru **%eax**. W języku asemblerowym, wiele instrukcji posiada *operandy*. **movl** ma dwa operandy - *źródło* i *cel*. W tym wypadku, źródłem jest liczba 1 a celem jest rejestr **%eax**. Operandy mogą być liczbami, wskaźnikami lokalizacji pamięci lub rejestrami. Różne instrukcje pozwalają na różne typy operandów. Zobacz Dodatek B, więcej informacji jakie instrukcje mają jakie typy operandów.
W większości instrukcji które mają dwa operandy, pierwszy jest operandem źródłowym a drugi docelowym. Zauważ, że w tych przypadkach, operand źródłowy w ogóle nie jest zmieniany. Inne instrukcje tego typu to, na przykład, **addl**, **subl** i **imull**. Dodają/odejmują/mnożą operand źródłowy do/od/przez operand docelowy i zachowują wynik w operandzie docelowym. Inne instrukcje mogą mieć operand wbudowany. **idivl**, na przykład, wymaga aby dzielna była w **%eax**, a **%edx** wynosił zero, i iloraz jest przesyłany do **%eax** a reszta do **%edx**. Jednakże, dzielnik może być jakimkolwiek rejestrem lub lokalizacją pamięci.
W procesorach x86, jest kilka rejestrów ogólnego przeznaczenia (wszystkie mogą być użyte z **movl**):
- **%eax**
- **%ebx**
- **%ecx**
- **%edx**
- **%edi**
- **%esi**

W dodatku do tych rejestrów ogólnego przeznaczenia, jest także kilka rejestrów specjalnego przeznaczenia:
- **%ebp**
- **%esp**
- **%eip**
- **%eflags**

Będziemy o nich dyskutować później, miej świadomość, że istnieją. Niektóre z tych rejestrów, jak **%eip** i **%eflags** mogą być dostępne tylko poprzez instrukcje specjalne. Inne mogą być dostępne używając tych samych instrukcji jak przy rejestrach ogólnego przeznaczenia, ale mają one specjalne znaczenie, specjalne użycie lub są po prostu szybsze kiedy są używane w specyficzny sposób.
Tak więc, instrukcja **movl** przenosi liczbę **1** do **%eax**. Znak dolara przed jedynką oznacza, że chcemy użyć trybu adresowania natychmiastowego (patrz Podrozdział *Metody Adresowania Danych* w Rozdziale 2). Bez znaku dolara mogłoby to oznaczać adresowanie bezpośrednie, ładując jakąkolwiek liczbę spod adresu **1**. Chcemy mieć załadowaną rzeczywistą liczbę **1**, więc musimy użyć trybu natychmiastowego.
Powód dla którego przesuwamy liczbę **1** do **%eax** jest taki, że zamierzamy wywołać Kernel Linuksa. Liczba **1** jest numerem *wywołania systemowego* **exit**. Będziemy dyskutować o wywołaniach systemowych bardziej głęboko wkrótce, ale generalnie są one prośbami o pomoc systemu operacyjnego. Normalne programy nie mogą zrobić wszystkiego. Wiele operacji takich jak wywołania innych programów, praca z plikami i wychodzenie muszą być robione przez system operacyjny poprzez wywołania systemowe. Kiedy zrobisz wywołanie systemowe, które wkrótce zrobimy, numer wywołania systemowego musi być załadowany do **%eax** (kompletna lista wywołań systemowych i ich numerów, patrz Dodatek C). W zależności od wywołania systemowego, inne rejestry mogą musieć mieć wartości także. Zauważ, że wywołania systemowe nie są jedynymi lub nawet nie głównymi użyciami rejestrów. To jest tylko jedno z działań w tym pierwszym programie. Późniejsze programy będą używały rejestrów do zwykłych obliczeń.
System operacyjny, zwykle jednak potrzebuje więcej informacji niż tylko które wywołanie zrobić. Na przykład, pracując z plikami, system operacyjny potrzebuje wiedzieć z którym plikiem, jakie dane chcemy zapisać, i inne szczegóły. Te dodatkowe szczegóły, zwane *parametrami*, są umieszczane w innych rejestrach. W przypadku wywołania systemowego **exit**, system operacyjny wymaga kodu statusu załadowanego do **%ebx**. Ta wartość jest wtedy zwracana do systemu. To jest wartość którą otrzymasz kiedy wpiszesz **echo $?**. Więc, ładujemy **0** do **%ebx** przez wpisanie następującego:

```
movl $0, %ebx
```

Teraz, ładując rejestry tymi liczbami nic nie zrobimy sami. Rejestry są używane do wszelakiego rodzaju rzeczy oprócz wywołań systemowych. Są one miejscem gdzie cała logika programu taka jak dodawanie, odejmowanie i porównywanie się odbywa. Linux po prostu potrzebuje tych konkretnych rejestrów załadowanych określonymi wartościami parametrów przed wykonaniem wywołania systemowego. **%eax** zawsze wymaga załadowania numerem wywołania systemowego. Dla innych rejestrów, jednakże, każde wywołanie systemowe ma inne wymagania. W wywołaniu systemowym **exit**, **%ebx** wymaga załadowania statusem wyjścia. Będziemy dyskutować różne wywołania systemowe kiedy będą potrzebne. Lista powszechnych wywołań systemowych i co jest wymagane w każdym rejestrze, patrz Dodatek C.
Następna instrukcja jest "magiczna". Wygląda tak:

```
int $0x80
```

int oznacza *przerwanie*. **0x80** jest użytym numerem przerwania. *Przerwanie* przerywa normalny tok programu i przekazuje kontrolę z naszego programu do Linuksa, to więc zrobi przerwanie systemowe. Możesz myśleć o nim jak o sygnałowym Batmanie. Potrzebujesz mieć coś zrobione, przesyłasz sygnał, i on przychodzi na ratunek. Nie dbasz o to jak on to robi - bardziej lub mniej jest to magiczne - i kiedy to zrobił masz z powrotem kontrolę. W tej sytuacji, wszystko co robimy to prosimy Linuksa do przeprowadzenia programu zanim będziemy mieli kontrolę z powrotem. Jeśli byśmy nie sygnalizowali przerwania, wtedy żadne wywołanie systemowe nie byłoby przeprowadzone.

Szybki Przegląd Wywołania Systemowego: Dla przypomnienia - własności Systemu Operacyjnego są osiągane poprzez wywołania systemowe. Są one przeprowadzane przez ustawienie rejestrów w specjalny sposób i wykonanie instrukcji **int $0x80**. Linux wie które wywołanie systemowe chcemy przeprowadzić poprzez to co umieściliśmy w rejestrze **%eax**. Każde wywołanie systemowe ma inne wymagania co powinno być umieszczone w innych rejestrach. Wywołanie systemowe numer 1 jest wywołaniem systemowym **exit**, które wymaga umieszczenia kodu statusu w **%ebx**.

Teraz kiedy zasemblowałeś, zlinkowałeś, uruchomiłeś i przetestowałeś ten program, powinieneś dokonać podstawowych edycji. Przemyśl jak zmienić liczbę ładowaną do **%ebx**, i zobacz co wyjdzie na końcu z **echo $?**. Nie zapomnij zasemblować i zlinkować powtórnie przed uruchomieniem. Dodaj kilka komentarzy. Nie przejmuj się, najgorsze co może się zdarzyć to, że program się nie zasembluje lub nie zlinkuje lub zamrozi twój ekran. To część nauki!

Planowanie Programu

W naszym następnym programie spróbujemy znaleźć wartość maksymalną z listy liczb. Komputery są bardzo zorientowane na szczegóły, więc w celu napisania tego programu będziemy musieli zaplanować szereg szczegółów. Te szczegóły zawierają:
- Gdzie oryginalna lista liczb będzie umieszczona?
- Jakiej procedury powinniśmy przestrzegać aby znaleźć liczbę maksymalną?
- Ile miejsca potrzebujemy aby przeprowadzić tę procedurę?
- Czy na miejsce potrzebne wystarczą rejestry, czy potrzebujemy użyć także trochę pamięci?

Mógłbyś pomyśleć, że coś tak prostego jak znalezienie liczby maksymalnej z listy nie zabrałoby wiele planowania. Zwykle możesz powiedzieć ludziom aby znaleźli liczbę maksymalną i mogą to zrobić bez trudu. Jednakże, nasze umysły nawykły do rozwiązywania złożonych zadań automatycznie. Komputery potrzebują być instruowane podczas procesu. W dodatku, zwykle możemy trzymać szereg rzeczy w naszej pamięci bez wielkiego trudu. Zwykle nawet nie dostrzegamy, że to robimy. Na przykład, jeśli przeglądasz listę liczb, prawdopodobnie będziesz trzymał w pamięci zarówno największą dotychczas widzianą i w którym miejscu jesteś listy. Podczas gdy twój umysł robi to automatycznie, w komputerach musisz ustawić pamięć dla trzymania bieżącej pozycji na liście i bieżącą największą liczbę. Masz także inne problemy takie jak w jaki sposób poznać kiedy skończyć. Kiedy czytasz kartkę papieru, możesz skończyć kiedy zabraknie liczb. Jednakże, komputer ma tylko liczby, więc nie ma pojęcia kiedy osiągnął twoją ostatnią liczbę.

W komputerach, musisz planować każdy krok w ten sposób. Więc, zróbmy małe planowanie. Przede wszystkim, tylko dla referencji, nazwijmy adres gdzie zaczyna się lista liczb jako data_items. Powiedzmy, że ostatnią liczbą na liście będzie zero, więc wiemy gdzie zakończyć. Potrzebujemy jeszcze wartości do trzymania bieżącej pozycji na liście, wartości do trzymania aktualnie testowanego elementu listy, i bieżącej największej wartości na liście. Przypiszmy każdej z nich rejestr:
- **%edi** będzie przechowywał bieżącą pozycję na liście
- **%ebx** będzie przechowywał bieżącą największą wartość na liście
- **%eax** będzie przechowywał aktualnie testowany element.

Kiedy zaczniemy program i spojrzymy na pierwszy element listy, ponieważ nie widzieliśmy żadnego innego, automatycznie będzie on aktualnie największym elementem na liście. Także, ustawimy bieżącą pozycję na liście na zero - pierwszy element. Odtąd, będziemy wykonywać następujące kroki:
1. Sprawdzanie bieżącego elementu listy (**%eax**) aby stwierdzić czy jest zerem (kończący element).
2. Jeśli jest zerem, wyjście.
3. Zwiększenie bieżącej pozycji (**%edi**).
4. Załadowanie następnej wartości z listy do rejestru bieżącej wartości (**%eax**). Jakiego trybu adresowania moglibyśmy tutaj użyć? Dlaczego?
5. Porównanie bieżącej wartości (**%eax**) z aktualnie największą wartością (**%ebx**).
6. Jeśli bieżąca wartość jest większa niż aktualnie największa wartość, wymiana największej aktualnie wartości na bieżącą wartość.
7. Powtórka.

Oto jest procedura. Wiele razy w tej procedurze użyłem słowa "jeśli". To miejsca gdzie podejmowane są decyzje. Widzisz, że komputer nie przestrzega dokładnie tych samych instrukcji za każdym razem. Zależnie od tego które "jeśli" jest poprawne, komputer może wykonywać różne zestawy instrukcji. Druga sprawa, może nie być największej wartości. W tym przypadku, będzie pominięty krok 6, ale powrót do kroku 7. W każdym przypadku oprócz ostatniego, będzie pominięty krok 2. W bardziej skomplikowanych programach, pominięcia rosną dramatycznie.

Te "jeśli" są klasą instrukcji zwaną *kontrolą przepływu* instrukcji, ponieważ mówią one komputerowi które kroki wykonywać i które ścieżki podjąć. W poprzednim programie, nie mieliśmy żadnej instrukcji kontroli przepływu, jako że była tylko jedna ścieżka możliwa - wyjście. Ten program jest dużo bardziej dynamiczny w tym że jest reżyserowany przez dane. Zależnie od tego jakie dane otrzyma, podąży różną ścieżką instrukcji.
W tym programie, będą temu towarzyszyły dwie różne instrukcje, skok warunkowy i skok bezwarunkowy. Skok warunkowy zmienia ścieżkę bazując na wynikach wcześniejszego porównania lub obliczenia. Skok bezwarunkowy idzie bezpośrednio do innej ścieżki bez względu na cokolwiek. Skok bezwarunkowy może się wydawać bezużyteczny, ale jest bardzo potrzebny skoro wszystkie instrukcje będą ułożone w linii. Jeśli ścieżka potrzebuje powrócić do głównej ścieżki, musi to zrobić poprzez skok bezwarunkowy. Będziemy oglądać więcej obydwu tych skoków w następnym podrozdziale.
Następne użycie kontroli przepływu jest w implementacji pętli. Pętla jest częścią kodu programu który ma być powtarzany. W naszym przykładzie, pierwsza część programu (ustawianie bieżącej pozycji na 0 i załadowanie aktualnej największej wartości wartością bieżącą) była zrobiona raz, więc nie była to pętla. Jednakże, następna część jest powtarzana za każdym razem dla każdej liczby na liście. Jest zarzucona tylko kiedy dochodzimy do ostatniego elementu, odznaczonego przez zero. Jest zwana *pętlą* ponieważ pojawia się raz za razem. Jest zaimplementowana przez wykonywanie skoków bezwarunkowych do początku pętli na końcu pętli, co sprawia, że startuje znowu. Jednakże, musisz zawsze pamiętać aby mieć skok warunkowy do wyjścia z pętli w którymś miejscu, lub pętla będzie kontynuowana bez końca! Ten warunek jest zwany *pętlą nieskończoną*. Jeśli przez przypadek zabrakłoby kroku 1,2 lub 3, pętla (i nasz program) mógłby się nigdy nie skończyć.
W następnym podrozdziale, będziemy implementować ten program który planowaliśmy. Planowanie programu brzmi skomplikowanie - i tak jest, do pewnego stopnia. Kiedy rozpoczynasz programowanie, często jest trudno zmienić nasz normalny proces myślenia na procedurę którą komputer może zrozumieć. Często zapominamy szeregu "tymczasowych lokalizacji pamięci" które nasz umysł używa do rozwiązania problemów. Jednak kiedy będziesz pisał i czytał programy, w końcu stanie się to bardzo naturalne dla ciebie. Tylko miej cierpliwość.

Szukanie Wartości Maksymalnej

Otwórz następujący program jako **maximum.s**:

```
#CEL: Program szuka największej liczby z zestawu danych elementów.

#ZMIENNE: Rejestry mają następujące użycie:
# %edi - Przechowuje indeks testowanego elementu danych
# %ebx - Największy znaleziony element danych
# %eax - Bieżący element danych
#
#Następujące lokalizacje pamięci są używane:
#
# data_items - zawiera elementy danych. 0 jest używane jako koniec danych
#

.section .data

data_items: #To są elementy danych
.long 3,67,34,222,45,75,54,34,44,33,22,11,66,0

.section .text
.globl _start
```

_start:
movl $0, %edi # przesuwa (kopiuje) 0 do rejestru indeksowego
movl data_items(,%edi,4), %eax # ładuje pierwszy bajt danych
movl %eax, %ebx # ponieważ to pierwszy element, %eax jest największy

start_loop: # początek pętli
cmpl $0, %eax # sprawdzenie czy osiągamy koniec
je loop_exit
incl %edi # załadowanie następnej wartości
movl data_items(,%edi,4), %eax
cmpl %ebx, %eax # porównuje wartości
jle start_loop # skok do początku pętli jeśli nowa wartość nie jest większa
movl %eax, %ebx # przesuwa (kopiuje) tę wartość jako największą
jmp start_loop # skok do początku pętli

loop_exit:
%ebx jest kodem statusu dla wywołania systemowego wyjścia (exit) i jednocześnie ma liczbę największą
movl $1, %eax # 1 jest wywołaniem systemowym wyjścia (exit)
int $0x80

Teraz, zasembluj i zlinkuj to tymi komendami:
as maximum.s -o maximum.o
ld maximum.o -o maximum
Teraz uruchom go, i sprawdź jego status.
./maximum
echo $?

Zauważysz, że zwraca wartość **222**. Spójrzmy na ten program i co on robi. Jeśli spojrzysz do komentarzy, zobaczysz, że program szuka największej z zestawu liczb (czyż komentarze nie są cudowne!). Możesz także zauważyć, że w tym programie mamy coś w sekcji danych. Oto linie w sekcji danych:

data_items: # To są elementy danych
.long 3,67,34,222,45,75,54,34,44,33,22,11,66,0

Zobaczmy to. **data_items** jest etykietą, która odnosi się do lokalizacji następującej po niej. Potem jest rozkaz, który zaczyna się od **.long**. Skłania to asembler do zarezerwowania pamięci na listę liczb, która następuje po nim. data_items odnosi się do lokalizacji pierwszego elementu. Ponieważ **data_items** jest etykietą, za każdym razem gdy potrzebujemy odnieść się do tego adresu możemy użyć symbolu **data_items**, a asembler podmieni go na adres gdzie zaczynają się liczby podczas asemblacji. Na przykład, instrukcja **movl data_items, %eax** mogłoby przenieść wartość 3 do **%eax**. Jest kilkanaście różnych typów lokalizacji pamięci innych niż **.long** które mogą być rezerwowane. Główne to następujące:

.byte
Bajty zabierają jedną lokalizację pamięci dla jednej liczby. Są ograniczone do liczb pomiędzy 0 i 255.

.int
Int (które różni się od instrukcji **int**) zabierają dwie lokalizacje pamięci dla każdej liczby. Są ograniczone do liczb pomiędzy 0 i 65535.

.long
Long zabierają cztery lokalizacje pamięci. Jest to taka sama przestrzeń jak używają rejestry, i dlatego są one użyte w tym programie. Mogą przechowywać liczby pomiędzy 0 i 4294967295.

.ascii
Dyrektywa **.ascii** jest aby umieścić znaki w pamięci. Każdy znak zajmuje jedną lokalizację pamięci (są one konwertowane w bajty wewnętrznie). Więc, jeśli dajesz komendę **.ascii "Hello there\0"**, asembler mógłby zarezerwować 12 lokalizacji pamięci (bajtów). Pierwszy bajt zawiera numeryczny kod dla **H**, drugi bajt zawiera numeryczny kod dla **e**, i tak dalej. Ostatni znak jest reprezentowany przez **\0**, i jest to znak końcowy (nie będzie wyświetlony, on tylko mówi innym częściom programu, że jest to koniec znaków). Litery i liczby które zaczynają się od backslasha reprezentują znaki które są niedrukowalne na klawiaturze lub łatwo oglądalne na ekranie. Na przykład, **\n** odnosi się do znaku "nowego wiersza" który powoduje, że komputer zaczyna wyjście w następnym wierszu i **\t** odnosi się do znaku tabulacji. Wszystkie litery w komendzie **.ascii** powinny być w cudzysłowie.
W naszym przykładzie, asembler rezerwuje 14 **.long**(ów), jeden za drugim. Ponieważ każdy long zajmuje 4 bajty, oznacza to, że cała lista zajmuje 56 bajtów. To są liczby wśród których będziemy poszukiwać największej. **data_items** jest użyta przez asembler do odniesienia się do adresu pierwszej z tych wartości.
Zauważ, że ostatnim elementem danych na liście jest zero. Zdecydowałem się użyć zera aby powiedzieć mojemu programowi, że osiągnął koniec listy. Mógłbym to zrobić w inny sposób. Mógłbym mieć rozmiar listy na stałe wpisany w program. Także, mógłbym umieścić długość listy jako pierwszy element, lub w osobnej lokalizacji. Mógłbym także umieścić symbol który oznaczałby ostatnią lokalizację elementu z listy. Nieważne jak to robię, muszę mieć kilka metod stwierdzenia końca listy. Komputer nic nie wie - może tylko zrobić co mu powiedziano. Nie zamierza zakończyć przetwarzania aż dam mu jakiegoś rodzaju sygnał. W innym razie mógłby kontynuować przetwarzanie danych poza koniec listy, w danych które następują po nim, i nawet w lokalizacjach gdzie nie umieściliśmy żadnych danych.
Zauważ, że nie mamy deklaracji **.globl** dla **data_items**. Jest tak ponieważ odnosimy się do lokalizacji wewnątrz programu. Żaden inny plik lub program nie potrzebuje wiedzieć gdzie się one znajdują. Stoi to w kontraście do symbolu **_start**, który Linux potrzebuje aby wiedzieć gdzie on jest a więc wie gdzie zacząć wykonywanie programu. Nie jest błędem napisać **.globl data_items**, tylko nie jest to konieczne. Jakkolwiek, poeksperymentuj z tym wierszem i dodaj własne liczby. Nawet jeśli są one **.long**, program będzie dawał dziwne wyniki jeśli którakolwiek liczba jest większa niż 255, ponieważ to jest największy dozwolony status wyjścia. Zauważ także, że jeśli przesuniesz 0 na pozycję wcześniejszą na liście, reszta będzie zignorowana. *Pamiętaj, że za każdym razem gdy zmienisz plik źródłowy, musisz zreasemblować i zrelinkować twój program. Zrób to teraz i zobacz wyniki*.
Dobrze, poeksperymentowaliśmy trochę z danymi. Teraz popatrzmy na kod. W komentarzach zauważysz, że zaznaczyliśmy kilka *zmiennych* które planujemy użyć. Zmienna jest dedykowaną lokalizacją pamięci używaną dla specjalnych celów, zwykle z wyróżnioną nazwą nadaną przez programistę. Mówiliśmy o nich w poprzednim podrozdziale, ale nie nadawaliśmy im nazwy. W tym programie, mamy kilka zmiennych:
- zmienną dla aktualnej największej znalezionej liczby
- zmienną dla tego numeru z listy który aktualnie testujemy, zwanego indeksem
- zmienną przechowującą aktualnie testowaną liczbę

W ten sposób, mamy kilka zmiennych które możemy przechowywać w rejestrach. W większych programach, musisz umieścić je w pamięci i wtedy przenosić do rejestrów kiedy jesteś gotów je użyć. Będziemy dyskutować jak to zrobić później. Kiedy ludzie zaczynają programować, zwykle niedoszacowują liczby zmiennych które będą potrzebować. Ludzie nie zwykli roztrząsać każdego detalu procesu, i dlatego opuszczają potrzebne zmienne w ich pierwszych programistycznych próbach.
W tym programie, używamy **%ebx** jako lokalizacji największego znalezionego elementu. **%edi** jest używany jako *indeks* do aktualnie przeglądanego elementu danych. Teraz, porozmawiajmy co to jest indeks. Kiedy czytamy informację z **data_items**, zaczynamy od pierwszej (element danych numer 0), potem idziemy do drugiej (element danych numer 1), potem do trzeciej (element danych numer 2), i tak dalej. Numer elementu danych jest *indeksem* **data_items**. Zauważysz, że pierwszą instrukcją którą dajemy komputerowi jest:

movl $0, %edi

Odkąd używamy **%edi** jako nasz indeks, i chcemy zacząć od pierwszego elementu, ładujemy 0 do **%edi**. Teraz, następna instrukcja jest sztuczką, ale kluczową dla tego co robimy. Mówi ona:

movl data_items(,%edi,4), %eax

Teraz aby zrozumieć ten wiersz, powinieneś mieć kilka rzeczy na uwadze:
- **data_items** jest numerem lokalizacji początku naszej listy liczb.
- Każda liczba jest przechowywana przez 4 lokalizacje pamięci (ponieważ zadeklarowaliśmy używanie **.long**)
- **%edi** przechowuje 0 w tym momencie

Tak więc, głównie co ten wiersz mówi, to "zacznij na początku data_items i pobierz pierwszy element liczbowy (ponieważ **%edi** wynosi 0), pamiętaj, że każda liczba zajmuje cztery lokalizacje pamięci". Wtedy liczba jest umieszczana w **%eax**. Oto jak wpisujesz instrukcje indeksowego trybu adresowania w języku asemblerowym. Ta instrukcja w formie ogólnej wygląda tak:

movl BEGINNINGADDRESS(,%INDEXREGISTER,WORDSIZE)

W naszym przypadku **data_items** było naszym adresem początkowym, **%edi** było naszym rejestrem indeksowym, i **4** było naszym rozmiarem słowa. Ten temat jest dyskutowany głębiej w Podrozdziale zwanym *Tryby Adresowania*.
Jeśli spojrzysz na liczby w **data_items**, zobaczysz, że liczba 3 jest teraz w **%eax**. Jeśli **%edi** byłby ustawiony na 1, liczba 67 byłaby w **%eax**, i jeśli **%edi** byłby 2, liczba 34 byłaby w **%eax**, i tak dalej. Bardzo dziwne rzeczy mogłyby się wydażyć jeśli użylibyśmy liczby innej niż 4 jako rozmiaru naszej lokalizacji pamięci. Sposób w jaki to piszemy jest bardzo nieprzyjemny, ale jeśli wiesz co każdy element robi, nie jest to zbyt trudne. Więcej informacji o tym, zobacz Podrozdział zwany *Tryby Adresowania*.
Spójrzmy na następny wiersz:

movl %eax, %ebx

Mamy do przeglądnięcia pierwszy element zachowany w **%eax**. Skoro jest to pierwszy element, wiemy, że jest to największy element który przeglądaliśmy. Umieszczamy go w **%ebx**, ponieważ tam trzymamy największe znalezione liczby. Także, pomimo tego, że **movl** oznacza przesunięcie, w rzeczywistości kopiuje wartość, więc **%eax** i **%ebx** obydwie zawierają początkową wartość.
Teraz przechodzimy do *pętli*. Pętla jest segmentem twojego programu, który może przebiegać więcej niż raz. Zaznaczyliśmy lokalizację początku pętli symbolem **start_loop**. Powodem dla którego robimy pętlę jest to, że nie wiemy ile elementów danych musimy przetworzyć, ale procedura będzie taka sama bez względu na to jak wiele ich jest. Nie chcemy przepisywać naszego programu dla każdej możliwej długości listy. Faktycznie, nie chcemy nawet napisać kodu dla porównania każdego elementu listy. Dlatego, mamy pojedynczą sekcję kodu (pętlę) którą wykonujemy raz za razem dla każdego elementu z **data_items**.
W poprzednim podrozdziale zarysowaliśmy co pętla powinna robić. Podsumujmy:
- Sprawdzanie czy aktualna wartość przetwarzana jest zerem. Jeśli tak, oznacza to, że jesteśmy na końcu naszych danych i powinniśmy wyjść z pętli.
- Musimy załadować następną wartość z naszej listy.
- Musimy sprawdzić czy następna wartość jest większa od naszej bieżącej największej wartości.
- Jeśli jest, musimy skopiować ją do lokalizacji gdzie przechowujemy największe wartości.
- Teraz potrzebujemy wrócić do początku pętli.

Dobrze, więc teraz przejdźmy do kodu. Mamy początek pętli oznaczony jako **start_loop**. Tak więc wiemy gdzie powrócić na końcu naszej pętli. Potem mamy te instrukcje:

cmpl $0, %eax
je end_loop

Instrukcja **cmpl** porównuje te dwie wartości. Tutaj, porównujemy liczbę 0 z liczbą przechowywaną w **%eax**. Ta instrukcja porównania także wiąże rejestr nie wspominany tutaj, rejestr **%eflags**. Jest on znany także jako rejestr statusu i ma wiele zastosowań które będziemy omawiać później. Bądź świadomy, że wynik porównania jest przechowywany w rejestrze statusu. Następny wiersz jest instrukcją warunkową która mówi *skocz* do lokalizacji końca pętli (**end_loop**) jeśli wartości które właśnie porównywaliśmy są równe (to jest to co oznacza **e** w **je**). Używa się rejestru statusu do przechowywania wartości ostatniego porównania. My użyliśmy **je**, ale jest wiele instrukcji skoku które możesz użyć:

je
Skocz jeśli wartości są równe

jg
skocz jeśli druga wartość jest większa niż pierwsza wartość

jge
Skocz jeśli druga wartość jest większa niż lub równa pierwsza wartość

jl
Skocz jeśli druga wartość jest mniejsza niż pierwsza wartość

jle
Skocz jeśli druga wartość jest mniejsza lub równa pierwszej wartości

jmp
Skocz bezwarunkowo. Nie potrzebuje porównania.

Kompletna lista jest udokumentowana w Dodatku B. W ten sposób, skaczemy jeśli **%eax** przechowuje wartość zero. Jeśli tak, skończyliśmy i idziemy do **loop_exit**.
Jeśli ostatni załadowany element nie był zerem, przechodzimy do następnych instrukcji:

incl %edi
movl data_item(,%edi,4), %eax

Jeśli pamiętasz z naszej poprzedniej dyskusji, **%edi** zawiera indeks do naszej listy wartości w **data_item**. **incl** zwiększa wartość **%edi** o jeden. Wtedy **movl** jest podobne jak poprzednio. Jednakże, ponieważ zwiększyliśmy **%edi**, **%eax** posiada następną wartość z listy. Teraz **%eax** ma następną wartość do testowania. Więc, przetestujmy ją!

cmpl %ebx, %eax
jle start_loop

Tutaj porównujemy naszą bieżącą wartość, umieszczoną w **%eax** z naszą największą wartością dotąd, umieszczoną w **%ebx**. Jeśli bieżąca wartość jest mniejsza lub równa naszej największej dotąd wartości, nie zwracamy na nią uwagi, skaczemy z powrotem do początku pętli. W przeciwnym razie, potrzebujemy zapisać tę wartość jako największa:

movl %eax, %ebx
jmp start_loop

co przesuwa bieżącą wartość do **%ebx**, które używamy do przechowywania aktualnie największej wartości, i zaczynamy pętlę znowu.
Dobrze, więc pętla działa aż do osiągnięcia 0, kiedy skaczemy do **loop_exit**. Ta część programu wywołuje kernel Linuksa celem wyjścia. Jeśli pamiętasz z ostatniego programu, kiedy wywołujesz system operacyjny (pamiętaj jest to jak wzywanie Batmana), umieszczasz numer wywołania systemowego w **%eax** (**1** dla wywołania **exit**-wyjście), i umieszczasz inne wartości w innych rejestrach. Wywołanie exit wymaga żebyśmy umieścili status wyjścia w **%ebx**. Już mamy tam status wyjścia skoro używamy **%ebx** jako naszej największej liczby, więc wszystko co musimy zrobić to załadować **%eax** liczbą jeden i wywołać kernel exit. W taki sposób:

movl $1, %eax
int 0x80

Dobrze, to było wiele pracy i tłumaczenia, specjalnie dla tak małego programu. Ale uczysz się wiele! Teraz, przeczytaj ten cały program ponownie, zwracając specjalną uwagę na komentarze. Upewnij się, że rozumiesz co się dzieje w każdym wierszu. Jeśli nie rozumiesz jakiegoś wiersza, powróć do tego podrozdziału i odnajdź co ten wiersz oznacza.
Mógłbyś także wziąć kartkę papieru, i przejść ten program krok po kroku, zapisując każdą zmianę w każdym rejestrze, tak możesz zobaczyć bardziej jasno co się dzieje.

Tryby Adresowania

W Podrozdziale nazwanym *Metody Adresowania Danych* w Rozdziale 2 uczyliśmy się różnych typów trybów adresowania dostępnych w języku asemblerowym. Ten podrozdział będzie się zajmować jak te tryby adresowania są reprezentowane w instrukcjach języka asemblerowego.
Ogólna forma referencji adresu pamięci jest taka:

ADRES_LUB_PRZESUNIĘCIE(%BAZA_LUB_PRZESUNIĘCIE,%INDEKS,WSPÓŁCZYNNIK_SKALOWANIA)

Wszystkie pola są opcjonalne. Aby wyliczyć adres, po prostu przeprowadź następujące obliczenie:

ADRES KOŃCOWY = ADRES_LUB_PRZESUNIĘCIE + %BAZA_LUB_PRZESUNIĘCIE + WSPÓŁCZYNNIK_SKALOWANIA * %INDEKS

ADRES_LUB_PRZESUNIĘCIE i **WSPÓŁCZYNNIK_SKALOWANIA** muszą być stałymi, podczas gdy te dwa inne muszą być rejestrami. Jeśli którykolwiek składnik nie występuje, jest zastąpiony przez zero w tym równaniu.
Wszystkie tryby adresowania wspominane w Podrozdziale zwanym *Metody Adresowania Danych* w Rozdziale 2 oprócz trybu natychmiastowego mogą być reprezentowane w tym stylu.

tryb adresowania bezpośredniego
Jest robiony przez użycie tylko części ADRES_LUB_PRZESUNIĘCIE. Przykład:
movl ADRES, %eax
To ładuje **%eax** wartością spod adresu pamięci ADRES.

tryb adresowania indeksowego
Jest robiony przez użycie części **ADRES_LUB_PRZESUNIĘCIE** i **%INDEKS**. Możesz użyć któregokolwiek rejestru

ogólnego przeznaczenia jako rejestru indeksowego. Możesz także mieć stały współczynnik skalowania 1, 2 lub 4 dla rejestru indeksowego, żeby prościej zaindeksować przez bajty, podwójne bajty, lub słowa. Na przykład, powiedzmy, że mamy łańcuch bajtów jako **string_start** i chcemy udostępnić trzeci element (o indeksie 2 ponieważ zaczynamy liczyć indeks od zera), i **%ecx** przechowuje tę wartość 2. Jeśli chciałbyś załadować ją do **%eax** mógłbyś zrobić co następuje:
movl string_start(,%ecx,1), %eax
To zaczyna w **string_start**, dodaje **1 * %ecx** do tego adresu i ładuje tę wartość do **%eax**.

tryb adresowania pośredni
Tryb adresowania pośredni ładuje wartość spod adresu wskazywanego przez rejestr. Na przykład, jeśli **%eax** przechowuje adres, moglibyśmy przesunąć tę wartość tego adresu do **%ebx** poprzez następujące działanie:
movl (%eax), %ebx

tryb adresowania wskaźnika bazowego
Adresowanie wskaźnika bazowego jest podobne do adresowania pośredniego, oprócz tego, że dodaje ono stałą wartość do adresu w rejestrze. Na przykład, jeśli masz rekord gdzie wartość wiek jest 4 bajty w rekordzie, i masz adres tego rekordu w **%eax**, możesz wyciągnąć wiek do **%ebx** poprzez przeprowadzenie następującej instrukcji:
movl 4(%eax), %ebx

tryb natychmiastowy
Tryb natychmiastowy jest bardzo prosty. Nie podlega ogólnej formie, której używamy. Tryb natychmiastowy jest używany do załadowania wartości bezpośredniej do rejestrów lub lokalizacji pamięci. Na przykład, jeśli chcesz załadować liczbę 12 do **%eax**, mógłbyś po prostu zrobić następująco:
movl $12, %eax
Zauważ, że aby wskazać tryb natychmiastowy, użyliśmy znaku dolara przed liczbą. Jeśli nie zrobilibyśmy tego, byłby to tryb adresowania bezpośredni, w tym przypadku wartość znajdująca się w lokalizacji pamięci 12 mogłaby być załadowana do **%eax** raczej niż liczba 12.

tryb adresowania rejestrowego
Tryb rejestrowy po prostu przenosi dane do lub z rejestru. We wszystkich naszych przykładach, tryb adresowania rejestrowego był używany dla innego operandu (gdy operandami są tylko rejestry).

Te tryby adresowania są bardzo ważne, jako, że każdy dostęp do pamięci będzie używał jednego z nich. Każdy tryb oprócz trybu natychmiastowego może być użyty zarówno jako operand źródła lub przeznaczenia. Tryb natychmiastowy może być tylko operandem źródła.
W dodatku do tych trybów, są także różne instrukcje dla różnych rozmiarów wartości do przenoszenia. Na przykład, używaliśmy **movl** do przenoszenia danych po słowie na raz. W wielu wypadkach, będziesz chciał przenosić dane tylko po bajcie na raz. Towarzyszy temu instrukcja **movb**.
Jednakże, ponieważ rejestry o których dyskutowaliśmy są rozmiaru słowa a nie bajta, nie możesz użyć całego rejestru. Faktycznie, musisz używać części rejestru.
Weź na przykład **%eax**. Jeśli chciałbyś pracować tylko z dwoma bajtami na raz, mógłbyś właśnie użyć **%ax**. **%ax** jest mniej znaczącą połową (t.j. - ostatnią częścią tej liczby) rejestru **%eax**, i jest użyteczna kiedy zajmujemy się wielkościami dwu-bajtowymi. **%ax** jest dalej dzielone na **%al** i **%ah**. **%al** jest mniej znaczącym bajtem **%ax**, a **%ah** jest bardziej znaczącym bajtem. Ładowanie wartości do **%eax** będzie czyściło to co było w **%al** i **%ah** (i także **%ax**, ponieważ **%ax** składa się z nich). Podobnie, ładowanie wartości do zarówno **%al** lub **%ah** będzie zaburzało wartość która poprzednio była w **%eax**. Ogólnie, mądrze jest używać rejestrów tylko dla bajtów lub tylko dla słów, ale nigdy dla obydwu w tym samym czasie.
Bardziej kompetentną listę instrukcji, zobacz Dodatek B.

Przegląd

Znajomość Koncepcji

- Co oznacza jeśli wiersz programu zaczyna się od znaku '#'?
- Jaka jest różnica pomiędzy plikiem języka asemblerowego a plikiem kodu obiektowego?
- Co robi linker?
- Jak sprawdzisz kod statusu wyniku ostatnio uruchomionego programu?
- Jaka jest różnica pomiędzy **movl $1, %eax** a **movl 1, %eax**?
- Jaki rejestr przechowuje numer wywołania systemowego?
- Po co są używane indeksy?
- dlaczego indeksy zwykle rozpoczynają od 0?
- Jeśli wpisałem komendę **movl data_item(,%edi,4), %eax** a data_item miało adres 3634 i **%edi** przechowuje wartość 13, jakiego adresu użyłbyś do przeniesienia do **%eax**?
- Wylicz rejestry ogólnego przeznaczenia.
- Jaka jest różnica pomiędzy **movl** i **movb**?
- Co to jest kontrola przepływu?
- Co robi skok warunkowy?
- Jakie sprawy musisz zaplanować kiedy piszesz program?
- Przejdź przez każdą instrukcję i wypisz jaki tryb adresowania był użyty dla każdego operandu?

Użycie Koncepcji

- Zmodyfikuj pierwszy program aby zwracał wartość 3.
- Zmodyfikuj program **maximum** aby szukał najmniejszej wartości.
- Zmodyfikuj program **maximum** aby używać liczby 255 jako koniec listy zamiast 0.
- Zmodyfikuj program **maximum** aby używać raczej końcowego adresu niż liczby 0 żeby wiedzieć kiedy zakończyć.
- Zmodyfikuj program **maximum** aby używać licznika długości raczej niż liczby 0 żeby wiedzieć kiedy zakończyć.
- Co mogłaby zrobić instrukcja **movl _start, %eax**? Bądź precyzyjny, bazuj na swojej wiedzy o zarówno trybach adresowania jak i znaczeniu **_start**. Jak mogłoby się to różnić od instrukcji **movl $_start, %eax**?

Idąc Dalej

- Zmodyfikuj pierwszy program opuszczając wiersz z instrukcją **int**. Zasembluj, zlinkuj, i wykonaj nowy program. Jaką informację o błędzie otrzymałeś. Jak myślisz dlaczego?
- Jak dotąd, przedyskutowaliśmy trzy metody szukania końca listy - użycie specjalnej liczby, użycie adresu końcowego, i użycie licznika długości. Jak myślisz, która metoda jest najlepsza? Dlaczego? Której metody użyłbyś jeśli wiedziałbyś, że lista była posortowana? Dlaczego?

Rozdział 4. Wszystko O Funkcjach

Traktowanie Złożoności

W Rozdziale 3, programy które napisaliśmy posiadały tylko jedną sekcję kodu. Jednakże, jeśli napisalibyśmy rzeczywiste programy w ten sposób, mogłoby być niemożliwe aby je uruchomić. Mogłoby być rzeczywiście trudno zebrać wiele osób pracujących nad projektem, jako że zmiana w jednej części może niekorzystnie wpływać na inną część nad którą pracuje inny twórca.

Aby pomóc programistom we wspólnym pracowaniu w grupach, konieczne jest dzielenie programów na osobne części, które komunikują się pomiędzy sobą poprzez dobrze zdefiniowane interfejsy. W ten sposób, każda część może być

tworzona i testowana niezależnie od innych, ułatwiając pracę wielu programistom nad tym projektem.
Programiści używają *funkcji* do rozdzielenia swoich programów na części które mogą być niezależnie tworzone i testowane. Funkcje są jednostkami kodu które wykonują zdefiniowaną część pracy na wyspecyfikowanych typach danych. Na przykład, w programie procesora tekstu, mogę mieć funkcję zwaną **handle_typed_character** która jest aktywowana kiedykolwiek użytkownik naciśnie klawisz. Dane, których funkcja używa mogłyby być naciśnięcie klawisza i dokument który użytkownik aktualnie otworzył. Funkcja mogłaby wtedy zmodyfikować dokument stosownie do naciśnięcia klawisza. Elementy danych przekazywanych funkcji do przetworzenia są zwane jej *parametrami*. W przykładzie procesora tekstu, klawisz który był naciśnięty i dokument mogłyby być uważane za parametry funkcji **handle_typed_characters**. Lista parametrów i oczekiwania przetwarzania funkcji (co jest oczekiwane do zrobienia z parametrami) są zwane interfejsem funkcji. Wiele uwagi zwraca się na projektowanie interfejsów funkcji, ponieważ jeśli są one wywoływane z wielu miejsc wewnątrz projektu, jest trudno zmieniać je w razie potrzeby.
Typowy program jest złożeniem setek lub tysięcy funkcji, każda z małym, dobrze określonym zadaniem do wykonania. Jednakże, ostatecznie są rzeczy dla których nie możesz napisać funkcji, które muszą być udostępniane przez system. Są one zwane *funkcjami prymitywnymi* (lub po prostu *prymitywami*) - są one podstawami z których wszystko inne jest zbudowane. Na przykład, wyobraź sobie program który rysuje graficzny interfejs użytkownika. Musi być funkcja do kreowania menu. Ta funkcja prawdopodobnie wywołuje inne funkcje do pisania tekstu, do rysowania ikon, do kolorowania tła, do obliczania gdzie jest wskaźnik myszy, itd. Jednakże, końcowo, sięgną one zestawu prymitywów udostępnianych przez system operacyjny do robienia podstawowych linii lub rysowania punktu. Programowanie może zarówno być widziane jako rozbieranie dużych programów na mniejsze części aż do osiągnięcia funkcji prymitywnych, lub wznosząco budowanie funkcji na prymitywach aż do osiągnięcia dużego. W języku asemblerowym, prymitywy zwykle są tym samym co wywołania systemowe, nawet pomimo że wywołania systemowe nie są prawdziwymi funkcjami o czym będziemy rozmawiać w tym rozdziale.

Jak Działają Funkcje

Funkcje składają się z kilku różnych części:

nazwy funkcji
Nazwa funkcji jest symbolem który reprezentuje adres gdzie zaczyna się kod funkcji. W języku asemblera, ten symbol jest zdefiniowany przez napisanie nazwy funkcji jako etykiety przed kodem funkcji. To jest tak samo jak etykiety których używałeś dla skoków.

parametry funkcji
Parametry funkcji są elementami danych które są wprost dawane funkcji dla przetworzenia. Na przykład, w matematyce, jest funkcja sinus. Jeśli poleciłbyś komputerowi znalezienie sinusa z 2, sinus byłby nazwą funkcji, a 2 byłoby parametrem. Niektóre funkcje mają wiele parametrów, inne nie mają żadnych.

zmienne lokalne
Zmienne lokalne są miejscem przechowywania danych których funkcja używa podczas przetwarzania i co jest wyrzucane po zakończeniu. Jest czymś w rodzaju kartek brudnopisu. Funkcje dostają nową kartkę za każdym razem kiedy są aktywowane, i muszą wyrzucić ją kiedy zakończą przetwarzanie. Zmienne lokalne funkcji nie są dostępne dla żadnej innej funkcji wewnątrz programu.

zmienne statyczne
Zmienne statyczne są miejscem przechowywania danych których funkcja używa podczas przetwarzania które nie są wyrzucane po zakończeniu, ale są ponownie używane za każdym razem gdy kod funkcji jest aktywowany. Te dane nie są dostępne dla żadnej innej części programu. Zmienne statyczne są generalnie nieużywane chyba że są absolutnie konieczne,

jako że mogą powodować problemy później.

zmienne globalne
Zmienne globalne są miejscem przechowywania danych których funkcja używa do przetwarzania i które są zarządzane z zewnątrz funkcji. Na przykład, prosty edytor tekstu może umieścić całą zawartość pliku na którym pracuje w zmienną globalną i tak plik nie musi być przesyłany do każdej funkcji która na nim pracuje. Wartości konfiguracyjne są także często umieszczane w zmiennych globalnych.

adres powrotu
Adres powrotu jest "niewidzialnym" parametrem który nie jest bezpośrednio używany podczas wykonywania funkcji. Adres powrotu jest parametrem który mówi funkcji gdzie powrócić z wykonywaniem po zakończeniu funkcji. Jest to potrzebne ponieważ funkcje mogą być wywoływane do wykonania przetwarzania z wielu różnych części twojego programu, i funkcja powinna być zdolna wrócić do miejsca skąd była wywołana. W większości języków programowania, ten parametr jest przekazywany automatycznie kiedy funkcja jest wywoływana. W języku asemblerowym, instrukcja wywołania **call** przechowuje adres powrotu dla ciebie, i **ret** używając tego adresu umożliwia powrót do miejsca skąd wywołałeś funkcję.

wartość powrotu
Wartość powrotu jest główną metodą przekazywania danych z powrotem do głównego programu. Większość języków programowania tylko pozwala na pojedynczą wartość powrotu dla funkcji.

Wymienione elementy są obecne w większości języków programowania. Każda część jest różna w każdym z nich, jednakże. Sposób w jaki zmienne są przechowywane a parametry i wartości powrotu są przekazywane przez komputer także zmienia się z języka na język. Ta zmienność jest znana jako konwencja wywoływania języka, ponieważ opisuje ona czego funkcje oczekują aby dostać i przyjąć dane kiedy są one wywoływane.
Język asemblerowy może użyć jakiejkolwiek konwencji wywoływania jakiej chce. Możesz nawet zrobić sobie własną. Jednakże, jeśli chcesz współpracować z funkcjami napisanymi w innych językach, musisz poddać się ich konwencjom wywoływania. Będziemy używać konwencji wywoływania języka programowania C w naszych przykładach ponieważ jest on najszerzej używany, i ponieważ jest on standardem dla platformy Linux.

Funkcje Języka Asemblerowego używające Konwencji Wywołań C

Nie możesz napisać funkcji języka asemblerowego bez zrozumienia jak działa *stos* komputera. Każdy program komputerowy który się uruchamia używa rejonu pamięci zwanego stosem do umożliwienia funkcjom poprawnej pracy. Myśl o stosie jak o stercie papierów na twoim biurku do której możesz dodawać. Generalnie trzymasz rzeczy nad którymi pracujesz na wierzchu, i zdejmujesz rzeczy z którymi skończyłeś pracować.
Twój komputer także ma stos. Stos komputera znajduje się na samym szczycie adresów pamięci. Możesz wpychać wartości na szczyt stosu poprzez instrukcję **pushl**, która wpycha zarówno rejestr lub wartość pamięci na szczyt stosu. Dobrze, powiedzieliśmy, że jest to szczyt, ale "szczyt" stosu jest jednocześnie podstawą pamięci stosu. Chociaż jest to mylące, powód tego jest taki, że kiedy myślimy o stosie czegokolwiek - talerzy, papierów, itd. - myślimy o dodawaniu do i zdejmowaniu z jego szczytu. Jednakże, w pamięci stos zaczyna się na szczycie pamięci i wzrasta w dół zgodnie z wymaganiami architektonicznymi. Dlatego, kiedy odnosimy się do "szczytu stosu" pamiętaj, że jest to spód pamięci stosu. Możesz także spychać wartości ze szczytu używając instrukcji **popl**. To ściąga szczytową wartość ze stosu i umieszcza ją w rejestrze lub lokalizacji pamięci wg twojego wyboru..
Kiedy wpychamy wartość na stos, szczyt stosu przemieszcza się aby pomieścić dodatkową wartość. Możemy w rzeczywistości stale wpychać wartości na stos i będzie on wzrastał coraz dalej w dół pamięci aż osiągniemy nasz kod lub dane. Więc skąd wiemy gdzie jest bieżący "wierzchołek" stosu? Rejestr stosu, **%esp**, zawsze zawiera wskaźnik na bieżący wierzchołek stosu, gdziekolwiek to jest.

Za każdym razem gdy wkładamy coś na stos z **pushl**, **%esp** zmniejsza się o **4** tak więc wskazuje nowy wierzchołek stosu (pamiętaj, każde słowo jest cztery bajty długie, a stos wzrasta w dół). Jeśli chcemy zdjąć coś ze stosu, po prostu używamy instrukcji **popl**, która dodaje **4** do **%esp** i odkłada poprzednią wartość wierzchołka w rejestrze który wybierzesz. **pushl** i **popl** każdy posiada jeden operand - rejestr do włożenia na stos dla **pushl**, lub przyjęcia danych zdejmowanych ze stosu dla **popl**.
Jeśli po prostu chcemy uzyskać dostęp do wartości na wierzchołku stosu bez zdejmowania jej, możemy użyć rejestru **%esp** w trybie pośrednim adresowania. Na przykład, następujący kod przenosi cokolwiek jest na wierzchołku stosu do **%eax**:

movl (%esp), %eax

Jeślibyśmy zrobili to tak:

movl %esp, %eax

wtedy **%eax** mógłby przechowywać wskaźnik na wierzchołek stosu raczej niż wartość na wierzchołku. Umieszczenie **%esp** w nawiasie powoduje, że komputer stosuje tryb pośredni adresowania, i dlatego dostajemy wartość wskazywaną przez **%esp**. Jeśli chcemy mieć dostęp do wartości zaraz poniżej wierzchołka stosu, możemy po prostu wydać taką instrukcję:

movl 4(%esp), %eax

Ta instrukcja używa trybu adresowania wskaźnika bazowego (zobacz Podrozdział *Metody Dostępu do Danych* w Rozdziale 2) który po prostu dodaje **4** do **%esp** przed sprawdzeniem wskazywanej wartości.
W konwencji wywoływania języka C, stos jest kluczowym elementem do implementacji zmiennych lokalnych, parametrów, i adresu powrotu funkcji.
Przed wykonaniem funkcji, program wkłada wszystkie parametry funkcji na stos w odwrotnej kolejności do zapisanej. Wtedy program przeprowadza instrukcję **call** (wywołania) wskazując którą funkcję życzy sobie rozpocząć. Instrukcja **call** robi dwie rzeczy. Po pierwsze wkłada adres następnej instrukcji, którą jest adres powrotu, na stos. Wtedy zmienia wskaźnik instrukcji (**%eip**) aby wskazywał miejsce rozpoczęcia funkcji. Więc, w czasie rozpoczęcia funkcji, stos wygląda tak ("wierzchołek" stosu jest na dole w tym przykładzie):

Parametr #N
...
Parametr 2
Parametr 1
Adres Powrotu <--- (%esp)

Każdy z tych parametrów funkcji został włożony na stos, i na końcu znajduje się adres powrotu. Teraz funkcja sama ma trochę pracy do wykonania.
Pierwszą rzeczą którą robi jest zapisanie bieżącego rejestru wskaźnika bazowego, **%ebp**, wykonując **pushl %ebp**. Wskaźnik bazowy jest rejestrem specjalnym używanym do dostępu parametrów funkcji i zmiennych lokalnych. Następnie, kopiuje wskaźnik stosu do **%ebp** wykonując **movl %esp, %ebp**. To pozwala być zdolnym do dostępu do parametrów funkcji jako stałych indeksów z wskaźnika bazowego. Mógłbyś pomyśleć, że możesz użyć wskaźnika stosu do tego. Jednakże, podczas twojego programu mógłbyś robić inne rzeczy ze stosem jak wkładanie argumentów do innych funkcji. Kopiując wskaźnik stosu do wskaźnika bazowego na początku funkcji pozwala zawsze wiedzieć gdzie twoje parametry są (i jak zobaczymy, także zmienne lokalne), nawet gdy być może wkładasz lub zdejmujesz coś ze stosu. **%ebp** zawsze będzie tam gdzie wskaźnik stosu był na początku funkcji, więc jest to bardziej lub mniej stałe odniesienie do *ramki stosu* (ramka stosu zawiera wszystkie zmienne stosu używane wewnątrz funkcji, włączając w to parametry, zmienne lokalne, i adres powrotu).
W tym momencie, stos wygląda tak:

Parametr #N <--- N*4+4(%ebp)
...
Parametr 2 <--- 12(%ebp)
Parametr 1 <--- 8(%ebp)
Adres Powrotu <--- 4(%ebp)
Stare %ebp <--- (%esp) i (%ebp)

Jak możesz zauważyć, do każdego parametru może być dostęp używając trybu adresowania wskaźnika bazowego przy użyciu rejestru **%ebp**.
Następnie, funkcja rezerwuje miejsce na stosie dla jakichkolwiek zmiennych lokalnych których potrzebuje. Jest to robione poprzez proste usuwanie wskaźnika stosu z drogi. Powiedzmy, że zamierzamy użyć dwu słów pamięci do uruchomienia funkcji. Możemy prosto przesunąć wskaźnik stosu w dół dwa słowa dla zarezerwowania miejsca. Jest to robione tak:

subl $8, %esp

To odejmuje **8** z **%esp** (pamiętaj, słowo ma długość czterech bajtów). W ten sposób, możemy użyć stosu do przechowywania zmiennych bez obaw o stracenie ich przez wkładania które moglibyśmy wykonywać dla wywołań funkcji. Także, odkąd jest to alokowane w ramce stosu dla tego wywołania funkcji, zmienna będzie istnieć tylko podczas funkcji. Kiedy powrócimy, ramka stosu zniknie, i to samo będzie ze zmiennymi. To jest dlaczego są one zwane lokalnymi - istnieją tylko gdy ta funkcja jest wywołana.
Teraz mamy dwa słowa na przechowywanie lokalne. Nasz stos wygląda teraz tak:

Parametr #N <--- N*4+4(%ebp)
...
Parametr 2 <--- 12(%ebp)
Parametr 1 <--- 8(%ebp)
Adres Powrotu <--- 4(%ebp)
Stare %ebp <--- (%ebp)
Zmienna Lokalna 1 <--- -4(%ebp)
Zmienna Lokalna 2 <--- -8(%ebp) i (%esp)

Teraz możemy mieć dostęp do wszystkich danych których potrzebujemy dla tej funkcji poprzez użycie adresowania wskaźnika bazowego używając różnych przesunięć od **%ebp**. **%ebp** było stworzone specjalnie do tego celu, dlatego jest zwane wskaźnikiem bazowym. Możesz używać innych rejestrów w trybie adresowania wskaźnika bazowego, ale architektura x86 sprawia, że użycie rejestru **%ebp** jest o wiele szybsze.
Zmienne globalne i zmienne statyczne są udostępniane tak jak udostępnialiśmy pamięć w poprzednich rozdziałach. Jedyna różnica pomiędzy zmiennymi globalnymi i statycznymi jest taka, że zmienne statyczne są używane tylko przez jedną funkcję, podczas gdy zmienne globalne są używane przez wiele funkcji. Język asemblerowy traktuje je dokładnie tak samo, chociaż większość innych języków rozróżnia je.
Kiedy funkcja wykonała zadanie, robi trzy rzeczy:
1. Umieszcza wartość powrotu w **%eax**.
2. Resetuje stos do tego co było kiedy była wywołana (pozbywa się bieżącej ramki stosu i z powrotem uaktywnia ramkę stosu kodu wywołującego).
3. Zwraca kontrolę z powrotem w miejsce skąd była wywołana. Jest to robione używając instrukcji **ret**, która zdejmuje wartość będącą na wierzchołku stosu, i ustawia wskaźnik instrukcji, **%eip**, na tą wartość.

Więc, zanim funkcja zwróci kontrolę do kodu który ją wywołał, musi odtworzyć poprzednią ramkę stosu. Zauważ także, że bez zrobienia tego, **ret** mogłoby nie zadziałać, ponieważ w naszej bieżącej ramce stosu, adres powrotu nie jest na

wierzchołku stosu. Dlatego, zanim powrócimy, musimy zresetować wskaźnik stosu **%esp** i wskaźnik bazowy **%ebp** do tych miejsc gdzie były one kiedy funkcja się zaczęła. Dlatego aby powrócić z funkcji musisz zrobić następująco:

movl %ebp, %esp
popl %ebp
ret

W tym miejscu, powinieneś uważać wszystkie zmienne lokalne za rozdysponowane. Powodem jest to, że po tym jak przesuniesz wskaźnik stosu z powrotem, przyszłe odkładania na stos prawdopodobnie nadpiszą wszystko co tam włożyłeś. Dlatego, nie powinieneś nigdy zapisywać adresu zmiennej lokalnej ponad czas trwania funkcji w której była utworzona, lub będzie ona nadpisana po zakończeniu trwania jej ramki stosu.
Kontrola teraz z powrotem została przekazana do kodu wywołującego, który może teraz przetestować **%eax** dla wartości powrotu. Kod wywołujący potrzebuje także zdjęcia wszystkich parametrów które włożył na stos w kolejności aby otrzymać wskaźnik stosu z powrotem tam gdzie był (możesz także po prostu dodać 4*liczba_parametrów do **%esp** używając instrukcji **addl**, jeśli nie potrzebujesz już wartości parametrów).

Destrukcja Rejestrów

Kiedy wywołujesz funkcję, powinieneś założyć, że wszystko co bieżąco jest w twoich rejestrach będzie wymiecione. Jedyny rejestr gdzie jest gwarancja, że będzie pozostawiony z wartością z którą wystartował jest **%ebp**. **%eax** jest gwarantowane, że będzie nadpisany, a inne prawdopodobnie też. Jeśli są rejestry które chcesz zachować przed wywołaniem funkcji, powinieneś zachować je przez włożenie ich na stos przed włożeniem parametrów funkcji. Możesz wtedy zdjąć je na powrót w odwrotnej kolejności po zdjęciu parametrów. Nawet jeśli wiesz, że funkcja nie nadpisuje rejestru powinieneś zachować go, ponieważ przyszłe wersje tej funkcji mogą.
Konwencje wywoływania innych języków mogą być różne. Na przykład, inne konwencje wywoływania mogą nakładać na funkcję zapisanie każdego rejestru którego używa. Upewnij się, że sprawdziłeś konwencje wywoływania swoich języków aby były kompatybilne przed próbowaniem połączenia języków. Lub, w przypadku języka asemblerowego, upewnij się, że wiesz jak wywołać funkcje innych języków.

Rozszerzona Specyfikacja: Szczegóły konwencji wywoływania języka C (zwanego także ABI, czyli Interfejs Binarny Aplikacji) jest dostępna w sieci. Uprościliśmy i pominęliśmy kilka ważnych spraw aby ułatwić zrozumienie przez nowych programistów. Aby poznać wszystkie szczegóły, powinieneś sprawdzić dokumenty dostępne na http://www.linuxbase.org /spec/refspecs/
Szczególnie, powinieneś szukać *System V Application Binary Interface - Intel386 Architecture Processor Supplement.*

Przykład Funkcji

Przyjrzyjmy się jak wywołanie funkcji działa w rzeczywistym programie. Funkcja którą zamierzamy napisać jest funkcją potęgowania (**power**). Damy funkcji potęgowania dwa parametry - liczbę i potęgę do której chcemy ją podnieść. Na przykład, jeśli dalibyśmy jej parametry 2 i 3, mogłaby podnieść 2 do potęgi 3, lub 2*2*2, co daje 8. Aby uprościć ten program, pozwolimy używać tylko liczb 1 i większych.
Poniżej jest kod dla kompletnego programu. Jak zwykle, podajemy wytłumaczenia. Nazwij ten plik **power.s**.

```
#CEL: Program ilustrujący jak działają funkcje
# Ten Program będzie obliczał wartość 2^3 + 5^2
#
```

```
#Wszystko w głównym programie jest umieszczone w rejestrach, więc w sekcji danych nie ma niczego.
.section .data

.section .text

.globl _start
_start:
pushl $3 #wkładamy drugi argument
pushl $2 #wkładamy pierwszy argument
call power #wywołanie funkcji
addl $8, %esp #przeniesienie z powrotem wskaźnika stosu
pushl %eax #zapisanie pierwszej odpowiedzi przed wywołaniem następnej funkcji
pushl $2 #włożenie drugiego argumentu
pushl $5 #włożenie pierwszego argumentu
call power #wywołanie funkcji
addl $8, %esp #przeniesienie z powrotem wskaźnika stosu
popl %ebx #Druga odpowiedź jest już w %eax.
#Zapisaliśmy pierwszą odpowiedź na stosie, więc teraz możemy zdjąć ją do %ebx
addl %eax, %ebx #dodanie odpowiedzi
#wynik jest w %ebx
movl $1, %eax #wyjście (exit) (%ebx jest zwracane)
int $0x80

#CEL: Ta funkcja jest użyta do obliczania wartości liczby podniesionej do potęgi
#
#WEJŚCIE: Pierwszy argument - liczba bazowa
# Drugi argument - potęga do której podnosimy
#
#WYJŚCIE: Będzie dawać wynik jako wartość zwracana
#
#UWAGI: Potęga musi być 1 lub większa
#
#ZMIENNE: %ebx - przechowuje liczbę podstawy
# %ecx - przechowuje potęgę
#
# -4(%ebp) - przechowuje bieżący wynik
#
# %eax jest użyty jako tymczasowe miejsce przechowywania
#
.type power, @function
power:
pushl %ebp #zapisanie starego wskaźnika bazowego
movl %esp, %ebp #robi wskaźnik stosu wskaźnikiem bazowym
subl $4, %esp #robi miejsce dla naszego lokalnego przechowywania
movl 8(%ebp), %ebx #wkłada pierwszy argument do %ebx
movl 12(%ebp), %ecx #wkłada drugi argument do %ecx
movl %ebx, -4(%ebp) #przechowuje bieżący wynik
power_loop_start:
cmpl $1, %ecx #jeśli potęga wynosi 1, kończymy
```

je end_power
movl -4(%ebp), %eax #przenosi bieżący wynik do %eax
imull %ebx, %eax #mnoży bieżący wynik przez liczbę bazową
movl %eax, -4(%ebp) #zachowuje bieżący wynik
decl %ecx #zmniejsza o 1 potęgę
jmp power_loop_start #uruchamia dla następnej potęgi
end_power:
movl -4(%ebp), %eax #wartość powrotu idzie do %eax
movl %ebp, %esp #odbudowa wskaźnika stosu
popl %ebp #odbudowa wskaźnika bazowego
ret

Wpisz ten program, zasembluj go, zlinkuj i uruchom. Spróbuj wywołać ten program dla różnych wartości ale pamiętaj, że wynik musi być mniejszy niż 256 kiedy jest przekazywany z powrotem do systemu operacyjnego. Także wypróbuj odejmowanie wyników tych dwu obliczeń. Wypróbuj dodawanie trzeciego wywołania funkcji **power**, i dodaj jego wynik.
Kod głównego programu jest bardzo prosty. Wkładasz argumenty na stos, wywołujesz funkcję i potem przenosisz wskaźnik stosu z powrotem. Wynik jest przechowywany w **%eax**. Zauważ, że pomiędzy dwoma wywołaniami do **power**, zapisujemy pierwszą wartość na stosie. Jest tak dlatego, że jedyny rejestr który ma gwarancję bycia zachowanym to **%ebp**. Dlatego wkładamy wartość na stos, i zdejmujemy z powrotem po skończeniu wywołania drugiej funkcji.
Zobaczmy jak funkcja jest napisana. Zauważ, że przed funkcją jest dokumentacja tego co funkcja robi, czym są jej argumenty, i co daje jako wartość powrotu. Jest to użyteczne dla programistów którzy używają tej funkcji. To jest interfejs funkcji. To pozwala programiście wiedzieć jakie wartości są potrzebne na stosie, i co będzie w **%eax** na końcu.
Wtedy mamy następujący wiersz:

.type power, @function

Mówi to linkerowi, że symbol **power** powinien być traktowany jako funkcja. Ponieważ program jest tylko w jednym pliku, mógłby działać tak samo z pominięciem tego. Jednakże, jest to dobra praktyka.
Po tym, definiujemy wartość etykiety **power**:

power:

Jak było wspominane poprzednio, definiuje to symbol **power** jako adres gdzie instrukcje następujące po tej etykiecie zaczynają się. Jest to jak wywołanie **call power** działa. Przekazuje kontrolę do tej części programu. Różnica pomiędzy **call** i **jmp** jest taka, że **call** także wkłada adres powrotu na stos więc funkcja może powrócić, podczas gdy **jmp** nie.
Następnie, mamy nasze instrukcje do ustawienia naszej funkcji:

pushl %ebp
movl %esp, %ebp
subl $4, %esp

W tym punkcie, nasz stos wygląda tak:

Liczba Bazowa <--- 12(%ebp)
Power <--- 8(%ebp)
Adres Powrotu <--- 4(%ebp)
Stare %ebp <--- (%ebp)
Bieżący Wynik <--- -4(%ebp) i (%esp)

Chociaż moglibyśmy używać rejestrów do przechowywania tymczasowego, ten program używa zmiennych lokalnych żeby pokazać jak je wprowadzać. Często nie ma wystarczająco dużo rejestrów do przechowywania wszystkiego, więc musisz wyładować je do zmiennych lokalnych. Innymi razy, twoja funkcja będzie potrzebować wywołania innej funkcji i przesłania jej wskaźnika do niektórych twoich danych. Nie możesz mieć wskaźnika do rejestru, więc musisz umieścić ją w zmiennej lokalnej aby przesłać wskaźnik do niej.
Generalnie, co robi ten program to zaczyna od liczby bazowej (podstawy), umieszcza ją raz jako mnożnik (w **%eax**) i raz jako bieżąca wartość (w **-4(%ebp)**). Ma także potęgę przechowywaną w **%ecx**. Wtedy ciągle mnoży bieżącą wartość przez mnożnik, zmniejszając potęgę, i opuszcza pętlę jeśli potęga (w **%ecx**) zmniejszy się do 1.
Od teraz, powinieneś być zdolny przejść ten program bez pomocy. Jedyne rzeczy które powinieneś poznać są te, że **imull** wykonuje mnożenie całkowitoliczbowe i umieszcza wynik w drugim operandzie, a **decl** zmniejsza dany rejestr o 1. Więcej informacji o tych i innych instrukcjach, zobacz Dodatek B.
Dobrym projektem do prób teraz jest rozszerzenie tego programu tak aby zwracał wartość liczby jeśli potęga wynosi 0 (podpowiedź, cokolwiek podniesione do potęgi 0 daje 1). Popróbuj. Jeśli nie działa za pierwszym razem, spróbuj przejść swój program ręcznie ze skrawkiem papieru, śledząc to co wskazują **%ebp** i **%esp**, co jest na stosie, i jakie wartości są w każdym rejestrze.

Funkcje Rekursywne

Następny program rozszerzy twoje horyzonty nawet więcej. Program będzie obliczał *silnię* z liczby. Silnia jest iloczynem liczby i wszystkich liczb pomiędzy nią i jedynką. Na przykład, silnia 7 to 7*6*5*4*3*2*1, a silnia 4 to 4*3*2*1. Teraz, jedna rzecz którą mógłbyś zauważyć to, że silnia liczby jest tym samym co iloczyn danej liczby i silni liczby o 1 mniejszej. Na przykład, silnia 4 wynosi 4 razy silnia 3. Silnia 3 wynosi 3 razy silnia 2. Silnia 2 wynosi 2 razy silnia 1. Silnia 1 wynosi 1. Ten typ definicji nosi nazwę definicji rekursywnej. Oznacza to, że definicja silni zawiera w sobie funkcję silni. Jednakże, ponieważ wszystkie funkcje muszą się kończyć, definicja rekursywna musi zawierać *podstawowy przypadek*. Przypadek podstawowy jest miejscem gdzie rekursja się zakończy. Bez przypadku podstawowego, funkcja mogłaby działać wiecznie wywołując się aż ewentualnie skończyłoby się miejsce na stosie. W przypadku silni, przypadkiem podstawowym jest liczba 1. Kiedy osiągamy liczbę 1, nie uruchamiamy silni ponownie, mówimy, że silnia 1 wynosi 1. Więc, przeprowadźmy to jak chcemy aby kod dla naszej funkcji silni wyglądał:
1. Sprawdź liczbę
2. Czy liczba wynosi 1?
3. Jeśli tak, odpowiedź jest jeden
4. W przeciwnym razie, odpowiedzią jest liczba razy silnia liczby minus jeden

Mogłoby być to problematyczne gdybyśmy nie mieli zmiennych lokalnych. W innych programach, umieszczanie wartości w zmiennych globalnych działało świetnie. Jednakże, zmienne globalne pozwalają tylko na pojedynczą kopię każdej zmiennej. W tym programie, będziemy mnożyć kopie danych! Odkąd zmienne lokalne egzystują w ramce stosu, i każde wywołanie funkcji ma własną ramkę stosu, jest w porządku.
Spójrzmy na kod jak on działa:

```
#CEL - Zadana liczba, ten program oblicza silnię. Na przykład, silnia 3 wynosi 3*2*1, czyli 6.
#Silnia 4 wynosi 4*3*2*1, czyli 24, i tak dalej.

#Ten program pokazuje jak wywoływać funkcję rekursywnie.

.section .data

#Ten program nie ma danych globalnych
```

```
.section .text
.globl _start
.globl factorial #to nie jest potrzebne chyba, że chcemy dzielić tę funkcję z innymi programami
_start:
pushl $4 #Silnia bierze jeden argument - liczbę którą chcemy poddać działaniu silni. Więc, została ona włożona
call factorial #uruchamia funkcję silni
addl $4, %esp #czyści parametr który był włożony na stos
movl %eax, %ebx #silnia zwraca odpowiedź w %eax,
#ale my chcemy ją mieć w %ebx aby wysłać ją jako nasz status wyjścia (exit)
movl $1, %eax #wywołanie funkcji wyjścia (exit) kernela
int $0x80
#To jest rzeczywista definicja funkcji
.type factorial, @function
factorial:
pushl %ebp #standardowe działanie przy funkcji - musimy odbudować %ebp do jej poprzedniego stanu przed powrotem,
#więc musimy włożyć ją
movl %esp, %ebp #Jest to dlatego, że nie chcemy zmieniać wskaźnika stosu, więc używamy %ebp.
movl 8(%ebp), %eax #przenosi pierwszy argument do %eax
#4(%ebp) przechowuje adres powrotu, a 8(%ebp) przechowuje pierwszy parametr
cmpl $1, %eax #Jeśli liczba równa się 1, to jest to nasza podstawa,
#i po prostu powracamy (1 jest już w %eax jako wartość powrotu)
je end_factorial
decl %eax #w przeciwnym razie, zmniejsza wartość
pushl %eax #wkłada ją dla naszego wywołania factorial(silnia)
call factorial #wywołanie factorial (silnia)
movl 8(%ebp), %ebx #%eax ma wartość powrotu, więc wyładowujemy nasz parametr do %ebx
imull %ebx, %eax #mnożenie przez wynik ostatniego wywołania factorial (w %eax)
#odpowiedź jest umieszczana w %eax, co jest dobre odkąd wartości powrotu tam idą.
end_factorial:
movl %ebp, %esp #standardowe wyjście z funkcji - musimy przestawić %ebp i %esp
popl %ebp #tam gdzie były przed zapoczątkowaniem funkcji
ret #powrót do funkcji (to zdejmuje także wartość powrotu)
```

Zasembluj, zlinkuj i uruchom to z tymi komendami:

```
as factorial.s -o factorial.o
ld factorial.o -o factorial
./factorial
echo $?
```

Powinieneś otrzymać wartość 24. 24 jest silnią 4, możesz to sprawdzić na kalkulatorze: 4*3*2*1=24.
Zakładam, że nie zrozumiałeś całego kodu programu. Przejdźmy go po jednym wierszu aby zobaczyć co się dzieje:

```
_start:
pushl $4
call factorial
```

No dobrze, ten program zamierza obliczać silnię liczby 4. Podczas programowania funkcji, oczekuje się od ciebie włożenia parametrów funkcji na wierzchołek stosu zaraz przed wywołaniem jej. Pamiętaj, *parametry* funkcji są danymi z którymi chcesz aby funkcja pracowała. W tym przypadku, funkcja silni bierze 1 parametr - liczbę dla której chcesz mieć silnię.

Instrukcja **pushl** wkłada zadaną wartość na wierzchołek stosu. Instrukcja **call** wtedy robi wywołanie funkcji.
Następnie mamy te wiersze:

```
addl $4, %esp
movl %eax, %ebx
movl $1, %eax
int $0x80
```

To ma miejsce gdy **factorial** zakończyła i obliczyła silnię 4 dla nas. Teraz musimy posprzątać stos. Instrukcja **addl** przenosi wskaźnik stosu z powrotem tam gdzie był przed włożeniem **$4** na stos. Zawsze powinieneś posprzątać swoje parametry ze stosu po powrocie wywołania funkcji.
Następna instrukcja przenosi **%eax** do **%ebx**. Co jest w **%eax**? Jest tam wartość powrotu funkcji **factorial**. W naszym przypadku, jest to wartość funkcji silni. Z 4 jako naszym parametrem, 24 powinno być naszą wartością powrotu. Pamiętaj, wartości powrotu są zawsze umieszczane w **%eax**. Chcemy zwrócić tę wartość jako kod statusu do systemu operacyjnego. Jednakże, Linux wymaga żeby status wyjścia (exit) programu był umieszczany w **%ebx**, nie w **%eax**, więc musimy ją przenieść. Wtedy robimy standardowe wywołanie systemowe wyjścia (exit).
Dobrą rzeczą w funkcjach jest to, że:
- Inni programiści nie muszą wiedzieć niczego o nich oprócz ich argumentów aby ich używać.
- Udostępniają one standaryzowane bloki z których możesz sformować program.
- Mogą być wywoływane wielokrotnie i z wielu lokalizacji i zawsze wiedzą jak powrócić do miejsca gdzie były ponieważ **call** wkłada adres powrotu na stos.

To są główne udogodnienia funkcji. Większe programy także używają funkcji aby podzielić kompleksowe części kodu na mniejsze, prostsze części. Faktycznie, prawie wszystko w programowaniu jest pisaniem i wywoływaniem funkcji.
Spójrzmy teraz jak funkcja **factorial** sama jest implementowana.
Przed rozpoczęciem funkcji, mamy tę dyrektywę:

```
.type factorial, @function
factorial:
```

Dyrektywa **.type** mówi linkerowi, że **factorial** jest funkcją. To nie jest koniecznie potrzebne chyba, że używamy **factorial** w innych programach. Dołączyliśmy ją dla kompletności. Wiersz mówiący **factorial:** daje symbolowi **factorial** lokalizację pamięci następnej instrukcji. To dlatego **call** wie gdzie iść kiedy mówimy **call factorial**.
Pierwszymi rzeczywistymi instrukcjami funkcji są:

```
pushl %ebp
movl %esp, %ebp
```

Jak było pokazane w poprzednim programie, tworzy to ramkę stosu dla tej funkcji. Te dwa wiersze będą formułką którą powinieneś zaczynać każdą funkcję.
Następna instrukcja jest taka:

```
movl 8(%ebp), %eax
```

Używa ona adresowania wskaźnika bazowego aby przenieść pierwszy parametr funkcji do **%eax**. Pamiętaj, **(%ebp)** ma stare **%ebp**, **4(%ebp)** ma adres powrotu, i **8(%ebp)** jest lokalizacją pierwszego parametru funkcji. Jeśli pomyślisz wstecz, to będzie wartość 4 w pierwszym wywołaniu, odkąd to było to co włożyliśmy na stos przed wywołaniem funkcji (**pushl $4**). Parametr w **%eax**. Jako, że ta funkcja wywołuje samą siebie, to będzie miała także inne wartości.
Następnie, sprawdzamy czy osiągnęliśmy podstawę (parametr o wartości 1). Jeśli tak, skaczemy do instrukcji o etykiecie **end_factorial**, gdzie będzie zwrócona. Jest już w **%eax** który, wspominaliśmy wcześniej, jest gdzie włożyłeś wartość

powrotu. Jest to odzwierciedlone przez te wiersze:

cmpl $1, %eax
je end_factorial

Jeśli nie jest to nasza podstawa, co powiedzielibyśmy aby zrobić? Moglibyśmy wywołać funkcję **factorial** na powrót z naszym parametrem minus jeden. Więc, po pierwsze zmniejszamy **%eax** o jeden:

decl %eax

decl oznacza dekrementację (zmniejszanie o jeden). Odejmuje 1 od danego rejestru lub lokalizacji pamięci (**%eax** w naszym przypadku). **incl** jest przeciwieństwem - dodaje 1. Po dekrementacji **%eax** wkładamy go na stos ponieważ będzie to parametr następnego wywołania funkcji. I wtedy wywołujemy **factorial** znowu!

pushl %eax
call factorial

Dobrze, teraz wywołaliśmy **factorial**. Jedna rzecz do zapamiętania jest taka, że po wywołaniu funkcji, nie możemy nigdy wiedzieć co jest w rejestrach (oprócz **%esp** i **%ebp**). Pomimo nawet tego, że mieliśmy wartość z którą wywoływaliśmy w **%eax**, nie ma jej tam już. Dlatego, potrzebujemy zdjąć ją ze stosu z tego samego miejsca z którego mieliśmy ją za pierwszym razem (z **8(%ebp)**). Mamy więc to:

movl 8(%ebp), %ebx

Teraz, chcemy pomnożyć tę liczbę przez wynik funkcji factorial. Jeśli pamiętasz naszą poprzednią dyskusję, wyniki funkcji są zostawiane w **%eax**. Więc, powinniśmy pomnożyć **%ebx** przez **%eax**. Jest to zrobione w tej instrukcji:

imull %ebx, %eax

To także umieszcza wynik w **%eax**, to jest dokładnie tam gdzie chcemy aby wartość powrotu dla tej funkcji była! Skoro wartość powrotu jest na miejscu potrzebujemy już tylko opuścić tę funkcję. Jeśli pamiętasz, na początku funkcji włożyliśmy **%ebp**, i przenieśliśmy **%esp** do **%ebp** aby utworzyć bieżącą ramkę stosu. Teraz odwracamy tę operację aby zniszczyć bieżącą ramkę stosu i reaktywować poprzednią:

end_factorial:
movl %ebp, %esp
popl %ebp

Teraz jesteśmy gotowi do powrotu, więc wydajemy następującą komendę

ret

To zdejmuje wierzchołkową wartość ze stosu, i wtedy skacze do niej. Jeśli pamiętasz naszą dyskusję o **call**, powiedzieliśmy, że **call** po pierwsze wkłada adres następnej instrukcji na stos przed tym jak skacze do początku funkcji. Więc, tutaj zdejmujemy ją z powrotem i możemy powrócić tam. Funkcja jest skończona i mamy naszą odpowiedź! Podobnie jak nasz poprzedni program, powinieneś przejrzeć go jeszcze raz, i upewnić się, że wiesz co która część robi. Przejrzyj z powrotem ten podrozdział i poprzednie podrozdziały dla wytłumaczenia wszystkiego czego nie rozumiesz. Potem, weź kartkę papieru, przejdź program krok po kroku śledząc jakie wartości są w rejestrach na każdym kroku, i jakie wartości są na stosie. Robienie tego powinno pogłębić twoje zrozumienie co się dzieje.

Przegląd

Znajomość koncepcji
- Co to są prymitywy?
- Co to są konwencje wywoływania?
- Co to jest stos?
- Jak **pushl** i **popl** działają na stos? Jaki rejestr specjalnego przeznaczenia one absorbują?
- Co to są zmienne lokalne i po co są używane?
- Dlaczego zmienne lokalne są tak potrzebne w funkcjach rekursywnych?
- Po co są używane **%ebp** i **%esp**?
- Co to jest ramka stosu?

Użycie Koncepcji
- Napisz funkcję zwaną **square** która przyjmuje jeden argument i zwraca kwadrat tego argumentu.
- Napisz program do testowania twojej funkcji **square**.
- Zmień program maximum podany w Podrozdziale zwanym *Szukanie Wartości Maksymalnej* w Rozdziale 3 tak żeby był funkcją która wskazuje kilkanaście wartości i zwraca ich maximum. Napisz program który wywołuje maximum z 3 różnych list, i zwraca wynik ostatniej jako kod statusu wyjścia (exit) programu.

Wytłumacz problemy które mogłyby narosnąć bez standardowej konwencji wywoływania.

Idąc Dalej
- Czy myślisz, że dla systemu jest lepiej mieć duży zestaw prymitywów czy mały, uwzględniając, że większy zestaw może być napisany jako złożenie mniejszych?
- Funkcja silni może być napisana nie-rekursywnie. Zrób to.
- Znajdź aplikację komputerową której używasz regularnie. Spróbuj zlokalizować jakąś właściwość i rozbierz tę właściwość na funkcje. Zdefiniuj interfejsy tej funkcji pomiędzy właściwością a resztą programu.
- Wystąp z własną konwencją wywoływania. Przepisz programy z tego rozdziału używając jej. Przykładem różnej konwencji mogłoby być przekazanie parametrów w rejestrach raczej niż stos, przekazanie ich w innej kolejności, wartości powrotu w innych rejestrach lub lokalizacji pamięci. Cokolwiek wybierzesz, bądź konsekwentny i wprowadź ją w całym programie.
- Czy możesz zbudować konwencję wywoływania bez używania stosu? Jakie może to mieć granice?
- Jakich przypadków testowych powinniśmy użyć w naszym programie aby sprawdzić czy działa poprawnie?

Rozdział 5. Postępowanie z Plikami

Wielka część programowania komputerowego dotyczy pracy z plikami. Poza wszystkim, kiedy rebootujemy nasze komputery, jedyne rzeczy które pozostają z poprzedniej sesji są to rzeczy które były odłożone na dysk. Dane które są przechowywane w plikach są zwane danymi wielokrotnymi, ponieważ pozostają one w plikach które zostają na dysku nawet kiedy program nie jest uruchomiony..

Koncepcja Pliku UNIX-owego

Każdy system operacyjny ma własny sposób postępowania z plikami. Jednakże, metoda UNIXa, która jest używana w

Linuksie, jest najprostsza i najbardziej uniwersalna. Pliki UNIX-owe, nieważne jaki program je stworzył, mogą wszystkie być dostępne jako sekwencyjny strumień bajtów. Kiedy dostajesz się do pliku, zaczynasz od otwarcia go przez nazwę. Wtedy system operacyjny nadaje numer, nazwany deskryptorem pliku, którego używasz do odnoszenia się do pliku dopóki z nim pracujesz. Możesz wtedy czytać i zapisywać do pliku używając jego deskryptora pliku. Kiedy zakończyłeś odczytywanie i zapisywanie, zamykasz plik, co czyni deskryptor pliku bezużytecznym.
W naszych programach będziemy postępować z plikami w następujący sposób:
1. Powiedz Linuksowi nazwę pliku do otwarcia, i w jakim trybie chcesz go otworzyć (odczytywanie, zapisywanie, obydwa odczytywanie i zapisywanie, utworzenie go jeśli nie stanieje, itd.). To jest przeprowadzane z wywołaniem systemowym **open** (otworzyć), które przybiera nazwę pliku, liczbę reprezentującą tryb, i zestaw uprawnień jako swoje parametry. **%eax** będzie przechowywał numer wywołania systemowego, wynoszącego 5. Adres pierwszego znaku nazwy pliku powinien być umieszczony w **%ebx**. Intencje odczyt/zapis, reprezentowane jako liczba, powinien być umieszczony w **%ecx**. Na teraz, używaj 0 dla plików z których chcesz czytać, i 03101 dla plików do których chcesz zapisywać (musisz włączyć wiodące zero). Na koniec, zestaw uprawnień powinien być umieszczony jako liczba w **%edx**. Jeśli jesteś nieobeznany z uprawnieniami UNIX, użyj 0666 jako uprawnienia (znowu, musisz włączyć wiodące zero).
2. Linux wtedy zwróci deskryptor pliku w **%eax**. Pamiętaj, to jest liczba której używasz do odnoszenia się do tego pliku podczas działania programu.
3. Następnie będziesz operował na pliku robiąc odczytywania i/lub zapisywania, za każdym razem dając Linuksowi deskryptor pliku którego chcesz użyć. **read** jest wywołaniem systemowym 3, i żeby je wywołać musisz mieć deskryptor pliku w **%ebx**, adres bufora dla umieszczenia danych które są odczytywane w **%ecx**, i rozmiar bufora w **%edx**. Bufory będą wytłumaczone w Podrozdziale zwanym *Bufory i .bss*. **read** zwróci albo liczbę znaków odczytanych z pliku lub kod błędu. Kody błędów mogą być rozpoznane ponieważ są zawsze liczbami ujemnymi (więcej informacji o liczbach ujemnych można znaleźć w Rozdziale 10). **write** jest wywołaniem systemowym 4, i wymaga tych samych parametrów co wywołanie systemowe **read**, oprócz tego, że bufor powinien być już wypełniony danymi do zapisania. Wywołanie systemowe **write** będzie oddawać liczbę bajtów zapisaną w **%eax** lub kod błędu.
4. Kiedy skończyłeś pracę ze swoimi plikami, możesz wtedy powiedzieć Linuksowi aby je zamknął. Poza tym, twój deskryptor pliku nie jest już dłużej ważny. Jest to robione z użyciem **close**, wywołania systemowego 6. Jedynym parametrem **close** jest deskryptor pliku, który jest umieszczony w **%ebx**.

Bufory i .bss

W poprzednim podrozdziale wspominaliśmy o buforach bez tłumaczenia czym one są. Bufor jest ciągłym blokiem bajtów używanym przy przenoszeniu porcji danych. Kiedy prosimy o czytanie pliku, system operacyjny potrzebuje miejsca na umieszczanie danych które czyta. To miejsce jest zwane buforem. Zwykle bufory są używane tylko do przechowywania danych tymczasowo, wtedy są one czytane z bufora i zmieniane do formy która jest prostsza dla programów do wykorzystania. Nasze programy nie będą aż tak skomplikowane aby potrzebować tego. Na przykład, powiedzmy, że chcesz czytać pojedynczy wiersz tekstu z pliku ale nie wiesz jak długi jest ten wiersz. Wtedy mógłbyś po prostu wczytać dużą liczbę bajtów/znaków z pliku do bufora, poszukać znaku końca wiersza, i skopiować wszystkie znaki do końca wiersza do innej lokalizacji. Jeśli nie znalazłbyś znaku końca wiersza, mógłbyś alokować następny bufor i kontynuować czytanie. Mógłbyś prawdopodobnie w ten sposób kończyć z kilkoma znakami pozostałymi w twoim buforze, które mógłbyś użyć jako punkt startowy kiedy będziesz potrzebował danych z tego pliku.
Inna sprawa do odnotowania jest taka, że bufory są stałego rozmiaru, ustalonego przez programistę. Więc, jeśli chcesz wczytać 500 bajtów danych na raz, przesyłasz wywołaniu systemowemu **read** adres nieużywanej 500-bajtowej lokalizacji i liczbę 500 więc wie on jak duże to jest. Możesz zrobić je mniejszym lub większym, zależnie od potrzeb twojej aplikacji.
Aby utworzyć bufor, musisz zarezerwować statyczne lub dynamiczne miejsce przechowywania. Statyczne miejsce przechowywania jest tym o czym mówiliśmy dotąd, lokalizacja pamięci zadeklarowana użyciem dyrektyw **.long** lub **.byte**. Dynamiczne miejsce przechowywania będzie dyskutowane w Podrozdziale zwanym *Osiąganie Większej Pamięci* w Rozdziale 9. Są problemy z deklaracją buforów z użyciem **.byte**. Po pierwsze, jest to uciążliwe do wpisywania. Musiałbyś

wpisać 500 liczb za deklaracją **.byte**, i one nie byłyby używane do czegokolwiek oprócz zajmowania miejsca. Po drugie, zużywa miejsce w obszarze wykonywania. W przykładach które używaliśmy dotąd, to nie zużywało zbyt wiele, ale to się może zmienić w większych programach. Jeśli chcesz 500 bajtów musisz wpisać 500 liczb i to marnuje 500 bajtów w obszarze wykonywania. Jest rozwiązanie obydwu tych problemów. Dotąd, dyskutowaliśmy dwie sekcje programu, sekcje **.text** i **.data**. Jest jeszcze sekcja zwana **.bss**. Ta sekcja jest jak sekcja danych, oprócz tego, że nie zabiera miejsca w obszarze wykonywania. Ta sekcja może rezerwować miejsce, ale nie może go inicjalizować. W sekcji **.data**, mógłbyś zarezerwować miejsce i ustawić je na wartość inicjalizującą. W sekcji **.bss**, nie możesz ustawić wartości inicjalizującej. To jest użyteczne dla buforów ponieważ nie potrzebujemy inicjalizować ich w ogóle, tylko potrzebujemy zarezerwować miejsce. Aby to zrobić, wykonujemy następujące komendy:

```
.section .bss
.lcomm my_buffer, 500
```

Ta dyrektywa, **.lcomm**, utworzy symbol, **my_buffer**, który referuje do 500-bajtowej lokalizacji pamięci której możemy użyć jako bufor. Możemy wtedy zrobić co następuje, przyjmując, że otworzyliśmy plik do czytania i umieściliśmy deskryptor pliku w **%ebx**:

```
movl $my_buffer, %ecx
movl 500, %edx
movl 3, %eax
int $0x80
```

To wczyta 500 bajtów do naszego bufora. W tym przykładzie, umieściłem znak dolara przed **my_buffer**. Pamiętaj, powodem tego jest to, że bez znaku dolara, **my_buffer** jest traktowany jako lokalizacja pamięci, i jest dostępna w trybie bezpośrednim adresowania. Znak dolara przełącza go w natychmiastowy tryb adresowania, który rzeczywiście ładuje liczbę reprezentowaną przez **my_buffer** (t.j. - adres początku naszego bufora, który jest adresem **my_buffer**) do **%ecx**.

Standardowe i Specjalne Pliki

Mógłbyś pomyśleć, że programy zaczynają się bez żadnych plików otwartych domyślnie. Nie jest to prawda. Programy Linuksa zwykle mają co najmniej trzy otwarte deskryptory plików kiedy zaczynają. Są to:

STDIN
To jest "standard input" (wejście standardowe). Jest to plik tylko do odczytu, i zwykle reprezentuje twoją klawiaturę. To jest zawsze deskryptor pliku 0.

STDOUT
To jest "standard output" (wyjście standardowe). Jest to plik tylko do zapisu, i zwykle reprezentuje twój ekran. To jest zawsze deskryptor pliku 1.

STDERR
To jest "standard error" (błąd standardowy). Jest to plik tylko do zapisu, i zwykle reprezentuje twój ekran. Większość regularnie procesującego wyjścia idzie do STDOUT, ale każda wiadomość o błędzie który zdarzył się w procesie idzie do STDERR. W ten sposób, jeśli chcesz, możesz podzielić je do osobnych miejsc. To jest zawsze deskryptor pliku 2.

Każdy z tych "plików" może być przekierowany z lub do rzeczywistego pliku, raczej niż ekran lub klawiatura. Jest to poza przedmiotem tej książki, ale każda dobra książka o wierszu poleceń UNIXa opisze to w szczegółach. Sam program nawet

nie potrzebuje się martwić o to przekierowanie - może używać standardowych deskryptorów plików jak zwykle.
Zauważ, że wiele z tych plików do których zapisujesz nie są w ogóle plikami. Systemy operacyjne na podstawie UNIX traktują wszystkie systemy wejścia/wyjścia jako pliki. Połączenia sieciowe (Network) są traktowane jako pliki, twój port szeregowy jest traktowany jak plik, nawet twoje urządzenia dźwiękowe (audio) są traktowane jako pliki. Komunikacja pomiędzy procesami jest zwykle robiona poprzez specjalne pliki zwane "pipes". Niektóre z tych plików mają różne metody otwierania i tworzenia ich niż pliki regularne (t.j. - nie używają wywołania systemowego wejścia (**open**)), ale wszystkie one mogą być wczytane z i zapisane do używając standardowych wywołań systemowych odczytu (**read**) i zapisu (**write**).

Używanie Plików w Programie

Zamierzamy napisać prosty program dla zilustrowania tych koncepcji. Program ten będzie brał dwa pliki, czytał z jednego, zmieniał wszystkie małe litery w duże litery, i zapisywał do innego pliku. Zanim to zrobimy, pomyślmy co potrzebujemy zrobić aby mieć to zadanie wykonane:
- Mieć funkcję która bierze blok pamięci i konwertuje go do dużych liter. Ta funkcja potrzebowałaby adresu bloku pamięci i jego rozmiaru jako parametrów.
- Mieć sekcję kodu który powtarzająco wczytuje do bufora, wywołuje naszą funkcję konwersji w buforze i wtedy zapisuje bufor z powrotem do innej funkcji.
- Zaczynamy program poprzez otwarcie potrzebnych plików.
Zauważ, że wyodrębniłem te rzeczy w odwrotnej kolejności niż będą robione. To użyteczna sztuczka w pisaniu złożonych programów - po pierwsze zdecyduj jaki cel jest osiągany. W tym przypadku, to konwertowanie bloków znaków do dużych liter. Wtedy, pomyśl o wszystkich potrzebach do utworzenia i przeprowadzenia aby to zaszło. W tym przypadku, musisz otworzyć pliki, i sukcesywnie czytać i zapisywać bloki na dysk. Jednym z kluczowych w programowaniu jest sukcesywne dzielenie problemu na mniejsze i mniejsze części aż jest on na tyle mały, że możesz łatwo go rozwiązać. Potem możesz składać te części z powrotem aż będziesz miał działający program.
Mógłbyś pomyśleć, że nigdy nie zapamiętasz tych wszystkich liczb które na ciebie spadły - liczby wywołania systemowego, liczba przerwania, itd. W tym programie także przedstawimy nową dyrektywę, **.equ**, która powinna pomóc. **.equ** pozwala przypisać nazwy do liczb. Na przykład, jeśli zrobiłbyś **.equ LINUX_SYSCALL, 0x80**, za każdym razem potem gdy napisałbyś LINUX_SYSCALL, asembler mógłby zastąpić to przez 0x80. Więc teraz, możesz napisać

int LINUX_SYSCALL

co jest znacznie łatwiejsze do czytania, i znacznie łatwiejsze do zapamiętania. Pisanie kodu jest złożone ale jest wiele takich rzeczy, które możemy zrobić aby uczynić je łatwiejszym.
Oto jest ten program. Zauważ, że mamy więcej etykiet niż rzeczywiście używamy do skoków, ponieważ niektóre z nich są tam tylko dla jasności. Spróbuj prześledzić ten program i zobacz co się dzieje w różnych sytuacjach.

```
#CEL: Ten program konwertuje plik wejściowy do pliku wyjściowego z wszystkimi literami przekształconymi do dużych.
#
#PROCESOWANIE: 1) Otwórz plik wejściowy
# 2) Otwórz plik wyjściowy
# 3) Jeśli nie jesteśmy na końcu pliku wejściowego
# a) wczytaj część pliku do naszego bufora pamięci
# b) przejdź przez każdy bajt pamięci
# jeśli bajt jest małą literą, # przekształć go do dużej litery
# c) zapisz bufor pamięci do pliku wyjściowego

.section .data
```

```
#####STAŁE##########

#liczby wywołań systemowych
.equ SYS_OPEN, 5
.equ SYS_WRITE, 4
.equ SYS_READ, 3
.equ SYS_CLOSE, 6
.equ SYS_EXIT, 1

#opcje dla "open" (zobacz do /usr/include/asm/fcntl.h po różnorodne wartości.
#Możesz połączyć je przez dodanie ich lub ORing ich)
#Jest to dyskutowane szerzej w Rozdziale "Licząc Jak Komputer"
.equ O_RDONLY, 0
.equ O_CREAT_WRONLY_TRUNC, 03101
#standardowe deskryptory pliku
.equ STDIN, 0
.equ STDOUT, 1
.equ STDERR, 2

#przerwanie wywołania systemowego
.equ LINUX_SYSCALL, 0x80

.equ END_OF_FILE, 0 #To jest wartość powrotu czytania która oznacza, że osiągnęliśmy koniec pliku

.equ NUMBER_ARGUMENTS, 2

.section .bss
#Bufor - to jest miejsce gdzie dane są ładowane z pliku danych i z którego są zapisywane do pliku wyjściowego.
#Nie powinien on przekroczyć 16000 z wielu powodów.
.equ BUFFER_SIZE, 500
.lcomm BUFFER_DATA, BUFFER_SIZE

.section .text

#POZYCJE STOSU
.equ ST_SIZE_RESERVE, 8
.equ ST_FD_IN, -4
.equ ST_FD_OUT, -8
.equ ST_ARGC, 0 #Liczba argumentów
.equ ST_ARGV_0, 4 #Nazwa programu
.equ ST_ARGV_1, 8 #Nazwa pliku wejściowego
.equ ST_ARGV_2, 12 #Nazwa pliku wyjściowego

.globl _start
_start:
###INICJALIZACJA PROGRAMU###
#zapisanie wskaźnika stosu
movl %esp, %ebp
```

```
#Alokacja miejsca dla deskryptora naszego pliku na stosie
subl $ST_SIZE_RESERVE, %esp
open_files:
open_fd_in:
###OTWARCIE PLIKU WEJŚCIOWEGO###
#otwarcie syscall
movl $SYS_OPEN, %eax
#nazwa pliku wejściowego do %ebx
movl ST_ARGV_1(%ebp), %ebx
#flaga tylko do odczytu
movl $O_RDONLY, %ecx
#to nie jest rzeczywiście ważne dla czytania
movl $0666, %edx
#wywołanie Linuksa
int $LINUX_SYSCALL

store_fd_in:
#zapisanie danego deskryptora pliku
movl %eax, ST_FD_IN(%ebp)

open_fd_out:
###OTWARCIE PLIKU WYJŚCIOWEGO###
#otwarcie tego pliku
movl $SYS_OPEN, %eax
#nazwa pliku wyjściowego do %ebx
movl ST_ARGV_2(%ebp), %ebx
#flagi dla zapisywania do pliku
movl $O_CREAT_WRONLY_TRUNC, %ecx
#tryb dla nowego pliku (jeśli jest utworzony)
movl $0666, %edx
#wywołanie Linuksa
int $LINUX_SYSCALL

store_fd_out:
#umieszczenia deskryptora pliku tutaj
movl %eax, ST_FD_OUT(%ebp)

###ROZPOCZĘCIE GŁÓWNEJ PĘTLI###
read_loop_begin:

###WCZYTYWANIE W BLOKACH Z PLIKU WEJŚCIOWEGO###
movl $SYS_READ, %eax
#podanie deskryptora pliku wejściowego
movl ST_FD_IN(%ebp), %ebx
#lokalizacja na wczytywanie do niej
movl $BUFFER_DATA, %ecx
#rozmiar bufora
movl $BUFFER_SIZE, %edx
#Rozmiar bufora czytania jest zwracany w %eax
```

```
int $LINUX_SYSCALL

###WYJŚCIE JEŚLI OSIĄGNĘLIŚMY KONIEC###
#sprawdzenie znacznika końca pliku
cmpl $END_OF_FILE, %eax
#jeśli znaleziony lub błąd, idź do końca
jle end_loop

continue_read_loop:
###KONWERSJA BLOKU DO DUŻYCH LITER###
pushl $BUFFER_DATA #lokalizacja bufora
pushl %eax #rozmiar bufora
call convert_to_upper
popl %eax #oddanie rozmiaru z powrotem
addl $4, %esp #odbudowa %esp

###ZAPIS BLOKU DO PLIKU WYJŚCIOWEGO###
#rozmiar bufora
movl %eax, %edx
movl $SYS_WRITE, %eax
#plik do użycia
movl ST_FD_OUT(%ebp), %ebx
#lokalizacja bufora
movl $BUFFER_DATA, %ecx
int $LINUX_SYSCALL

###KONTYNUACJA PĘTLI###
jmp read_loop_begin

end_loop:
###ZAMKNIĘCIE PLIKÓW###
#UWAGA - nie potrzebujemy sprawdzania błędu ponieważ warunki błędu nie wskazują niczego specjalnego tutaj
movl $SYS_CLOSE, %eax
movl ST_FD_OUT(%ebp), %ebx
int $LINUX_SYSCALL

movl $SYS_CLOSE, %eax
movl ST_FD_IN(%ebp), %ebx
int $LINUX_SYSCALL

###WYJŚCIE###
movl $SYS_EXIT, %eax
movl $0, %ebx
int $LINUX_SYSCALL

#CEL: Ta funkcja rzeczywiście robi konwersję do dużych liter dla bloku
#
#WEJŚCIE: Pierwszy parametr jest lokalizacją bloku pamięci do konwersji
# Drugi parametr jest długością tego bufora
```

```
#
#WYJŚCIE: Ta funkcja nadpisuje bieżący bufor wersją z dużymi literami.
#
#ZMIENNE: # %eax - początek bufora
# %ebx - długość bufora
# %edi - przesunięcie bieżącego bufora
# %cl - bieżący bajt będący testowanym (pierwsza część %ecx)
#

###STAŁE###
#Dolna granica naszych poszukiwań
.equ LOWERCASE_A, 'a'
#Górna granica naszych poszukiwań
.equ LOWERCASE_Z, 'z'
#Konwersja pomiędzy dużymi i małymi literami
.equ UPPER_CONVERSION, 'A' - 'a'

###ELEMENTY STOSU###
.equ ST_BUFFER_LEN, 8 #Długość bufora
.equ ST_BUFFER, 12 #rzeczywisty bufor
convert_to_upper:
pushl %ebp
movl %esp, %ebp

###USTAWIANIE ZMIENNYCH###
movl ST_BUFFER(%ebp), %eax
movl ST_BUFFER_LEN(%ebp), %ebx
movl $0, %edi

#jeśli bufor z zerową długością był nam dany, po prostu opuszczamy
cmpl $0, %ebx
je end_convert_loop

convert_loop:
#bierzemy bieżący bajt
movb (%eax,%edi,1), %cl
#idź do następnego bajta aż dotąd jak jest pomiędzy 'a' i 'z'
cmpb $LOWERCASE_A, %cl
jl next_byte
cmpb $LOWERCASE_Z, %cl
jg next_byte

#w przeciwnym razie konwertuje do dużych liter
addb $UPPER_CONVERSION, %cl
#i zachowuje ją z powrotem
movb %cl, (%eax,%edi,1)
next_byte:
incl %edi #następny bajt
cmpl %edi, %ebx #kontynuuje chyba, że osiągnęliśmy koniec
```

jne convert_loop

end_convert_loop:
#nie ma wartości powrotu, tylko opuszczamy
movl %ebp, %esp
popl %ebp
ret

Zapisz ten program jako **toupper.s**, i wtedy wykonaj następujące komendy:

as toupper.s -o toupper.o
ld toupper.o -o toupper

Buduje to program nazwany **toupper**, który konwertuje wszystkie znaki małych liter w pliku do dużych liter. Na przykład, aby konwertować plik **toupper.s** do dużych liter, wypisz następującą komendę:

./toupper toupper.s toupper.uppercase

Teraz znajdziesz w pliku **toupper.uppercase** wersję z dużymi literami swojego oryginalnego pliku.
Przetestujmy jak ten program działa.
Pierwsza sekcja programu jest oznaczona STAŁE. W programowaniu, stała jest wartością która jest przypisywana podczas asemblacji lub kompilacji programu, i nigdy nie jest zmieniana. Nabyłem zwyczaju umieszczania wszystkich moich stałych razem na początku programu. Jest to potrzebne tylko do ich deklaracji przed ich użyciem, ale zebranie ich wszystkich na początku ułatwia ich odnalezienie. Pisząc je wszystkie dużymi literami powodujemy, że jest oczywiste w twoim programie które wartości są stałymi i gdzie je znaleźć. W języku asemlerowym, deklarujemy stałe dyrektywą **.equ** jak było wspomniane wcześniej. Tutaj, po prostu dajemy nazwy wszystkim standardowym liczbom których używaliśmy do tej pory, jak liczby wywołania systemowego, liczbę przerwania "syscall", i opcje otwarcia pliku.
Następna sekcja jest oznaczona BUFORY. Używamy tylko jednego bufora w tym programie, który nazywamy BUFFER_DATA. Definiujemy także stałą, BUFFER_SIZE, która przechowuje rozmiar bufora. Jeśli zawsze referujemy do tej stałej raczej niż wypisujemy liczbę 500 kiedykolwiek potrzebujemy użyć rozmiaru bufora, to jeśli później się ona zmieni, potrzebujemy tylko zmodyfikować tę wartość, raczej niż musieć przechodzić przez cały program zmieniając wszystkie te wartości indywidualnie.
Zamiast iść do sekcji programu **_start**, idziemy na koniec gdzie definiujemy funkcję **convert_to_upper**. To jest ta część która rzeczywiście wykonuje konwersję.
Ta sekcja zaczyna się od listy stałych których będziemy używać. Powód dla którego umieszczamy je tutaj raczej niż na początku jest taki, że działają one tylko z tą jedyną funkcją. Oto mamy te definiuje:

.equ LOWERCASE_A, 'a'
.equ LOWERCASE_Z, 'z'
.equ UPPER_CONVERSION, 'A' - 'a'

Pierwsze dwie prosto definiują litery które są granicami tego czego szukamy. Pamiętaj, że w komputerze, litery są reprezentowane przez liczby. Dlatego, możemy używać **LOWERCASE_A** w porównaniach, dodawaniach, odejmowaniach, lub w czymkolwiek innym gdzie możemy używać liczb. Także, zauważ, że definiujemy stałą **UPPER_CONVERSION**. Ponieważ litery są reprezentowane przez liczby, możemy je odejmować. Odejmowanie dużej litery od tej samej małej litery daje nam ile potrzebujemy dodać do małej litery aby zrobić z niej dużą literę. Jeśli to nie ma sensu, zobacz sam na tabelę kodów ASCII (patrz Dodatek D). Zauważysz, że liczbą dla znaku A jest 65 a dla znaku a jest 97. Współczynnik konwersji jest więc -32. Dla każdej małej litery jeśli dodasz -32, otrzymasz jej wielkoliterowy ekwiwalent.

Poza tym, mamy kilka stałych zaetykietowanych POZYCJE STOSU. Pamiętaj, że parametry tej funkcji są wkładane na stos przed wywołaniami funkcji. Te stałe (z prefiksem **ST** dla jasności) definiują gdzie na stosie powinniśmy oczekiwać znalezienia każdego elementu danych. Adres powrotu jest w pozycji **4 + %esp**, długość bufora jest w pozycji **8 + %esp**, i adres bufora jest w pozycji **12 + %esp**. Używając symboli dla tych liczb zamiast liczb samych ułatwia zobaczenie które dane są użyte i przenoszone.
Następnie wchodzi etykieta **convert_to_upper**. To jest punkt otwarcia funkcji. Pierwsze dwa wiersze są naszymi standardowymi wierszami funkcji do zachowania wskaźnika stosu. Następne dwa wiersze

movl ST_BUFFER(%ebp), %eax
movl ST_BUFFER_LEN(%ebp), %ebx

przenoszą parametry funkcji do odpowiednich rejestrów dla użycia. Wtedy, ładujemy zero do **%edi**. Co zamierzamy zrobić to iterować każdy bajt bufora poprzez ładowanie z lokalizacji **%eax + %edi**, inkrementując **%edi**, i powtarzając aż **%edi** będzie równy długości bufora przechowywanego w **%ebx**. Wiersze

```
cmpl $0, %ebx
je end_convert_loop
```

są sprawdzianem błędu aby upewnić się, że nic nie daje nam bufora o zerowym rozmiarze. Jeśli daje, czyścimy i opuszczamy. Ochrona przeciwko potencjalnym błędom użytkownika lub programistycznym jest ważnym zadaniem programisty. Możesz zawsze sprecyzować, że twoja funkcja nie powinna brać buforów o zerowym rozmiarze, ale jest nawet lepiej mieć sprawdzanie funkcji i realistyczny plan jeśli to się zdarzy.
Teraz zaczynamy naszą pętlę. Po pierwsze, przenosi bajt do **%cl**. Kod do tego jest

movb (%eax,%edi,1), %cl

Używa on trybu adresowania indeksowego pośredniego. Mówi aby zacząć w **%eax** i przejść lokalizacje **%edi**, z każdą lokalizacją będącą o 1 bajt większą. Pobiera wartość tam znalezioną i odkłada ją w **%cl**. Po tym sprawdza czy wartość jest w granicach małoliterowego **a** do małoliterowego **z**. Aby sprawdzić granice, po prostu sprawdza czy dana litera jest mniejsza od **a**. Jeśli tak, nie może być to małoliterowa litera. Analogicznie, jeśli jest większa od **z**, nie może być to małoliterowa litera. Więc, w każdym z tych przypadków, po prostu przechodzi dalej. Jeśli jest w prawidłowych granicach, wtedy dodaje konwersję do dużych liter, i umieszcza z powrotem w buforze.
Obydwie sytuacje, wtedy idą do następnej wartości przez inkrementację **%cl**. Następnie sprawdza czy jesteśmy na końcu bufora. Jeśli nie jesteśmy na końcu, skaczemy z powrotem do początku pętli (etykieta **convert_loop**). Jeśli jesteśmy na końcu, po prostu kontynuuje do końca funkcji. Ponieważ modyfikujemy bufor bezpośrednio, nie potrzebujemy zwracać niczego do programu wywołującego - zmiany są już w buforze. Etykieta **end_convert_loop** nie jest potrzebna, ale jest tam więc jest łatwo zobaczyć gdzie części programu są.
Teraz wiemy jak działa proces konwersji. Obecnie potrzebujemy zrozumieć jak otrzymujemy dane w plikach i z plików. Przed czytaniem i zapisywaniem plików musimy je otworzyć. UNIX-owe wywołanie systemowe **open** jest tym co udostępnia to. Przyjmuje następujące parametry:
- **%eax** zawiera liczbę wywołania systemowego jak zwykle - 5 w tym przypadku.
- **%ebx** zawiera wskaźnik do łańcucha który jest nazwą pliku do otwarcia. Łańcuch ten musi kończyć się znakiem "null".
- **%ecx** zawiera opcje używane dla otwarcia tego pliku. Mówią one Linuksowi jak otworzyć ten plik. Mogą one zaznaczać takie rzeczy jak otwarty do odczytu, otwarty do zapisu, otwarty do odczytu i zapisu, utwórz jeśli nie istnieje, usuń plik jeśli już istnieje, etc. Nie będziemy wchodzić w to jak tworzyć liczby dla tych opcji aż do Podrozdziału nazwanego *Prawda, Fałsz i Liczby Binarne* w Rozdziale 10. Na teraz, zaufaj liczbom które opisaliśmy.
- **%edx** zawiera uprawnienia które są używane do otwarcia tego pliku. Jest to używane w przypadku gdy plik musi być utworzony najpierw, więc Linux wie z jakimi uprawnieniami utworzyć ten plik. Są one wyrażone w systemie ósemkowym, właśnie jak regularne uprawnienia UNIXa.

Po wykonaniu wywołania systemowego, deskryptor pliku nowo-otwartego pliku jest umieszczany w **%eax**.
Więc, jakie pliki my otwieramy? W tym przykładzie, będziemy otwierać pliki sprecyzowane w wierszu poleceń. Fortunnie, parametry wiersza poleceń są już umieszczane przez Linux w lokalizacji o łatwym dostępie, i już kończą się znakiem "null". Kiedy zaczyna się program Linuksa, wszystkie wskaźniki do argumentów wiersza poleceń są umieszczone na stosie. Liczba argumentów jest umieszczana w **8(%esp)**, nazwa programu jest umieszczana w **12(%esp)**, a argumenty są umieszczane od **16(%esp)**. W języku Programowania C, jest to oznaczane jako tablica **argv**, więc będziemy referować do niej w ten sposób w naszym programie.
Pierwszą rzeczą którą robi nasz program jest zachowanie pozycji bieżącego stosu w **%ebp** i wtedy zarezerwowanie jakiegoś miejsca na stosie do umieszczenia deskryptorów pliku. Po tym, zaczyna otwieranie plików.
Pierwszym plikiem który program otwiera jest plik wejściowy, co jest pierwszym argumentem wiersza poleceń. Robimy to przez ustawienie wywołania systemowego. Umieszczamy nazwę pliku w **%ebx**, liczbę trybu tylko do odczytu w **%ecx**, tryb domyślny **$0666** w **%edx**, i liczbę wywołania systemowego w **%eax**. Po wywołaniu systemowym, plik jest otwarty i deskryptor pliku jest umieszczany w **%eax**. Deskryptor pliku jest wtedy przenoszony do jego właściwego miejsca na stosie.
To samo jest wtedy robione dla pliku wyjściowego, oprócz tego, że jest on utworzony z trybami tylko do zapisu, utwórz-jeśli-nie-istnieje, przytnij-jeśli-istnieje. Jego deskryptor pliku jest zachowywany także.
Teraz przechodzimy do głównej części - pętla odczyt/zapis. Generalnie, będziemy wczytywać stało-rozmiarowe kęsy danych z pliku wejściowego, wywoływać naszą funkcję konwersji na nich, i zapisywać z powrotem do pliku wyjściowego. Chociaż czytamy stało-rozmiarowe kęsy, rozmiar kęsów nie wpływa na ten program - my tylko operujemy na prostych sekwencjach znaków. Moglibyśmy wczytywać tak małe lub tak duże kęsy jak chcemy, i wciąż to mogłoby pracować poprawnie.
Pierwsza część pętli jest do wczytywania danych. Używa wywołania systemowego **read**. To wywołanie właśnie pobiera deskryptor pliku z którego czyta, bufor do zapisu, i rozmiar bufora (t.j. - maksymalną liczbę bajtów która mogłaby być zapisana). Wywołanie systemowe zwraca liczbę bajtów aktualnie wczytanych, lub "end-of-file" (koniec pliku - liczba 0).
Po wczytaniu bloku, sprawdzamy **%eax** dla znacznika końca pliku "end-of-file". Jeśli znaleziony, wyjście z pętli. W przeciwnym razie kontynuujemy.
Po wczytaniu danych, funkcja **convert_to_upper** jest wywoływana z buforem który właśnie wczytaliśmy i liczbą znaków wczytanych w poprzednim wywołaniu systemowym. Po wykonaniu tej funkcji, bufor powinien być z dużymi literami i gotowy do zapisania. Rejestry są wtedy odbudowywane do tego co było w nich przedtem.
Ostatecznie, wydajemy wywołanie systemowe **write**, które jest dokładnie takie same jak wywołanie systemowe **read**, oprócz tego, że przesyła dane z bufora do pliku. Teraz idziemy z powrotem do początku pętli.
Po wyjściu z pętli (pamiętaj, wychodzi jeśli, po wczytaniu, wykryje koniec pliku), po prostu zamyka jego deskryptory plików i wychodzi. Wywołanie systemowe zamknięcia pobiera deskryptor pliku do zamknięcia w **%ebx**.
Program jest wtedy zakończony!

Przegląd

Znajomość Koncepcji
- Opisz cykl życia deskryptora pliku.
- Co to są standardowe deskryptory plików i po co są używane?
- Co to jest bufor?
- Jaka jest różnica pomiędzy sekcją **.data** i sekcją **.bss**?
- Jakie są relacje wywołań systemowych do czytania i zapisywania plików?

Użycie Koncepcji
- Zmodyfikuj program **toupper** tak aby czytał z STDIN i zapisywał do STDOUT zamiast używać plików w wierszu poleceń.
- Zmień rozmiar bufora.
- Przepisz program tak żeby używał sekcji **.bss** raczej niż stosu do przechowywania deskryptorów plików.

- Napisz program który utworzy plik zwany **heynow.txt** i zapisz w nim słowa "Hey diddle diddle!".

Idąc Dalej

- Jaką różnicę robi rozmiar bufora?
- Jakie wyniki błędu mogą być zwrócone przez każdy z tych wywołań systemowych?
- Zrób program zdolny do zarówno operowania na argumentach wiersza poleceń lub użycia STDIN lub STDOUT bazujący na liczbie argumentów wiersza poleceń sprecyzowanych przez ARGC.
- Zmodyfikuj program tak żeby sprawdzał wyniki każdego wywołania systemowego, i wpisywał wiadomości błędu do STDOUT kiedy się pojawią.

Rozdział 6. Odczytywanie i Zapisywanie Prostych Rekordów

Jak wspomniano w Rozdziale 5, wiele aplikacji pracuje z danymi które są "persistent" (na stałe) - w znaczeniu, że dane żyją dłużej niż program przez umieszczenie na dysku w plikach. Możesz zamknąć program i otworzyć go z powrotem, i jesteś tam gdzie zacząłeś. Teraz, są dwa podstawowe rodzaje danych "persistent" - zorganizowane i niezorganizowane. Niezorganizowane dane są takie z jakimi mieliśmy do czynienia w programie **toupper**. One właśnie mają do czynienia z plikami tekstowymi które były otwarte przez osobę. Zawartość plików nie była użyteczna dla programu ponieważ program nie może zinterpretować tego co użytkownik próbuje powiedzieć w tekście bezcelowym.

Zorganizowane dane, z drugiej strony, są tym co komputery robią najlepiej. Zorganizowane dane są danymi które są podzielone na pola i rekordy. Dla większości, pola i rekordy są stałej długości. Ponieważ dane są podzielone na stałej długości rekordy i stałego formatu pola, komputer może zinterpretować dane. Zorganizowane dane mogą zawierać zmiennej długości pola, ale w tym momencie zwykle jesteś w bardziej komfortowej sytuacji z bazą danych.

Ten rozdział zajmuje się odczytywaniem i zapisywaniem prostych, stałej długości rekordów. Powiedzmy, że chcielibyśmy przechować kilka podstawowych informacji o ludziach, których znamy. Moglibyśmy wyobrazić sobie następujący przykład, stałej długości rekord o osobach:

- Imię - 40 bajtów
- Nazwisko - 40 bajtów
- Adres - 240 bajtów
- Wiek - 4 bajty

W tym, wszystko jest daną znakową oprócz wieku, który jest prostym polem numerycznym, używającym standardowego 4-bajtowego słowa (moglibyśmy użyć tylko pojedynczego bajtu do tego, ale trzymanie tego w słowie ułatwia procesowanie).

W programowaniu, często masz konkretne definicje których będziesz używał ciągle wewnątrz programu, lub prawdopodobnie wewnątrz wielu programów. Dobrze jest odseparować je w pliki które są po prostu włączane w pliki języka asemblerowego kiedy potrzeba. Na przykład, w naszych następnych programach będziemy potrzebować dostępu do różnych części rekordu. To oznacza, że powinniśmy znać przesunięcia każdego pola od początku rekordu w kolejności dostępu do nich używając adresowania wskaźnika bazowego. Następujące stałe opisują przesunięcia do struktur. Umieść je w pliku nazwanym **record-def.s**:

```
.equ RECORD_FIRSTNAME, 0
.equ RECORD_LASTNAME, 40
.equ RECORD_ADDRESS, 80
.equ RECORD_AGE, 320

.equ RECORD_SIZE, 324
```

W dodatku, jest kilkanaście stałych które definiujemy za każdym razem w naszych programach, i użytecznie jest umieścić je w pliku, tak, że nie musimy ciągle ich definiować na nowo. Umieść następujące stałe w pliku nazwanym **linux.s**:

#Powszechne Definicje Linuksa

#Liczby Wywołania Systemowego
.equ SYS_EXIT, 1
.equ SYS_READ, 3
.equ SYS_WRITE, 4
.equ SYS_OPEN, 5
.equ SYS_CLOSE, 6
.equ SYS_BRK, 45

#liczba Przerwania Wywołania Systemowego
.equ LINUX_SYSCALL, 0x80

#Deskryptory Pliku Standardowego
.equ STDIN, 0
.equ STDOUT, 1
.equ STDERR, 2

#Kody Statusu Powszechnego
.equ END_OF_FILE, 0

Napiszemy trzy programy w tym rozdziale używając struktur zdefiniowanych w **record-def.s**. Pierwszy program będzie budował plik zawierający kilka rekordów jak zdefiniowano powyżej. Drugi program będzie wyświetlał rekordy w tym pliku. Trzeci program będzie dodawał 1 rok do wieku w każdym rekordzie.
W dodatku do standardowych stałych będziemy używać w tych programach, są jeszcze dwie funkcje które będziemy używać w kilku programach - jeden który odczytuje rekord i drugi który zapisuje rekord.
Jakich parametrów te funkcje potrzebują w kolejności operowania? Generalnie potrzebujemy:
- Lokalizacji bufora do którego możemy wczytać rekord
- Deskryptora pliku z którego chcemy czytać lub do którego chcemy zapisywać
Pierwsze zobaczmy naszą funkcję czytającą:

.include "record-def.s"
.include "linux.s"

#CEL: Ta funkcja czyta rekord z deskryptora pliku
#
#WEJŚCIE: Deskryptor pliku i bufor
#
#WYJŚCIE: Ta funkcja zapisuje dane do bufora i zwraca kod statusu.
#
#ZMIENNE LOKALNE STOSU
.equ ST_READ_BUFFER, 8
.equ ST_FILEDES, 12
.section .text
.globl read_record
.type read_record, @function
read_record:
pushl %ebp
movl %esp, %ebp

```
pushl %eax
movl ST_FILEDES(%ebp), %ebx
movl ST_READ_BUFFER(%ebp), %ecx
movl $RECORD_SIZE, %edx
movl $SYS_READ, %eax
int $LINUX_SYSCALL

#UWAGA - %eax ma wartość powrotu, którą będziemy oddawać z powrotem do naszego programu wywołującego
popl %ebx

movl %ebp, %esp
popl %ebp
ret
```

To jest bardzo prosta funkcja. Czyta dany rozmiar naszej struktury do odpowiedniego rozmiaru bufora z danego deskryptora pliku. Zapisywanie jest podobne:

```
.include "linux.s"
.include "record-def.s"
#CEL: Ta funkcja zapisuje rekord do danego deskryptora pliku
#
#WEJŚCIE: Deskryptor pliku i bufor
#
#WYJŚCIE: Ta funkcja produkuje kod statusu
#
#ZMIENNE LOKALNE STOSU
.equ ST_WRITE_BUFFER, 8
.equ ST_FILEDES, 12
.section .text
.globl write_record
.type write_record, @function
write_record:
pushl %ebp
movl %esp, %ebp

pushl %ebx
movl $SYS_WRITE, %eax
movl ST_FILEDES(%ebp), %ebx
movl ST_WRITE_BUFFER(%ebp), %ecx
movl $RECORD_SIZE, %edx
int $LINUX_SYSCALL

#UWAGA - %eax ma wartość powrotu, którą oddamy z powrotem do naszego programu wywołującego
popl %ebx

movl %ebp, %esp
popl %ebp
ret
```

Teraz skoro mamy napisane nasze podstawowe definicje, jesteśmy gotowi do napisania naszych programów.

Zapisywanie Rekordów

Ten program będzie po prostu zapisywał rekordy na dysk. Będzie:
- Otwierał plik
- Zapisywał trzy rekordy
- Zamykał plik

Napisz następujący kod do pliku zwanego **write-records.s**:

```
.include "linux.s"
.include "record-def.s"

.section .data

#Stałe dane rekordów które chcemy zapisać
#Każdy element danych tekstowych jest wypełniany do prawidłowej długości znakami null (t.j. 0) bajtów.

#.rept jest użyte do wypełniania każdego elementu. .rept mówi asemblerowi aby powtarzał sekcję pomiędzy
#.repr i .endr wyspecyfikowaną liczbę razy.
#To jest używane w tym programie aby dodać dodatkowe znaki null na końcu każdego pola do wypełnienia
record1:
.ascii "Frederick\0"
.rept 31 #Wypełnianie do 40 bajtów
.byte 0
.endr

.ascii "Bartlett\0"
.rept 31 #Wypełnianie do 40 bajtów
.byte 0
.endr

.ascii "4242 S Prairie\nTulsa, OK 55555\0"
.rept 209 #Wypełnianie do 240 bajtów
.byte 0
.endr

.long 45

record2:
.ascii "Marilyn\0"
.rept 32 #Wypełnianie do 40 bajtów
.byte 0
.endr

.ascii "Taylor\0"
```

```
.rept 33 #Wypełnianie do 40 bajtów
.byte 0
.endr

.ascii "2224 S Johannan St\nChicago, IL 12345\0"
.rept 203 #Wypełnianie do 240 bajtów
.byte 0
.endr

.long 29

record3:
.ascii "Derrick\0"
.rept 32 #Wypełnianie do 40 bajtów
.byte 0
.endr

.ascii "McIntire\0"
.rept 31 #Wypełnianie do 40 bajtów
.byte 0
.endr

.ascii "500 W Oakland\nSan Diego, CA 54321\0"
.rept 206 #Wypełnianie do 240 bajtów
.byte 0
.endr

.long 36

#To jest nazwa pliku do którego będziemy zapisywać
file_name:
.ascii "test.dat\0"

.equ ST_FILE_DESCRIPTOR, -4
.globl _start
_start:
#Kopiuje wskaźnik stosu do %ebp
movl %esp, %ebp
#Alokacja przestrzeni do umieszczenia deskryptora pliku
subl $4, %esp

#Otwarcie pliku
movl $SYS_OPEN, %eax
movl $file_name, %ebx
movl $0101, %ecx #Mówi, żeby utworzyć jeśli nie istnieje, i otworzyć do zapisu
movl $0666, %edx
int $LINUX_SYSCALL

#Zachowanie deskryptora pliku
```

```
movl %eax, ST_FILE_DESCRIPTOR(%ebp)

#Zapisanie pierwszego rekordu
pushl ST_FILE_DESCRIPTOR(%ebp)
pushl $record1
call write_record
addl $8, %esp

#Zapisanie drugiego rekordu
pushl ST_FILE_DESCRIPTOR(%ebp)
pushl $record2
call write_record
addl $8, %esp

#Zapisanie trzeciego rekordu
pushl ST_FILE_DESCRIPTOR(%ebp)
pushl $record3
call write_record
addl $8, %esp

#Zamknięcie deskryptora pliku
movl $SYS_CLOSE, %eax
movl ST_FILE_DESCRIPTOR(%ebp), %ebx
int $LINUX_SYSCALL

#WYJŚCIE z programu
movl $SYS_EXIT, %eax
movl $0, %ebx
int $LINUX_SYSCALL
```

To jest całkiem prosty program. Zawiera jedynie definiowanie danych które chcemy zapisać w sekcji **.data**, i wtedy wywoływanie właściwych wywołań systemowych i wywołań funkcji z nimi związanych. Dla przypomnienia wszystkich używanych wywołań systemowych, zobacz Dodatek C.
Mógłbyś zauważyć wiersze:

```
.include "linux.s"
.include "record-def.s"
```

Te sentencje powodują, że dane pliki generalnie są przekazane bezpośrednio tam w kodzie. Nie potrzebujesz tego robić za pomocą funkcji ponieważ linker może się zająć powiązaniem funkcji wyeksportowanych z **.globl**. Jednakże, stałe zdefiniowane w następnym pliku powinny być importowane w ten sposób.
Także, mógłbyś zauważyć użycie nowej dyrektywy asemblera, **.rept**. Ta dyrektywa powtarza zawartość pliku pomiędzy dyrektywami **.rept** i **.endr** liczbę razy sprecyzowaną po **.rept**. Jest ona używana w sposób który my jej użyliśmy - wypełnianie wartościami w sekcji **.data**. W naszym przypadku, dodajemy znaki null na końcu pola aż wypełnią ich zdefiniowane długości.
Aby zbudować te aplikacje, uruchom komendy:

```
as write-records.s -o write-records.o
as write-record.s -o write-record.o
```

ld write-record.o write-records.o -o write-records

Tutaj asemblujemy dwa pliki osobno, i wtedy łączymy je razem używając linkera.
Aby uruchomić ten program, napisz co następuje:

./write-records

To spowoduje utworzenie pliku nazwanego **test.dat** zawierającego te rekordy. Jednakże, ponieważ zawierają one znaki niedrukowalne (szczególnie, znak null), mogą one nie być widzialne w edytorze tekstu. Dlatego potrzebujemy następnego programu aby odczytał je dla nas.

Odczytywanie Rekordów

Teraz rozpatrzymy proces czytania rekordów. W tym programie, będziemy czytać każdy rekord i wyświetlać pierwszą nazwę z każdego rekordu.
Ponieważ każda nazwa osoby jest różnej długości, będziemy potrzebować funkcji do zliczania liczby znaków które chcemy zapisać. Ponieważ dopełniamy każde pole znakami null, możemy po prostu zliczać znaki aż osiągniemy znak null. Zauważ, że to znaczy, że każdy z naszych rekordów musi zawierać co najmniej jeden znak null.
Oto jest ten kod. Umieść go w pliku zwanym **count-chars.s**:

```
#CEL: Zlicza znaki aż bajt null jest osiągnięty.
#
#WEJŚCIE: Adres łańcucha znakowego
#
#WYJŚCIE: Zwraca obliczenie w %eax
#
#PROCESOWANIE:
# Użyte rejestry:
# %ecx - licznik znaków
# %al - bieżący znak
# %edx - adres bieżącego znaku

.type count_chars, @function
.globl count_chars

#To jest gdzie nasz jeden parametr jest na stosie
.equ ST_STRING_START_ADDRESS, 8
count_chars:
pushl %ebp
movl %esp, %ebp

#Licznik zaczyna w zerze
movl $0, %ecx

#Początkowy adres danych
movl ST_STRING_START_ADDRESS(%ebp), %edx
```

```
count_loop_begin:
#Pobierz bieżący znak
movb (%edx), %al
#Czy jest to null?
cmpb $0, %al
#Jeśli tak, kończymy
je count_loop_end
#W przeciwnym razie, inkrementacja licznika i wskaźnika (zwiększenie o 1)
incl %ecx
incl %edx
#Powrót do początku pętli
jmp count_loop_begin

count_loop_end:
#Kończymy. Przeniesienie zliczenia do %eax i powrót
movl %ecx, %eax

popl %ebp
ret
```

Jak możesz zobaczyć, jest to całkiem zrozumiała funkcja. Prosto przebiega pętlę poprzez bajty, zliczając przebiegi, aż natrafi na znak null. Wtedy zwraca zliczenie.
Nasz rekord-czytający program będzie także całkiem zrozumiały. Będzie robił co następuje:
- Otwierał plik
- Przystąpi do czytania rekordu
- Jeśli jesteśmy na końcu pliku, wyjście
- W przeciwnym razie, liczy znaki pierwszej nazwy
- Zapisuje pierwszą nazwę do STDOUT
- Zapisuje nową linię do STDOUT
- Powraca do czytania innego rekordu

Aby to zapisać, potrzebujemy jeszcze jednej prostej funkcji - funkcji zapisu nowej linii do STDOUT.
Umieść następujący kod w **write-newline.s**:

```
.include "linux.s"
.globl write_newline
.type write_newline, @function
.section .data
newline:
.ascii "\n"
.section .text
.equ ST_FILEDES, 8
write_newline:
pushl %ebp
movl %esp, %ebp

movl $SYS_WRITE, %eax
movl ST_FILEDES(%ebp), %ebx
movl $newline, %ecx
movl $1, %edx
int $LINUX_SYSCALL
```

```
movl %ebp, %esp
popl %ebp
ret
```

Teraz jesteśmy gotowi do napisania głównego programu. Tutaj jest kod w **read-records.s**:

```
.include "linux.s"
.include "record-def.s"

.section .data
file_name:
.ascii "test.dat\0"

.section .bss
.lcomm record_buffer, RECORD_SIZE

.section .text
#Główny program
.globl _start
_start:
#To są lokalizacje na stosie gdzie będziemy przechowywać deskryptory wejścia i wyjścia
#(moglibyśmy używać adresów pamięci w sekcji .data zamiast tego)
.equ ST_INPUT_DESCRIPTOR, -4
.equ ST_OUTPUT_DESCRIPTOR, -8

#Kopiuje wskaźnik stosu do %ebp
movl %esp, %ebp
#Alokuje miejsce do trzymania deskryptorów pliku
subl $8, %esp

#Otwarcie pliku
movl $SYS_OPEN, %eax
movl $file_name, %ebx
movl $0, %ecx #To mówi aby otworzyć tylko do odczytu
movl $0666, %edx
int $LINUX_SYSCALL

#Zapis deskryptora pliku
movl %eax, ST_INPUT_DESCRIPTOR(%ebp)

#Nawet pomimo, że jest to stała, zapisujemy deskryptor pliku wyjściowego w zmiennej lokalnej tak więc
#jeśli zdecydujemy później, że nie zawsze będzie to STDOUT, możemy łatwo to zmienić.
movl $STDOUT, ST_OUTPUT_DESCRIPTOR(%ebp)

record_read_loop:
pushl ST_INPUT_DESCRIPTOR(%ebp)
pushl $record_buffer
call read_record
addl $8, %esp
```

```
#Zwraca liczbę bajtów wczytanych.
#Jeśli nie jest to ta sama liczba co poszukiwana, wtedy jest to albo end-of-file lub błąd, więc kończymy
cmpl $RECORD_SIZE, %eax
jne finished_reading

#W przeciwnym razie, wypisuje pierwszą nazwę
#ale najpierw, musimy znać jej rozmiar
pushl $RECORD_FIRSTNAME + record_buffer
call count_chars
addl $4, %esp

movl %eax, %edx
movl ST_OUTPUT_DESCRIPTOR(%ebp),%ebx
movl $SYS_WRITE, %eax
movl $RECORD_FIRSTNAME + record_buffer, %ecx
int $LINUX_SYSCALL

pushl ST_OUTPUT_DESCRIPTOR(%ebp)
call write_newline
addl $4, %esp

jmp record_read_loop

finished_reading:
movl $SYS_EXIT, %eax
movl $0, %ebx
int $LINUX_SYSCALL
```

Aby zbudować ten program, potrzebujemy zasemblować wszystkie części i zlinkować je razem:

```
as read_record.s -o read_record.o
as count_chars.s -o count_chars.o
as write_newline.s -o write_newline.o
as read_records.s -o read_records.o
ld read_record.o count_chars.o write_newline.o read_records.o -o read_records
```

Możesz uruchomić swój program przez **./read_records**.
Jak możesz zobaczyć, ten program otwiera plik i wtedy uruchamia pętlę czytającą, sprawdzając czy nie ma końca pliku, i zapisując nazwisko. Jedyną konstrukcją która mogłaby być nowa jest wiersz mówiący:

```
pushl $RECORD_FIRSTNAME + record_buffer
```

Wygląda to jakbyśmy łączyli i dodawali instrukcję z instrukcją "pushl", ale nie robimy tego. Zauważ, obydwie **RECORD_FIRSTNAME** i **record_buffer** są stałymi. Pierwsza jest stałą bezpośrednią, utworzoną poprzez użycie dyrektywy **.equ**, podczas gdy litera jest zdefiniowana automatycznie przez asembler poprzez użycie jej jako etykiety (jej wartość będąca adresem w którym będzie początek danych następujących po nim). Ponieważ obydwie są stałymi które asembler zna, jest on zdolny dodać je razem podczas asemlowania twojego programu, więc cała instrukcja jest pojedynczym trybem natychmiastowym wkładania pojedynczej stałej.
Stała **RECORD_FIRSTNAME** jest liczbą bajtów po początku rekordu zanim dotrzemy do pierwszej nazwy.

record_buffer jest nazwą naszego bufora do przechowywania rekordów. Dodanie ich razem daje nam adres pierwszej nazwy członka przechowywanego rekordu w **record_buffer**.

Modyfikacja Rekordów

W tym Podrozdziale, napiszemy program który:
- Otwiera plik wejściowy i wyjściowy
- Czyta rekordy z wejścia
- Inkrementuje wiek
- Zapisuje nowy rekord do pliku wyjściowego

Jak większość programów które zrobiliśmy ostatnio, ten program jest całkiem zrozumiały.

```
.include "linux.s"
.include "record-def.s"

.section .data
input_file_name:
.ascii "test.dat\0"

output_file_name:
.ascii "testout.dat\0"

.section .bss
.lcomm record_buffer, RECORD_SIZE

#Przesunięcia stosu zmiennych lokalnych
.equ ST_INPUT_DESCRIPTOR, -4
.equ ST_OUTPUT_DESCRIPTOR, -8

.section .text
.globl _start
_start:
#Kopiuje wskaźnik stosu i robi miejsce dla zmiennych lokalnych
movl %esp, %ebp
subl $8, %esp

#Otwiera plik do czytania
movl $SYS_OPEN, %eax
movl $input_file_name, %ebx
movl $0, %ecx
movl $0666, %edx
int $LINUX_SYSCALL

movl %eax, ST_INPUT_DESCRIPTOR(%ebp)

#Otwiera plik do zapisu
movl $SYS_OPEN,%eax
```

```
movl $output_file_name, %ebx
movl $0101, %ecx
movl $0666, %edx
int $LINUX_SYSCALL

movl %eax, ST_OUTPUT_DESCRIPTOR(%ebp)

loop_begin:
pushl ST_INPUT_DESCRIPTOR(%ebp)
pushl $record_buffer
call read_record
addl $8, %esp

#Zwraca liczbę bajtów czytanych.
#Jeśli nie jest to liczba o którą pytaliśmy, wtedy jest to albo koniec pliku albo błąd, więc opuszczamy
cmpl $RECORD_SIZE, %eax
jne loop_end

#Inkrementacja wieku
incl record_buffer + RECORD_AGE

#Zapisuje rekord
pushl ST_OUTPUT_DESCRIPTOR(%ebp)
pushl $record_buffer
call write_record
addl $8, %esp

jmp loop_begin

loop_end:
movl $SYS_EXIT, %eax
movl $0, %ebx
int $LINUX_SYSCALL
```

Możesz wpisać to jako **add-year.s**. Aby zbudować to, napisz następujące:

```
as add-year.s -o add-year.o
ld add-year.o read-record.o write-record.o -o add-year
```

Aby uruchomić ten program, napisz następujące:

```
./add-year
```

To doda rok do każdego rekordu w **test.dat** i zapisze nowe rekordy do pliku **testout.dat**.
Jak możesz zobaczyć, zapisywanie rekordów stałej długości jest całkiem proste. Musisz tylko wczytać bloki danych do bufora, spreparować je, i zapisać je z powrotem. Niefortunnie, ten program nie zapisuje nowych liczb lat na ekran abyś mógł zweryfikować efektywność swojego programu. Jest tak dlatego ponieważ nie dostaniemy liczb wyświetlających aż do Rozdziału 8 i Rozdziału 10. Po ich przeczytaniu może będziesz chciał wrócić i przepisać ten program aby wyświetlać numeryczne dane które modyfikujemy.

Przegląd

Znajomość koncepcji

- Co to jest rekord?
- Jaka jest przewaga rekordów stałej długości nad rekordami zmiennej długości?
- Jak włączysz stałe w wieloelementowe asemblerowe pliki źródłowe?
- Dlaczego mógłbyś chcieć podzielić projekt na wieloelementowe pliki źródłowe?
- Co robi instrukcja **incl record_buffer + RECORD_AGE**? Jakiego trybu adresowania używa? Jak wiele operandów ma instrukcja **incl** w tym przypadku? Jakie części są kontrolowane przez asembler a jakie części są zarządzane gdy program jest uruchomiony?

Użycie Koncepcji

- Dodaj następnego członka danych do struktury personalnej zdefiniowanej w tym rozdziale, przepisz funkcje czytające, zapisujące i programy aby brały to pod uwagę. Pamiętaj aby zreasemblować i zrelinkować swoje pliki przed uruchomieniem swoich programów.
- Stwórz program który używa pętli do zapisania 30 identycznych rekordów do pliku.
- Stwórz program do szukania największego wieku (liczby lat) w pliku i zwróć ten wiek jako kod statusu tego programu.
- Stwórz program do szukania najmniejszego wieku w pliku i zwróć ten wiek jako kod statusu tego programu.

Idąc Dalej

- Przepisz te programy w tym rozdziale aby używać argumentów wiersza poleceń do precyzowania nazw plików.
- Zbadaj wywołanie systemowe **lseek**. Przepisz program **add-year** aby otworzyć plik źródłowy dla obydwu czytania i zapisywania (użyj $2 dla trybu odczyt/zapis), i zapisz zmodyfikowane rekordy z powrotem do tego samego pliku z którego były wczytane.
- Zbadaj różne kody błędów które mogą być zwrócone przez wywołania systemowe robione w tych programach. Weź jeden do przepisania, i dodaj kod który sprawdza **%eax** na warunki błędu, i, jeśli błąd jest znaleziony, zapisuje wiadomość o nim do STDERR i wychodzi.
- Napisz program który będzie dodawał pojedynczy rekord do pliku przez wczytywanie danych z klawiatury. Pamiętaj, będziesz musiał się upewnić, że dane mają co najmniej jeden znak null na końcu, i powinieneś mieć sposób dla użytkownika powiadomienia, że skończył wpisywanie. Ponieważ nie posiadamy konwersji znaków w liczby, nie będziemy zdolni wczytać wieku z klawiatury, więc będziesz musiał mieć domyślny wiek.
- Napisz funkcję zwaną **compare-strings** która będzie porównywać dwa łańcuchy do 5 znaków. Potem napisz program który pozwala użytkownikowi wprowadzić 5 znaków, a program zwraca wszystkie rekordy których pierwsza nazwa zaczyna się na te 5 znaków.

Rozdział 7. Rozwijanie Solidnych Programów

Ten rozdział zajmuje się rozwijaniem programów które są *solidne*. Solidne programy są zdolne do kontrolowania warunków błędów elegancko. Są to programy które się nie wykolejają bez względu na to co zrobi użytkownik. Budowanie solidnych programów jest kwintesencją praktyki programowania. Pisanie solidnych programów wymaga dyscypliny i pracy - zwykle wymaga szukania każdego możliwego problemu który może się pojawić, i wychodzenia z planem akcji dla twojego programu do podjęcia.

Gdzie Idzie Czas?

Programiści słabo prognozują. W prawie każdym projekcie programistycznym, programistom zabierze dwa, cztery lub nawet osiem razy dłużej skonstruowanie programu lub funkcji niż oryginalnie założyli. Jest wiele powodów tego problemu, włączając w to:
- Programiści nie zawsze przewidują czas na spotkania lub inne nie kodujące czynności z których składa się każdy dzień.
- Programiści często nie doceniają czasów poprawek (jak długo zabierze przeprowadzenie zmian żądanych i zaaprobowanych) dla projektów.
- Programiści nie zawsze rozumieją pełny zakres tego co produkują.
- Programiści często muszą ocenić czas wykonania totalnie różnego rodzaju projekt niż te które wykonywali dotąd, i przez to są niezdolni do dokładnej prognozy.
- Programiści często nie doceniają ilości czasu jaki zabierze otrzymanie programu w pełni solidnego.
Ostatnia pozycja jest tym czym jesteśmy zainteresowani tutaj. *Wiele czasu i wysiłku zabiera rozwój solidnych programów.* Więcej nawet niż ludzie zwykle zgadują, włączając w to doświadczonych programistów. Programiści są tak skoncentrowani na prostym rozwiązywaniu problemu od ręki, że nie dostrzegają możliwych konsekwencji.
W programie **toupper**, nie mamy żadnej akcji jeśli plik który użytkownik wybierze nie istnieje. Program będzie szedł dalej i spróbuje działać mimo wszystko. Nie raportuje żadnej wiadomości o błędzie więc użytkownik nawet nie będzie wiedział, że wpisał nazwę źle. Powiedzmy, że plik docelowy jest na dysku sieciowym i sieć tymczasowo przestała działać. System operacyjny zwraca nam kod statusu w **%eax**, ale my nie sprawdzamy go. Dlatego, jeśli pojawi się brak działania, użytkownik jest totalnie nieświadomy. Ten program jest zdecydowanie niesolidny. Jak możesz zobaczyć, nawet w prostym programie jest wiele rzeczy mogących pójść źle z którymi programista musi się zmagać.
W dużych programach, staje się to o wiele bardziej problematyczne. Zwykle jest wiele więcej możliwych błędnych warunków niż poprawnych warunków. Dlatego, zawsze powinieneś oczekiwać spędzania większości swojego czasu nad sprawdzaniem kodów statusu, pisaniem wychwytywaczy błędów, i rozwiązywaniem podobnych zadań aby uczynić swój program solidnym. Jeśli dwa tygodnie zabierze napisanie programu, prawdopodobnie co najmniej dwa więcej zabierze zrobienie go solidnym. Pamiętaj, że każda wiadomość o błędzie która wyskakuje na twoim ekranie musiała być przez kogoś zaprogramowana.

Kilka Sposobów na Rozwijanie Solidnych Programów

Testowanie Użytkownika
Testowanie jest jedną z najbardziej fundamentalnych rzeczy które robi programista. Jeśli nie testowałeś czegoś, powinieneś przyjąć, że to nie działa. Jednakże, testowanie nie jest tylko upewnianiem się, że twój program działa, jest upewnianiem się, że twój program nie psuje się. Na przykład, jeśli mam program który ma działać tylko z liczbami dodatnimi, powinieneś przetestować co się stanie gdy użytkownik wpisze liczby ujemne. Lub literę. Lub liczbę zero. Musisz przetestować co się stanie jeśli umieszczą spacje przed swoimi liczbami, spacje po liczbach i inne drobne możliwości. Powinieneś się upewnić, że przechowujesz dane użytkownika w sposób który ma sens dla niego, i że przesyłasz te dane w sposób który ma sens dla reszty twojego programu. Kiedy twój program znajdzie wejście które nie ma sensu, powinien on podjąć odpowiednie akcje. Zależnie od twojego programu, to może zawierać zakończenie programu, poproszenie użytkownika o przepisanie ponowne wartości, odnotowanie logu centralnego błędu, cofnięcie operacji, lub zignorowanie go i kontynuacja.
Nie tylko powinieneś przetestować swój program, potrzebujesz także przetestowania go przez innych. Powinieneś włączyć innych programistów i użytkowników twojego programu aby pomogli testować twój program. Jeśli coś jest problemem dla twoich użytkowników, nawet jeśli wydaje się w porządku dla ciebie, powinno być rozwiązane. Jeśli użytkownik nie wie jak używać poprawnie twojego programu, powinno być to traktowane jako błąd który trzeba rozwiązać.
Odkryjesz, że użytkownicy znajdują o wiele więcej błędów w twoim programie niż ty kiedykolwiek mógłbyś. Powodem jest to, że użytkownicy nie wiedzą czego komputer oczekuje. Ty wiesz jakiego rodzaju danych oczekuje komputer, i dlatego jest o wiele bardziej prawdopodobne, że wpiszesz dane które mają sens dla komputera. Użytkownicy wpisują dane które mają sens dla nich. Pozwolenie nie programistom na użycie twojego programu w celu testowania zwykle daje ci o wiele bardziej

dokładne wyniki jak solidny jest rzeczywiście twój program.

Testowanie Danych

Podczas projektowania programów, każda z twoich funkcji powinna być bardzo precyzyjna co do typu i zakresu danych które będzie lub nie będzie akceptować. Wtedy potrzebujesz przetestować te funkcje aby upewnić się, że działają one według dokumentacji kiedy podawane są odpowiednie dane. Najważniejsze jest testowanie *warunków granicznych* lub *brzegowych*. Warunki graniczne to wejścia które są najbardziej prawdopodobne, że spowodują problemy lub działania nieoczekiwane.

Podczas testowania danych numerycznych jest kilka warunków granicznych które zawsze powinieneś przetestować:

- Liczba 0
- Liczba 1
- Liczba wewnątrz oczekiwanego zakresu
- Liczba poza oczekiwanym zakresem
- Pierwsza liczba w oczekiwanym zakresie
- Ostatnia liczba w oczekiwanym zakresie
- Pierwsza liczba poniżej oczekiwanego zakresu
- Pierwsza liczba powyżej oczekiwanego zakresu

Na przykład, jeśli mam program który powinien akceptować wartości pomiędzy 5 i 200, powinienem przetestować 0, 1, 4, 5, 153, 200, 201, i 255 jako minimum (153 i 255 były wybrane przypadkowo wewnątrz i na zewnątrz zakresu). To samo dotyczy każdej listy danych jakie masz. Powinieneś przetestować czy twój program zachowuje się jak oczekiwano dla list: 0 elementów, 1 element, wielka liczba elementów, itd. W dodatku, powinieneś także testować każdy zwrotny punkt jaki masz. Na przykład, jeśli masz różny kod dla osób poniżej wieku 30 lat i różny dla osób powyżej 30, przykładowo, potrzebowałbyś przetestować go dla osób w wieku 29, 30 i 31 lat, co najmniej.

Będzie kilka wewnętrznych funkcji które uznasz, że dają dobre dane ponieważ sprawdziłeś je na błędy do tej pory. Jednakże, podczas rozwoju często powinieneś sprawdzać bezbłędność mimo tego, jako że twój inny kod może mieć błędy w sobie. Aby zweryfikować skład i walidację danych w czasie rozwoju, większość języków posiada narzędzia do łatwego sprawdzania przypuszczeń co do poprawności danych. W języku C jest makro **assert**. Możesz po prostu umieścić w swoim kodzie **assert (a > b);**, i będzie to dawało błąd jeśli osiągnie taki kod kiedy ten warunek nie jest prawdziwy. W dodatku, ponieważ takie sprawdzenie jest stratą czasu po tym jak twój kod jest stabilny, makro **assert** pozwala wyłączyć warunek deklaracji na czas kompilacji. To upewnia, że twoje funkcje przyjmują poprawne dane bez powodowania niepotrzebnych opóźnień wydań publicznych kodu.

Testowanie Modulu

Powinieneś nie tylko testować swój program jako całość, potrzebujesz także testować indywidualne części swojego programu. Kiedy rozwijasz swój program, powinieneś testować indywidualne funkcje przez dostarczanie im danych aby upewnić się, że odpowiadają prawidłowo.

W celu wykonania tego efektywnie, musisz rozwijać funkcje których podstawowym celem jest wywoływanie funkcji dla testowania. Są one zwane *drivers* (nie mylić z "hardware drivers"). One po prostu ładują twoją funkcję, wyposażają ją w dane, i sprawdzają wyniki. Jest to specjalnie użyteczne jeśli pracujesz na częściach niedokończonego programu. Ponieważ nie możesz testować wszystkich części razem, możesz stworzyć program (driver) napędu który będzie testował każdą funkcję indywidualnie.

Także, kod który testujesz może wywoływać funkcje jeszcze nie rozwinięte. W celu uporania się z tym problemem, możesz napisać małą funkcję zwaną *stub* która po prostu zwraca wartości potrzebne funkcji do działania. Na przykład, w aplikacji e-commerce, miałem funkcję nazwaną **is_ready_to_checkout**. Przed tym jak miałem czas aby rzeczywiście napisać tę funkcję ustawiłem ją na zwracanie prawdy na każde wywołanie więc te funkcje które na niej polegały mogły mieć odpowiedź. To pozwoliło mi testować funkcje które polegały na **is_ready_to_checkout** bez tej funkcji zaimplementowanej całkowicie.

Efektywna Obsługa Błędów

Ważne jest wiedzieć nie tylko jak testować ale co robić kiedy błąd zostanie wykryty.

Posiadanie Kodu Błędu na Wszystko
Prawdziwie solidne oprogramowanie ma unikalny kod błędu dla każdej możliwej sytuacji. Przez prostą znajomość kodu błędu, powinieneś być zdolny do znalezienia miejsca w twoim kodzie gdzie ten błąd był zasygnalizowany.
Jest to ważne ponieważ kod błędu zwykle jest wszystkim co użytkownik musi przekazać kiedy raportuje błędy.
Dlatego, powinno być to tak użyteczne jak tylko możliwe.
Kodom błędów powinny także towarzyszyć opisowe wiadomości o błędzie. Jednakże, tylko w rzadkich okolicznościach wiadomość o błędzie powinna zgadywać dlaczego błąd się pojawił. Powinna tylko relacjonować co się stało. W 1995 pracowałem dla "Internet Service Provider". Jedna z przeglądarek sieciowych które wspieraliśmy starała się zgadnąć raczej przyczynę każdego błędu internetowego niż raportować błąd. Jeśli komputer nie był podłączony do Internetu i użytkownik próbował się podłączyć do sieci, mówiła że jest problem z "Internet Service Provider", że serwer nie działa, i że użytkownik powinien się skontaktować z "Internet Service Provider" aby rozwiązać ten problem. Niemal czwarta część naszych zapytań była od osób które otrzymały tę wiadomość, ale jedynie potrzeba było podłączyć się do Internetu przed próbą użycia przeglądarki. Jak mogłeś zobaczyć, próba zdiagnozowania co jest problemem może prowadzić do jeszcze większej liczby problemów niż je rozwiązać. Lepiej jest tylko raportować kody błędu i wiadomości, i mieć osobne źródła dla użytkowników do rozwiązywania problemów aplikacji. Przewodnik rozwiązywania problemów, nie sam program, jest odpowiednim miejscem do wyliczania możliwych powodów i kierunków akcji dla każdej wiadomości o błędzie.

Punkty Naprawy
W celu uproszczenia zarządzania błędami, często użyteczne jest podzielenie twojego programu na odróżniające się jednostki, gdzie każda jednostka psuje się i jest naprawiana jako całość. Na przykład, mógłbyś podzielić swój program tak, że odczyt pliku konfiguracyjnego byłby jednostką. Jeśli odczyt pliku konfiguracyjnego pada w jakimś punkcie (otwieranie pliku, odczyt pliku, próba dekodowania pliku, itd.) wtedy program mógłby po prostu traktować to jako problem pliku konfiguracyjnego i wskoczyć do *punktu naprawy* dla rozwiązania tego problemu. W ten sposób redukujesz liczbę mechanizmów zarządzania błędami których potrzebujesz dla swojego programu, ponieważ naprawa błędów jest robiona na znacznie bardziej generalnym poziomie.
Zauważ, że nawet z punktami naprawy, twoje wiadomości o błędach muszą być precyzyjne co było problemem. Punkty naprawy są podstawowymi jednostkami do naprawy błędów, nie do wykrywania błędów. Wykrywanie błędów ciągle musi być wyjątkowo dokładne, a raporty błędów potrzebują dokładnych kodów błędów i wiadomości.
Podczas używania punktów naprawy, często potrzebujesz włączyć kod czyszczący aby zarządzać różnymi zdarzeniami. Na przykład, w naszym przykładowym pliku konfiguracyjnym, funkcja naprawy potrzebowałaby zawierać kod do sprawdzenia i stwierdzenia czy plik konfiguracyjny jest ciągle otwarty. Zależnie od tego gdzie błąd się pojawił, plik mógł pozostać otwarty. Funkcja naprawy musi sprawdzić ten warunek i każdy inny warunek mogący prowadzić do niestabilności systemu, i wrócić program do regularnego stanu.
Najprostszym sposobem zarządzania punktami naprawy jest opakowanie całego programu w pojedynczy punkt naprawy. Mógłbyś mieć prostą funkcję raportującą błędy którą możesz wywołać z kodem błędu i wiadomością. Funkcja mogłaby wypisać je i po prostu opuścić program. Zwykle nie jest to najlepsze rozwiązanie dla rzeczywistych sytuacji, ale jest to dobra ucieczka, mechanizm ostatniej szansy.

Robienie Naszych Programów Bardziej Solidnymi

Ten podrozdział będzie przerabiał program z Rozdziału 6 **add-year.s** na trochę bardziej solidny.
Ponieważ jest to bardzo prosty program, ograniczymy się do pojedynczego punktu naprawy który pokryje cały program.
Jedyną rzecz którą zrobimy aby naprawić jest wypisanie błędu i wyjście. Kod do zrobienia tego jest bardzo prosty:

```
.include "linux.s"
.equ ST_ERROR_CODE, 8
.equ ST_ERROR_MSG, 12
.globl error_exit
.type error_exit, @function
error_exit:
pushl %ebp
movl %esp, %ebp

#Wypisz kod błędu
movl ST_ERROR_CODE(%ebp), %ecx
pushl %ecx
call count_chars
popl %ecx
movl %eax, %edx
movl $STDERR, %ebx
movl $SYS_WRITE, %eax
int $LINUX_SYSCALL

#Wypisuje wiadomość błędu
movl ST_ERROR_MSG(%ebp), %ecx
pushl %ecx
call count_chars
popl %ecx
movl %eax, %edx
movl $STDERR, %ebx
movl $SYS_WRITE, %eax
int $LINUX_SYSCALL

pushl $STDERR
call write_newline

#Wyjście ze statusem 1
movl $SYS_EXIT, %eax
movl $1, %ebx
int $LINUX_SYSCALL
```

Umieść to w pliku zwanym **error-exit.s**. Aby wywołać, musisz tylko włożyć adres wiadomości błędu i potem kod błędu na stos, i wywołać funkcję.
Teraz poszukajmy potencjalnych punktów błędu w naszym programie **add-year**. Przede wszystkim, nie sprawdzamy czy nasze wywołania systemowe wejścia (**open**) rzeczywiście prawidłowo działa. Linux zwraca swój kod statusu w **%eax**, więc powinniśmy sprawdzić czy tam jest błąd.

```
#Otwarcie pliku do odczytu
movl $SYS_OPEN, %eax
movl $input_file_name, %ebx
movl $0, %ecx
movl $0666, %edx
```

```
int $LINUX_SYSCALL

movl %eax, ST_INPUT_DESCRIPTOR(%ebp)

#To będzie testować i sprawdzać czy %eax jest ujemne. Jeśli nie jest ujemne, skok do continue_processing.
#W przeciwnym razie będzie zarządzać warunkiem błędu który ta ujemna liczba reprezentuje.
cmpl $0, %eax
jl continue_processing

#Prześlij błąd
.section .data
no_open_file_code:
.ascii "0001: \0"
no_open_file_msg:
.ascii "Can't Open Input File\0"

.section .text
pushl $no_open_file_msg
pushl $no_open_file_code
call error_exit

continue_processing:
#Reszta programu
```

Więc, po wywołaniu tego wywołania systemowego, sprawdzamy czy mamy błąd przez sprawdzenie czy wynik wywołania systemowego jest mniejszy od zera. Jeśli tak, wywołujemy naszą procedurę raportującą błędy i wyjścia.
Po każdym wywołaniu systemowym, wywołaniu funkcji lub instrukcji która może dawać błędne wyniki powinieneś dodać kod sprawdzający błędy i zarządzający nimi.
Aby zasemblować i zlinkować te pliki, zrób:

```
as add-year.s -o add-year.o
as error-exit.s -o error-exit.o
ld add-year.o write-newline.o error-exit.o read-record.o write-record.o count-chars.o -o add-year
```

Teraz spróbuj uruchomić program bez potrzebnych plików. Obecnie wychodzi czysto i elegancko!

Przegląd

Znajomość Koncepcji

- Jakie są powody trudności programistów z dotrzymywaniem terminów?
- Znajdź swój ulubiony program, i spróbuj używać go w kompletnie zły sposób. Otwórz pliki niewłaściwego typu, wybierz niewłaściwe opcje, zamknij okna które przypuszcza się, że są otwarte, itd. Oblicz ile różnych scenariuszy błędu muszą tłumaczyć.
- Co to są przypadki brzegowe? czy możesz podać przykłady numerycznych przypadków brzegowych?
- Dlaczego testowanie użytkownika jest tak ważne?
- Po co są używane "stubs" i "drivers"? Jaka jest różnica między tymi dwoma?
- Po co są używane punkty naprawy?

- Ile różnych kodów błędu powinien mieć program?

Użycie Koncepcji

- Przejdź program **add-year.s** i dodaj kod sprawdzający błędy po każdym wywołaniu systemowym.
- Wybierz inny program który zrobiliśmy do tej pory, i dodaj sprawdzanie błędów do tego programu.
- Dodaj mechanizm naprawczy dla **add-year.s** który pozwala na odczyt z STDIN jeśli nie można otworzyć standardowego pliku.

Idąc Dalej

- Co, jeśli w ogóle, powinieneś zrobić jeżeli twoja funkcja raportująca błędy padnie? Dlaczego?
- Spróbuj znaleźć błędy w co najmniej jednym programie o otwartym kodzie. Umieść raport błędu w pliku dla niego.
- Spróbuj naprawić błąd który znalazłeś w poprzednim ćwiczeniu.

Rozdział 8. Dzielenie Funkcji z Kodem Bibliotek

Do teraz powinieneś uświadomić sobie, że komputer musi wykonywać mnóstwo pracy nawet przy prostych zadaniach. Dlatego właśnie, musisz wykonać wiele pracy aby napisać kod dla komputera nawet do wykonania prostych zadań. W dodatku, zadania programistyczne zwykle nie są bardzo proste. Dlatego, potrzebujemy sposobu uczynienia tego łatwiejszym dla nas samych. Jest kilka dróg zrobienia tego, włączając w to:

- Pisanie kodu w języku wysokiego poziomu zamiast języka asemblerowego
- Posiadanie wielkiej ilości kodu napisanego wcześniej który możesz wycinać i wklejać do twoich własnych programów
- Posiadanie zestawu funkcji w systemie które są dzielone przez każdy program który życzy sobie ich użycia

Wszystkie trzy są zwykle używane do pewnego stopnia w każdym projekcie. Pierwsza opcja będzie eksplorowana głębiej w Rozdziale 11. Druga opcja jest użyteczna ale cierpi na kilka niedogodności, włączając w to:

- Kod który jest kopiowany musi być znacznie zmodyfikowany aby dopasował się do otaczającego kodu.
- Każdy program zawierający skopiowany kod ma ten sam kod w sobie, w ten sposób marnotrawi wiele miejsca.
- Jeśli jest znaleziony błąd w jakimś kopiowanym kodzie musi być naprawiony w każdym programie aplikacyjnym.

Dlatego, druga opcja jest zwykle używana ostrożnie. Jest zwykle używana tylko w przypadkach gdzie kopiujesz i wklejasz szkieletowy kod dla specyficznego typu zadania, i dodajesz twoje specyficzne programowo szczegóły. Trzecia opcja jest tą która jest używana najczęściej. Trzecia opcja zawiera posiadanie centralnego repozytorium dzielonego kodu. Wtedy, zamiast marnotrawienia miejsca na przechowywanie tych samych kopii funkcji przez każdy program, mogą one prosto wskazywać biblioteki dzielone które zawierają potrzebne funkcje. Jeśli jest znaleziony błąd w jednej z tych funkcji, musi być on naprawiony tylko wewnątrz jednego pliku biblioteki funkcji, a wszystkie aplikacje które jej używają są automatycznie uaktualnione. Główna niedogodność z tym związana jest taka, że tworzy kilka problemów z zależnościami, włączając w to:

- Jeśli wiele aplikacji używa pliku dzielonego, skąd wiemy kiedy jest bezpiecznie usunąć ten plik? Na przykład, jeśli trzy aplikacje dzielą plik funkcji i 2 z tych programów są usuwane, skąd system wie, że ciągle egzystuje aplikacja która używa tego kodu, i dlatego nie powinien być on usunięty?
- Niektóre programy bezmyślnie polegają na błędach wewnątrz dzielonych funkcji. Dlatego, jeśli uaktualnienie dzielonego programu naprawia błąd od którego program zależał, mogłoby to spowodować, że ta aplikacja przestanie funkcjonować.

Te problemy prowadzą do tego co jest znane jako "piekło DLL". Jednakże, generalnie przyznaje się, że plusy przewyższają minusy.

W programowaniu, te pliki dzielonego kodu są określane jako *biblioteki dzielone*, *obiekty dzielone*, *biblioteki dynamicznie wiązane*, *DLL*, lub *pliki .so*. My będziemy je określać jako *biblioteki dzielone*.

Użycie Biblioteki Dzielonej

Program który będziemy analizować tutaj jest prosty - wypisuje znaki **hello world** na ekranie i wychodzi. Regularny program, **helloworld-nolib.s**, wygląda tak:

#CEL: Ten program wypisuje wiadomość "hello world" i wychodzi

```
.include "linux.s"
.section .data

helloworld:
.ascii "hello world\n"
helloworld_end:

.equ helloworld_len, helloworld_end - helloworld

.section .text
.globl _start
_start:
movl $STDOUT, %ebx
movl $helloworld, %ecx
movl $helloworld_len, %edx
movl $SYS_WRITE, %eax
int $LINUX_SYSCALL

movl $0, %ebx
movl $SYS_EXIT, %eax
int $LINUX_SYSCALL
```

Nie jest to zbyt długie. Jednakże, zobacz jak krótki jest **helloworld-lib** który używa biblioteki:

#CEL: Ten program wypisuje wiadomość **hello world** i wychodzi

```
.section .data

helloworld:
.ascii "hello world\n\0"

.section .text
.globl _start
_start:
pushl $helloworld
call printf

pushl $0
call exit
```

Jest to nawet krótsze!
Teraz, budowanie programów które używają bibliotek dzielonych jest trochę różne niż normalnie. Możesz zbudować pierwszy program normalnie przez użycie:

as helloworld-nolib.s -o helloworld-nolib.o
ld helloworld-nolib.o -o helloworld-nolib

Jednakże, w celu zbudowania drugiego programu, musisz zrobić to:

as helloworld-lib.s -o helloworld-lib.o
ld -dynamic-linker /lib/ld-linux.so.2 -o helloworld-lib helloworld-lib.o -lc

Opcja **-dynamic-linker /lib/ld-linux.so.2** pozwala naszemu programowi dowiązać biblioteki. To buduje wykonywanie tak, że przed wykonywaniem, system operacyjny będzie ładował program **/lib/ld-linux.so.2** który załaduje zewnętrzne biblioteki i powiąże je z programem. Ten program jest znany jako *linker dynamiczny*.
Opcja **-lc** mówi aby dowiązać do biblioteki **c**, nazwanej **libc.so** w systemie GNU/Linux. Nadana nazwa biblioteki, **c** w tym przypadku (zwykle nazwy bibliotek są dłuższe niż pojedyncza litera), linker GNU/Linux poprzedza łańcuchem **lib** na początku nazwy biblioteki i dodaje **.so** na jej końcu formując nazwę pliku biblioteki. Ta biblioteka zawiera wiele funkcji do zautomatyzowania wszystkich typów zadań. Dwie które używamy to **printf**, która drukuje łańcuchy, i **exit**, która wychodzi z programu.
Zauważ, że symbole **printf** i **exit** są prosto powiązane przez nazwę wewnątrz programu. W poprzednich rozdziałach, linker mógł powiązać wszystkie nazwy do adresów pamięci fizycznej, i nazwy mogłyby być wyrzucone. Kiedy używamy dynamicznego dowiązania, nazwa sama rezyduje wewnątrz wykonawcy i jest dowiązywana przez linker dynamiczny kiedy jest uruchamiana. Kiedy program jest uruchomiony przez użytkownika, linker dynamiczny ładuje biblioteki dzielone wypisane w naszych rozkazach linkujących, i wtedy szuka wszystkich nazw funkcji i zmiennych które były nazwane przez nasz program ale nie znalezione w czasie linkowania, i dopasowuje je do korespondujących wejść w bibliotekach dzielonych które ładuje. Wtedy wymienia wszystkie nazwy z adresami do których są ładowane. Brzmi to jak pożerające czas. Jest tak w niewielkim stopniu, ale to zachodzi tylko raz - w czasie rozpoczęcia programu.

Jak Działają Biblioteki Dzielone

W naszych pierwszych programach, cały kod był zawarty wewnątrz pliku źródłowego. Takie programy są zwane *programami statycznie-linkowanymi*, ponieważ zawierają one wszystkie potrzebne programowi funkcjonalności które nie są zarządzane przez kernel. W programach które napisaliśmy w Rozdziale 6, używaliśmy zarówno pliku głównego programu i plików zawierających procedury używane przez wiele programów. W tych przypadkach, łączyliśmy cały kod razem używając linkera w czasie linkowania, więc było to ciągle statyczne linkowanie. Jednakże, w programie **helloworld-lib**, zaczęliśmy używać bibliotek dzielonych. Kiedy używasz bibliotek dzielonych, twój program jest wtedy *linkowany dynamicznie*, co oznacza, że nie cały kod potrzebny do uruchomienia programu jest rzeczywiście zawarty wewnątrz samego pliku programu, ale w zewnętrznych bibliotekach.
Kiedy umieścimy **-lc** w komendzie do linkowania programu **helloworld**, to powie linkerowi aby użył biblioteki **c** (**libc.so**) do szukania innych symboli które nie były jeszcze zdefiniowane w **helloworld.o**. Jednakże, to w rzeczywistości nie dodaje żadnego kodu do naszego programu, tylko zaznacza w programie gdzie szukać. Kiedy program **helloworld** zaczyna się, plik **/lib/ld-linux.so.2** jest ładowany najpierw. To jest linker dynamiczny. Patrzy w nasz program **helloworld** i widzi, że potrzebuje on biblioteki **c** do uruchomienia. Więc, szuka pliku zwanego **libc.so** w standardowych miejscach (wypisanych w **/etc/ld.so.conf** i w zawartości zmiennej środowiskowej **LD_LIBRARY_PATH**), wtedy zagląda tam po wszystkie potrzebne symbole (**printf** i **exit** w tym przypadku), i wtedy ładuje bibliotekę do pamięci wirtualnej programu. Ostatecznie, zamienia wszystkie instancje **printf** w programie na rzeczywistą lokalizację **printf** w bibliotece.
Uruchom następującą komendę:

ldd ./helloworld-nolib

Powinno to dać raport **not a dynamic executable**. Właśnie tak jak mówiliśmy - **helloworld-nolib** jest statycznie łączonym programem. Jednakże, spróbuj tego:

ldd ./helloworld-lib

To zaraportuje coś takiego

libc.so.6 => /lib/libc.so.6 (0x4001d000)
/lib/ld-linux.so.2 => /lib/ld-linux.so.2 (0x400000000)

Liczby w nawiasach mogą być różne w twoim systemie. To oznacza, że program **helloworld** jest dowiązany do **libc.so.6** (**.6** jest numerem wersji), która znajduje się w **/lib/libc.so.6**, i **/lib/ld-linux.so.2** znajduje się w **/lib/ld-linux.so.2**. Te biblioteki muszą być załadowane zanim program może być uruchomiony. Jeśli jesteś zainteresowany, uruchom program **ldd** dla różnych programów które są w twojej dystrybucji Linuksa, i zobacz na jakich bibliotekach one polegają.

Szukanie Informacji o Bibliotekach

Dobrze, więc teraz kiedy już wiesz o bibliotekach, pytaniem jest, jak odszukać które biblioteki masz w swoim systemie i co one robią? Odsuńmy na minutę to pytanie i zadajmy następne: Jak programiści opisują funkcje jeden drugiemu w swoich dokumentacjach?
Spójrzmy na funkcję **printf**. Jej interfejs wywołania (zwykle określany jako prototyp) wygląda tak:

int printf(char *string, ...);

W Linuksie, funkcje są opisane w języku programowania C. Faktycznie, większość programów Linuksa jest napisana w C. To dlatego większość dokumentacji i binarnych porównań jest zdefiniowana z użyciem języka C. Interfejs funkcji **printf** powyżej jest opisany z użyciem języka programowania C.
Ta definicja oznacza, że to jest funkcja **printf**. Rzeczy w środku nawiasów są parametrami lub argumentami funkcji. Pierwszy parametr tutaj to **char *string**. Oznacza to, że jest parametr nazwany **string** (nazwa nie jest ważna, oprócz użycia do mówienia o tym), który ma typ **char ***. **char** oznacza, że wymaga znaku pojedynczo-bajtowego. **** po tym oznacza, że nie wymaga w rzeczywistości znaku jako argumentu, ale zamiast tego chce adresu znaku lub sekwencji znaków. Jeśli popatrzysz w tył na nasz program **helloworld**, zauważysz,że wywołanie funkcji wygląda tak:

pushl $hello
call printf

Więc, włożyliśmy adres łańcucha **hello**, raczej niż rzeczywiste znaki. Mógłbyś zauważyć, że nie włożyliśmy długości łańcucha. Sposobem w jaki **printf** znalazła koniec łańcucha było to, że zakończyliśmy go znakiem null (\0). Wiele funkcji działa w ten sposób, specjalnie funkcje języka C. **int** przed definicją funkcji mówi jaki typ wartości funkcja będzie zwracać w **%eax** kiedy wraca. **printf** będzie zwracać **int** kiedy zakończy. Teraz, po **char *string**, mamy serię kropek, ... To oznacza, że może wziąć niezdefiniowaną liczbę dodatkowych argumentów po tym łańcuchu. Większość funkcji może wziąć tylko określoną liczbę argumentów. **printf**, jednakże, może wziąć wiele. Będzie ona patrzyła na parametry **string**, i gdzie zobaczy znaki **%s**, będzie szukać następnego łańcucha ze stosu do umieszczenia, i gdzie zobaczy **%d** będzie szukać liczby ze stosu do umieszczenia. Najlepiej jest to opisać używając przykładu:

#CEL: Ten program jest do zademonstrowania jak wywołać "printf"

```
.section .data

#Ten łańcuch jest zwany łańcuchem formatującym.
#Jego pierwszy parametr, a printf używa go do przeszukania ile parametrów było dane, i jakiego są one rodzaju.
firststring:
.ascii "Hello! %s is a %s who loves the number %d\n\0"
name:
.ascii "Jonathan\0"
personstring:
.ascii "person\0"
#To mogłoby być także .equ, ale zdecydowaliśmy się dać rzeczywistą lokalizację pamięci tak dla jaj
numberloved:
.long 3
.section .text
.globl _start
_start:
#zauważ, że parametry są przekazywane w odwrotnej kolejności niż są wypisane w prototypie funkcji.
pushl numberloved #To jest %d
pushl $personstring #To jest drugi %s
pushl $name #To jest pierwszy %s
pushl $firststring #To jest łańcuch formatujący w prototypie
call printf

pushl $0
call exit
```

Wpisz to z nazwą pliku **printf-example.s**, i wtedy wykonaj następujące komendy:

```
as printf-example.s -o printf-example.o
ld printf-example.o -o printf-example -lc -dynamic-linker /lib/ld-linux.so.2
```

Wtedy uruchom ten program z **./printf-example**, i powinno to wyświetlić:

```
Hello! Jonathan is a person who loves the number 3
```

Teraz, jeśli popatrzysz w kod, zobaczysz, że włożyliśmy łańcuch formatujący jako ostatni, chociaż jest wypisany jako pierwszy parametr. Zawsze wkładasz parametry funkcji w odwrotnej kolejności. Możesz być ciekaw skąd funkcja **printf** wie ile jest parametrów. Cóż, przeszukuje twój łańcuch, i liczy ile **%d** i **%s** znajdzie, i wtedy ściąga taką liczbę parametrów ze stosu. Jeśli parametr pasuje do **%d**, traktuje go jak liczbę, i jeśli pasuje do **%s**, traktuje to jako wskaźnik do łańcucha kończącego się znakiem null. **printf** posiada wiele więcej atrybutów niż te, ale to są najbardziej użyteczne. Więc, jak możesz zobaczyć, **printf** może utworzyć wyjście o wiele łatwiej, ale ma także wiele komplikacji, ponieważ musi zliczyć znaki w łańcuchu, przeszukać go dla wszystkich znaków kontrolnych potrzebnych do zamiany, ściągnąć je ze stosu, konwertować je do odpowiedniej reprezentacji (liczby muszą być konwertowane do łańcuchów, itd.), i połączyć je wszystkie razem odpowiednio.
Widzieliśmy jak używać prototypów języka programowania C do wywoływania funkcji biblioteki. Aby używać je efektywnie, jednakże, powinieneś znać kilkanaście dalszych możliwych typów danych dla odczytu funkcji. Tutaj są główne:

int

int jest liczbą całkowitą (4 bajty na procesorze x86).

long
long jest także liczbą całkowitą (4 bajty na procesorze x86).

long long
long long jest liczbą całkowitą dłuższą niż **long** (8 bajtów na procesorze x86).

short
short jest liczbą całkowitą krótszą niż **int** (2 bajty na procesorze x86).

char
char jest liczbą całkowitą jedno bajtową. Jest to najczęściej używane do przechowywania danych znakowych, ponieważ łańcuchy ASCII zwykle są reprezentowane przez jeden bajt na znak.

float
float jest liczbą zmiennoprzecinkową (4 bajty na procesorze x86). Liczby zmiennoprzecinkowe będą wytłumaczone głębiej w Podrozdziale nazwanym *Liczby Zmiennopozycyjne* w Rozdziale 10.

double
double jest liczbą zmiennoprzecinkową która jest większa niż **float** (8 bajtów na procesorze x86).

unsigned
unsigned jest modyfikatorem używanym dla każdego z powyższych typów który uniemożliwia użycie ich jako liczb ze znakiem. Różnica pomiędzy liczbami ze znakiem i bez znaku będą dyskutowane w Rozdziale 10.

*
Gwiazdka (*) jest używana aby odnotować, że ta dana nie jest rzeczywistą wartością, ale zamiast tego jest wskaźnikiem na lokalizację przechowującą daną wartość (4 bajty na procesorze x86). Więc, powiedzmy, że w lokalizacji pamięci **my_location** masz umieszczoną liczbę 20. Jeśli prototyp mówi aby przesłać **int**, mógłbyś użyć trybu adresowania bezpośredniego i wykonać **pushl my_location**. Jednakże, jeśli prototyp powiedział aby przesłać **int ***, mógłbyś wykonać **pushl $my_location** - tryb natychmiastowy wkładania adresu w którym rezyduje ta wartość. W dodatku do wskazywania adresu pojedynczej wartości, wskaźniki mogą także być używane do przekazywania sekwencji konsekwentnych lokalizacji, zapoczątkowanych przez wartość tej na którą wskazuje wskaźnik. Jest to zwane tablicą.

struct
struct jest zestawem elementów danych które są złożone razem pod jedną nazwą. Na przykład mógłbyś zadeklarować:
```
struct teststruct {
int a;
char *b;
};
```
i za każdym razem gdy uruchamiasz **struct teststruct** mógłbyś wiedzieć, że są to aktualnie dwa słowa jeden po drugim, pierwszy jest liczbą całkowitą, drugi wskaźnikiem do znaku lub grupy znaków. Nigdy nie widzisz struktur przekazanych jako argumenty do funkcji. Zamiast tego, zwykle widzisz wskaźniki do struktur przekazane jako argumenty. Jest tak ponieważ przekazywanie struktur do funkcji jest znacznie skomplikowane, odkąd mogą one zabierać tak wiele lokalizacji przechowywania.

typedef
typedef generalnie pozwala ci zmienić nazwę typu. Na przykład, mogę zrobić **typedef int myowntype;** w programie C, i za

każdym razem gdy napiszę **myowntype**, mogłoby być tak jakbym napisał **int**. Może to powodować rodzaj zniechęcenia, ponieważ musisz zwracać uwagę co wszystkie te **typedef**s i **struct**s w prototypie funkcji rzeczywiście znaczą. Jednakże, **typedef**s są użyteczne nadając typom bardziej znaczące i opisowe nazwy.

Uwaga o Kompatybilności: Wypisane rozmiary są dla maszyn kompatybilnych z Intelem (x86). Inne maszyny będą miały różne rozmiary. Także, nawet kiedy parametry krótsze niż słowo są przekazane do funkcji, są one przekazywane jako długie (long) na stos.

Jest to jak czytać dokumentację funkcji. Teraz, powróćmy do pytania jak szukać czegoś o bibliotekach. Większość twoich bibliotek systemowych jest w **/usr/lib** lub **/lib**. Jeśli chcesz tylko zobaczyć jakie symbole one definiują, uruchom **objdump -R FILENAME** gdzie **FILENAME** jest pełną ścieżką do biblioteki. Wynik tego nie jest zbyt pomocny, chociaż, do znalezienia interfejsu którego mógłbyś potrzebować. Zwykle, na początku musisz wiedzieć jakiej chcesz biblioteki, i wtedy właśnie przeczytać dokumentację. Większość bibliotek ma manuale i strony man dla swoich funkcji. Internet jest najlepszym źródłem dokumentacji do bibliotek. Większość bibliotek z projektu GNU ma także strony info, które są nieco gładsze niż strony man.

Użyteczne Funkcje

Kilka użytecznych funkcji z biblioteki **c** których chciałbyś być świadom zawiera:
- **size_t strlen (const char *s)** oblicza rozmiar łańcuchów kończących się znakiem null.
- **int strcmp (const char *s1, const char *s2)** porównuje dwa łańcuchy alfabetycznie.
- **char * strdup (const char *s)** pobiera wskaźnik na łańcuch, i tworzy nową kopię w nowej lokalizacji, i zwraca nową lokalizację.
- **FILE * fopen (const char *filename, const char *opentype)** otwiera zarządzany, zbuforowany plik (pozwala na łatwiejszy odczyt i zapis niż użycie bezpośrednie deskryptorów pliku).
- **int fclose (FILE *stream)** zamyka plik otwarty z **fopen**.
- **char * fgets (char *s, int count, FILE *stream)** formuje wiersz znaków w łańcuch **s**.
- **int fputs (const char *s, FILE *stream)** zapisuje łańcuch do danego otwartego pliku.
- **int fprintf (FILE *stream, const char *template, ...)** jest jak **printf**, ale używa raczej otwartego pliku niż domyślnego użycia standardowego wyjścia.

Możesz znaleźć kompletny manual do tej biblioteki idąc do
http://www.gnu.org/software/libc/manual/

Budowanie Biblioteki Dzielonej

Powiedzmy, że chcielibyśmy wziąć cały nasz dzielony kod z Rozdziału 6 i zbudować z niego bibliotekę dzieloną do użytku w naszych programach. Pierwsza rzecz którą moglibyśmy zrobić jest zasemblowanie go jak zwykle:

```
as write-record.s -o write-record.o
as read-record.s -o read-record.o
```

Teraz, zamiast zlinkować je w program, chcemy zlinkować je w bibliotekę dzieloną. To zmienia naszą komendę linkera do takiej:

```
ld -shared write-record.o read-record.o -o librecord.so
```

To wiąże obydwa pliki razem w bibliotekę dzieloną nazwaną **librecord.so**. Ten plik może teraz być użyty do wielu programów. Jeśli chcemy uaktualnić funkcje zawarte w niej, możemy właśnie uaktualnić ten jeden plik i nie martwić się o to które programy go używają.
Zobaczmy jak moglibyśmy zlinkować coś z tą biblioteką. Aby zlinkować program **write-records**, moglibyśmy zrobić co następuje:

```
as write-record.s -o write-records
ld -L . -dynamic-linker /lib/ld-linux.so.2 -o write-records -lrecord write-records.o
```

W tej komendzie, **-L .** mówi linkerowi żeby szukał bibliotek w bieżącym katalogu (zwykle przeszukuje tylko katalog **/lib**, katalog **/usr/lib**, i kilka innych). Jak widzieliśmy, opcja **-dynamic-linker /lib/ld-linux.so.2** określa linker dynamiczny. Opcja **-lrscord** mówi linkerowi aby szukał funkcji w pliku nazwanym **librecord.so**.
Teraz program **write-records** jest zbudowany, ale się nie uruchomi. Jeśli spróbujemy uruchomić, dostaniemy następujący błąd:

```
./write-records: error while loading shared libraries:
librecord.so: cannot open shared object file: No such file or directory
```

Jest tak ponieważ, domyślnie, linker dynamiczny szuka bibliotek tylko w **/lib**, **/usr/lib**, i w katalogach zapisanych w **/etc/ld.so.conf**. W przypadku uruchamiania programu, powinieneś albo przenieść bibliotekę do jednego z tych katalogów, lub wykonać następującą komendę:

```
LD_LIBRARY_PATH=.
export LD_LIBRARY_PATH
```

Alternatywnie, jeśli daje to błąd, zamiast tego zrób:

```
setenv LD_LIBRARY_PATH .
```

Teraz, możesz uruchomić **write-records** normalnie przez napisanie **./write-records**. Ustawienie **LD_LIBRARY_PATH** mówi linkerowi aby dodał ścieżki które mu podasz do biblioteki ścieżek szukania bibliotek dynamicznych.
Więcej informacji o linkowaniu dynamicznym, zobacz następujące źródła w Internecie:
- Strona man dla **ld.so** zawiera wiele informacji o tym jak działa linker dynamiczny Linuksa.
- http://www.benyossef.com/presentations/dlink/ jest wspaniałą prezentacją linkowania dynamicznego w Linuksie.
- http://www.linuxjournal.com/article.php?sid=1059 i http://www.linuxjournal.com/article.php?sid=1060 dają dobre wprowadzenie do formatu pliku ELF, więcej szczegółów dostępne na http://www.cs.ucdavis.edu/~haungs/paper/node10.html
- http://www.iecc.com/linker/linker10.html zawiera wspaniały opis jak działa linkowanie dynamiczne z plikami ELF.

Przegląd

Znajomość Koncepcji
- Jakie są plusy i minusy bibliotek dzielonych?
- Dana jest biblioteka nazwana 'foo', jaka mogłaby być nazwa pliku biblioteki?
- Co robi komenda **ldd**?
- Powiedzmy, że mamy pliki **foo.o** i **bar.o**, i chciałbyś połączyć je razem, i dynamicznie zlinkować je z biblioteką 'kramer'.

Jaka mogłaby być komenda linkująca do wygenerowania finalnego programu?
- Po co jest **typedef**?
- Po co jest **struct**?
- Jaka jest różnica między elementami danych *int* i *int* *? Jak mógłbyś różnie udostępnić je w swoim programie?
- Jeśli miałbyś plik obiektowy nazwany **foo.o**, jaka mogłaby być komenda do utworzenia biblioteki dzielonej nazwanej 'bar'?
- Jaki jest cel LD_LIBRARY_PATH?

Użycie Koncepcji

- Przepisz jeden lub więcej programów z poprzednich rozdziałów aby wypisywać ich wyniki na ekranie używając raczej **printf** niż zwracanie wyniku jako kod statusu wyjścia. Także, zrób kod statusu wyjścia 0.
- Użyj funkcji **factorial** którą zrobiłeś w Podrozdziale nazwanym *Funkcje Rekursywne* w Rozdziale 4 do utworzenia biblioteki dzielonej. Wtedy przepisz program główny tak aby połączyć go dynamicznie z biblioteką.
- Przepisz powyższy program tak żeby połączyć go z biblioteką 'c'. Użyj funkcji bibliotecznej 'c' **printf** do wyświetlenia wyniku wywołania **factorial**
- Przepisz program **toupper** tak żeby używał funkcji biblioteki **c** dla plików raczej niż wywołań systemowych.

Idąc Dalej

- Zrób listę wszystkich zmiennych środowiskowych używanych przez linker dynamiczny GNU/Linux.
- Przejrzyj różne typy formatów plików wykonywalnych używanych obecnie i w historii informatyki. Omów silne i słabe strony każdej z nich.
- Jaki rodzaj programowania cię interesuje (grafika, bazy danych, nauka, etc.)? Znajdź bibliotekę do pracy w tym obszarze, i napisz program który używa podstaw tej biblioteki.
- Sprawdź użycie **LD_PRELOAD**. Po co jest używane? Spróbuj zbudować bibliotekę dzieloną która zawiera funkcję **exit**, i ma napisaną wiadomość do STDERR przed wychodzeniem. Użyj **LD_PRELOAD** i uruchom różne programy z nim. Jakie są wyniki?

Rozdział 9. Średnio Zaawansowane Zagadnienia Pamięci

Jak Komputer Widzi Pamięć

Przeglądnijmy jak działa pamięć wewnątrz komputera. Możesz także przeczytać ponownie Rozdział 2.
Komputer patrzy na pamięć jak na długą sekwencję numerowanych lokalizacji. Sekwencję *milionów* numerowanych lokalizacji. Wszystko jest umieszczane w tych lokalizacjach. Twoje programy są tam umieszczane, twoje dane tam są umieszczane , wszystko. Każda lokalizacja wygląda jak każda inna. Lokalizacje przechowujące twój program są takie same jak przechowujące twoje dane. W rzeczywistości, komputer nie ma pojęcia które są które, oprócz tego, że plik wykonawczy powie mu gdzie zapoczątkować wykonywanie.
Te lokalizacje są zwane bajtami. Komputer może połączyć do czterech z nich razem w pojedyncze słowo. Normalnie dane numeryczne są przetwarzane po słowie za jednym razem. Jak wspomnieliśmy, instrukcje są także przechowywane w tej samej pamięci. Każda instrukcja jest różnej długości. Większość instrukcji zabiera jedną lub dwie lokalizacje dla samej instrukcji, i potem lokalizacje dla argumentów instrukcji. Na przykład, instrukcja

movl data_item(,%edi,4), %ebx

zabiera do 7 lokalizacji. Pierwsze dwie przechowują instrukcję, trzecia mówi których rejestrów użyć, i następne cztery przechowują lokalizacje **data_item**. W pamięci, instrukcje wyglądają jak wszystkie inne liczby, i instrukcje same mogą być przenoszone do i z rejestrów właśnie jak liczby, dlatego, że tym są.

Ten rozdział skupia się na szczegółach pamięci komputera. Na początek przejrzyjmy kilka podstawowych terminów których będziemy używać w tym rozdziale:

Bajt
Jest to rozmiar lokalizacji przechowywania. Na procesorach x86, bajt może przechowywać liczby pomiędzy 0 i 255.

Słowo
To jest rozmiar zwykłego rejestru. Na procesorach x86, słowo jest cztery bajty długie. Większość operacji komputera przeprowadza się po jednym słowie na raz.

Adres
Adres jest liczbą która odnosi się do bajta w pamięci. Na przykład, pierwszy bajt w komputerze ma adres 0, drugi ma adres 1, i tak dalej. Każdy kawałek danych w komputerze poza rejestrami ma adres. Adres który spaja kilka bajtów jest taki sam jak adres pierwszego bajta.
Normalnie, nawet nie wpisujemy numerycznego adresu czegokolwiek, ale pozwalamy asemblerowi zrobić to dla nas. Kiedy używamy etykiet w kodzie, symbol użyty w etykiecie będzie równoważnikiem adresu który etykietuje. Asembler wtedy będzie wymieniał ten symbol z jego adresem gdziekolwiek używasz go w swoim programie. Na przykład, masz następujący kod:

```
.section .data
my_data:
.long 2, 3, 4
```

Teraz, za każdym razem w programie gdzie **my_data** jest użyta, będzie ona wymieniona przez adres pierwszej wartości dyrektywy **.long**

Wskaźnik
Wskaźnik jest rejestrem lub słowem pamięci którego wartością jest adres. W naszych programach używamy **%ebp** jako wskaźnik do bieżącej ramki stosu. Wszystkie adresowania wskaźnika bazowego dotyczą wskaźników. Programowanie używa mnóstwa wskaźników, więc jest to ważna koncepcja do zrozumienia.

Plan Pamięci Programu Linuksowego

Kiedy twój program jest ładowany do pamięci, każda **.section** jest ładowana do swojego własnego rejonu pamięci. Cały kod i dane deklarowane w każdej sekcji są zebrane razem, nawet jeśli były odseparowane w twoim kodzie źródłowym. Rzeczywiste instrukcje (sekcja **.text**) są ładowane do adresu 0x08048000 (liczby rozpoczynające się od **0x** są heksadecymalnymi, co będzie dyskutowane w Rozdziale 10). Sekcja **.data** jest ładowana natychmiast po tym, a następnie sekcja **.bss**.
Ostatni bajt który może być zaadresowany w Linuksie jest zlokalizowany pod 0xbfffffff. Linux rozpoczyna stos tutaj i wzrasta on w dół w kierunku innych sekcji. Pomiędzy nimi jest ogromna szczelina. Inicjalizacyjny plan stosu jest następujący: na dnie stosu (dno stosu jest najwyższym adresem pamięci - zobacz Rozdział 4), jest słowo pamięci które jest zerem. Po tym przychodzi zakończona znakiem null nazwa programu używająca znaków ASCII. Po nazwie programu przychodzą zmienne środowiskowe programu (te nie są ważne dla nas w tej książce). Wtedy przychodzą argumenty programowe wiersza poleceń. Są to wartości które użytkownik wpisał w wierszu poleceń dla uruchomienia programu. Kiedy uruchamiamy **as**, na przykład, dajemy mu kilka argumentów - **as**, **sourcefile.s**, **-o**, i **objectfile.o**. Po tym, mamy liczbę argumentów które były użyte. Kiedy program się zaczyna, to jest to na co wskaźnik stosu, **%esp**, wskazuje. Dalsze odkładania na stos przesuwa **%esp** w dół pamięci. Na przykład, instrukcja

```
pushl %eax
```

jest równoważne temu

movl %eax, (%esp)
subl $4, %esp

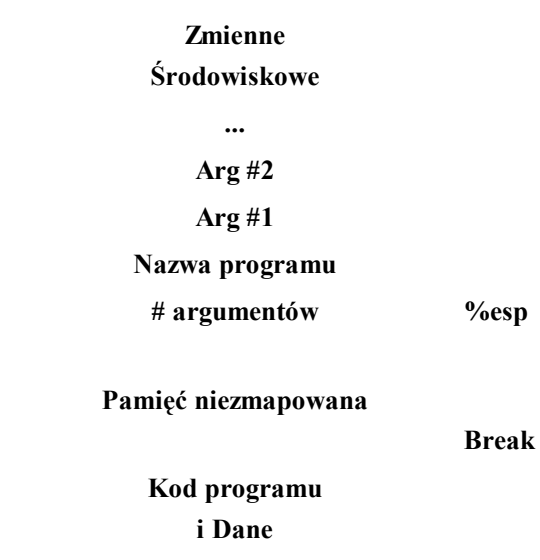
Podobnie, instrukcja

popl %eax

jest tym samym co

movl (%esp), %eax
addl $4, %esp

Rejon danych twojego programu zaczyna się na dnie pamięci i idzie w górę. Stos zaczyna się na wierzchołku pamięci, i przesuwa się w dół za każdym odłożeniem. Ta środkowa część pomiędzy stosem a sekcjami danych twojego programu jest niedostępną pamięcią - nie masz pozwolenia na dostęp do niej aż dotąd jak powiesz kernelowi, że potrzebujesz tego. Jeśli spróbujesz, otrzymasz błąd (wiadomością o błędzie jest zwykle "segmentation fault"). To samo się zdarzy jeśli spróbujesz dostać się do danych przed początkiem twojego programu, 0x0804800. Ostatni dostępny adres pamięci do twojego programu jest zwany *system break* (także nazywany *current break* lub prosto *break*).



Plan Pamięci Programu Linuksowego na początku

Każdy Adres Pamięci to Kłamstwo

Więc, dlaczego komputer nie pozwala na dostęp do pamięci w obszarze "break"? Aby odpowiedzieć na to pytanie, będziemy musieli pogrzebać głęboko jak rzeczywiście twój komputer przetwarza pamięć.
Mógłbyś być ciekaw, ponieważ każdy program jest ładowany do tego samego miejsca w pamięci, czy nie zachodzą one na siebie, lub nadpisują się wzajemnie? Wygląda, że tak. Jednakże, jako piszący programy, masz tylko dostęp do *pamięci wirtualnej*.

Pamięć fizyczna odnosi się do rzeczywistych chipów RAM w środku twojego komputera i do tego co one zawierają. Jest to zwykle pomiędzy 16 a 512 Megabajtów we współczesnych komputerach. Jeśli mówimy o *adresie pamięci fizycznej*, mówimy o tym gdzie dokładnie jest zlokalizowana cząstka pamięci w tych chipach. Pamięć wirtualna jest sposobem w jaki *twój program* myśli o pamięci. Przed załadowaniem programu, Linux szuka pustego obszaru pamięci fizycznej na tyle wielkiego aby zmieścił twój program, i wtedy mówi procesorowi żeby udawał, że ta pamięć jest obecnie pod adresem 0x08048000 do załadowania w nią twojego programu. Mylące? Pozwól mi wytłumaczyć dalej.

Każdy program ma własną piaskownicę do zabawy w niej. Każdy program uruchomiony na twoim komputerze myśli, że był załadowany pod adresem pamięci 0x08048000, i jego stos zaczyna się w 0xbfffffff. Kiedy Linux ładuje program, szuka sekcji nieużywanej pamięci, i wtedy mówi procesorowi aby użył tej sekcji pamięci jako zaadresowanej pod 0x08048000 dla tego programu. Ten adres który program wierzy, że używa jest zwany adresem wirtualnym, podczas gdy adres rzeczywisty na chipie do którego się on odnosi jest zwany adresem fizycznym. Proces przypisywania adresów wirtualnych do adresów fizycznych jest nazywany *mapowaniem* (*mapping*).

Wcześniej mówiliśmy o niedostępności pamięci pomiędzy **.bss** a stosem, ale nie mówiliśmy o tym dlaczego tak jest. Powód jest taki, że ten rejon adresów pamięci wirtualnej nie był zmapowany na adresy pamięci fizycznej. Proces mapowania zabiera znacząco czas i miejsce, więc jeśli każdy możliwy adres wirtualny każdego możliwego programu byłby mapowany, mógłbyś nie mieć dość pamięci fizycznej nawet do uruchomienia pojedynczego programu. Więc, "break" jest początkiem obszaru który zawiera niezmapowaną pamięć. Ze stosem, jednakże, Linux będzie automatycznie mapował pamięć która jest dostępna ze stosowych odkładań.

Oczywiście, jest to bardzo uproszczona wizja pamięci wirtualnej. Pełna koncepcja jest dużo bardziej zaawansowana. Na przykład, pamięć wirtualna może być mapowana do czegoś więcej niż tylko pamięć fizyczna, może być mapowana na dysk również. Partycja swap w Linuksie pozwala systemowi pamięci wirtualnej Linuksa na mapowanie pamięci nie tylko na fizyczny RAM, ale także na bloki dysku również. Na przykład, powiedzmy masz tylko 16 Megabajtów pamięci fizycznej. Powiedzmy także, że 8 Megabajtów jest używane przez Linux i kilka podstawowych aplikacji, a ty chcesz uruchomić program który wymaga 20 Megabajtów pamięci. Czy możesz? Odpowiedź brzmi tak, ale tylko jeśli masz ustawioną partycję swap. Co się dzieje, po tym jak całe pozostałe 8 Megabajtów pamięci fizycznej zostało zmapowane na pamięć wirtualną, Linux zaczyna mapować części pamięci wirtualnej twojej aplikacji na bloki dysku. Więc, jeśli dostajesz się do lokalizacji "pamięć" w twoim programie, ta lokalizacja może nie być w ogóle rzeczywiście w pamięci, ale na dysku. Jako programista nie poznasz różnicy, ponieważ to wszystko jest przeprowadzane poza sceną przez Linux.

Teraz, procesory x86 nie mogą uruchamiać instrukcji bezpośrednio z dysku, ani nie mogą mieć dostępu do danych bezpośrednio z dysku. To wymaga pomocy systemu operacyjnego. Kiedy próbujesz mieć dostęp do pamięci która jest mapowana na dysk, procesor spostrzega, że nie może obsłużyć bezpośrednio twojego żądania pamięci. Wtedy prosi Linuksa o wkroczenie. Linux zauważa, że ta pamięć jest rzeczywiście na dysku. Dlatego, przesuwa on część danych które są bieżąco w pamięci na dysk aby zrobić miejsce, i wtedy przesuwa tą pamięć będącą do udostępnienia z dysku do pamięci fizycznej. Wtedy reguluje tablice przeszukiwania pamięci wirtualnej-do-fizycznej procesora tak żeby mógł on znaleźć tę pamięć w nowej lokalizacji. Ostatecznie, Linux zwraca kontrolę do programu i restartuje go na instrukcji która próbowała uzyskać dostęp do danych w pierwszym miejscu. Ta instrukcja może teraz być wykonana z sukcesem, ponieważ pamięć jest teraz w fizycznym RAM.

Tutaj jest przegląd sposobów przetwarzania dostępu do pamięci pod Linuksem:

- Program próbuje załadować pamięć z adresu wirtualnego.
- Procesor, używając tablic wspieranych przez Linuksa, przekształca adres pamięci wirtualnej w adres pamięci fizycznej w locie.
- Jeśli procesor nie ma adresu fizycznego dla adresu pamięci, przesyła żądanie do Linuksa aby załadował go.
- Linux testuje adres. Jeśli jest on zmapowany na lokalizację dyskową, kontynuuje do następnego kroku. W przeciwnym razie, kończy program z błędem "segmentation fault".
- Jeśli nie ma wystarczająco dużo miejsca do załadowania pamięci z dysku, Linux przesunie inną część programu lub inny program na dysk aby zrobić miejsce.
- Linux wtedy przesuwa dane do wolnego adresu pamięci fizycznej.
- Linux uaktualnia tablice mapowania pamięci wirtualnej-do-fizycznej procesora celem odzwierciedlenia zmian.
- Linux oddaje kontrolę do programu, zmuszając go do ponowienia instrukcji która spowodowała wystąpienie tego procesu.
- Procesor może teraz przetworzyć tę instrukcję używając nowo załadowanej pamięci i tablic translacji.

Jest wiele do zrobienia dla systemu operacyjnego, ale to daje użytkownikowi i programiście wielką dowolność kiedy przychodzi do zarządzania pamięcią.
Teraz, w celu uczynienia procesu bardziej efektywnym, pamięć jest podzielona na grupy zwane stronami. Kiedy uruchamiamy Linuksa na procesorze x86, strona ma 4096 bajtów pamięci. Wszystkie mapowania pamięci są robione po stronie na raz. Przypisywanie pamięci fizycznej, swapowanie, mapowanie, etc. są wszystkie robione stronom pamięci zamiast adresom indywidualnym pamięci. To oznacza dla ciebie jako programisty, że cokolwiek programujesz, powinieneś utrzymywać większość dostępów do pamięci w ramach tego samego podstawowego obszaru pamięci, tak będziesz potrzebował tylko stronę lub dwie za jednym razem. W przeciwnym razie, Linux może musieć przenosić strony na dysk i z dysku aby zadowolić twoje potrzeby pamięci. Dostęp do dysku jest powolny, więc to może rzeczywiście spowolnić twój program.
Czasami tak wiele programów może być załadowanych, że nie starcza pamięci fizycznej dla nich. Kończą one spędzając więcej czasu tylko swapując pamięć na dysk i z dysku niż wykonując rzeczywiste procesy. To prowadzi do kondycji zwanej *swap death* która prowadzi do tego, że twój system nie odpowiada i staje się nieproduktywnym. Jest to zwykle naprawialne jeśli zaczniesz kończyć swoje pamięciożerne programy, ale jest to bolesne.

Resident Set Size: Wielkość pamięci którą twój program bieżąco posiada w pamięci fizycznej jest nazwana jego "resident set size", i może być wyświetlany przez użycie programu **top**. "Resident set size" jest wypisany w kolumnie z etykietą "RSS".

Osiąganie Większej Pamięci

Wiemy teraz, że Linux mapuje całą naszą pamięć wirtualną na pamięć fizyczną lub swap. Jeśli próbujesz dostać się do części pamięci wirtualnej która nie jest jeszcze zmapowana, powoduje to błąd znany jako "segmentation fault", który zakończy twój program. Punkt "break" programu, jeśli pamiętasz, jest ostatnim ważnym adresem którego możesz użyć. Teraz, jest wszystko w porządku jeśli wiesz wcześniej jak wiele miejsca będziesz potrzebował. Możesz właśnie dodać całą pamięć której potrzebujesz do twojej sekcji **.data** lub **.bss**, i wszystko będzie tam. Jednakże, powiedzmy, że nie wiesz jak wiele pamięci będziesz potrzebował. Na przykład, z edytorem tekstu, nie wiesz jak duży będzie plik osobowy. Mógłbyś próbować znaleźć maksymalny rozmiar pliku, i powiedzieć użytkownikom, że nie mogą go przekraczać, ale jest to strata jeśli plik jest mały. Dlatego Linux ma możliwości do przeniesienia punktu "break" dla zaspokojenia potrzeb pamięci aplikacji.
Jeśli potrzebujesz więcej pamięci, możesz powiedzieć Linuksowi gdzie chcesz aby nowy punkt "break" był, i Linux zmapuje całą pamięć której potrzebujesz pomiędzy bieżącym i nowym punktem "break", i wtedy przeniesie punkt "break" do miejsca które wybrałeś. Ta pamięć jest teraz dostępna dla twojego programu do użycia. Sposób w jaki mówimy Linuksowi aby przeniósł punkt "break" jest poprzez wywołanie systemowe **brk**. Wywołanie systemowe **brk** jest wywołaniem o numerze 45 (który będzie w **%eax**). **%ebx** powinien być załadowany żądanym punktem przerwania. Wtedy wywołujesz **int $0x80** dla zasygnalizowania Linuksowi aby wykonał swoją pracę. Po mapowaniu w twojej pamięci, Linux zwróci nowy punkt przerwania w **%eax**. Nowy punkt przerwania może być większy niż ten o który prosiłeś, ponieważ Linux zaokrągla go w górę do najbliższej strony. Jeśli nie ma wystarczającej pamięci fizycznej lub swap aby wypełnić twoje żądanie, Linux zwróci zero w **%eax**. Także, jeśli wywołasz **brk** z zerem w **%ebx**, po prostu zwróci ostatni użyteczny adres pamięci.
Problemem z tą metodą jest utrzymywanie śladu pamięci której żądamy. Powiedzmy potrzebuję przenieść "break" żeby zrobić miejsce na załadowanie pliku, i wtedy potrzebuję znowu przenieść "break" aby załadować następny plik. Powiedzmy potem wracam do pierwszego pliku. Masz teraz wielką dziurę w pamięci która jest zmapowana, ale której nie używasz. Jeśli kontynuujesz przenoszenie "break" w ten sposób dla każdego załadowanego pliku, możesz łatwo wyczerpać pamięć. Więc, to co jest potrzebne to *zarządca pamięci*.
Zarządca pamięci jest zestawem procedur który opiekuje się brudną robotą dostarczania twojemu programowi pamięci dla ciebie. Większość zarządców pamięci posiada dwie podstawowe funkcje - **allocate** i **deallocate**. Kiedykolwiek

potrzebujesz określonej ilości pamięci, możesz po prostu powiedzieć **allocate** jak wiele potrzebujesz, i ona zwróci adres do tej pamięci. Kiedy skończyłeś, mówisz **deallocate**, że skończyłeś. **allocate** będzie wtedy zdolna ponownie użyć tej pamięci. Ten wzór zarządzania pamięcią jest zwany *dynamiczną alokacją pamięci*. To minimalizuje liczbę "dziur" w twojej pamięci, potwierdzając, że robisz najlepszy użytek z niej jaki możesz. Pula pamięci używana przez zarządców pamięci jest popularnie referowana jako *heap*
Sposób w jaki zarządcy pamięci pracują jest taki, że śledzą oni gdzie jest systemowy "break" a gdzie jest pamięć którą alokowałeś. Zaznaczają każdy blok pamięci w "heap" jako użytą lub nieużytą. Kiedy żądasz pamięci, zarządca pamięci sprawdza czy są jakieś nieużyte bloki o odpowiednim rozmiarze. Jeśli nie, wywołuje wywołanie systemowe **brk** aby prosić o więcej pamięci. Kiedy uwalniasz pamięć zaznacza bloki jako nieużywane więc przyszłe żądania mogą je brać pod uwagę. W następnym podrozdziale zobaczymy budowanie naszego własnego zarządcy pamięci.

Prosty Zarządca Pamięci

Tutaj pokażę prostego zarządcę pamięci. Jest on bardzo prymitywny ale pokazuje bardzo dobrze zasady. Jak zwykle, najpierw podam program do wglądu. Później nastąpi pogłębione wytłumaczenie. Wygląda na długi, ale w większości to komentarze.

```
#CEL: Program do zarządzania użyciem pamięci - alokuje i dealokuje pamięć na żądanie
#
#UWAGI: Programy używające tych procedur będą prosić o określonego rozmiaru pamięć.
#Rzeczywiście używamy więcej niż ten rozmiar, ale na początku umieścimy tę, przed wskaźnikiem który oddamy.
#Dodajemy pole rozmiaru i znacznik OSIĄGALNA/NIEOSIĄGALNA. Więc, pamięć wygląda tak
#
############################################################
##Znacznik Osiągalna#Rozmiar pamięci#Rzeczywiste lokalizacje pamięci#
############################################################
#######################################^--Wskaźnik zwrócony
#########################################wskazuje tutaj
#Wskaźnik który zwracamy wskazuje tylko na rzeczywiste lokalizacje żądane dla ułatwienia programowi wywołującemu.
#Pozwala nam także na zmianę naszej struktury bez zmieniania czegokolwiek przez wywołujący program

.section .data

######ZMIENNE GLOBALNE############

#To wskazuje na początek pamięci którą zarządzamy
heap_begin:
.long 0

#To wskazuje na pierwszą lokalizację za zarządzaną
current_break:
.long 0


########INFORMACJE STRUKTURALNE#########
#rozmiar przestrzeni dla nagłówka rejonu pamięci
.equ HEADER_SIZE, 8
```

```
#Lokalizacja flagi "available" w nagłówku
.equ HDR_AVAIL_OFFSET, 0
#Lokalizacja pola rozmiaru w nagłówku
.equ HDR_SIZE_OFFSET, 4


#########STAŁE###################
.equ UNAVAILABLE, 0 #To jest liczba której będziemy używać do zaznaczenia przestrzeni która była wydana
.equ AVAILABLE, 1 #To jest liczba której użyjemy do zaznaczenia, że przestrzeń została oddana i jest dostępna do dania
jej
.equ SYS_BRK, 45 #Numer wywołania systemowego dla wywołania systemowego "break"
.equ LINUX_SYSCALL, 0x80 #ułatwia czytanie wywołań systemowych

.section .text


###########FUNKCJE#########################

##allocate_init##
#CEL: wywołaj tę funkcję dla zainicjalizowania funkcji (specjalnie, ustawia "heap_begin" i "current_break").
#Nie ma parametrów i wartości zwracanej.

.globl allocate_init
.type allocate_init, @function
allocate_init:
pushl %ebp #standardowe działania funkcyjne
movl %esp, %ebp

#Jeśli wywołanie systemowe "brk" jest wywoływane z 0 w %ebx, zwraca ostatni ważny użyteczny adres
movl $SYS_BRK, %eax #wyszukuje gdzie jest "break"
movl $0, %ebx
int $LINUX_SYSCALL

incl %eax #%eax teraz ma ostatni ważny adres, a my chcemy lokalizacji pamięci za nim

movl %eax, current_break #przechowuje "current break"

movl %eax, heap_begin #przechowuje "current break" jako nasz pierwszy adres.
#To spowoduje, że funkcja "allocate" dostanie
#więcej pamięci od Linuksa podczas pierwszego uruchomienia

movl %ebp, %esp #wyjście z funkcji
popl %ebp
ret
##############KONIEC FUNKCJI####################


##allocate##
#CEL: Funkcja jest użyta do pobrania sekcji pamięci.
```

#Sprawdza ona czy są jakieś wolne bloki, i jeśli nie, prosi Linuksa o nową sekcję.
#
#PARAMETRY: Funkcja ma jeden parametr - rozmiar bloku pamięci który chcemy alokować.
#
#ZWRACANA WARTOŚĆ: Funkcja zwraca adres alokowanej pamięci w %eax.
#Jeśli nie ma dostępnej pamięci, zwróci 0 w %eax.

#############PROCESOWANIE###################
#Użyte zmienne: # %ecx - przechowuje rozmiar żądanej pamięci (pierwszy/jedyny parametr)
%eax - bieżący rejon pamięci będący testowanym
%ebx - bieżąca pozycja "break"
%edx - rozmiar bieżącego rejonu pamięci
#
#Przeglądamy każdy rejon pamięci zaczynając od "heap_begin". Patrzymy na rozmiar każdego i czy jest alokowany.
#Jeśli jest wystarczająco duży dla żądanego rozmiaru, i jest dostępny, bierzemy ten.
#Jeśli nie znajdzie się rejonu wystarczająco dużego, prosimy Linuksa o więcej pamięci.
#W ten sposób, "current_break" przesuwa się w górę

.globl allocate
.type allocate, @function
.equ ST_MEM_SIZE, 8 #pozycja stosu rozmiaru pamięci do alokowania
allocate:
pushl %ebp #działanie standardowe funkcji
movl %esp, %ebp

movl ST_MEM_SIZE(%ebp), %ecx #%ecx przechowa rozmiar który szukamy (który jest pierwszym i jedynym parametrem)

movl heap_begin, %eax #%eax przechowa bieżącą lokalizację przeszukiwania

movl current_break, %ebx #%ebx przechowa bieżący "break"

alloc_loop_begin: #tutaj iterujemy po każdym rejonie pamięci

cmpl %ebx, %eax #potrzeba więcej pamięci jeśli są równe
je move_break

#bierze rozmiar tej pamięci
movl HDR_SIZE_OFFSET(%eax), %edx

#Jeśli ta przestrzeń jest niedostępna, idzie do następnej
cmpl $UNAVAILABLE, HDR_SIZE_OFFSET(%eax)
je next_location #następna

cmpl %edx, %ecx #Jeśli przestrzeń jest dostępna, porównaj rozmiar do potrzebnego rozmiaru.
jle allocate_here #Jeśli jest wystarczająco duży, przejdź do "allocate_here"

next_location:

addl $HEADER_SIZE, %eax #Całkowity rozmiar rejonu pamięci jest sumą żądanego rozmiaru (bieżąco umieszczonego w %edx),
addl %edx, %eax #plus następne 8 bajtów na nagłówek (4 na flagę AVAILABLE/UNAVAILABLE, i 4 na rozmiar rejonu).
#Więc, dodając %edx i $8 do %eax da adres następnego rejonu pamięci

jmp alloc_loop_begin #idź do przejrzenia następnej lokalizacji

allocate_here: #jeśli zrobilibyśmy to tutaj, znaczyłoby to, że nagłówek rejonu do alokacji jest w %eax

#zaznacz przestrzeń jako niedostępna
movl $UNAVAILABLE, HDR_AVAIL_OFFSET(%eax)
addl $HEADER_SIZE, %eax #przesuń %eax poza nagłówek do użytecznej pamięci (ponieważ to jest to co zwracamy)

movl %ebp, %esp #powrót z funkcji
popl %ebp
ret

move_break: #jeśli zrobilibyśmy to tutaj, to znaczyłoby,
#że wyczerpaliśmy całą adresowalną pamięć i musimy poprosić o więcej.
#%ebx przechowuje bieżący końcowy punkt danych, a %ecx przechowuje jego rozmiar

addl $HEADER_SIZE, %ebx #potrzebujemy zwiększyć %ebx do miejsca gdzie chcemy mieć koniec pamięci,
#więc dodajemy przestrzeń na strukturę nagłówków
addl %ecx, %ebx #dodaje przestrzeń do "break" dla żądanych danych

#teraz jest czas na proszenie Linuksa o więcej pamięci

pushl %eax #zachowuje potrzebne rejestry
pushl %ecx
pushl %ebx

movl $SYS_BRK, %eax #zresetowanie "break" (%ebx ma żądany punkt "break")
int $LINUX_SYSCALL
#w normalnych warunkach, to powinno zwrócić nowy "break" w %eax, #który będzie albo 0 jeśli porażka, lub będzie równy lub większy niż prosiliśmy.
#Nie zwracamy uwagi w tym programie gdzie rzeczywiście ustawia "break",
#więc tak długo jak %eax nie jest 0, nie zwracamy uwagi co to jest

cmpl $0, %eax #sprawdza warunki błędu
je error

popl %ebx #odnawia zachowane rejestry
popl %ecx
popl %eax

#ustawia tę pamięć jako niedostępną, ponieważ zamierzamy ją odrzucić
movl $UNAVAILABLE, HDR_AVAIL_OFFSET(%eax)

```
#ustawia rozmiar pamięci
movl %ecx, HDR_SIZE_OFFSET(%eax)

#przesuwa %eax do obecnego początku użytecznej pamięci.
#%eax teraz przechowuje zwracaną wartość
addl $HEADER_SIZE, %eax

movl %ebx, current_break #zachowuje nowy "break"

movl %ebp, %esp #powrót funkcji
popl %ebp
ret

error:
movl $0, %eax #przy błędzie, zwraca zero
movl %ebp,%esp
popl %ebp
ret
############KONIEC FUNKCJI##################


##deallocate##
#CEL: Celem tej funkcji jest oddanie z powrotem rejonu pamięci do puli po skończeniu używania go
#
#PARAMETRY: Jedynym parametrem jest adres pamięci którą chcemy zwrócić do puli pamięci.
#
#ZWRACANA WARTOŚĆ: Nie ma zwracanej wartości
#
#PROCESOWANIE: Jeśli pamiętasz, podajemy programowi początek pamięci której można użyć,
#to jest 8 lokalizacji przechowywania po obecnym początku rejonu pamięci.
#Wszystko co musimy zrobić to iść w tył 8 lokalizacji i zaznaczyć tę pamięć jako dostępną,
#więc tak funkcja "allocate" wie, że może jej użyć.
.globl deallocate
.type deallocate, @function
#pozycja stosu rejonu pamięci do uwolnienia
.equ ST_MEMORY_SEG, 4
deallocate:
#ponieważ funkcja jest tak prosta, nie potrzebujemy niczego z tych śmiesznych działań funkcji
#dostać adres pamięci do uwolnienia (normalnie jest to 8(%ebp), ale ponieważ nie odłożyliśmy %ebp
#lub przesunęliśmy %esp do %ebp, możemy zrobić 4(%esp))
movl ST_MEMORY_SEG(%esp), %eax

#uzyskanie wskaźnika do rzeczywistego początku pamięci
subl $HEADER_SIZE, %eax

#zaznaczenie jej jako dostępnej
movl $AVAILABLE, HDR_AVAIL_OFFSET(%eax)

#powrót
```

ret
############KONIEC FUNKCJI#######################

Pierwszą rzeczą do zauważenia jest to, że nie ma symbolu **_start**. Powód jest taki, że to tylko zestaw funkcji. Zarządca pamięci sam z siebie nie jest pełnym programem - nie robi niczego. Jest po prostu pomocą do użytku przez inne programy. Dla zasemblowania tego programu, wykonaj co następuje:

as alloc.s -o alloc.o

Dobrze, teraz popatrzmy w kod.

Zmienne i Stałe
Na początku programu, mamy ustawione dwie lokalizacje:

heap_begin:
.long 0

current_break:
.long 0

Pamiętaj, że sekcja pamięci będąca zarządzaną jest powszechnie referowana jako *heap*. Kiedy asemblujemy program, nie mamy pojęcia gdzie początek "heap" jest, ani gdzie jest bieżący "break". Dlatego, rezerwujemy dla nich adresy, ale wypełniamy przez 0 na razie.
Następnie mamy zestaw stałych do zdefiniowania struktury "heap". Sposób w jaki ten zarządca pamięci pracuje jest, że przed każdym rejonem pamięci alokowanej, będziemy mieli krótki rekord opisujący tę pamięć. Ten rekord ma słowo zarezerwowane na flagę dostępności i słowo na rozmiar rejonu. Rzeczywista alokowana pamięć następuje zaraz po tym rekordzie. Flaga dostępu jest użyta do zaznaczenia czy ten rejon jest dostępny do alokacji, lub jest obecnie w użyciu. Pole rozmiaru pozwala nam poznać zarówno czy ten rejon jest wystarczająco duży dla żądania alokacyjnego, jak i lokalizację następnego rejonu pamięci. Następujące stałe opisują ten rekord:

.equ HEADER_SIZE, 8
.equ HDR_AVAIL_OFFSET, 0
.equ HDR_SIZE_OFFSET, 4

Mówią one, że nagłówek ma w całości 8 bajtów, flaga dostępu jest przesunięta 0 bajtów od początku, i pole rozmiaru jest przesunięte 4 bajty od początku. Jeśli jesteśmy uważni i zawsze zastosujemy te stałe, wtedy uchronimy się od wykonywania zbyt wielkiej pracy kiedy później zdecydujemy się dodać więcej informacji do nagłówka.
Wartości których będziemy używać dla naszego pola **available** są albo 0 dla nieosiągalnych, albo 1 dla osiągalnych. Aby ułatwić odczytywanie, mamy następujące definicje:

.equ UNAVAILABLE, 0
.equ AVAILABLE, 1
Ostatecznie, mamy nasze definicje wywołań systemowych Linuksa:

.equ BRK, 45
.equ LINUX_SYSCALL, 0x80

Funkcja "allocate_init"
Jest to prosta funkcja. Wszystko co robi to ustawia zmienne **heap_begin** i **current_break** dyskutowanych wcześniej. Więc, jeśli pamiętasz dyskusję wcześniejszą, bieżący "break" może być znaleziony przez użycie wywołania systemowego **brk**. Więc, początek funkcji wygląda tak:

```
pushl %ebp
movl %esp, %ebp

movl $SYS_BRK, %eax
movl $0, %ebx
int $LINUX_SYSCALL
```

Jakkolwiek, po **int $LINUX_SYSCALL**, **%eax** przechowuje ostatni ważny adres. W rzeczywistości chcemy pierwszego nieważnego adresu zamiast ostatniego ważnego adresu, więc inkrementujemy **%eax**. Wtedy przenosimy tę wartość do lokalizacji **heap_begin** i **current_break**. Potem opuszczamy funkcję. Kod wygląda tak:

```
incl %eax
movl %eax, current_break
movl %eax, heap_begin
movl %ebp, %esp
popl %ebp
ret
```

"heap" zawiera pamięć pomiędzy **heap_begin** a **current_break**, więc mówi to, że startujemy z "heap" mającym zero bajtów. Nasza funkcja **allocate** będzie wtedy rozszerzać "heap" aż do potrzeb określonych podczas wywoływania.

Funkcja "allocate"
Zacznijmy od przeglądnięcia zarysu funkcji:
1. Start w początku "heap".
2. Sprawdzenie czy jesteśmy na końcu "heap".
3. Jeśli jesteśmy na końcu "heap", zabranie pamięci którą potrzebujemy od Linuksa, oznaczenie jej jako "unavailable" i zwrócenie jej. Jeśli Linux nie da nam więcej, zwraca 0.
4. Jeśli bieżący rejon pamięci jest oznaczony "unavailable", idziemy do następnego, i wracamy do kroku 2.
5. Jeśli bieżący rejon pamięci jest zbyt mały żeby przechowywać żądaną wielkość przestrzeni, wracamy do kroku 2.
6. Jeśli rejon pamięci jest dostępny i wystarczająco duży, oznaczamy go jako "unavailable" i zwracamy go.
Teraz, przeglądnij kod z powrotem z powyższym w myślach. Upewnij się, że przeczytałeś komentarze więc będziesz wiedział który rejestr przechowuje którą wartość.
Teraz skoro na powrót przeglądnąłeś kod, sprawdźmy go po jednym wierszu. Startujemy stąd:

```
pushl %ebp
movl %esp, %ebp
movl ST_MEM_SIZE(%ebp), %ecx
movl heap_begin, %eax
movl current_break, %ebx
```

Ta część inicjalizuje wszystkie nasze rejestry. Pierwsze dwa wiersze standardowymi działaniami funkcji. Następne przeniesienie wciąga rozmiar pamięci do zaalokowania na stos. To jest nasz jedyny parametr funkcji. Po tym, przesuwa adres początku "heap" i koniec "heap" do rejestrów. Jestem gotowy teraz do przeprowadzenia procesu.

Następna sekcja jest oznaczona **alloc_loop_begin**. W tej pętli zamierzamy przetestować rejony pamięci tak, że albo znajdziemy rejon wolnej pamięci lub stwierdzimy, że potrzebujemy więcej pamięci. Nasze pierwsze instrukcje sprawdzają czy potrzebujemy więcej pamięci:

```
cmpl %ebx, %eax
je move_break
```

%eax przechowuje bieżący rejon pamięci będącej testowaną i **%ebx** przechowuje lokalizację za końcem "heap". Dlatego jeśli następny rejon do testowania jest za końcem "heap", oznacza to, że potrzebujemy więcej pamięci do alokacji rejonu tego rozmiaru. Przesuńmy się do **move_break** i zobaczmy co tam się dzieje:

```
move_break:
addl $HEADER_SIZE, %ebx
addl %ecx, %ebx
pushl %eax
pushl %ecx
pushl %ebx
movl $SYS_BRK, %eax
int $LINUX_SYSCALL
```

Kiedy osiągnęliśmy ten punkt w kodzie, **%ebx** przechowuje miejsce gdzie chcemy żeby był następny rejon pamięci. Więc, dodajemy nasz rozmiar nagłówka i rozmiar rejonu do **%ebx**, i to jest gdzie chcemy aby systemowe "break" było. Wtedy odkładamy wszystkie rejestry które chcemy zachować na stos, i wywołujemy wywołanie systemowe **brk**. Po tym sprawdzamy błędy:

```
cmpl $0, %eax
je error
```

Jeśli nie było błędów zdejmujemy rejestry z powrotem ze stosu, zaznaczając pamięć jako niedostępna, zapisujemy rozmiar pamięci, i upewniamy się, że **%eax** wskazuje początek użytecznej pamięci (która jest za nagłówkiem).

```
popl %ebx
popl %ecx
popl %eax
movl $UNAVAILABLE, HDR_AVAIL_OFFSET(%eax)
movl %ecx, HDR_SIZE_OFFSET(%eax)
addl $HEADER_SIZE, %eax
```

Wtedy umieszczamy nowe "break" programu i zwracamy wskaźnik na alokowaną pamięć.

```
movl %ebx, current_break
movl %ebp, %esp
movl %ebp
ret
```

Kod **error** zwraca 0 w **%eax**, więc nie dyskutujemy go.
Powróćmy i zobaczmy resztę pętli. Co się dzieje jeśli bieżąca pamięć będąca pod obserwacją nie jest poza końcem "heap"? Dobrze, zobaczmy.

movl HDR_SIZE_OFFSET(%eax), %edx
cmpl $UNAVAILABLE, HDR_AVAIL_OFFSET(%eax)
je next_location

Pierwsza instrukcja pobiera rozmiar rejonu pamięci i umieszcza go w **%edx**. Wtedy obserwuje flagę dostępności czy jest ustawiona na "UNAVAILABLE". Jeśli tak, oznacza to, że rejon pamięci jest w użyciu, więc musimy ją pominąć. Więc, jeśli flaga dostępności jest ustawiona na "UNAVAILABLE", idziemy do kodu z etykietą **next_location**. Jeśli flaga dostępności jest ustawiona na "AVAILABLE", wtedy idziemy dalej.
Powiedzmy, że ta przestrzeń jest dostępna, i idziemy dalej. Wtedy sprawdzamy czy ta przestrzeń jest wystarczająco duża do przechowywania żądanej wielkości pamięci. Rozmiar tego rejonu jest przechowywany w **%edx**, więc robimy to:

cmpl %edx, %ecx
jle allocate_here

Jeśli żądany rozmiar jest mniejszy lub równy rozmiarowi bieżącego rejonu, możemy użyć tego bloku. Nie ma znaczenia jeśli bieżący rejon jest większy od żądanego, ponieważ ta ekstra przestrzeń po prostu nie będzie używana. Więc, przeskoczmy do **allocate_here** i zobaczmy co się dzieje:

movl $UNAVAILABLE, HDR_AVAIL_OFFSET(%eax)
addl $HEADER_SIZE, %eax
movl %ebp, %esp
popl %ebp
ret

To zaznacza pamięć jako niedostępna. Wtedy przenosi wskaźnik **%eax** za nagłówek, i używa go jako wartości powrotu dla funkcji. Pamiętaj, osoba używająca tej funkcji nie potrzebuje nawet wiedzieć o naszym rekordzie nagłówka pamięci. On tylko potrzebuje wskaźnika do użytecznej pamięci.
Dobrze, więc powiedzmy rejon nie był wystarczająco duży. Co wtedy? Tak, wtedy moglibyśmy być w kodzie z etykietą **next_location**. Ta sekcja kodu jest używana za każdym razem gdy stwierdzimy, że bieżący rejon pamięci nie będzie pracował jako alokowana pamięć. Wszystko co on robi to przeniesienie **%eax** do następnego możliwego rejonu pamięci, i powrócenie do początku pętli. Pamiętaj, że **%edx** przechowuje rozmiar bieżącego rejonu pamięci, i **HEADER_SIZE** jest symbolem dla rozmiaru nagłówka rejonu pamięci. Więc ten kod przeniesie nas do następnego rejonu pamięci:

addl $HEADER_SIZE, %eax
addl %edx, %eax
jmp alloc_loop_begin

A teraz funkcja uruchamia inną pętlę.
Kiedykolwiek masz pętlę, musisz się upewnić, że *zawsze* będzie ona miała koniec. Najlepszy sposób aby to zrobić jest przetestowanie wszystkich możliwości, i upewnienie się, że wszystkie one ostatecznie prowadzą do zakończenia pętli. W naszym przypadku, mamy następujące możliwości:
- Osiągniemy koniec "heap"
- Znajdziemy rejon pamięci który jest dostępny i wystarczająco duży
- Przejdziemy do następnej lokalizacji

Pierwsze dwa elementy są warunkami które będą powodować zakończenie pętli. Trzeci będzie utrzymywał działanie. Jednakże, nawet jeśli nigdy nie znajdziemy wolnego rejonu, ostatecznie osiągniemy koniec "heap", ponieważ jest on skończonego rozmiaru. Dlatego, wiemy, że bez względu na to który warunek jest prawdziwy, pętla ostatecznie osiągnie warunek zakończenia.

Funkcja "deallocate"
Funkcja **deallocate** jest znacznie prostsza niż **allocate**. Jest tak ponieważ nie musi ona w ogóle wykonywać żadnego przeszukiwania. Może ona właśnie zaznaczyć bieżący rejon pamięci jako **AVAILABLE**, i **allocate** będzie szukać go następnym razem kiedy jest wywoływana. Więc mamy:

```
movl ST_MEMORY_SEG(%esp), %eax
subl $HEADER_SIZE, %eax
movl $AVAILABLE, HDR_AVAIL_OFFSET(%eax)
ret
```

W tej funkcji, nie musimy zachowywać **%ebp** lub **%esp** odkąd nie zmieniamy ich, ani nie musimy odbudowywać ich na końcu. Wszystko co robimy to czytanie adresu rejonu pamięci ze stosu, powracanie do początku nagłówka, i oznaczanie rejonu jako dostępny. Ta funkcja nie ma wartości powrotu, więc nie zwracamy uwagi co pozostawiamy w **%eax**.

Właściwości Wykonania i Inne Problemy
Nasz uproszczony zarządca pamięci nie jest realnie użyteczny do niczego więcej niż ćwiczenia akademickie.
Ta sekcja przygląda się problemom z tak uproszczonym alokatorem.
Największy problem tutaj to prędkość. Teraz, jeśli jest wykonywanych tylko kilka alokacji, prędkość nie będzie wielką własnością. Ale pomyśl co się dzieje jeśli wykonujesz tysiąc alokacji. Przy alokacji numer 1000, musisz przeszukać 999 rejonów pamięci aby odkryć, że musisz zażądać więcej pamięci. Jak możesz zobaczyć, to daje spore spowolnienie. W dodatku, pamiętaj, że Linux może trzymać strony pamięci na dysku zamiast w pamięci. Więc, odkąd musisz przejść poprzez każdą część pamięci twojego programu, oznacza to, że Linux musi załadować każdą część pamięci która jest obecnie na dysku do sprawdzenia czy jest dostępna. Możesz zobaczyć jak jest to bardzo, bardzo powolne. Ta metoda jest powiedziane, że działa w czasie *linearnym*, co oznacza, że każdy element którym musisz zarządzać powoduje, że twój program wydłuża się. Program który działa w *stałym* czasie zabiera taką samą ilość czasu bez względu na ilość elementów do zarządzania. Weźmy funkcję **deallocate** dla przykładu. Uruchamia tylko 4 instrukcje, bez względu jak wieloma elementami zarządzamy, lub gdzie one są w pamięci. Faktycznie, chociaż nasza funkcja **allocate** jest jedną z najwolniejszych funkcji zarządcy pamięci, funkcja **deallocate** jest jedną z najszybszych.
Następnym problemem działania jest ile razy wywołujemy wywołanie systemowe **brk**. Wywołania systemowe zabierają wiele czasu. Nie są one jak funkcje, ponieważ procesor musi przełączyć tryb. Twój program nie ma pozwolenia na mapowanie swojej pamięci, ale kernel Linuksa ma. Więc, procesor musi się przełączyć w *tryb kernela*, wtedy Linux mapuje pamięć, a potem przełącza się z powrotem w *tryb użytkownika* dla twojej aplikacji aby kontynuować działanie. Jest to także zwane *przełączaniem kontekstu*. Przełączniki kontekstu są relatywnie wolne w procesorach x86. Generalnie, powinieneś unikać wywoływania kernela chyba, że tego rzeczywiście potrzebujesz.
Następnym problemem jest, że nie zapisujemy gdzie Linux rzeczywiście ustawia "break". Poprzednio wspominaliśmy, że Linux może w rzeczywistości ustawić "break" za miejscem którego żądaliśmy. W tym programie, nawet nie obserwujemy gdzie Linux rzeczywiście ustawia "break" - tylko przypuszczamy, że ustawia go tam gdzie żądamy. To nie jest realny błąd, ale będzie to prowadzić do niepotrzebnych wywołań systemowych **brk** kiedy już mamy pamięć zmapowaną.
Następnym problemem który mamy jest to, że jeżeli szukamy 5-cio bajtowego rejonu pamięci, i pierwszy wolny do którego dochodzimy jest 1000 bajtowy, po prostu zaznaczamy całą wielkość jako alokowaną i zwracamy ją. Prowadzi to do 995 bajtowej nieużywanej, ale alokowanej, pamięci. Byłoby miło w takich sytuacjach podzielić ją tak żeby 995 bajtów mogło być użyte później. Byłoby także dobrze połączyć wolne przestrzenie kiedy szukamy wielkich alokacji.

Używanie naszego Alokatora

Programy które robimy w tej książce nie są wystarczająco skomplikowane aby potrzebować zarządcy pamięci. Dlatego, użyjemy naszego zarządcy pamięci do alokowania bufora dla jednego z naszych plików programów czytająco/zapisujących zamiast wpisywania go w **.bss**.
Program na którym to zademonstrujemy to **read-records.s** z Rozdziału 6. Ten program używa bufora nazwanego **record_buffer** na swoje potrzeby wejścia/wyjścia. Po prostu zmienimy to z bycia buforem zdefiniowanym w **.bss** na bycie wskaźnikiem do dynamicznie alokowanego bufora używając naszego zarządcy pamięci. Będziesz musiał mieć kod z tego programu pod ręką jako, że będziemy tylko dyskutować o zmianach w tej sekcji.
Pierwsza zmiana którą powinniśmy zrobić jest w deklaracji. Obecnie wygląda ona tak:

```
.section .bss
.lcomm, record_buffer, RECORD_SIZE
```

Byłoby nieporozumieniem utrzymywanie tej samej nazwy, odkąd zmieniamy ją z rzeczywistego bufora na wskaźnik do bufora. W dodatku, teraz potrzebuje być tylko wielkości słowa (wystarczy do przechowywania wskaźnika). Nowa deklaracja będzie umieszczona w sekcji **.data** i będzie wyglądać tak:

```
record_buffer_ptr:
.long 0
```

Naszą następną zmianą jest to, że potrzebujemy zainicjalizować naszego zarządcę pamięci natychmiast po zapoczątkowaniu programu. Dlatego, zaraz po ustawieniu stosu, następujące wywołanie musi być dodane:

```
call allocate_init
```

Po tym, zarządca pamięci jest gotowy do wykonywania żądań alokacji pamięci. Potrzebujemy alokować wystarczająco dużo pamięci aby przechowywać te rekordy które wczytujemy. Dlatego, będziemy wywoływać **allocate** do alokacji tej pamięci, i potem zapiszemy wskaźnik który zwraca w **record_buffer_ptr**. w taki sposób:

```
pushl $RECORD_SIZE
call allocate
movl %eax, record_buffer_ptr
```

Teraz, kiedy wykonaliśmy wywołanie do **read_record**, jest oczekiwany wskaźnik. W starym kodzie, wskaźnik był odniesieniem trybu natychmiastowego do **record_buffer**. Obecnie, **record_buffer_ptr** przechowuje raczej wskaźnik niż bufor sam w sobie. Dlatego, musimy zrobić ładowanie trybu bezpośredniego aby otrzymać wartość w **record_buffer_ptr**. Musimy wymazać ten wiersz:

```
pushl $record_buffer
```

I umieścić ten wiersz na jego miejsce:

```
pushl record_buffer_ptr
```

Następna zmiana przychodzi kiedy próbujemy znaleźć adres pola nazwiska naszego rekordu. W starym kodzie było **$RECORD_FIRSTNAME + record_buffer**. Jednakże, to działa tylko dlatego, że jest stałe przesunięcie od stałego adresu. W nowym kodzie, przesunięcie adresu jest przechowywane w **record_buffer_ptr**. Aby otrzymać tę wartość, będziemy potrzebować przenieść wskaźnik do rejestru, i wtedy dodać **$RECORD_FIRSTNAME** do niego żeby otrzymać wskaźnik. Więc w miejscach gdzie mamy następujący kod:

pushl $RECORD_FIRSTNAME + record_buffer

Musimy wymienić go na taki:

movl record_buffer_ptr, %eax
addl $RECORD_FIRSTNAME, %eax
pushl %eax

Podobnie, musimy zmienić wiersz mówiący:

movl $RECORD_FIRSTNAME + record_buffer, %ecx

na takie:

movl record_buffer_ptr, %ecx
addl $RECORD_FIRSTNAME, %ecx

Ostatecznie, jedną zmianą którą potrzebujemy zrobić jest zdealokowanie pamięci z którą skończyliśmy działanie (w tym programie nie jest to niezbędne, ale jest dobrą praktyką). Aby to zrobić, przesyłamy **record_buffer_ptr** do funkcji **deallocate** zaraz przed wyjściem:

pushl record_buffer_ptr
call deallocate

Teraz możesz zbudować swój program poprzez następujące komendy:

as read-records.s -o read-records.o
ld alloc.o read-record.o read-records.o write-newline.o count-chars.o -o read-records

Możesz wtedy uruchomić swój program poprzez **./read-records**.
Użycia dynamicznej alokacji pamięci mogą nie być dla ciebie zbyt przejrzyste w tym momencie, ale jak przejdziesz od ćwiczeń akademickich do programów realnych będziesz używał jej ciągle.

Więcej Informacji

Więcej informacji na temat zarządzania pamięcią w Linuksie i innych systemach operacyjnych może być znalezione w następujących miejscach:
- Więcej informacji o planie pamięci w programach Linuksowych można znaleźć w dokumencie Konstantina Boldyszewa "Startup state of a Linux/i386 ELF binary", dostępnym na http://linuxassembly.org/startup.html
- Dobry przegląd pamięci wirtualnej w wielu różnych systemach jest dostępny na http://cne.gmu.edu/modules/vm/
- Kilka pogłębionych artykułów o podsystemie pamięci wirtualnej Linuksa jest dostępna na http://www.nongnu.org/ikdp/files.html
- Doug Lea opisał swój popularny alokator pamięci na http://gee.cs.oswego.edu/dl/html/malloc.html
- Dokument o alokatorze pamięci w 4.4 BSD jest dostępny na http://docs.freebsd.org/44doc/papers/malloc.html

Przegląd

Znajomość Koncepcji

- Co to jest "heap" (sterta)?
- Co to jest bieżący "break"?
- W którym kierunku wzrasta stos?
- W którym kierunku wzrasta sterta (heap)?
- Co się dzieje kiedy dostajesz się do niezmapowanej pamięci?
- Jak system operacyjny unika żeby procesy nadpisywały sobie wzajemnie pamięć?
- Opisz proces który pojawia się gdy część pamięci którą używasz jest obecnie na dysku?
- Dlaczego potrzebujesz alokatora?

Użycie Koncepcji

- Zmodyfikuj zarządcę pamięci tak żeby wywoływał **allocate_init** automatycznie jeśli nie była zainicjalizowana.
- Zmodyfikuj zarządcę pamięci tak że jeśli żądany rozmiar pamięci jest mniejszy od wybranego rejonu, będzie on dzielił ten rejon na wiele części. Upewnij się aby wziąć pod uwagę rozmiar nowego nagłówka rekordu kiedy to zrobisz.
- Zmodyfikuj jeden z twoich programów który używa buforów tak żeby używał zarządcę pamięci do otrzymania bufora pamięci raczej niż używał **.bss**.

Idąc Dalej

- Wymyśl *kolektor śmieci*. Jakie plusy i minusy to posiada w porównaniu ze stylem zarządcy pamięci użytym tutaj?
- Wymyśl *licznik odniesień*. Jakie plusy i minusy to posiada w porównaniu ze stylem zarządcy pamięci użytym tutaj?
- Zmień nazwę funkcji na **malloc** i **free**, i wbuduj je w bibliotekę dzieloną. Użyj **LD_PRELOAD** do przeforsowania użycia ich jako twój zarządca pamięci zamiast domyślnego. Dodaj kilka **write** wywołań systemowych do "STDOUT" żeby zweryfikować, że twój zarządca pamięci jest używany zamiast domyślnego.

Rozdział 10. Licząc Jak Komputer

Liczenie

Licząc Jak Człowiek

W wielu sytuacjach, komputery liczą właśnie tak jak ludzie. Wiec, zanim zaczniemy się uczyć jak komputery liczą, spójrzmy głębiej jak my liczymy.

Ile masz palców? Nie, nie jest to podchwytliwe pytanie. Ludzie (zwykle) mają dziesięć palców. Dlaczego jest to istotne? Spójrz na nasz system liczbowy. Od którego momentu liczba jednocyfrowa staje się dwucyfrową? Tak, to prawda, od dziesięciu. Ludzie liczą używając dziesiętnego systemu liczbowego. Podstawa dziesięć oznacza, że grupujemy wszystko w dziesiątki. Powiedzmy, że liczymy owce. 1, 2, 3, 4, 5, 6, 7, 8, 9, 10. Dlaczego nagle mamy dwie cyfry, i ponownie używamy 1? Jest tak dlatego, ponieważ grupujemy nasze liczby w dziesiątki, i mamy 1 grupę dziesięciu owiec. Dobrze, przejdźmy do następnej liczby 11. To oznacza, że mamy 1 grupę dziesięciu owiec, i 1 owcę pozostałą niezgrupowaną. Więc kontynuujemy - 12, 13, 14, 15, 16, 17, 18, 19, 20. Teraz mamy 2 grupy dziesięciu. 21 - 2 grupy dziesięciu, i 1 owcę niezgrupowaną. 22 - 2 grupy dziesięciu, i 2 owce niezgrupowane. Więc, powiedzmy kontynuujemy dalej, i dochodzimy do 97, 98, 99, i 100. Patrz, to zdarzyło się ponownie! Co dzieje się przy 100? Teraz mamy dziesięć grup po dziesięć. Przy 101 mamy dziesięć grup po dziesięć, i 1 niezgrupowaną owcę. Więc możemy rozpatrywać każdą liczbę w ten sposób. Jeśli naliczyliśmy 60879 owiec, mogłoby to oznaczać, że mamy 6 grup po dziesięć grup po dziesięć grup po dziesięć grup po dziesięć, 0 grup po dziesięć grup po dziesięć grup po dziesięć, 8 grup po dziesięć grup po dziesięć, 7 grup po dziesięć, i 9 owiec pozostaje niezgrupowanych.

Więc, czy jest coś istotnego w grupowaniu rzeczy w dziesiątki? Nie! Właśnie grupowanie w dziesiątki jest tym co zawsze robiliśmy, ponieważ mamy dziesięć palców. Moglibyśmy grupować w dziewiątki lub w jedenastki (w tym wypadku

musielibyśmy zrobić nowy symbol). Jedyną różnicą pomiędzy różnymi grupowaniami liczb jest to, że musimy nauczyć się od nowa naszych tablic mnożenia, dodawania, odejmowania i dzielenia dla każdego grupowania. Zasady nie zmieniły się, tylko sposób w jaki je prezentujemy. Także, niektóre nasze sztuczki których się nauczyliśmy nie zawsze zadziałają. Na przykład, powiedzmy, że grupujemy po dziewięć zamiast po dziesięć. Przesunięcie przecinka dziesiętnego o jedną cyfrę w prawo już nie mnoży przez dziesięć, teraz mnoży przez dziewięć. Przy podstawie dziewięć, 500 jest tylko dziewięć razy większe od 50.

Licząc Jak Komputer

Jest pytanie, ile palców ma komputer na których liczy? Komputer ma tylko dwa palce. Więc to oznacza, że wszystkie te grupy są grupami dwójek. Więc, liczmy binarnie - 0 (zero), 1 (jeden), 10 (dwa - jedna grupa dwójek), 11 (trzy - jedna grupa dwójek i jeden pozostały ponad), 100 (cztery - dwie grupy dwójek), 101 (pięć - dwie grupy dwójek i jeden pozostały ponad), 110 (sześć - dwie grupy dwójek i jedna grupa dwójek), i tak dalej. Przy podstawie dwa, przesunięcie dziesiętne jednej cyfry w prawo mnoży przez dwa, a przesunięcie jej w lewo dzieli przez dwa. Podstawa dwa jest także określana jako binarna.

Miłą rzeczą w podstawie dwa jest to, że tabele podstaw matematycznych są bardzo krótkie. Przy podstawie dziesięć, tablice mnożenia są na dziesięć kolumn szerokie i na dziesięć wierszy wysokie. Przy podstawie dwa, jest to bardzo proste:

Tabela dodawania binarnego

+	0	1
0	0	1
1	1	10

Tabela mnożenia binarnego

*	0	1
0	0	0
1	0	1

Więc, dodajmy liczby 10010101 i 1100101:

```
  10010101
+ 1100101
----------
  11111010
```

Teraz, pomnóżmy je:

```
                  10010101
                * 1100101
                ----------
                  10010101
                 00000000
                10010101
               00000000
              00000000
             10010101
            10010101
           ----------------------------
```

11101011001001

Konwersje Pomiędzy Liczbami Binarnymi i Decymalnymi

Nauczmy się konwersji liczb binarnych (podstawa dwa) na decymalne (podstawa dziesięć). Jest to w rzeczywistości raczej prosty proces. Jeśli pamiętasz, każda cyfra oznacza jakąś grupę dwójek. Więc, musimy właśnie pododawać wszystkie reprezentacje cyfr, i będziemy mieli liczbę decymalną. Weź liczbę binarną 10010101. Aby znaleźć ile to jest decymalnie, rozkładamy ją tak:

1	0	0	1	0	1	0	1
|	|	|	|	|	|	|	Jednostka (2^0)
|	|	|	|	|	|	0 grup 2 (2^1)	
|	|	|	|	|	1 grupa 4 (2^2)		
|	|	|	|	0 grup 8 (2^3)			
|	|	|	1 grupa 16 (2^4)				
|	|	0 grup 32 (2^5)					
|	0 grup 64 (2^6)						
1 grupa 128 (2^7)							

i wtedy dodajemy wszystkie części razem, w ten sposób:
1*128 + 0*64 + 0*32 + 1*16 + 0*8 + 1*4 + 0*2 + 1*1 = 128 + 16 + 4 + 1 = 149
Więc 10010101 binarnie to 149 decymalnie. Popatrzmy na 1100101. Może być to zapisane jako
1*64 + 1*32 + 0*16 + 0*8 + 1*4 + 0*2 + 1*1 = 64 + 32 + 4 + 1 = 101
Więc widzimy, że 1100101 binarnie to 101 decymalnie. Popatrzmy na jeszcze jedną liczbę, 11101011001001. Możesz zamienić ją na decymalną poprzez
1*8192 + 1*4096 + 1*2048 + 0*1024 + 1*512 + 0*256 + 1*128 + 1*64 + 0*32 + 0*16 + 1*8 + 0*4 + 0*2 + 1*1 =
8192 + 4096 + 2048 + 512 + 128 + 64 + 8 + 1 = 15049
Teraz, jeśli uważałeś, odnotowałeś, że liczby które zamienialiśmy są tymi samymi których użyliśmy wcześniej do mnożenia. Więc, sprawdźmy nasze rezultaty: 101*149=15049. To działa!
Teraz przypatrzmy się przejściu od liczb decymalnych z powrotem do binarnych. Żeby dokonać tej konwersji, musisz *podzielić* tę liczbę na grupy dwójek. Więc, powiedzmy masz liczbę 17. Jeśli podzielisz ją przez dwa, otrzymasz 8 z 1 pozostałym ponad. Więc to oznacza, że jest 8 grup dwójek, i 1 niezgrupowane. To oznacza, że cyfrą najbardziej na prawo będzie 1. Teraz, mamy cyfrę najbardziej na prawo ustaloną, i 8 grup po 2 pozostałych dalej. Teraz, zobaczmy jak wiele mamy grup dwójek z grup dwójek, poprzez dzielenie 8 przez 2. Otrzymujemy 4, bez reszty. To oznacza, że wszystkie grupy mogą być dalej dzielone na więcej grup dwójek. Więc, mamy 0 grup jedynie dwójek. Więc następna cyfra w lewo to 0. Więc, dzielimy 4 przez 2 i otrzymujemy dwa, i 0 reszty, więc następną cyfrą jest 0. Wtedy, dzielimy 2 przez 2 i otrzymujemy 1, i 0 reszty. Więc następną cyfrą jest 0. Ostatecznie, dzielimy 1 przez 2 i otrzymujemy 0 i 1 reszty, więc następną cyfrą w lewo jest 1. Teraz, nic nie pozostało, skończyliśmy. Więc, liczbą którą otrzymaliśmy jest 10001.
Poprzednio, zamieniliśmy binarne 11101011001001 na decymalne 15049. Wykonajmy odwrotność aby potwierdzić, że zrobiliśmy to dobrze:

15049 / 2 = 7524	Reszta 1
7524 / 2 = 3762	Reszta 0

3762 / 2 = 1881	Reszta 0
1881 / 2 = 940	Reszta 1
940 / 2 = 470	Reszta 0
470 / 2 = 235	Reszta 0
235 / 2 = 117	Reszta 1
117 / 2 = 58	Reszta 1
58 / 2 = 29	Reszta 0
29 / 2 = 14	Reszta 1
14 / 2 = 7	Reszta 0
7 / 2 = 3	Reszta 1
3 / 2 = 1	Reszta 1
1 / 2 = 0	Reszta 1

Wtedy, układamy reszty liczb z powrotem razem, i mamy liczbę oryginalną! Pamiętaj, że pierwsza reszta z dzielenia idzie najdalej w prawo, więc od dołu do góry mamy
11101011001001.
Każda cyfra liczby binarnej jest zwana *bitem*, pochodzi to od *cyfry binarnej*. Pamiętaj, że komputery dzielą swoją pamięć na lokalizacje zwane bajtami. Każda lokalizacja w procesorach x86 (i większości innych) jest długości 8 bitów. Wcześniej powiedzieliśmy, że bajt może przechowywać jakąkolwiek liczbę pomiędzy 0 i 255. Powodem tego jest to, że największą liczbą jaką możesz zmieścić w 8 bitach jest 255. Możesz sam to sprawdzić konwertując binarne 11111111 do liczby decymalnej:
11111111 = (1 * 2^7) + (1 * 2^6) + (1 * 2^5) + (1 * 2^4) + (1 * 2^3) + (1 * 2^2) + (1 * 2^1) + (1 * 2^0) = 128 + 64 + 32 + 16 + 8 + 4 + 2 + 1 = 255
Największą liczbą którą możesz przechowywać w 16 bitach jest 65535. Największą liczbą którą możesz przechowywać w 32 bitach jest 4294967295 (4 miliardy). Największą liczbą którą możesz przechowywać w 64 bitach jest 18,446,744,073,709,551,615. Największą liczbą którą możesz przechowywać w 128 bitach jest 340,282,366,920,938,463,374,607,431,768,211,456. Jakkolwiek, daje to jakieś wyobrażenie. Dla procesorów x86, większość czasu będziesz pracował z liczbami 4-bajtowymi (32 bity), ponieważ to jest rozmiar rejestrów.

Prawda, Fałsz i Liczby Binarne

Teraz możemy stwierdzić, że komputer przechowuje wszystko jako sekwencje jedynek i zer. Popatrzmy na niektóre inne użycia tego. Co jeśli, zamiast patrzenia na sekwencję bitów jako liczby, patrzymy na nią jak na zestaw przełączników. Na przykład, powiedzmy są cztery przełączniki które kontrolują oświetlenie w domu. Mamy przełącznik dla oświetlenia zewnętrznego, przełącznik dla oświetlenia korytarza, przełącznik dla oświetlenia jadalni, przełącznik dla oświetlenia sypialni. Moglibyśmy zrobić małą tabelę pokazującą które z nich są włączone a które wyłączone, w taki sposób:
Zewnętrzne Korytarz Jadalnia Sypialnia
Włączone Wyłączone Włączone Włączone
Jest oczywiste, patrząc na to, że wszystkie oświetlenia oprócz korytarza są jedynkami. Teraz, zamiast używania słów "Włączone" i "Wyłączone", użyjmy liczb 1 i 0. 1 będzie reprezentował włączone, a 0 wyłączone. Więc, moglibyśmy zaprezentować tę samą informację jako

Zewnętrzne	Korytarz	Jadalnia	Sypialnia
włączone	wyłączone	włączone	włączone

Jest oczywiste po spojrzeniu na to, że wszystkie światła są włączone za wyjątkiem tego w korytarzu .Teraz, zamiast używania słów "włączone" i "wyłączone", użyjmy liczb 1 i 0. 1 będzie reprezentował włączone, a 0 będzie reprezentowało wyłączone. Wtedy, ta sama informacja mogłaby być zaprezentowana jako

Zewnętrzne	Korytarz	Jadalnia	Sypialnia
1	0	1	1

Teraz, zamiast posiadania etykiet na przełączniki światła, powiedzmy tylko zapamiętujemy pozycję idącą z danym przełączeniem. Wtedy, ta sama informacja mogłaby być reprezentowana tak

1 0 1 1

lub jako

1011

To jest tylko jeden z wielu sposobów w jaki możesz użyć lokalizacji komputerowych do reprezentowania czegoś więcej niż tylko liczby. Pamięć komputera widzi tylko liczby, ale programiści mogą użyć tych liczb do reprezentowania wszystkiego co im podpowie wyobraźnia. Czasami muszą oni być kreatywni kiedy ustalają najlepszą reprezentację.
Możesz wykonywać nie tylko zwykłą arytmetykę z liczbami binarnymi, jest także kilka własnych operacji, zwanych binarnymi lub logicznymi operacjami. Standardowymi operacjami binarnymi są
- AND
- OR
- NOT
- XOR

Zanim przejrzymy przykłady, opiszę je dla ciebie. "AND" pobiera dwa bity i zwraca jeden bit. "AND" zwróci 1 tylko jeśli obydwa bity są 1, i 0 w innym przypadku. Na przykład, 1 AND 1 daje 1, ale 1 AND 0 daje 0, 0 AND 1 daje 0, i 0 AND 0 daje 0.
"OR" pobiera dwa bity i zwraca jeden bit. Zwróci 1 jeśli któryś z oryginalnych bitów wynosi 1. Na przykład, 1 OR 1 daje 1, 1 OR 0 daje 1, 0 OR 1 daje 1, ale 0 OR 0 daje 0.
"NOT" pobiera tylko jeden bit, i zwraca jego przeciwieństwo, NOT 1 daje 0 i NOT 0 daje 1.
Ostatecznie, "XOR" jest takie same jak "OR", oprócz tego, że zwraca 0 jeśli obydwa bity są 1.
Komputery mogą robić te operacje na całych rejestrach na raz. Na przykład, jeśli rejestr ma
10100010101010010101101100101010 i inny ma
10001000010101010101010101111010, możesz uruchomić którąś z tych operacji na całych rejestrach. Na przykład, jeśli mielibyśmy dokonać operacji "AND" na nich, komputer będzie przechodził od pierwszego do 32-go bitu i wykona operację "AND" na bitach w obydwu rejestrach. W tym przypadku:
10100010101010010101101100101010 AND
10001000010101010101010101111010

10000000000000010101000100101010
Zobaczysz, że wynikowy zestaw bitów ma jedynki tylko tam gdzie *obydwie* liczby są jedynkami, i w każdej innej pozycji ma zero. Przypatrzmy się jak "OR" wygląda:
10100010101010010101101100101010 OR
10001000010101010101010101111010

10101010111111010101111101111010
W tym przypadku, liczba wynikowa ma 1 tam gdzie chociaż jedna z liczb jest 1. Przypatrzmy się operacji "NOT":
NOT 10100010101010010101101100101010

01011101010101101010010011010101
To odwraca każdy bit. Ostatecznie, mamy "XOR", która jest taka sama jak "OR", oprócz tego gdy *obydwie* cyfry są 1, wtedy zwraca 0.
10100010101010010101101100101010 XOR
10001000010101010101010101111010

00101010111111000000111001010000
To są te same dwie liczby co użyte w operacji "OR", więc możesz porównać jak działają te operacje.
Także, jeśli zrobisz "XOR" dla liczby z nią samą, zawsze otrzymasz 0, tak jak:
10100010101010010101101100101010 XOR
10100010101010010101101100101010

00000000000000000000000000000000

Te operacje są użyteczne z dwu powodów:
- Komputer może wykonywać je niezwykle szybko
- Możesz używać ich do porównywania wielu wartości prawdy w tym samym czasie
Możesz nie wiedzieć, że różne instrukcje wykonują się z różną szybkością. To prawda, tak robią. I te operacje są najszybsze na większości procesorów. Na przykład, widziałeś, że "XORowanie" liczby z nią samą daje 0. Dobrze, operacja "XOR" jest szybsza niż operacja ładowania, więc wielu programistów używa jej do ładowania rejestru zerem. na przykład, kod:

movl $0, %eax

jest często wymieniany na

xorl %eax, %eax

Będziemy dyskutować o szybkości w Rozdziale 12, ale chcę żebyś zobaczył jak programiści robią często sztuczki, specjalnie z tymi operacjami binarnymi, żeby uczynić je szybkimi. Teraz popatrzmy jak możemy użyć tych operacji do manipulowania wartościami prawda/fałsz. Wcześniej dyskutowaliśmy jak liczby binarne mogą być używane do reprezentowania jakiejkolwiek liczby rzeczy. Użyjmy liczb binarnych do reprezentowania tego co mój Tata i ja lubimy.
Najpierw, zobaczmy co ja lubię:
Jedzenie: tak
Muzyka Heavy Metal: tak
Nosić Eleganckie Ubrania: nie
Piłka Nożna: tak

Teraz, zobaczmy co mój Tata lubi:
Jedzenie: tak
Muzyka Heavy Metal: nie
Nosić Eleganckie Ubrania: tak
Piłka Nożna: tak

Teraz, użyjmy 1 aby powiedzieć tak gdy coś lubimy, i 0 aby powiedzieć nie gdy nie lubimy. Teraz mamy:

Ja
Jedzenie: 1
Muzyka Heavy Metal: 1
Nosić Eleganckie Ubrania: 0

Piłka Nożna: 1

Tata
Jedzenie: 1
Muzyka Heavy Metal: 0
Nosić Eleganckie Ubrania: 1
Piłka Nożna: 1

Teraz, jeśli tylko zapamiętamy co któraś pozycja oznacza, mamy:

Ja
1101

Tata
1011

Teraz, chcemy otrzymać listę rzeczy które obydwaj mój Tata i ja lubimy. Mógłbyś użyć operacji "AND". Więc

1101 AND
1011

1001

Co tłumaczy się na

Rzeczy które obydwaj lubimy
Jedzenie: tak
Muzyka Heavy Metal: nie
Nosić Eleganckie Ubrania: nie
Piłka Nożna: tak

Pamiętaj, komputer nie ma pojęcia co jedynki i zera reprezentują. To twoja praca i praca twojego programu. Jeśli napisałbyś program wokół tej reprezentacji twój program mógłby w jakimś zakresie przetestować każdy bit i mógłby posiadać kod który powie użytkownikowi co to jest (jeśli zapytałbyś komputer na co dwoje ludzi się zgodziło i uzyskałbyś odpowiedź 1001, nie byłoby to zbyt użyteczne). Tak czy inaczej, powiedzmy, że chcemy poznać w jakich rzeczach się nie zgadzamy. W tym celu moglibyśmy użyć "XOR", ponieważ ona zwróci 1 tylko wtedy gdy jeden lub drugi jest 1, ale nie obydwa. Więc

1101 XOR
1011

0110

Poprzednie operacje: "AND", "OR", "NOT", i "XOR" są zwane *operatorami boolowskimi* ponieważ były one po raz pierwszy studiowane przez George'a Booe'a. Więc, jeśli ktoś wspomina o operatorach boolowskich lub algebrze boolowskiej, wiesz teraz o czym jest mowa.
Oprócz operatorów boolowskich, są także dwa operatory binarne które nie są boolowskimi, "shift" i "rotate". "Shifts" (przesunięcia) i "rotates" (obroty) robią to co sugerują ich nazwy, i mogą to robić w prawo lub w lewo. Lewy "shift" przesuwa każdą cyfrę liczby binarnej o jedno miejsce w lewo, dokładając zero w puste miejsce, i obcinając najdalszą cyfrę na lewo. Lewy "rotate" wykonuje to samo, ale zabiera najdalszą cyfrę w lewo i kładzie ją w puste miejsce po prawej. Na przykład,

Shift left 10010111 = 00101110
Rotate left 10010111 = 00101111
Zauważ, że jeśli wykonujesz "rotate" liczby dla każdej cyfry którą ona ma (t.j. - obracając liczbą 32-bitową 32 razy), zakończysz z tą samą liczbą z którą zacząłeś. Jednakże, jeśli wykonujesz "shift" liczby dla każdej cyfry którą ona ma, zakończysz z 0. Więc, do czego są użyteczne "shifts"? Cóż, jeśli masz liczby binarne reprezentujące rzeczy, używasz "shifts" do spojrzenia na każdą indywidualną wartość. Powiedzmy, na przykład, że mamy upodobania mojego Taty przechowywane w rejestrze (32 bity). Mogłoby to wyglądać tak:
00000000000000000000000000001011
Teraz, jak powiedzieliśmy poprzednio, to nie działa jako wyjście programu. Więc, aby wykonać wyjście, będziemy potrzebowali wykonania przesunięcia ("shifting") i *maskowania* ("masking"). Maskowanie jest to proces eliminowania wszystkiego czego nie chcesz. W tym przypadku, dla każdej wartości której szukamy, będziemy przesuwać liczbę tak, że ta wartość jest na pierwszym miejscu, i wtedy maskujemy tę cyfrę tak, że jest to wszystko co widzimy. Maskowanie jest przeprowadzane poprzez wykonanie "AND" z liczbą która ma interesujące nas bity ustawione na 1. Na przykład, powiedzmy, że chcemy wypisać czy mój Tata lubi eleganckie ubrania czy nie. Ta dana jest drugą od prawej wartością. Więc, musimy przesunąć liczbę o 1 cyfrę w prawo, i wygląda ona tak:

00000000000000000000000000000101

i wtedy, chcemy właśnie spojrzeć na tę cyfrę, więc maskujemy ją przez ANDowanie jej z 00000000000000000000000000000001.

00000000000000000000000000000101 AND
00000000000000000000000000000001

00000000000000000000000000000001

To spowoduje, że wartość rejestru będzie 1 jeśli mój Tata lubi eleganckie ubrania, i 0 jeśli nie lubi. Wtedy możemy dokonać porównania do 1 i wypisać wyniki. Kod ten mógłby wyglądać tak:

#UWAGA - zakładamy, że rejestr %ebx przechowuje preferencje mojego Taty

movl %ebx, %eax #To kopiuje tę informację do %eax żebyśmy nie stracili oryginalnej dany

shrl $1, %eax #To jest operator przesunięcia ("shift"). Oznacza on "Shift Right Long". #Pierwsza liczba jest liczbą pozycji do przesunięcia, a druga jest rejestrem w którym jest liczba przesuwana

#To wykonuje maskowanie
andl $0b00000000000000000000000000000001, %eax

#Sprawdzenie czy wynik jest 1 lub 0
cmpl $0b00000000000000000000000000000001, %eax
je yes_he_likes_dressy_clothes
jmp no_he_doesnt_like_dressy_clothes

I wtedy moglibyśmy mieć dwie etykiety które wypisują coś o tym czy on lubi czy nie lubi eleganckie ubrania i potem wychodzą. Notacja **0b** oznacza, że to co następuje po tym jest liczbą binarną. W tym przypadku nie było to potrzebne, ponieważ 1 jest tym samym w każdym systemie numerycznym, ale umieściłem to tutaj dla jasności. Także nie potrzebowaliśmy tych 31 zer, ale umieściłem je tutaj aby zaznaczyć, że liczba której używamy jest 32 bitowa.
Kiedy liczba reprezentuje zestaw opcji dla funkcji lub wywołania systemowego, indywidualne elementy prawda/fałsz są

zwane *flagami*. Wiele wywołań systemowych posiada sporo opcji które są wszystkie ustawione w tym samym rejestrze używając mechanizmu podobnego do tego który opisaliśmy. Wywołanie systemowe **open**, na przykład, posiada jako drugi parametr listę flag mówiących systemowi jak otworzyć ten plik. Zawartość niektórych flag:

O_WRONLY
Ta flaga wynosi 0b00000000000000000000000000000001 binarnie, lub 01 oktalnie (lub inna liczba systemu dla tego celu). Mówi ona aby otworzyć plik w trybie tylko do zapisu.

O_RDWR
Ta flaga wynosi 0b00000000000000000000000000000010 binarnie, lub 02 oktalnie. Mówi ona aby otworzyć plik dla obydwu odczytu i zapisu.

O_CREAT
Ta flaga wynosi 0b00000000000000000000000001000000 binarnie, lub 0100 oktalnie. Oznacza to aby utworzyć plik jeśli jeszcze nie istnieje.

O_TRUNC
Ta flaga wynosi 0b00000000000000000000001000000000 binarnie, lub 01000 oktalnie. Oznacza to aby wymazać zawartość pliku jeśli ten plik już istnieje.

O_APPEND
Ta flaga wynosi 0b00000000000000000000010000000000 binarnie, lub 02000 oktalnie. Oznacza to aby zacząć zapis na końcu pliku raczej niż na początku.

Aby użyć tych flag, po prostu wykonaj "OR" na nich w kombinacji której chcesz. Na przykład, aby otworzyć plik w trybie tylko do zapisu, i utworzyć ten plik jeśli nie istnieje, mógłbym użyć O_WRONLY (01) i O_CREAT (0100). Po wykonaniu "OR", miałbym 0101.
Zauważ, że jeśli nie ustawisz O_WRONLY albo O_RDWR, wtedy ten plik zostanie otwarty automatycznie w trybie tylko do odczytu (O_RDONLY, oprócz tego nie jest to rzeczywista flaga ponieważ wynosi zero).
Wiele funkcji i wywołań systemowych używa flag dla opcji, pozwala to na przechowywanie w pojedynczym słowie do 32 możliwych opcji jeśli każda opcja jest reprezentowana przez pojedynczy bit.

Rejestr Statusu Programu

Widzieliśmy jak bity w rejestrze mogą być użyte do dawania odpowiedzi tak/nie i prawda/fałsz. Na twoim komputerze, jest rejestr zwany *rejestrem statusu programu*. Ten rejestr przechowuje mnóstwo informacji o tym co dzieje się podczas obliczeń. Na przykład, czy byłeś kiedykolwiek ciekaw co mogłoby się stać jeślibyś dodał dwie liczby i wynik był większy niż rozmiar rejestru? Rejestr statusu programu ma flagę zwaną flagą przeniesienia (carry flag). Możesz ją przetestować aby zobaczyć czy ostatnia operacja przepełniła rejestr. Są flagi dla wielu różnych statusów. W rzeczywistości, kiedy wykonujesz instrukcję porównania (**cmpl**), wynik jest przechowywany w tym rejestrze. Instrukcje skoków warunkowych (**jge**, **jne**, itd.) używają tych wyników do odpowiedzi czy powinny wykonać skok czy nie. **jmp**, skok bezwarunkowy, nie zwraca uwagi co jest w rejestrze statusu, ponieważ jest bezwarunkowy.
Powiedzmy, że potrzebujesz przechować liczbę większą niż 32 bity. Więc, powiedzmy ta liczba jest rozmiaru 2 rejestrów, lub 64 bitów. Jak mógłbyś to rozwiązać? Jeśli chciałbyś dodać dwie liczby 64 bitowe, mógłbyś dodać najpierw najmniej znaczące rejestry. Potem, jeśli stwierdziłbyś przeniesienie, mógłbyś dodać 1 do najbardziej znaczącego rejestru. W rzeczywistości, prawdopodobnie to jest sposób w jaki nauczyłeś się dodawania dziesiętnego. Jeśli wynik w jednej kolumnie jest większy niż 9, po prostu przenosisz liczbę do następnej najbardziej znaczącej kolumny. Jeśli dodałeś 65 i 37, pierwsze

dodajesz 7 i 5 otrzymując 12. Zatrzymujesz 2 w prawej kolumnie, i przenosisz jeden do następnej kolumny. Tam dodajesz 6, 3, i 1 który przeniosłeś. Daje to wynik 10. Więc, zatrzymujesz zero w tej kolumnie i przenosisz jeden do następnej najbardziej znaczącej kolumny, która jest pusta, więc tylko umieszczasz tam to jeden. Szczęśliwie, 32 bity są zwykle wystarczająco duże do przechowywania liczb których używamy regularnie.
Dodatkowe flagi rejestru statusu programu są testowane w Dodatku B.

Inne Systemy Liczbowe

To co studiowaliśmy do tej pory dotyczy liczb całkowitych dodatnich. Jednakże, liczby realnego świata nie zawsze są całkowitymi dodatnimi. Liczby ujemne i liczby z przecinkami także są używane.

Liczby Zmiennoprzecinkowe
Do tej pory, jedyne liczby z jakimi mieliśmy do czynienia to liczby całkowite - liczby bez przecinka dziesiętnego. Komputery mają generalnie problem z liczbami z przecinkami dziesiętnymi, ponieważ komputery mogą tylko przechowywać wartości stałorozmiarowe, skończone. Liczby rzeczywiste mogą być dowolnej długości, włączając w to długość nieskończoną (pomyśl o liczbach rzeczywistych powtarzalnych, takich jak wynik z 1/3).
Sposobem w jaki komputer radzi sobie z liczbami rzeczywistymi jest przechowywanie ich w stałej precyzji (liczba znaczących bitów). Komputer przechowuje liczby rzeczywiste w dwu częściach - *eksponenta* i *mantysa*. Mantysa zawiera rzeczywiste cyfry których będziemy używać, i eksponenta jest potęgą liczby. Na przykład, 12345,2 jest przechowywane jako 1,23452*10^4. Mantysą jest 1,23452 i eksponentą jest 4. Wszystkie liczby są przechowywane jako X,XXXXX*10^XXXX. Liczba 1 jest przechowywana jako 1,00000*10^4.
Teraz, mantysa i eksponenta są tylko określonej długości, co prowadzi do kilku interesujących problemów. Na przykład, kiedy komputer przechowuje liczbę całkowitą, jeśli dodasz do niej 1, liczba wynikowa jest o jeden większa. To niekoniecznie się zdarza z liczbami zmiennoprzecinkowymi. Jeśli liczba jest odpowiednio duża, jak 5,234*10^5000, dodanie 1 do niej może nawet nie być zarejestrowane w mantysie (pamiętaj, obydwie części są tylko ustalonej długości). To uwrażliwia na kilka szczegółów, specjalnie na kolejność operacji. Powiedzmy, że dodałem 1 do 5,234*10^5000 kilka miliardów lub trylionów razy. Zgadujemy że - liczba w ogóle się nie zmieni. Jednakże, jeśli dodam jeden do jeden wystarczającą ilość razy, i potem dodam to do oryginalnej liczby, to może zrobić efekt.
Powinieneś zauważyć, że większości komputerów zabiera dużo więcej czasu robienie arytmetyki zmiennoprzecinkowej niż robienie arytmetyki liczb całkowitych. Więc, dla programów które rzeczywiście potrzebują szybkości, liczby całkowite są częściej używane.

Liczby Ujemne
Jak myślisz mogłyby być reprezentowane liczby ujemne na komputerze? Jednym pomysłem mogłoby być użycie pierwszej cyfry liczby jako znaku, więc
00000000000000000000000000000001 mogłoby reprezentować liczbę 1, i
10000000000000000000000000000001 mogłoby reprezentować -1. To ma wiele sensu, i w rzeczywistości niektóre stare komputery działały w ten sposób. Jednakże, powoduje on kilka problemów. Przede wszystkim, pobiera o wiele więcej prądu aby dodawać i odejmować liczby ze znakiem taką drogą. Nawet bardziej problematyczne, ta reprezentacja ma problem z liczbą 0. W ten sposób, mógłbyś mieć zarówno ujemne i dodatnie 0. To prowadzi do wielu pytań, w rodzaju "czy ujemne zero powinno być równe dodatniemu zeru?", i "jaki powinien być znak zera w różnorodnych warunkach?".
Problemy te były przezwyciężone przez użycie reprezentacji liczb ujemnych zwanej *uzupełnieniem dwójkowym*. Aby otrzymać reprezentację liczby ujemnej w formie uzupełnienia dwójkowego, musisz wykonać następujące kroki:
1. Przeprowadź operację "NOT" na liczbie
2. Dodaj jeden do liczby wynikowej

Więc, aby uzyskać ujemne 00000000000000000000000000000001, mógłbyś po pierwsze wykonać operację "NOT", co

daje 11111111111111111111111111111110, i wtedy dodać jeden, otrzymując 11111111111111111111111111111111. Aby otrzymać ujemne dwa, po pierwsze weź 00000000000000000000000000000010. "NOT" tej liczby to 11111111111111111111111111111101. Dodanie jednego daje 11111111111111111111111111111110. Z tą reprezentacją, możesz dodawać liczby właśnie tak jakby były one liczbami dodatnimi, i wychodzisz z dobrymi odpowiedziami. Na przykład, jeśli dodasz dodatni jeden do jeden ujemnego binarnie, zauważysz, że wszystkie cyfry zmierzają do zera. Także, pierwsza cyfra ciągle niesie bit znaku, upraszczając zdeterminowanie czy liczba jest dodatnia czy ujemna. Liczba ujemna będzie zawsze miała 1 w skrajnym lewym bicie. To także zmienia które liczby są ważne dla danej liczby bitów. Dla liczb ze znakiem, możliwa potęga wartości jest podzielona na obydwie dodatnie i ujemne liczby. Na przykład, bajt może zwykle mieć wartości do 255. Bajt ze znakiem, jednakże, może przechowywać wartości od -128 do 127.

Jedna rzecz do zauważenia w reprezentacji uzupełnienia dwójkowego liczb ze znakiem jest taka, że w przeciwieństwie do liczb bez znaku, jeśli zwiększysz liczbę bitów, nie możesz tak po prostu dodać zer po lewej stronie liczby. Na przykład, powiedzmy, że zajmujemy się wielkościami cztero-bitowymi i mamy liczbę -3, 1101. Jeśli rozszerzylibyśmy to do rejestru ośmio-bitowego, nie moglibyśmy reprezentować jej jako 00001101 bo to reprezentuje 13, nie -3. Kiedy rozszerzasz rozmiar wielkości ze znakiem w reprezentacji uzupełnienia dwójkowego, musisz przeprowadzić *rozszerzenie znaku*. Rozszerzenie znaku oznacza, że musisz dopełnić lewą stronę wielkości cyframi takimi samymi jakie są na miejscu cyfry znaku kiedy dodajesz bity. Więc, jeśli rozszerzamy liczbę ujemną 4 cyframi, powinniśmy wypełnić nowe cyfry przez 1. Jeśli rozszerzamy liczbę dodatnią 4 cyframi, powinniśmy wypełnić nowe cyfry przez 0. Więc, rozszerzanie -3 z czterech do ośmiu bitów da 11111101.

Procesor x86 ma różne formy kilku instrukcji zależnie od tego czy oczekują one operowania na wielkościach ze znakiem czy bez znaku. Są one wymienione w Dodatku B. Na przykład, procesor x86 posiada obydwie "shift-right" uwzględniająca znak, **sarl**, i "shift-right" która nie uwzględnia bitu znaku, **shrl**.

Liczby Oktalne i Heksadecymalne

Systemami liczbowymi dyskutowanymi do tej pory były decymalny (dziesiętny) i binarny. Jednakże, dwa inne są używane powszechnie w informatyce - oktalny i heksadecymalny. W rzeczywistości, występują one prawdopodobnie w zapisie znacznie częściej niż binarny. System oktalny jest reprezentacją która używa tylko liczb od 0 do 7. Więc oktalna liczba 10 jest rzeczywiście 8 w decymalnym ponieważ jest to jedna grupa ósemek. Oktalne 121 jest decymalnym 81 (jedna grupa 64 (8^2), dwie grupy 8, i jeden pozostały). Co czyni liczby oktalne interesującymi jest to, że każde 3 binarne cyfry dają jedną oktalną cyfrę (nie ma takich zgrupowań cyfr binarnych w decymalnych). Więc 0 to 000, 1 to 001, 2 to 010, 3 to 011, 4 to 100, 5 to 101, 6 to 110, i 7 to 111.

Uprawnienia w Linuksie są zrobione używając liczb oktalnych. Jest tak dlatego, że Linuksowskie uprawnienia opierają się na zdolności do odczytu, zapisu i wykonywania. Pierwsza cyfra jest uprawnieniem odczytu, druga cyfra jest uprawnieniem zapisu, i trzecia cyfra jest uprawnieniem wykonywania. Więc, 0 (000) nie daje żadnych uprawnień, 6 (110) daje uprawnienia odczytu i zapisu, i 5 (101) daje uprawnienia odczytu i wykonywania. Te liczby są potem używane dla trzech różnych zestawów uprawnień - właściciela, grupy, i innych. Liczba 0644 oznacza odczyt i zapis dla pierwszego zestawu uprawnień, i tylko odczyt dla drugiego i trzeciego zestawu. Pierwszy zestaw uprawnień jest dla właściciela pliku. Drugi zestaw uprawnień jest dla grupy właścicieli pliku. Ostatni zestaw uprawnień jest dla innych. Więc, 0751 oznacza, że właściciel pliku może czytać, zapisywać i wykonywać ten plik, członkowie grupy mogą czytać i wykonywać ten plik, a inni mogą tylko wykonywać tan plik.

Tak czy inaczej, jak możesz widzieć, cyfry oktalne są używane do grupowania bitów (cyfry binarne) w trójki. Sposób w jaki asembler wie, że liczba jest oktalną jest taki, że liczby oktalne są poprzedzone zerem. Na przykład 010 oznacza 10 oktalnie, co wynosi 8 decymalnie. Jeśli zapiszesz tylko 10 oznacza to 10 decymalnie. Początkowe zero jest tym co rozróżnia te dwie reprezentacje. Więc, *bądź ostrożny aby nie umieszczać prowadzących zer na początku liczb decymalnych, lub będą one zinterpretowane jako liczby oktalne!*

Liczby heksadecymalne (zwane także tylko "hex") używają liczb 1-15 dla wszystkich cyfr. Jednakże, ponieważ 10-15 nie

ma swoich własnych cyfr, liczby heksadecymalne używają liter od a do f aby je reprezentować. Na przykład, litera **a** reprezentuje 10, litera **b** reprezentuje 11, i tak dalej. 10 heksadecymalnie jest 16 decymalnie. Oktalnie, każda cyfra reprezentowała trzy bity. Heksadecymalnie, każda cyfra reprezentuje cztery bity. Każde dwie cyfry to pełny bajt, a osiem cyfr to 32-bitowe słowo. Więc widzisz, jest wyraźnie łatwiej zapisywać liczbę heksadecymalną niż liczbę binarną, ponieważ ma ona tylko ćwiartkę ilości cyfr liczby binarnej. Najważniejsza cyfra do zapamiętania w heksadecymalnych to f, co oznacza, że wszystkie bity są ustawione. Więc, jeśli chcę ustawić wszystkie bity rejestru na 1, mogę właśnie wykonać

movl $0xFFFFFFFF, %eax

Co jest wyraźnie łatwiejsze i mniej narażone na błędy niż zapisywanie

movl $0b11111111111111111111111111111111, %eax

Zauważ także, że liczby heksadecymalne są poprzedzone przez 0x. Więc, kiedy wykonujemy

int $0x80

Wywołujemy przerwanie numer 128 (8 grup 16), lub przerwanie numer 0b00000000000000000000000010000000.
Liczby heksadecymalne i oktalne zabierają trochę czasu aby się z nimi oswoić, ale są one bardzo często używane w programowaniu komputerowym. Wartościowe mogłoby być napisanie kilku liczb heksadecymalnie i spróbowanie konwersji ich do i z powrotem liczb binarnych , decymalnych, i oktalnych.

Porządek Bajtów w Słowie

Jedna rzecz która myli wiele osób kiedy zajmują się bitami i bajtami na niskim poziomie jest to, że kiedy bajty są zapisywane z rejestru do pamięci, te bajty są zapisywane najpierw-najmniej-znacząca-część. To czego większość ludzi oczekuje jest, że jeśli mają oni słowo w rejestrze, powiedzmy **0x5d 23 ef ee** (spacje są abyś mógł widzieć gdzie są bajty), bajty będą zapisane do pamięci w takim porządku. Jednakże, na procesorach x86, bajty są rzeczywiście zapisywane w odwrotnej kolejności. W pamięci te bajty mogą być **0xee ef 23 5d** na procesorze x86. Bajty są zapisywane w odwrotnej kolejności od tej w jakiej mogłyby się pojawiać, ale bity w bajtach są w zwykłej kolejności.
Nie wszystkie procesory zachowują się w taki sposób. Procesor x86 jest procesorem *little-endian*, co oznacza, że umieszcza "mały koniec", lub najmniej-znaczący bajt swojego słowa jako pierwszy.

Innymi procesorami są procesory *big-endian*, co oznacza, że przechowują one "wielki koniec", lub najbardziej znaczący bajt swojego słowa jako pierwszy, sposób w jaki moglibyśmy naturalnie czytać liczbę.

Ta różnica nie jest normalnie problemem (chociaż zaiskrzyła wieloma kontrowersjami technicznymi przez te lata). Ponieważ bajty są odwracane ponownie (lub nie, jeśli jest to procesor "big-endian") kiedy są czytane z powrotem do rejestru, programista nigdy zwykle nie zauważa jaki jest porządek bajtów. Magia przełączania bajtów dzieje się automatycznie poza sceną podczas transferu rejestr-pamięć. Jednakże, porządek bajtów może powodować problemy w kilku sytuacjach:
- Jeśli próbujesz wczytać kilka bajtów jednocześnie używając **movl** ale zajmujesz się nimi na bazie bajta po bajcie używając najmniej znaczącego bajta (t.j. - przez użycie **%al** i/lub przesunięcia rejestru), to będzie w różnym porządku niż pojawią się one w pamięci.
- Jeśli wczytujesz lub zapisujesz plik napisany dla różnych architektur, możesz być zmuszony brać pod uwagę w jakim porządku zapisują one swoje bajty.

- Jeśli wczytujesz lub zapisujesz do gniazd sieci, możesz być zmuszony brać pod uwagę porządek bajtów w protokole.

Dopóki jesteś świadomy tego, zwykle nie jest to wielki problem. Dla bardziej pogłębionego spojrzenie na sprawy porządku bajtów, powinieneś przeczytać DAV's Endian FAQ na http://www.rdrop.com/~cary/html/endian_faq.html, specjalnie artykuł "On Holy Wars and a Plea for Peace" Daniela Cohena.

Przekształcanie Liczb do Wyświetlania

Jak dotąd, nie byliśmy w stanie wyświetlić żadnej liczby przechowywanej do użytkownika, oprócz w znaczeniu ekstremalnie ograniczonym przekazywanie jej poprzez kody wyjścia. W tym podrozdziale, przedyskutujemy konwertowanie dodatnich liczb w łańcuch dla wyświetlania.
Funkcja będzie się nazywać **integer2string**, i będzie pobierać dwa parametry - liczbę całkowitą do konwersji i bufor łańcucha wypełniony znakami null (zerami). Bufor przyjmiemy, że będzie wystarczająco duży aby przechować daną liczbę jako łańcuch (co najmniej 11 znaków długi, włączając kończący znak null).
Pamiętaj, że sposób w jaki widzimy liczby jest podstawowo 10. Dlatego, aby dotrzeć do indywidualnych decymalnych cyfr liczby, potrzebujemy dzielić przez 10 i wyświetlać resztę dla każdej cyfry. Dlatego, proces będzie wyglądał tak:
- Podziel liczbę przez dziesięć.
- Resztą jest bieżąca cyfra. Zamień ją na znak i zachowaj ją.
- Skończyliśmy jeśli reszta wynosi zero.
- W przeciwnym razie, pobierz tę resztę i następną lokalizację w buforze i powtórz proces.
Jedynym problemem jest to, że odkąd ten proces zajmuje się pierwsze jednym miejscem, będzie on zostawiał liczbę wspak. Dlatego, będziemy musieli zakończyć poprzez odwrócenie znaków. Zrobimy to poprzez umieszczenie znaków na stosie kiedy obliczamy je. W ten sposób, kiedy zdejmujemy je z powrotem aby wypełnić bufor, będzie to w odwrotnej kolejności niż je odkładaliśmy.
Kod dla tej funkcji powinien być umieszczony w pliku zwanym **integer-to-string.s** i powinien się zaczynać następująco:

```
#CEL: Konwersja liczby całkowitej do łańcucha decymalnego dla wyświetlania
#
#WEJŚCIE: Bufor dostatecznie duży aby przechować największą możliwą liczbę
# Liczba całkowita do konwersji
#
#WYJŚCIE: Bufor będzie nadpisany przez łańcuch decymalny
#
#Zmienne:
# %ecx będzie przechowywał licznik znaków w procesie
# %eax będzie przechowywał bieżącą wartość
# %edi będzie przechowywał podstawę (10)
#
.equ ST_VALUE, 8
.equ ST_BUFFER, 12

.globl integer2string
.type integer2string, @function
integer2string:
#Zwykły początek funkcji
pushl %ebp
```

```
movl %esp, %ebp

#Bieżący licznik znaków
movl $0, %ecx

#Przesuwa wartość do pozycji
movl ST_VALUE(%ebp), %eax

#Kiedy dzielimy przez 10, 10 musi być w rejestrze lub w lokalizacji pamięci
movl $10, %edi

conversion_loop:
#Dzielenie jest aktualnie przedstawiane na rejestrze połączonym %edx:%eax, więc pierwsze czyścimy %edx
movl $0, %edx

#Dzieli %edx:%eax (co jest implementowane) przez 10.
#Przechowuje resztę z dzielenia w %eax i resztę w %edx (obydwie są implementowane).
divl %edi

#Reszta z dzielenia jest na właściwym miejscu. %edx ma resztę, która teraz powinna być konwertowana do liczby.
#Więc, %edx ma liczbę od 0 do 9. Mógłbyś także zinterpretować to jako indeks tablicy ASCII zaczynający się od znaku '0'.
#Kod ascii dla '0' plus zero jest ciągle kodem ascii dla '0'. Kod ascii dla '0' plus '1' jest kodem ascii dla znaku '1'.
#Dlatego, następująca instrukcja da nam znak dla liczby przechowywanej w %edx
addl $'0', %edx

#Teraz pobierzemy tę wartość i odłożymy ją na stosie.
#W ten sposób, kiedy skończymy, możemy tylko zdjąć znaki jeden po drugim i będą one w prawidłowej kolejności.
#Zauważ, że odkładamy cały rejestr, ale potrzebujemy tylko bajta w %dl (ostatni bajt rejestru %edx) dla tego znaku.
pushl %edx

#Inkrementacja licznika cyfr
incl %ecx

#Sprawdza czy %eax wynosi już zero, idzie do następnego kroku jeśli tak.
cmpl $0, %eax
je end_conversion_loop

#%eax ma już swoją nową wartość.

jmp conversion_loop

end_conversion_loop:
#Łańcuch jest teraz na stosie, jeśli zdejmiemy go po jednym znaku możemy go skopiować do bufora i zakończyć.

#Daje wskaźnik na bufor w %edx
movl ST_BUFFER(%ebp), %edx

copy_reversing_loop:
#Odłożyliśmy cały rejestr, ale potrzebujemy tylko ostatni bajt. Więc zamierzamy zdjąć cały rejestr %eax,
#ale potem przenieść tylko małą część (%al) do łańcucha znakowego.
```

```
popl %eax
movb %al, (%edx)

#Zmniejszenie %ecx więc wiemy kiedy skończyliśmy
decl %ecx
#Zwiększenie %edx tak, że będzie wskazywało na następny bajt
incl %edx
#Sprawdza czy skończyliśmy
cmpl $0, %ecx
#Jeśli tak, skok na koniec funkcji
je end_copy_reversing_loop
#W przeciwnym razie, powtarza pętlę
jmp copy_reversing_loop

end_copy_reversing_loop:
#Kopiowanie zakończone. Teraz zapis bajta null i powrót
movb $0, (%edx)

movl %ebp, %esp
popl %ebp
ret
```

Aby pokazać jak to działa w pełnym programie, użyj następującego kodu, wspólnie z funkcjami **count_chars** i **write_newline** napisanymi w poprzednich rozdziałach. Kod powinien być w pliku nazwanym **conversion-program.s**.

```
.include "linux.s"

.section .data

#To jest miejsce gdzie będzie to przechowywane
tmp_buffer:
.ascii "\0\0\0\0\0\0\0\0\0\0"

.section .text

.globl _start
_start:
movl %esp, %ebp

#Przechowalnia dla wyników
pushl $tmp_buffer
#Liczba do konwersji
pushl $824
call integer2string
addl $8, %esp

#Otrzymuje licznik znaków dla naszych wywołań systemowych
pushl $tmp_buffer
call count_chars
```

```
addl $4, %esp

#Licznik idzie w %edx dla SYS_WRITE
movl %eax, %edx

#Wykonuje wywołanie systemowe
movl $SYS_WRITE, %eax
movl $STDOUT, %ebx
movl $tmp_buffer, %ecx

int $LINUX_SYSCALL

#Zapisuje powrót karetki
pushl $STDOUT
call write_newline

#Wyjście
movl $SYS_EXIT, %eax
movl $0, %ebx
int $LINUX_SYSCALL
```

Aby zbudować ten program, wpisz następujące komendy:

```
as integer-to-string.s -o integer-to-number.o
as count_chars.s -o count_chars.o
as write_newline.s -o write_newline.o
as conversion-program.s -o conversion-program.o
ld integer-to-number.o count_chars.o write_newline.o conversion-program.o -o conversion-program
```

Aby uruchomić tylko wpisz **./conversion-program** i wyjście powinno powiedzieć **824**.

Przegląd

Znajomość Koncepcji

- Przekonwertuj liczbę decymalną 5294 na binarną.
- Jaką liczbę reprezentuje 0x0234aeff? Napisz ją binarnie, oktalnie i decymalnie.
- Dodaj liczby binarne 10111001 i 101011.
- Pomnóż liczby binarne 1100 i 1010110.
- Przekonwertuj wyniki dwu poprzednich zadań na decymalne.
- Opisz jak działają AND, OR, NOT, i XOR.
- Po co jest maskowanie?
- Jakiej liczby użyłbyś dla flag wywołania systemowego **open** jeśli chciałbyś otworzyć plik dla zapisu, i utworzenia pliku jeśli nie istnieje?
- Jak mógłbyś zaprezentować -55 w rejestrze trzydziestodwu bitowym?
- Rozszerz ze znakiem poprzednią wielkość do rejestru 64-bitowego.
- Opisz różnice przechowywania słów w pamięci pomiędzy "little-endian" i "big-endian".

Użycie Koncepcji

- Powróć do poprzednich programów które zwracały wyniki numeryczne poprzez kod statusu wyjścia, i przepisz je tak aby zamiast tego wypisywały wyniki używając naszej funkcji konwersji liczb całkowitych do łańcucha.
- Zmodyfikuj kod **integer2string** aby zwracał wyniki w liczbach oktalnych raczej niż decymalnych.
- Zmodyfikuj kod **integer2string** tak, że podstawą konwersji jest raczej parametr niż liczba na stałe zapisana.
- Napisz funkcję zwaną **is_negative** która pobiera pojedynczą liczbę całkowitą jako parametr i zwraca 1 jeśli ten parametr jest ujemny, a 0 jeśli parametr jest dodatni.

Idąc Dalej

- Zmodyfikuj kod **"integer2string** tak, że podstawa konwersji może być większa niż 10 (to wymaga od ciebie użycia liter dla cyfr przekraczających 9).
- Utwórz funkcję odwracającą **integer2string** zwaną **number2integer** która pobiera znak łańcucha i konwertuje go do liczby całkowitej rozmiaru rejestru. Przetestuj ją przez uruchomienie tej liczby całkowitej z powrotem z funkcji **integer2string** i wyświetlając wyniki.
- Napisz program który przechowuje rzeczy lubiane i nie lubiane w pojedynczym słowie maszyny, a potem porównuje dwa zestawy lubianych i nie lubianych rzeczy.
- Napisz program który czyta łańcuch znaków z STDIN i konwertuje je w liczbę.

Rozdział 11. Języki Wysokiego Poziomu

W tym rozdziale zaczniemy przyglądać się naszemu pierwszemu językowi programowania "świata rzeczywistego". Język asemblerowy jest językiem używanym na poziomie maszynowym, ale większość ludzi znajduje kodowanie w języku asemblerowym zbyt ciężkim dla codziennego użycia. Wiele języków komputerowych było wynalezionych aby uczynić zadanie programowania łatwiejszym. Znajomość szerokiej gamy języków jest użyteczna z wielu powodów, włączając w to
- Różne języki bazują na różnych koncepcjach, które pomogą nauczyć się różnych i lepszych metod programistycznych i idei.
- Różne języki są dobre dla różnych typów projektów.
- Różne firmy mają różne standardy języków, więc znajomość większej liczby języków czyni twoje zdolności bardziej sprzedawalnymi.
- Im więcej języków znasz, tym łatwiej poznać nowy język.

Jako programista, będziesz często musiał poznawać nowe języki. Profesjonalni programiści mogą zwykle poznać nowy język w ciągu tygodni studiowania i praktyki. Języki są prostymi narzędziami, a uczenie się używania nowego narzędzia nie powinno być czymś czego programista unika. W rzeczywistości, jeśli wykonujesz konsultacje komputerowe będziesz często musiał uczyć się nowych języków na bieżąco żeby utrzymać się w pracy. To często będzie twój klient, nie ty, kto decyduje jaki język jest używany. Ten rozdział wprowadzi cię do kilku języków dostępnych dla ciebie. Zachęcam cię do zgłębiania tak wielu języków jak wieloma jesteś zainteresowany. Ja osobiście próbuję nauczyć się nowego języka co kilka miesięcy.

Języki Kompilowane i Interpretowane

Wiele języków jest językami *kompilowanymi*. Kiedy piszesz w języku asemblerowym, każda instrukcja którą napiszesz jest tłumaczona na dokładnie jedną instrukcję maszynową dla procesowania. Z kompilatorami, instrukcja może być tłumaczona w jedną lub w setki instrukcji maszynowych. Faktycznie, zależnie od tego jak zaawansowany jest twój kompilator, może on nawet restrukturyzować części twojego kodu aby zrobić go szybszym. W języku asemblerowym co napiszesz to otrzymasz. Są także języki które są językami *interpretowanymi*. Te języki wymagają żeby użytkownik uruchomił program zwany *interpreterem* który z kolei uruchamia dany program. Są one zwykle wolniejsze niż programy kompilowane, ponieważ

interpreter musi wczytać i zinterpretować ten kod jak on idzie w całości. Jednakże, w dobrze zrobionych interpreterach, ten czas może być całkiem pomijalny. Jest także klasa języków hybrydowych które częściowo kompilują program przed wykonaniem w bajtkody. To jest robione ponieważ interpreter może czytać bajtkody znacznie szybciej niż czytać język regularny.
Jest wiele powodów wybierania tego czy innego. Programy kompilowane są miłe, ponieważ nie musisz mieć zainstalowanego interpretera na maszynie użytkownika. Musisz mieć kompilator dla tego języka, ale użytkownicy twojego programu nie muszą. W językach interpretowanych, musisz być pewien, że użytkownik ma zainstalowany interpreter dla twojego programu, i że komputer wie z którym interpreterem uruchomić twój program. Jednakże, języki interpretowane maja tendencję do bycia bardziej elastycznymi, podczas gdy języki kompilowane są bardziej sztywne.
Wybór języka jest zwykle podyktowany dostępnymi narzędziami i wsparciem dla metod programistycznych raczej niż tym czy jest to język kompilowany lub interpretowany. Faktycznie wiele języków ma opcje dla obydwu.
Języki wysokiego poziomu, kompilowane czy interpretowane, są zorientowane wokół ciebie, programisty, zamiast wokół maszyny. To otwiera je na szeroką gamę możliwości, które mogą zawierać następujące:
- Zdolność do grupowania wielu operacji w pojedyncze wyrażenie.
- Zdolność do używania "wielkich wartości" - wartości które są o wiele większe niż słowo 4-bajtowe które komputer normalnie przetwarza (na przykład, zdolność do widzenia łańcuchów tekstowych jako pojedynczej wartości raczej niż jako łańcucha bajtów).
- Posiadanie dostępu do lepszych konstrukcji warunkowych niż tylko skoki.
- Posiadanie kompilatora do sprawdzania typów podanych wartości i innych żądań.
- Posiadanie pamięci zarządzanej automatycznie.
- Zdolność do pracy w języku który jest podobny do istoty problemu raczej niż do komputerowego sprzętu.
Więc dlaczego ktoś przedkłada jeden język nad inne? Na przykład, wielu wybiera Perl ponieważ posiada rozległą bibliotekę funkcji do zarządzania niemal każdym protokołem lub typem danych na tej planecie. Python, jednakże, ma klarowniejszą składnię i często nadaje się do bardziej zrozumiałych rozwiązań. Jego między platformowe narzędzia GUI są także wyśmienite. PHP czyni pisanie aplikacji sieciowych prostymi. Common LISP ma więcej mocy i możliwości niż jakiekolwiek inne środowisko dla tych którzy życzą sobie nauczyć się tego. Scheme jest przykładem prostoty i mocy połączonych razem. C jest łatwy do łączenia z innymi językami.
Każdy język jest inny, i im więcej języków znasz tym lepszym programistą będziesz. Znajomość koncepcji różnych języków pomoże tobie w całym programowaniu, ponieważ możesz lepiej dopasować język programowania do problemu, i masz większy zestaw narzędzi pracy. Nawet jeśli szczególne możliwości nie są bezpośrednio wspierane w języku którego używasz, często mogą one być symulowane. Jednakże, jeśli nie masz szerokiego doświadczenia z językami, nie będziesz znał wszystkich możliwości z których musisz wybrać.

Twój Pierwszy Program C

Tutaj jest twój pierwszy program C, który wypisuje "Hello world" na ekranie i wychodzi. Zapisz go, i nadaj mu nazwę **Hello-World.c**

```
#include < stdio.h >

/*CEL: Ten program jest pokazem podstaw programu C.*/
/*Wszystko co robi to wypisuje "Hello World!" na ekran i wychodzi.*/

/* Główny Program */
int main(int argc, char **argv)
{
/* Wypisuje nasz łańcuch do standardowego wyjścia */
```

```
puts("Hello World!\n");

/* Wyjście ze statusem 0 */
return 0;
}
```

Jak widzisz, jest to bardzo prosty program. Aby go skompilować, uruchom komendę

```
gcc -o HelloWorld Hello-World.c
```

Aby uruchomić ten program, zrób

```
./HelloWorld
```

Zobaczmy jak ten program był składany.
Komentarze w C zaczynają się od **/*** a kończą ***/**. Komentarze mogą wiązać kilka wierszy, ale wiele osób preferuje zaczynanie i kończenie komentarzy w tym samym wierszu żeby nie powodować omyłek.
#include < stdio.h > jest pierwszą częścią programu. Jest to *dyrektywa preprocesora*. Kompilacja C jest podzielona na dwa stopnie - preprocesor i główny kompilator. Ta dyrektywa mówi preprocesorowi aby szukał pliku **stdio.h** i wkleił go do twojego programu. Preprocesor jest odpowiedzialny za złożenie tekstu programu razem. To zawiera zlepianie różnych plików razem, uruchamianie makr w tekście twojego programu, itd. Po tym jak tekst jest złożony razem, preprocesor kończy działanie i główny kompilator przystępuje do działania.
Teraz, wszystko z **stdio.h** jest obecnie w twoim programie tak jakbyś sam tam to wpisał. Nawiasy kątowe wokół nazwy pliku mówią kompilatorowi aby szukał tego pliku w swoich standardowych ścieżkach (**/usr/include** i **/usr/local/include**, zwykle). Jeśli byłoby to w cudzysłowie, jak **#include "stdio.h"** mógłby szukać tego pliku w bieżącym katalogu.
Jakkolwiek, **stdio.h** zawiera deklaracje dla funkcji standardowego wejścia i wyjścia i zmiennych. Te deklaracje mówią kompilatorowi jakie funkcje są dostępne dla wejścia i wyjścia. Następne kilka wierszy jest prostym komentarzem o tym programie.
Potem jest wiersz **int main(int argc, char **argv)**. To jest początek funkcji. Funkcje C są deklarowane ze swoimi nazwami, argumentami i typami powrotu. Ta deklaracja mówi, że nazwą funkcji jest **main**, zwraca **int** (liczba całkowita - długości 4 bajtów na platformie x86), i ma dwa argumenty - **int** zwany **argc** i **char **** zwany **argv**. Nie musisz się martwić o to gdzie te argumenty są umieszczone na stosie - kompilator C opiekuje się tym dla ciebie. Nie musisz także się martwić o ładowanie wartości do i z rejestrów ponieważ kompilator opiekuje się tym, także.
Funkcja **main** jest funkcją specjalną w języku C - to jest początek wszystkich programów C (podobnie jak **_start** w naszych programach języka asemblerowego). Zawsze pobiera dwa parametry. Pierwszy parametr jest liczbą argumentów dawanych tej komendzie, a drugi parametr jest listą argumentów które były dane.
Następny wiersz jest wywołaniem funkcji. W języku asemblerowym, musiałeś odłożyć argumenty funkcji na stos, i wtedy wywołać funkcję. C opiekuje się tym kompleksowo dla ciebie. Musisz po prostu wywołać tę funkcję z parametrami w nawiasach. W tym przypadku, wywołujemy funkcję **puts**, z pojedynczym parametrem. Ten parametr jest łańcuchem znakowym który chcemy wypisać. Musimy właśnie wpisać ten łańcuch w cudzysłowie, a kompilator zaopiekuje się definiowaniem miejsca przechowywania i przenoszeniem wskaźników do tego miejsca na stos przed wywołaniem funkcji. Jak możesz zobaczyć, jest tu o wiele mniej pracy.
Ostatecznie nasza funkcja zwraca liczbę **0**. w języku asemblerowym, umieszczaliśmy naszą wartość powrotu w **%eax**, ale w C tylko używamy komendy **return** i przejmuje ona opiekę nad tym za nas. Wartość powrotu funkcji **main** jest tym co było użyte jako kod wyjścia dla tego programu.
Jak widzisz, używając języków wysokiego poziomu znacznie ułatwia życie. Pozwala to także naszym programom łatwiej się uruchamiać na wielorakich platformach. W języku asemblerowym, twój program jest przywiązany do obydwu systemu operacyjnego i platformy sprzętowej, podczas gdy w językach kompilowanych i interpretowanych ten sam kod może zwykle być uruchomiony na różnorodnych systemach operacyjnych i platformach sprzętowych. Na przykład, ten program

może być zbudowany i wykonywany na sprzęcie x86 z Linuksem, Windowsem, UNIX-em, lub z większością innych systemów operacyjnych.
Dodatkowe informacje o języku programowania C mogą być znalezione w Dodatku E.

Perl

Perl jest językiem interpretowanym, egzystującym głównie na platformach Linuksowych i bazujących na UNIX-ie. W rzeczywistości uruchamia się na prawie wszystkich platformach, ale znajdziesz go najczęściej na Linuksie i systemach bazujących na UNIX-ie. Tutaj jest wersja Perlowska tego programu, która powinna być wpisana do pliku nazwanego **Hello-World.pl**:

```
#!/usr/bin/perl

print("Hello world!\n");
```

Ponieważ Perl jest interpretowany, nie musisz go kompilować i linkować. Tylko uruchom go następującą komendą:

```
perl Hello-World.pl
```

Jak widzisz, wersja Perlowska jest nawet krótsza niż wersja C. W Perlu nie musisz deklarować żadnych funkcji lub punktów otwarcia programu. Możesz właśnie rozpocząć wpisując komendy i interpreter uruchomi je jak przyjdzie do nich. w rzeczywistości ten program ma tylko dwa wiersze kodu, jeden z nich jest opcjonalny.
Pierwszy, opcjonalny wiersz jest użyty dla maszyn UNIX-owych aby powiedzieć którego interpretera użyć żeby uruchomić ten program. **#!** mówi komputerowi, że to jest interpretowany program, a **/usr/bin/perl** mówi komputerowi żeby użył programu **/usr/bin/perl** do zinterpretowania tego programu. Jednakże, ponieważ uruchomiliśmy program przez wypisanie **perl Hello-World.pl**, już określiliśmy, że używaliśmy interpretera Perla.
Następny wiersz wywołuje funkcję wbudowaną Perla, "print". Ma ona jeden parametr, łańcuch do wypisania. Ten program nie ma jawnej komendy powrotu - wie on żeby powrócić po prostu ponieważ osiąga koniec pliku. On także wie żeby zwrócić 0 ponieważ nie było błędów podczas uruchamiania. Widzisz, że języki interpretowane są często skupione na dostarczeniu ci działającego kodu tak szybko jak to możliwe, bez wykonywania mnóstwa dodatkowej pracy.
Jedną rzeczą w Perlu, które nie jest tak ewidentne w tym przykładzie, jest to, że Perl traktuje łańcuchy jako pojedyncze wartości. W języku asemblerowym, musieliśmy programować zgodnie z architekturą pamięci komputera, co oznaczało, że łańcuchy musiały być traktowane jako sekwencja wielu wartości, z wskaźnikiem na pierwszą literę. Perl udaje, że łańcuchy mogą być przechowywane bezpośrednio jako wartości, i tak ukrywa komplikację manipulowania nimi przed tobą. W rzeczywistości, jedną z głównych zalet Perla jest jego zdolność i szybkość w manipulowaniu tekstem.

Python

Wersja Pythona tego programu wygląda prawie dokładnie tak jak Perla. Jednakże, Python jest rzeczywiście bardzo różnym językiem w porównaniu do Perla, nawet jeśli na to nie wygląda z tego trywialnego przykładu. Wpisz ten program do pliku nazwanego **Hello-World.py**. Program wygląda następująco:

```
#!/usr/bin/python

print "Hello World"
```

Powinieneś być w stanie powiedzieć co różne wiersze programu robią.

Przegląd

Znajomość Koncepcji
- Jaka jest różnica pomiędzy językiem interpretowanym i językiem kompilowanym?
- Jakie powody mogą skłaniać cię do nauki nowych języków programowania?

Użycie Koncepcji
- Naucz się podstaw składni nowego języka programowania. Przekoduj jeden z programów z tej książki w tym języku.
- W tym programie który napisałeś w pytaniu powyższym, jakie specyficzne rzeczy były zautomatyzowane w tym języku programowania który wybrałeś?
- Zmodyfikuj swój program tak, że uruchamia się on 10000 razy z rzędu, w języku asemblerowym i w twoim nowym języku. Wtedy uruchom komendę **time** aby zobaczyć który jest szybszy. Który był szybszy? Dlaczego myślisz, że tak jest?
- Jak metody wejścia/wyjścia języka programowania różnią się od tych z wywołań systemowych Linuksa?

Idąc Dalej
- Widząc języki które mają taką zwięzłość jak Perl, dlaczego, myślisz, ta książka zaczęła się od języka tak dosłownego jak język asemblerowy?
- Jak języki wysokiego poziomu udają proces programowania, co myślisz?
- Dlaczego występuje tak wiele języków, co myślisz?
- Naucz się dwu nowych języków wysokiego poziomu. Jak one różnią się między sobą? Na ile są one podobne? Jakie przybliżenie do rozwiązania problemu każdy z nich osiąga?

Rozdział 12. Optymalizacja

Optymalizacja jest to proces sprawiania, że twoja aplikacja uruchamia się bardziej efektywnie. Możesz optymalizować wiele rzeczy - szybkość, używanie przestrzeni pamięci, używanie przestrzeni dysku, itd. Ten rozdział, jednak, skupia się na optymalizacji szybkości.

Kiedy Optymalizować

Lepiej jest nie optymalizować w ogóle niż optymalizować zbyt szybko. Kiedy optymalizujesz, twój kod generalnie staje się mniej przejrzysty, ponieważ staje się bardziej kompleksowy. Czytający twój kod będą mieli więcej kłopotu odkrywając dlaczego zrobiłeś to co zrobiłeś co zwiększy koszt utrzymania twojego projektu. Nawet kiedy wiesz jak i dlaczego twój program uruchamia się w sposób w jaki się uruchamia, zoptymalizowany kod jest trudniejszy do debugowania i rozszerzania. Spowalnia on proces rozwoju znacząco, zarówno z powodu czasu zabieranego przez optymalizację kodu, i czasu zabieranego na modyfikację twojego zoptymalizowanego kodu.
Podsumowanie tego problemu jest takie, że nawet nie wiesz zawczasu gdzie będą punkty mające wpływ na szybkość w twoim programie. Nawet doświadczeni programiści mają kłopot z przewidywaniem które części programu będą wąskim gardłem wymagającym optymalizacji, więc prawdopodobnie skończysz marnotrawiąc swój czas optymalizując niewłaściwe części. Podrozdział nazwany *Gdzie Optymalizować* będzie omawiać jak znaleźć części twojego programu które potrzebują optymalizacji.

Kiedy udoskonalasz swój program, potrzebujesz mieć następujące priorytety:
- Wszystko jest udokumentowane
- Wszystko działa jak udokumentowano
- Kod jest napisany w modularnej, łatwo modyfikowalnej formie
Dokumentacja jest kluczowa, zwłaszcza kiedy pracujemy w grupach. Prawidłowe funkcjonowanie programu jest kluczowe. Zauważysz, że szybkości aplikacji nie ma na żadnej takiej liście. Optymalizacja nie jest potrzebna podczas wczesnego rozwoju z następujących powodów:
- Mniejsze problemy z szybkością mogą być zwykle rozwiązane poprzez sprzęt, co jest znacznie tańsze niż czas programisty.
- Twoja aplikacja zmieni się dramatycznie kiedy zrewidujesz ją, dlatego to strata większości twoich zasobów na optymalizowanie jej teraz.
- Problemy szybkości są zwykle zlokalizowane w kilku miejscach twojego kodu - odnalezienie ich jest trudne przed ukończeniem większości programu.
Dlatego, czas na optymalizację jest bliżej końca projektu, kiedy odkryłeś, że twój poprawny kod obecnie ma problemy wykonawcze.
W projekcie sieciowym, komercyjnym (e-commerce) w który byłem zaangażowany, skupiłem się kompletnie na poprawności. Było to przedmiotem dużej obawy moich kolegów, którzy byli przestraszeni faktem, że każda strona zabiera dwanaście sekund na procesowanie zanim nawet zacznie się ładować (większość stron sieciowych uruchamia się poniżej sekundy). Jednakże, byłem zdeterminowany zrobić to w sposób właściwy najpierw, i umieścić optymalizację jako ostatni priorytet. Kiedy kod był ostatecznie poprawny po 3 miesiącach pracy, znalezienie i eliminacja wąskich gardeł zajęła tylko trzy dni, przynosząc średni czas procesowania poniżej ćwiartki sekundy. Przez skupienie się na prawidłowym porządku, byłem w stanie ukończyć projekt który był zarówno poprawny jak i wydajny.

Gdzie Optymalizować

Kiedy już stwierdziłeś, że masz problem wykonawczy powinieneś sprawdzić gdzie w kodzie pojawiają się te problemy. Możesz to zrobić przez uruchomienie *profilera*. "Profiler" jest programem który pozwoli ci uruchomić twój program, i powie jak wiele czasu jest spędzane w każdej funkcji, i jak wiele razy są one uruchamiane. **gprof** jest standardowym narzędziem "profilingu" GNU/Linux, ale dyskusja o używaniu "profilerów" jest poza przedmiotem tego tekstu. Po uruchomieniu "profilera", możesz stwierdzić które funkcje są wywoływane najczęściej lub mają najwięcej czasu spędzonego w nich. To są te na których powinieneś skupić swoje optymalizacyjne trudy.
Jeśli program spędza tylko 1% swojego czasu w danej funkcji, wtedy nie ma znaczenia jak bardzo ją przyspieszysz osiągniesz tylko *maksymalnie* 1% poprawy całej szybkości. Jednakże, jeśli program spędza 20% swojego czasu w danej funkcji, wtedy nawet mniejsza poprawa szybkości tej funkcji będzie zauważalna. Dlatego, "profiling" daje ci informację której potrzebujesz aby dokonać dobrych wyborów gdzie spędzić swój programistyczny czas.
W przypadku optymalizowania funkcji, powinieneś rozumieć w jaki sposób są one wywoływane i używane. Im więcej wiesz o tym jak i kiedy funkcja jest wywoływana, tym lepsza będzie twoja pozycja w optymalizowaniu jej odpowiednio. Są dwie główne kategorie optymalizacji - optymalizacja lokalna i optymalizacja globalna. Optymalizacja lokalna zawiera optymalizacje które są zarówno specyficznie sprzętowe - tak jak najszybszy sposób przeprowadzenia danego obliczenia - jak i specyficzne programowo - tak jak budowanie specyficznych części kodu działającego najlepiej dla najczęściej pojawiających się przypadków. Na przykład, jeśli próbujesz znaleźć najlepszy sposób dla trzech osób w różnych miastach aby spotkali się w St.Louis, optymalizacją lokalną mogłoby być znalezienie lepszej drogi aby się tam dostać, podczas gdy optymalizacją globalną mogłoby być zdecydowanie się na telekonferencję zamiast osobistego spotkania. Optymalizacja globalna często obejmuje restrukturyzację kodu aby uniknąć problemów wykonywania, niż raczej próbowanie znalezienia najlepszego sposobu wykonywania go.

Optymalizacje Lokalne

Kilka następujących metod to dobrze znane metody optymalizowania części kodu. Kiedy używasz języków wysokiego poziomu, niektóre z nich mogą być wykonywane automatycznie przez twój optymalizator kompilatora.

Kalkulacje Przedobliczeniowe
Czasami funkcja ma ograniczoną liczbę możliwych wejść i wyjść. W rzeczywistości, może być ich tak niewiele, że możesz przeliczyć wszystkie możliwe odpowiedzi przed, i po prostu sprawdzić odpowiedź kiedy funkcja jest wywołana. To zajmuje trochę miejsca ponieważ musisz przechowywać wszystkie odpowiedzi, ale dla małego zestawu danych działa to całkiem dobrze, specjalnie jeśli obliczenia normalnie zabierają długi czas.

Zapamiętywanie Wyników Obliczeń
Jest to podobne do poprzedniej metody, ale zamiast obliczania wyników przed, wynik każdego żądanego obliczenia jest przechowywany. W ten sposób kiedy funkcja się rozpoczyna, jeśli wynik był obliczony przed będzie po prostu zwracana poprzednia odpowiedź, w przeciwnym razie będzie wykonywane pełne obliczenie i przechowanie wyniku do późniejszego wglądu. To ma tę zaletę, że wymaga mniej miejsca ponieważ nie obliczasz wszystkich wyników. Czasami jest to nazywane *"caching"* lub *"memoizing"*.

Lokalność Odniesienia
Lokalność odniesienia jest terminem na to gdzie są w pamięci elementy danych do których masz dostęp. Z pamięcią wirtualną, możesz mieć dostęp do stron pamięci które są przechowywane na dysku. W takim przypadku, system operacyjny musi załadować tę stronę pamięci z dysku, i przenieść inne na dysk. Powiedzmy, na przykład, że system operacyjny pozwoli mieć 20k pamięci w fizycznej pamięci i nakłania jej resztę żeby była na dysku, a twoja aplikacja używa 60k pamięci. Powiedzmy, że twój program musi wykonać 5 operacji na każdej części danych. Jeśli wykona on jedną operację na każdej części danych, i wtedy idzie dalej i wykonuje następną operację na każdej części danych, ewentualnie każda strona danych będzie ładowana z dysku i na dysk 5 razy. Zamiast tego, jeśli wykonałbyś wszystkie 5 operacji na zadanym elemencie danych, musiałbyś tylko załadować każdą stronę z dysku raz. Kiedy zbierzesz tak wiele operacji na danej która jest fizycznie blisko do innej w pamięci, wtedy korzystasz z zalet lokalności odniesienia. W dodatku, procesory zwykle przechowują niektóre dane na "chipie" w "cache". Jeśli trzymasz wszystkie swoje operacje w granicach małego obszaru pamięci fizycznej, twój program może nawet zignorować główną pamięć i używać tylko "chipowej" ultra-szybkiej pamięci "cache". To wszystko jest robione za ciebie - wszystko co musisz zrobić to próbować operować na małych sekcjach pamięci za jednym razem, niż raczej zbierać wszystko razem.

Użycie Rejestru
Rejestry są najszybszymi lokalizacjami pamięci na komputerze. Kiedy dostajesz się do pamięci, procesor musi czekać aż jest ona załadowana z magistrali pamięci. Jednakże, rejestry są zlokalizowane na samym procesorze, więc dostęp jest ekstremalnie szybki. Dlatego robienie mądrego użytku z rejestrów jest nadzwyczaj ważne. Jeśli masz wystarczająco niedużo elementów danych z którymi pracujesz, spróbuj umieścić je wszystkie w rejestrach. W językach wysokiego poziomu, nie zawsze masz tę opcję - kompilator decyduje co idzie do rejestrów a co nie.

Funkcje Liniowane
Funkcje są wielkie z punktu widzenia zarządzania programem - ułatwiają one podział twojego programu na niezależne, zrozumiałe i re używalne części. Jednakże, wywołania funkcji angażują pominięcie odkładanych argumentów na stosie i wykonywanie skoków (przypomnij sobie lokalność odniesienia - twój kod może być przeniesiony na dysk zamiast do pamięci). Dla języków wysokiego poziomu, jest często niemożliwe dla kompilatora zrobienie optymalizacji poza granicami wywołań funkcji. Jednakże, niektóre języki wspierają funkcje liniowane lub funkcje makro. Te funkcje wyglądają, pachną, smakują i działają jak funkcje rzeczywiste, oprócz tego, że kompilator ma opcję prostego włączenia kodu dokładnie tam gdzie funkcja była wywołana. To robi program szybszym, ale także zwiększa rozmiar kodu. Jest także wiele funkcji, jak funkcje rekursywne, które nie mogą być uliniowione ponieważ wywołują się zarówno bezpośrednio jak i pośrednio.

Instrukcje Zoptymalizowane
Często jest wiele instrukcji języka asemblerowego które służą do tego samego celu. Doświadczony programista języka asemlerowego wie które instrukcje są najszybsze. Jednakże, to może się zmieniać z procesora na procesor. Dla uzyskania więcej informacji na ten temat, musisz zobaczyć podręcznik użytkownika który jest udostępniany dla specyficznego chipu którego używasz. Jako przykład, zobaczmy na proces ładowania liczby 0 do rejestru. Na większości procesorów, wykonanie **movl $0, %eax** nie jest najszybszym sposobem. Najszybszym sposobem jest xor-owanie rejestru z samym sobą, **xorl %eax, %eax**. Jest tak dlatego, że to musi mieć tylko dostęp do rejestru, a nie musi przesyłać żadnych danych. Dla użytkowników języków wysokiego poziomu, kompilator zarządza tymi rodzajami optymalizacji dla ciebie. Dla programistów języka asemlerowego, powinieneś znać dobrze swój procesor.

Tryby Adresowania
Różne tryby adresowania działają z różną prędkością. Najszybsze są tryby adresowania natychmiastowy i rejestrowy. Bezpośredni jest następny, pośredni po nim a wskaźnika bazowego i pośredni indeksowany są najwolniejsze. Próbuj używać szybszych trybów adresowania, kiedy tylko to jest możliwe. Jedną interesującą konsekwencją tego jest to, że kiedy masz strukturyzowaną część pamięci do której się dostajesz używając adresowania wskaźnika bazowego, pierwszy element może być osiągnięty najszybciej. Ponieważ jego przesunięcie wynosi 0, możesz osiągnąć go używając adresowania pośredniego zamiast adresowania wskaźnika bazowego, który robi to szybciej.

Wyrównywanie Danych
Niektóre procesory mogą mieć szybszy dostęp do danych w wyrównanych do słowa granicach pamięci (t.j. - adresach podzielnych przez rozmiar słowa) niż do danych niewyrównanych. Więc, podczas ustawiania struktur w pamięci, najlepiej jest utrzymywać je wyrównane do słowa. Niektóre nie-x86 procesory, w rzeczywistości, nie mogą mieć dostępu do niewyrównanych danych w niektórych trybach.

To są tylko niektóre możliwe przykłady rodzajów optymalizacji lokalnych. Jednakże, pamiętaj, że utrzymywalność i czytelność kodu jest znacznie bardziej ważna oprócz warunków ekstremalnych.

Optymalizacja Globalna

Optymalizacja globalna ma dwie zalety. Pierwszą jest ułożenie twojego kodu w formie gdzie łatwo jest przeprowadzać optymalizacje lokalne. Na przykład, jeśli masz wielką procedurę która wykonuje kilka powolnych, kompleksowych obliczeń, mógłbyś zobaczyć czy możesz podzielić części tej procedury na ich własne funkcje gdzie te wartości mogą być obliczone wcześniej lub zapamiętane.
Funkcje nieustalone (funkcje które operują tylko na parametrach które były im przekazane - t.j. nie globalne lub wywołania systemowe) są najłatwiejszym typem funkcji do optymalizacji w komputerze. Im więcej masz części nieustalonych w twoim programie, tym więcej masz możliwości do optymalizacji. W sytuacji e-commerce o której pisałem wcześniej, komputer musiał znaleźć wszystkie powiązane części dla specyficznej listy elementów. To wymagało około 12 wywołań bazodanowych, i w najgorszym przypadku zajmowało około 20 sekund. Jednakże, zaletą tego programu była interaktywność, i długi czas oczekiwania mógł zniszczyć tę zaletę. Jednakże, wiedziałem, że takie konfiguracje list nie zmieniają się. Dlatego, przekonwertowałem wywołania bazodanowe w ich własne funkcje, które były nieustalone. Wtedy byłem w stanie zapamiętywać te funkcje. Na początku każdego dnia, wyniki funkcji były kasowane aby uniknąć tego, że ktoś mógł je zmienić, i kilkanaście elementów listy było automatycznie wstępnie ładowanych. Od tego momentu podczas dnia, pierwszy raz gdy ktoś dostawał się do elementu listy, mogło to zabrać 20 sekund na załadowanie, ale potem zabierało mniej niż sekundę ponieważ wyniki bazy danych były zapamiętywane.
Optymalizacje globalne zwykle często wymagają osiągnięcia następujących właściwości w twoich funkcjach:

Paralelizacja
Paralelizacja oznacza, że twój algorytm może efektywnie być podzielony na różnorodne procesy. Na przykład, ciąża nie jest bardzo paralelizowalna ponieważ nie ważne jak wiele weźmiemy kobiet, zabiera on ciągle dziewięć miesięcy. Jednakże, budowanie samochodów jest paralelizowalne ponieważ możesz mieć jednego pracownika pracującego nad silnikiem podczas gdy inny pracuje nad wystrojem wnętrza. Zwykle, aplikacje mają granicę jak bardzo są paralelizowalne. Im bardziej paralelizowalna jest aplikacja, tym większe korzyści może odnieść z konfiguracji multiprocesorowej i klastrowej komputera.

Nieustaloność
Jak omawialiśmy, nieustalone funkcje i programy są tymi które polegają kompletnie na danych jawnie przekazanych im dla funkcjonowania. Większość procesów nie jest całkowicie nieustalona, ale mogą być w pewnych granicach. W moim przykładzie e-commerce, funkcja nie była całkowicie nieustalona, ale była w granicach pojedynczego dnia. Dlatego, zoptymalizowałem ją tak jakby była funkcją nieustaloną, ale pozwalała na zmiany w nocy. Dwa wielkie udogodnienia wynikające z nieustaloności są takie, że większość nieustalonych funkcji jest paralelizowalna i częste są korzyści z zapamiętywania.

Optymalizacja globalna wymaga dość sporej praktyki aby wiedzieć co działa a co nie. Decydując jak traktować problemy optymalizacji w kodzie wymaga spojrzenia na wszystkie wyjścia, i wiedzy, że poprawienie niektórych właściwości może spowodować inne.

Przegląd

Znajomość Koncepcji
- Na jakim poziomie ważności znajduje się optymalizacja w porównaniu z innymi priorytetami w programowaniu?
- Jaka jest różnica pomiędzy lokalna i globalną optymalizacją?
- Wymień kilka typów optymalizacji lokalnych?
- Jak stwierdzisz które części twojego programu wymagają optymalizacji?
- Na jakim poziomie ważności znajduje się optymalizacja w porównaniu z innymi priorytetami w programowaniu?
Jak sądzisz dlaczego powtórzyłem to pytanie?

Użycie Koncepcji
- Przejdź każdy program z tej książki i spróbuj zrobić optymalizacje zgodnie z procedurami zarysowanymi w tym rozdziale
- Weź jakiś program z poprzedniego ćwiczenia i spróbuj obliczyć wpływ wykonywania na twój kod pod specyficznymi wejściami.

Idąc Dalej
- Znajdź program open-source który uważasz za specjalnie szybki. Skontaktuj się z jednym z projektantów i zapytaj jakie rodzaje optymalizacji zastosowali dla poprawy szybkości.
- Znajdź program open-source który uważasz za szczególnie wolny, i spróbuj sobie wyobrazić powody tej powolności. Potem, ściągnij ten kod i spróbuj sprofilować go używając **gprof** lub podobnego narzędzia. Znajdź gdzie kod spędza najwięcej czasu i spróbuj zoptymalizować go. Czy powód powolności był różny niż sobie wyobraziłeś?
- Czy kompilator wyeliminował potrzebę optymalizacji lokalnych? Dlaczego lub dlaczego nie?
- Jaki rodzaj problemów mógłby uruchamiać kompilator jeśli spróbowałby optymalizować kod poza granice wywołania funkcji?

Rozdział 13. Perspektywy

Gratuluję dojścia tak daleko. Powinieneś teraz mieć podstawy do zrozumienia zagadnień związanych z wieloma obszarami programowania. Nawet jeśli nigdy nie użyjesz języka asemblerowego znowu, uzyskałeś wartościową perspektywę i mentalny obraz dla zrozumienia reszty nauki komputerowej.
Głównie są trzy metody uczenia się programowania:
- Od Podstaw Wzwyż - To jest tak jak uczy ta książka. Zaczyna od programowania niskopoziomowego, i zmierza ku bardziej generalnemu nauczaniu.
- Z Wierzchołka W Dół - Jest to kierunek przeciwny. Skupia się na tym co chcesz zrobić na komputerze, i uczy jak zagłębić się coraz niżej aż osiągniesz niskie poziomy.
- Od Środka - Jest to charakterystyczne dla książek które uczą specyficznego języka programowania lub API. Nie są one tak skoncentrowane na koncepcjach jak na specyfikacjach.
Różni ludzie lubią różne metody, ale dobry programista bierze wszystkie pod uwagę. Metody od podstaw pomagają zrozumieć aspekty maszynowe, metody od góry pomagają zrozumieć aspekty problemów przestrzeni, a metody od środka pomagają przy praktycznych pytaniach i odpowiedziach. Pozostawienie któregoś z tych aspektów poza mogłoby być błędem.
Programowanie komputerowe jest obszernym tematem. Jako programista, będziesz musiał być przygotowany na ciągłe uczenie się i poszerzenie swoich granic. Te książki pomogą ci to zrobić. One nie tylko uczą swoich tematów, ale także uczą różnych dróg i metod *myślenia*. Jak powiedział Alan Perlis, "Język który nie wyraża sposobu w jaki myślisz o programowaniu nie jest wart poznawania" (http://www.cs.yale.edu/homes/perlis-alan/quotes.html). Jeśli stale poszukujesz nowych i lepszych sposobów wykonania i myślenia, staniesz się pełnym sukcesów programistą. Jeśli nie szukasz wzbogacania samego siebie, "Trochę snu, trochę drzemania, trochę założenia rąk - a przyjdzie na ciebie nędza jak bandyta i niedostatek jak człowiek zbrojny." (Księga Przysłów 24:33-34). Pewnie nie całkiem tak srogo, ale ciągle, najlepiej jest zawsze się uczyć.
Te książki były wybrane z powodu swojej zawartości i wielkiego szacunku jaki one mają w świecie nauki o komputerach. Każda z nich wnosi coś unikalnego. Tutaj jest wiele książek. Najlepszym sposobem aby zacząć mogłoby być przejrzenie online skrótów z kilkunastu z tych książek, i znalezienie punktu początkowego który cię interesuje.

Od Podstaw Wzwyż

Ta lista jest w najlepszym porządku jaki mogłem znaleźć. Nie koniecznie jest od najłatwiejszych do najtrudniejszych, ale bazowała na sprawach tematycznych.
- "Programming from the Ground Up" Jonathan Bartlett
- "Introduction to Algorithms" Thomas H. Cormen, Charles E. Leiserson, i Ronald L. Rivest
- "The Art of Computer Programming" Donald Knuth (zestaw 3 tomów - tom 1 jest najważniejszy)
- "Programming Languages" Samuel N. Kamin
- "Modern Operating Systems" Andrew Tanenbaum
- "Linkers and Loaders" John Levine
- "Computer Organization and Design: The Hardware/Software Interface" David Patterson i John Hennessy

Ze Szczytu W Dół

Te książki są ułożone od najprostszych do najtrudniejszych. Jednakże, mogą one być czytane w jakiejkolwiek kolejności z którą się wygodnie czujesz.
- "How to Design Programs" Matthias Felleisen, Robert Bruce Findler, Matthew Flatt, i Shiram Krishnamurthi, dostępna online na http://www.htdp.org/

- "Simply Scheme: An Introduction to Computer Science" Brian Harvey i Matthew Wright
- "How to Think Like a Computer Scientist: Learning with Python" Allen Downey, Jeff Elkner, i Chris Meyers, dostępna online na http://www.greenteapress.com/thinkpython/
- "Structure and Interpretation of Computer Programs" Harold Abelson i Gerald Jay Sussman z Julie Sussman, dostępna online na http://mitpress.mit.edu/sicp/
- "Design Patterns" Erich Gamma, Richard Helm, Ralph Johnson, i John Vlissides
- "What not How: The Rules Approach to Application Development" Chris Date
- "The Algorithm Design Manual" Steve Skiena
- "Programming Language Pragmatics" Michael Scott
- "Essentials of Programming Langauges" Daniel P. Friedman, Mitchell Wand, i Christopher T. Haynes

Od Środka Na Zewnątrz

Każda z tych książek jest najlepszą w swoim temacie. Jeśli potrzebujesz nauczyć się tych języków, powiedzą one wszystko czego potrzebujesz wiedzieć.
- "Programming Perl" Larry Wall, Tom Christiansen, i Jon Orwant
- "Common LISP: The Language" Guy R. Steele
- "ANSI Common LISP" Paul Graham
- "The C Programming Language" Brian W. Kernighan i Dennis M. Ritchie
- "The Waite Group's C Primer Plus" Stephen Prata
- "The C++ Programming Language" Bjarne Stroustrup
- "Thinking in Java" Bruce Eckel, dostępna online na http://www.mindview.net/Books/TIJ/
- "The Scheme Programming Language" Kent Dybvig
- "Linux Assembly Language Programming" Bob Neveln

Zagadnienia Wyspecjalizowane

Książki te są najlepszymi książkami omawiającymi swoje dziedziny. Są one szczegółowe i autorytatywne. Aby posiąść szerokie podstawy wiedzy, powinieneś przeczytać kilka z poza tych obszarów w których normalnie programujesz.
- "Practical Programming - Programming Pearls and More Programming Pearls" Jon Louis Bentley
- "Databases - Understanding Relational Databases" Fabian Pascal
- "Project Management - The Mythical Man-Month" Fred P. Brooks
- "UNIX Programming - The Art of UNIX Programming" Eric S. Raymond, dostępna online na http://www.catb.org /~esr/writings/taoup/
- "UNIX Programming - Advanced Programming in the UNIX Environment" W. Richard Stevens
- "Network Programming - UNIX Network Programming" (2 tomy) W. Richard Stevens
- "Generic Programming - Modern C++ Design" Andrei Alexandrescu
- "Compilers - The Art of Compiler Design: Theory and Practice" Thomas Pittman i James Peters
- "Compilers - Advanced Compiler Design and Implementation" Steven Muchnick
- "Development Process - Refactoring: Improving the Design of Existing Code" Martin Fowler, Kent Beck, John Brant, William Opdyke, i Don Roberts
- "Typesetting - Computers and Typesetting" (5 tomów) Donald Knuth
- "Cryptography - Applied Cryptography" Bruce Schneier
- "Linux - Professional Linux Programming" Neil Matthew, Richard Stones, i 14 współautorów
- "Linux Kernel - Linux Device Drivers" Alessandro Rubini i Jonathan Corbet

- "Open Source Programming - The Cathedral and the Bazaar: Musings on Linux and Open Source by an Accidental Revolutionary" Eric S. Raymond
- "Computer Architecture - Computer Architecture: A Quantitative Approach" David Patterson i John Hennessy

Dalsze Źródła Języka Asemblerowego

W języku asemblerowym, twoje najlepsze źródła są w sieci.
- http://www.linuxassembly.org/ - wspaniałe źródło dla programistów języka asemblerowego Linuksa
- http://www.sandpile.org/ - repozytorium materiałów referencyjnych na x86, x86-64, i procesorów kompatybilnych
- http://www.x86.org/ - Dr. Dobb's Journal Microprocessor Resources
- http://www.drpaulcarter.com/pcasm/ - Strona Języka Asemblerowego PC Dr. Paula Cartera
- http://webster.cs.ucr.edu/ - Strona Domowa "The Art of Assembly"
- http://www.intel.com/design/pentium/manuals/ - manuale Intela dla swoich procesorów
- http://www.janw.easynet.be/ - przykłady z języka asemblerowego Jana Wegemakera
- http://www.azillionmonkeys.com/qed/asm.html - Strona Asemlera x86 Paula Hsieha

Dodatek A. Programowanie GUI

Wprowadzenie do Programowania GUI
Celem tego dodatku nie jest nauczenie cię jak robić Graficzne Interfejsy Użytkownika. Jest on po prostu pomyślany aby pokazać jak pisanie aplikacji graficznych jest tym samym co pisanie innych aplikacji, oprócz użycia dodatkowych bibliotek do utrzymywania części graficznych. Jako programista musisz nabrać nawyku uczenia się nowych bibliotek. Większość twojego czasu będzie spędzana na przesyłaniu danych od jednej biblioteki do drugiej.

Biblioteki Gnome
Projekt GNOME jest jednym z kilku projektów prowadzącym kompletny pulpit dla użytkowników Linuksa. Projekt GNOME zawiera panel do przechowywania zestawów aplikacji i mini-aplikacji zwanych apletami, kilku standardowych aplikacji do wykonywania zadań takich jak zarządzanie plikami, zarządzanie sesją, i konfiguracja, i API dla tworzenia aplikacji które dopasowuje je do sposobu działania reszty systemu.
Jedną rzeczą do odnotowania o bibliotekach GNOME jest to, że stale tworzą one i przekazują tobie wskaźniki do wielkich struktur danych, ale ty nigdy nie potrzebujesz wiedzieć jak są one ułożone w pamięci. Wszystkie manipulacje struktur danych GUI są robione całkowicie poprzez wywołania funkcji. Jest to charakterystyczne dla dobrego projektu biblioteki. Biblioteki zmieniają się z wersji na wersję, i tak samo robią dane które każda struktura danych przechowuje. Jeśli musiałbyś sam dostawać się i manipulować tymi danymi, to wtedy kiedy biblioteka jest uaktualniana musiałbyś modyfikować swoje programy do współpracy z nową biblioteką lub przynajmniej je przekompilować. Kiedy dostajesz się do danych poprzez funkcje, funkcje biorą na siebie znajomość gdzie jest w strukturze każda część danych. Wskaźniki które otrzymujesz od biblioteki są *opakowane* - nie potrzebujesz wiedzieć specjalnie jak struktury na które one wskazują wyglądają, musisz tylko znać funkcje które będą prawidłowo nimi manipulować. Podczas projektowania bibliotek, nawet dla użytku w ramach jednego programu, dobrą praktyką jest pamiętanie o tym.
Ten rozdział nie będzie wchodził w szczegóły jak działa GNOME. Jeśli chciałbyś wiedzieć więcej, odwiedź witrynę projektantów GNOME na http://developer.gnome.org/. Witryna zawiera podręczniki, listy mailingowe, dokumentację API, i wszystko czego potrzebujesz aby zacząć programować w środowisku GNOME.

Prosty program GNOME w Kilku Językach

Ten program będzie po prostu pokazywał Okno które ma przycisk do opuszczenia aplikacji. Kiedy ten przycisk zostaje kliknięty będzie pytanie czy jesteś pewny, a jeśli klikniesz "yes" nastąpi zamknięcie aplikacji. Dla uruchomienia tego programu, zapisz następujące jako **gnome-example.s**:

```
#CEL: Ten program jest pomyślany jako przykład jak wyglądają programy GUI napisane z bibliotekami GNOME
#
#WEJŚCIE: Użytkownik może tylko kliknąć na przycisk "Quit" lub zamknąć okno
#
#WYJŚCIE: Aplikacja będzie zamknięta
#
#PROCES: Jeśli użytkownik klika na przycisk "Quit", program wyświetli dialog pytając czy jest pewien.
#Jeśli kliknie "Yes", zamknie aplikację.
#W przeciwnym razie będzie kontynuował
#

.section .data
###Definicje GNOME - Były one znalezione w plikach nagłówkowych GNOME dla języka C
#i przekonwertowane w ich asemblerowe odpowiedniki

#Nazwy Przycisków GNOME
GNOME_STOCK_BUTTON_YES:
.ascii "Button_Yes\0"
GNOME_STOCK_BUTTON_NO:
.ascii "Button_No\0"

#Typy "MessageBox" Gnome
GNOME_MESSAGE_BOX_QUESTION:
.ascii "question\0"

#Standardowa definicja NULL
.equ NULL, 0

#Definicje sygnałów GNOME
signal_destroy:
.ascii "destroy\0"
signal_delete_event:
.ascii "delete_event\0"
signal_clicked:
.ascii "clicked\0"

###Definicje specyficznie aplikacyjne

#Informacje aplikacyjne
app_id:
.ascii "gnome-example\0"
app_version:
.ascii "1.000\0"
app_title:
.ascii "Gnome Example Program\0"
```

```
#Tekst dla Przycisków i okien
button_quit_text:
.ascii "I Want to Quit the GNOME Example Program\0"
quit_question:
.ascii "Are you sure you want to quit?\0"

.section .bss

#Zmienne do zachowania utworzonych widżetów
.equ WORD_SIZE, 4
.lcomm appPtr, WORD_SIZE
.lcomm btnQuit, WORD_SIZE

.section .text

.globl main
.type main, @function
main:
pushl %ebp
movl %esp, %ebp

#Inicjalizacja bibliotek GNOME
pushl 12(%ebp) #argv
pushl 8(%ebp) #argc
pushl $app_version
pushl $app_id
call $16, %esp #odbudowa stosu

#Tworzenie nowego okna aplikacji
pushl $app_title #Tytuł (nazwa) okna
pushl $app_id #ID aplikacji
call gnome_app_new
addl $8, %esp #odtworzenie stosu
movl %eax, appPtr #zapisuje wskaźnik okna

#Tworzenie nowego przycisku
pushl $button_quit_text #tekst przycisku
call gtk_button_new_witz_label
addl $4, %esp #odtworzenie stosu
movl %eax, btnQuit #zapisanie wskaźnika przycisku

#Powoduje pojawienie się przycisku wewnątrz okna aplikacji
pushl btnQuit
pushl appPtr
call gnome_app_set_contents
addl $8, %esp

#Powoduje pojawienie się przycisku (jedynie po pojawieniu się okna)
pushl btnQuit
```

```
call gtk_widget_show
addl $4, %esp

#Powoduje pojawienie się okna aplikacji
pushl appPtr
call gtk_widget_show
addl $4, %esp

#Posiadanie wywołania GNOME naszej funkcji delete_handler kiedy pojawia się zdarzenie "delete"
pushl $NULL #dodatkowa dana do przekazania do naszej funkcji (nie używamy żadnej)
pushl $delete_handler #adres funkcji do wywołania
pushl $signal_delete_event #nazwa sygnału
pushl appPtr #widżet do nasłuchiwania zdarzeń
call gtk_signal_connect
addl $16, %esp #odtworzenie stosu

#Posiadanie wywołania GNOME naszej funkcji destroy_handler kiedy pojawia się zdarzenie "destroy"
pushl $NULL #dodatkowa dana do przekazania do naszej funkcji (nie używamy żadnej)
pushl $destroy_handler #adres funkcji do wywołania
pushl $signal_destroy #nazwa sygnału
pushl appPtr #widżet do nasłuchiwania zdarzeń
call gtk_signal_connect
addl $16, %esp #odtworzenie stosu

#Posiadanie wywołania GNOME naszej funkcji click_handler kiedy pojawia się zdarzenie "click".
#Zauważ, że poprzednie sygnały były nasłuchiwane w oknie aplikacji,
#podczas gdy ten jest tylko nasłuchiwany w przycisku
pushl $NULL
pushl $click_handler
pushl $signal_clicked
pushl btnQuit
call gtk_signal_connect
addl $16, %esp

#Przekazanie kontroli do GNOME.
#Wszystko co się wydarza od tego miejsca jest reakcją na zdarzenia użytkownika,
#który wywołuje uchwyty sygnałów.
#Ta główna funkcja właśnie ustawia główne okno i łączy uchwyty sygnałów,
#a uchwyty sygnałów zajmują się resztą
call gtk_main

#Po zakończeniu programu, opuszczamy
movl $0, %eax
leave
ret

#Zdarzenie "destroy" zachodzi gdy widżet jest usuwany.
#W tym przypadku, kiedy okno aplikacji jest usuwane,
#po prostu chcemy aby zakończyć pętlę zdarzenia
```

```
destroy_handler:
pushl %ebp
movl %esp, %ebp

#To powoduje, że gtk opuszcza swoją pętlę zdarzenia tak szybko jak może.
call gtk_main_quit

movl $0, %eax
leave
ret

#Zdarzenie "delete" zachodzi gdy okno aplikacji jest kliknięte w "x"
#który normalnie jest używany do zamykania okna
delete_handler:
movl $1, %eax
ret

#Zdarzenie "click" zachodzi gdy widżet jest kliknięty
click_handler:
pushl %ebp
movl %esp, %ebp

#Tworzenie dialogu "Are you sure"
pushl $NULL #Koniec przycisku
pushl $GNOME_STOCK_BUTTON_NO #Przycisk 1
pushl $GNOME_STOCK_BUTTON_YES #Przycisk 0
pushl $GNOME_MESSAGE_BOX_QUESTION #Typ dialogu
pushl $quit_question #Wiadomość dialogu
call gnome_message_box_new
addl $16, %esp #odtworzenie stosu

#%eax przechowuje teraz wskaźnik na okno dialogowe

#Ustawienie Modal na 1 zapobiega interakcji jakiegoś innego użytkownika podczas wyświetlania dialogu
pushl $1
pushl %eax
call gtk_window_set_modal
popl %eax
addl $4, %esp

#Teraz wyświetlamy dialog
pushl %eax
call gtk_widget_show
popl %eax

#To ustawia wszystkie konieczne uchwyty sygnałów na potrzeby wyświetlenia dialogu,
#zamknięcia go gdy jeden z przycisków został kliknięty,
#i zwrócenie numeru przycisku który kliknął użytkownik.
#Numer przycisku bazuje na porządku w jakim przyciski były
```

```
#odłożone w funkcji gnome_message_box_new
pushl %eax
call gnome_dialog_run_and_close
addl $4, %esp

#Przycisk 0 jest przyciskiem "Yes".
#Jeśli to jest przycisk który kliknięto, mówi GNOME aby opuścić jego pętlę zdarzenia.
#W przeciwnym przypadku nic nie robi
cmpl $0, %eax
jne click_handler_end

call gtk_main_quit

click_handler_end:
leave
ret
```

Aby zbudować tę aplikację, wykonaj następujące komendy:

```
as gnome-example.s -o gnome-example.o
gcc gnome-example.o 'gnome-config --libs gnomeui' -o gnome-example
```

Potem wpisz **./gnome-example** aby ją uruchomić.
Ten program, jak większość programów GUI, robi wielki użytek z przesyłania wskaźników do funkcji jako parametry. W tym programie tworzysz widżety z funkcjami GNOME i potem ustawiasz funkcje aby były wywoływane kiedy zajdą konkretne zdarzenia. Te funkcje są zwane funkcjami *callback*. Wszystkie zdarzenia są utrzymywane przez funkcję **gtk_main**, więc nie musisz się martwić o to jak zdarzenia są przeprowadzane. Wszystko co musisz zrobić to mieć "callbacks" ustawione na oczekiwanie na nie.
Tutaj jest jest krótki opis wszystkich funkcji GNOME które były użyte w tym programie:

gnome_init
Pobiera argumenty wiersza komend, licznik argumentów, id aplikacji, i wersję aplikacji i inicjalizuje biblioteki GNOME.

gnome_app_new
Tworzy okno nowej aplikacji i zwraca wskaźnik do niego. Pobiera id aplikacji i tytuł okna jako argumenty.

gtk_button_new_with_label
Tworzy nowy przycisk i zwraca wskaźnik do niego. Pobiera jeden argument - tekst który jest w tym przycisku.

gnome_app_set_contents
Pobiera wskaźnik do okna aplikacji gnome i jakikolwiek widżet który chcesz (w tym przypadku przycisk) i powoduje, że ten widżet jest zawartością okna aplikacji.

gtk_widget_show
Musi być wywoływany przy utworzeniu każdego widżetu (okna aplikacji, przycisków, okienek tekstowych, itd) aby je uwidocznić. Jednakże, w przypadku danego widżetu aby był widoczny, wszyscy jego rodzice muszą być także widoczni.

gtk_signal_connect
To jest funkcja która łączy widżety i ich funkcje sygnału utrzymującego wywołania powrotne. Ta funkcja pobiera wskaźnik

widżetu, nazwę sygnału, funkcję wywołania powrotnego i wskaźnik dodatkowych danych. Po wywołaniu tej funkcji, za każdym razem gdy dane zdarzenie jest uruchomione, wywołanie powrotne będzie wywołane z widżetem który produkuje sygnał i wskaźnik dodatkowych danych. W tej aplikacji, nie używamy wskaźnika danych dodatkowych, więc ustawiamy go właśnie na NULL, to jest 0.

gtk_main
Ta funkcja powoduje wejście GNOME w jego główną pętlę. Aby ułatwić programowanie aplikacji, GNOME utrzymuje główną pętlę programu dla nas. GNOME będzie sprawdzał zdarzenia i wywoływał odpowiednie funkcje wywołań powrotnych gdy się pojawią. Ta funkcja będzie kontynuować przetwarzanie zdarzeń do momentu wywołania **gtk_main_quit** przez uchwyt sygnału.

gtk_main_quit
Ta funkcja powoduje wyjście GNOME z jego głównej pętli przy najwcześniejszej okazji.

gnome_message_box_new
Ta funkcja tworzy okno dialogowe zawierające pytanie i przyciski odpowiadające. Pobiera jako parametry wiadomość do wyświetlenia, typ wiadomości (ostrzeżenie, pytanie, itd) i listę przycisków do wyświetlenia. Końcowym parametrem powinien być NULL aby zaznaczyć, że nie ma więcej przycisków do wyświetlenia.

gtk_window_set_modal
Ta funkcja przerabia dane okno na okno modalne. W programowaniu GUI, okno modalne to takie które wstrzymuje przetwarzanie zdarzeń w innych oknach do momentu gdy to okno się zamknie. Jest to często używane w oknach dialogowych.

gnome_dialog_run_and_close
Ta funkcja pobiera wskaźnik dialogowy (wskaźnik zwrócony przez **gnome_message_box_new** może być tutaj użyty) i będzie ustawiać wszystkie odpowiednie uchwyty sygnałów tak, że będzie uruchomiony aż do naciśnięcia przycisku. W tym momencie zamknie dialog i zwróci który przycisk został naciśnięty. Numer przycisku odnosi się do porządku w którym przyciski były ustawione w **gnome_message_box_new**.

Następujący program jest tym samym programem napisanym w języku C. Zapisz go jako **gnome-example-c.c**:

```
/* CEL: Ten program jest pomyślany jako przykład jak wyglądają programy GUI napisane z bibliotekami GNOME */
#include < gnome.h >
/* Definicje programu */
#define MY_APP_TITLE "Gnome Example Program"
#define MY_APP_ID "gnome-example"
#define MY_APP_VERSION "1.000"
#define MY_BUTTON_TEXT "I Want to Quit the Example Program"
#define MY_QUIT_QUESTION "Are you sure you want to quit?"
/* Muszą być zadeklarowane funkcje zanim będą użyte */
int destroy_handler(gpointer window, GdkEventAny *e, gpointer data);
int delete_handler(gpointer window, GdkEventAny *e, gpointer data);
int click_handler(gpointer window, GdkEventAny *e, gpointer data);

int main(int argc, char **argv)
{
gpointer appPtr; /* okno aplikacji */
gpointer btnQuit; /* przycisk "quit" */
```

```
/* Inicjalizacja bibliotek GNOME */
gnome_init(MY_APP_ID, MY_APP_VERSION, argc, argv);

/* Tworzy nowe okno aplikacji */
appPtr = gnome_app_new(MY_APP_ID, MY_APP_TITLE);

/* Tworzy nowy przycisk */
btnQuit = gtk_button_new_with_label(MY_BUTTON_TEXT);

/* Powoduje wyświetlenie przycisku wewnątrz okna aplikacji */
gnome_app_set_contents(appPtr, btnQuit);

/* Powoduje wyświetlenie przycisku */
gtk_widget_show(btnQuit);

/* Powoduje wyświetlenie okna aplikacji */
gtk_widget_show(appPtr);

/* Łączy uchwyty sygnałów */
gtk_signal_connect(appPtr, "delete_event", GTK_SIGNAL_FUNC(delete_handler), NULL);
gtk_signal_connect(appPtr, "destroy", GTK_SIGNAL_FUNC(destroy_handler), NULL);
gtk_signal_connect(btnQuit, "clicked", GTK_SIGNAL_FUNC(click_handler), NULL);

/* Przekazuje kontrolę do GNOME */
gtk_main();

return 0;
}

/* Funkcja do przyjmowania sygnału "destroy" */
int destroy_handler(gpointer window, GdkEventAny *e, gpointer data)
{
/* Opuszcza pętlę zdarzeń GNOME */
gtk_main_quit();
return 0;
}

/* Funkcja do przyjmowania sygnału "delete_event" */
int delete_handler(gpointer window, GdkEventAny *e, gpointer data)
{
return 0;
}

/* Funkcja do przyjmowania sygnału "clicked" */
int click_handler(gpointer window, GdkEventAny *e, gpointer data)
{
gpointer msgbox;
int buttonClicked;
```

```
/* Tworzy dialog "Are you sure" */
msgbox = gnome_message_box_new(MY_QUIT_QUESTION, GNOME_MESSAGE_BOX_QUESTION,
GNOME_STOCK_BUTTON_YES, GNOME_STOCK_BUTTON_NO, NULL);
gtk_window_set_modal(msgbox, 1);
gtk_widget_show(msgbox);

/* Uruchomienie okienka dialogowego */
buttonClicked = gnome_dialog_run_and_close(msgbox);

/* Przycisk 0 jest przyciskiem "Yes". Jeśli jest to przycisk kliknięty, mówi GNOME aby opuścić pętlę zdarzeń. W
przeciwnym razie nie robi nic */
if(buttonClicked == 0)
{
gtk_main_quit();
}
return 0;
}
```

Aby to skompilować, wpisz

```
gcc gnome-example-c.c 'gnome-config --cflags --libs gnomeui' -o gnome-example-c
```

Uruchom go przez wpisanie **./gnome-example-c**

Ostatecznie, mamy wersję w Pythonie. Zapisz go jako **gnome-example.py**:

```
#CEL: Ten program jest pomyślany jako przykład jak wyglądają programy GUI napisane z bibliotekami GNOME
#

#Importowanie bibliotek GNOME
import gtk
import gnome.ui

####NAJPIERW DEFINIOWANIE FUNKCJI WYWOŁAŃ POWROTNYCH####

#W Pythonie, funkcje muszą być zdefiniowane przed ich użyciem,
#więc musimy najpierw zdefiniować nasze funkcje wywołań powrotnych.

def destroy_handler(event):
gtk.mainquit()
return 0

def delete_handler(window, event):
return 0

def click_handler(event):
#Tworzenie dialogu "Are you sure"
msgbox = gnome.ui.GnomeMessageBox("Are you sure you want to quit?",
gnome.ui.MESSAGE_BOX_QUESTION,
```

```
gnome.ui.STOCK_BUTTON_YES, gnome.ui.STOCK_BUTTON_NO)
msgbox.set_modal(1)
msgbox.show()

result = msgbox.run_and_close()

#Przycisk 0 jest przyciskiem "Yes". Jeśli jest to przycisk kliknięty, mówi GNOME aby opuścił pętlę zdarzeń.
#W przeciwnym razie nie robi nic.
if (result == 0):
gtk.mainquit()

return 0

####PROGRAM GŁÓWNY####

#Tworzenie nowego okna aplikacji
myapp = gnome.ui.GnomeApp("gnome-example", "Gnome Example Program")

#tworzenie nowego przycisku
mybutton = gtk.GtkButton("I Want to Quit the GNOME Example Program")
myapp.set_contants(mybutton)

#Wyświetlenie przycisku
mybutton.show()

#Wyświetlenie okna aplikacji
myapp.show()

#Łączenie uchwytów sygnałów
myapp.connect("delete_event", delete_handler)
myapp.connect("destroy", destroy_handler)
mybutton.connect("clicked", click_handler)

#Przekazanie kontroli do GNOME
gtk.mainloop()
```

Aby go uruchomić wpisz **python gnome-example.py**

Budowniczowie GUI

W poprzednim przykładzie, stworzyłeś interfejs użytkownika dla aplikacji poprzez wywoływanie utworzonych funkcji dla każdego widżetu i umieszczeniu go tam gdzie chciałeś. Jednakże, może być to całkiem bałaganiarskie dla bardziej kompleksowych aplikacji. Wiele środowisk programistycznych, włączając w to GNOME, ma programy zwane budowniczowie GUI, które mogą być użyte do automatycznego tworzenia twojego GUI dla ciebie. Musisz tylko napisać kod dla uchwytów sygnałów i zainicjalizować swój program. Główny budowniczy GUI dla aplikacji GNOME jest zwany GLADE. GLADE jest dostarczany z większością dystrybucji Linuksa.

Są budowniczowie GUI dla większości środowisk programistycznych. Borland ma gamę narzędzi które będą budować GUI szybko i łatwo w systemach linux i Win32. Środowisko KDE ma narzędzie zwane Projektant QT który pomaga automatycznie organizować GUI dla swojego systemu.

Jest szeroka gama wyboru dla organizowania aplikacji graficznych, ale ten dodatek dał tobie przedsmak tego jak wygląda

programowanie GUI.

Dodatek B. Powszechne Instrukcje x86

Odczytywanie Tablic

Tablice instrukcji prezentowane w tym dodatku zawierają:

- kod instrukcji
- używane operandy
- używane flagi
- krótki opis co te instrukcje robią

W sekcji operandów, będą wypisane typy operandów jakie instrukcje przyjmują. Jeśli instrukcja przyjmuje więcej niż jeden operand, każdy operand będzie oddzielony przecinkiem. Każdy operand będzie miał listę znaków które powiedzą czy te operandy mogą być wartością trybu natychmiastowego (I), rejestrem (R), lub adresem pamięci (M). Na przykład, instrukcja **movl** jest wypisana jako I/R/M, R/M. To oznacza, że pierwszy operand może być każdego typu wartością, podczas gdy drugi operand musi być rejestrem lub lokalizacją pamięci. Zauważ, jednakże, że w języku asemblera x86 nie możesz mieć więcej niż jeden operand będący lokalizacją pamięci.

W sekcji flag, będą wypisane flagi w rejestrze **%eflags** zaangażowane przez te instrukcje. Następujące flagi są wymienione:

O

Flaga "overflow" (przepełnienia). Jest ustawiona na wartość prawdy jeśli operand przeznaczenia nie był wystarczająco duży aby zmieścić wynik instrukcji.

S

Flaga "sign" (znaku). Ta flaga jest ustawiona na znak ostatniego wyniku.

Z

Flaga "zero" (zera). Jest ustawiona na wartość prawdy jeśli wynik instrukcji jest zero.

A

Flaga "auxiliary carry" (przeniesienia pomocniczego). Jest ustawiona dla przenoszenia i pożyczek pomiędzy trzecim i czwartym bitem. Nie jest ona często używana.

P

Flaga "parity" (parzystości). Jest ustawiona na wartość prawdy jeśli niski bajt ostatniego wyniku miał parzystą liczbę bitów 1.

C

Flaga "carry" (przeniesienia). Używana w arytmetyce do powiedzenia czy wynik powinien być przeniesiony do dodatkowego bajta czy nie. Jeśli flaga przeniesienia jest ustawiona, to zwykle oznacza, że rejestr przeznaczania mógłby przechować cały wynik. Decyzja należy do programisty jaką akcję podjąć (t.j. - rozprzestrzenić wynik na następny bajt, zasygnalizować błąd, lub zignorować kompletnie).

Inne flagi istnieją, ale są o wiele mniej istotne.

Instrukcje Przesyłania Danych

Te instrukcje przeprowadzają niewiele, jeśli jakiekolwiek obliczenia. Zamiast tego są one używane do przenoszenia danych z jednego miejsca do drugiego.

Tabela B-1. Instrukcje Przesyłania Danych

Instrukcja	**Operandy**	**Z**
movl	I/R/M, I/R/M	(
To kopiuje słowo danych z jednej lokalizacji do innej.**movl %eax, %ebx** kopiuje zawartość **%eax** do **%ebx**		

movb	I/R/M, I/R/M	
Tak samo jak **movl**, ale operuje na indywidualnych bajtach.		
leal	M, I/R/M	
Pobiera daną lokalizację pamięci w formacie standardowym, i, zamiast załadować zawartość tej lokalizacji pamięci, ładuje ob **%eax** ładuje obliczony adres poprzez **5 + %ebp + 1*%ecx** i zachowuje go w **%eax**		
popl	R/M	
Zdejmuje wierzchołek stosu do danej lokalizacji. Jest to równoważne przeprowadzeniu **movl (%esp), R/M** po czym następu wierzchołek stosu do rejestru **%eflags.**		
pushl	I/R/M	
Wkłada daną wartość na stos. Jest to równoważne przeprowadzeniu **subl $4, %esp** po czym następuje **movl I/R/M, (%esp)** rejestru **%eflags** na wierzchołek stosu.		
xchgl	R/M, R/M	
Zamienia wartości danych operandów.		

Instrukcje Całkowitoliczbowe

Tutaj są podstawowe instrukcje obliczeniowe które operują na liczbach całkowitych ze znakiem i bez znaku.

Tabela B-2. Instrukcje Całkowitoliczbowe

Instrukcja	**Operandy**	**Z**
adcl	I/R/M, R/M	
Dodawanie z przeniesieniem. Dodaje bit przeniesienia i pierwszy operand do drugiego, i, jeśli jest przepełnienie, ustawia prz stosowane dla operacji większych niż słowo maszynowe. Dodawanie na słowie mniej znaczącym mogłoby mieć miejsce z uży mogłoby używać instrukcji **adcl** aby wziąć pod uwagę przeniesienie z poprzedniego dodawania. Dla zwykłego przypadku, ni		
addl	I/R/M, R/M	
Dodawanie. Dodaje pierwszy operand do drugiego, zachowując wynik w drugim. Jeśli wynik jest większy niż rejestr przezna wartość prawdy. Ta instrukcja operuje zarówno na liczbach ze znakiem i bez znaku.		
cdq		
Przemienia słowo **%eax** w słowo podwójne zawierające **%edx:%eax** z rozszerzeniem znakiem. **q** zaznacza, że jest to *słowo* jest zwane poczwórnym słowem z powodu terminologii używanej w czasach 16-bitowych. Jest zwykle używana przed przep		
cmpl	I/R/M, R/M	
Porównuje dwie liczby całkowite. Robi to poprzez odejmowanie pierwszego operandu od drugiego. Pomija wyniki, ale ustaw warunkowym.		
decl	R/M	
Dekrementuje (zmniejsza o jeden) rejestr lub lokalizację pamięci. Użyj **decb** do dekrementacji bajta zamiast słowa.		
divl	R/M	
Wykonuje dzielenie bez znaku. Dzieli zawartość słowa podwójnego umieszczonego w połączonym rejestrze **%edx:%eax** pr pamięci. Rejestr **%eax** zawiera wynikowy iloraz, a rejestr **%edx** zawiera wynikową resztę. Jeśli iloraz jest za duży aby się zm		
idivl	R/M	
Wykonuje dzielenie ze znakiem. Działa tak samo jak **divl** powyżej.		
imull	R/M/I, R	
Wykonuje mnożenie ze znakiem i zachowuje wynik w drugim operandzie. Jeśli drugi operand nie występuje, przyjmuje się, ż podwójnym słowie **%edx:%eax**.		
mull	R/M/I, R	
Przeprowadza mnożenie bez znaku. Takie same reguły jak przy **imull**.		

incl	R/M	
Inkrementuje (zwiększa o jeden) dany rejestr lub lokalizację pamięci. Użyj **incb** do inkrementacji bajta zamiast słowa.		
negl	R/M	
Neguje (daje inwersję dopełnienia dwójkowego) danego rejestru lub lokalizacji pamięci.		
sbbl	R/M/I, R/M	
Odejmowanie z pożyczaniem. Jest to używane w taki sam sposób jak **adc**, dla odejmowania. Zwykle tylko **subl** jest używane		
subl	R/M/I, R/M	
Odejmuje dwa operandy. Odejmuje pierwszy operand od drugiego, i zachowuje wynik w drugim operandzie. Instrukcja ta mo znaku.		

Instrukcje Logiczne

Te instrukcje operują na pamięci jako bitach zamiast słów.

Tabela B-3. Instrukcje Logiczne

Instrukcja	**Operandy**	
andl	I/R/M, R/M	
Wykonuje logiczne "and" (i) zawartości dwu operandów, i zachowuje wynik w drugim operandzie. Ustawia flagi przepełnien		
notl	R/M	
Wykonuje logiczne "not" (nie) na każdym bicie operanda. Znana także jako dopełnienie jedynkowe.		
orl	R/M/I, R/M	
Wykonuje logiczne "or" (lub) pomiędzy dwoma operandami, i zachowuje wynik w drugim operandzie. Ustawia flagi przepeł		
rcll	I/%cl, R/M	
Przesuwa dane położenie bitów w lewo o ilość miejsc podaną w pierwszym operandzie, który jest albo wartością trybu natyc włączona w ten obrót, powodując użycie 33 bitów zamiast 32. Ustawia także flagę przepełnienia.		
rcrl	I/%cl, R/M	
To samo co powyżej, ale obraca w prawo.		
roll	I/%cl, R/M	
Przesuwa bity w lewo. Ustawia flagi przepełnienia i przeniesienia, ale nie wlicza flagi przeniesienia jako część obrotu. Ilość n lub jest zawarta w rejestrze **%cl**.		
rorl	I/%cl, R/M	
To samo co powyżej, ale obraca w prawo.		
sall	I/%cl, R/M	
Arytmetyczne przesunięcie w lewo. Bit znaku jest przesunięty do flagi przeniesienia, a bit zero jest umieszczany w najmniej z Jest to to samo co zwykłe przesunięcie w lewo. Ilość bitów do przesunięcia jest podana albo w trybie natychmiastowym albo		
sarl	I/%cl, R/M	
Arytmetyczne przesunięcie w prawo. Najmniej znaczący bit jest przesuwany do flagi przeniesienia. Bit znaku jest wsuwany i w prawo. Ilość bitów do przesunięcia jest podana albo w trybie natychmiastowym albo jest zawarta w rejestrze **%cl**.		
shll	I/%cl, R/M	
Logiczne przesunięcie w lewo. Przesuwa wszystkie bity w lewo (bit znaku nie jest traktowany specjalnie). Bit najbardziej w l przesunięcia jest podana w trybie natychmiastowym lub jest zawarty w rejestrze **%cl**.		
shrl	I/%cl, R/M	
Logiczne przesunięcie w prawo. Przesuwa wszystkie bity w prawo (bit znaku nie jest traktowany specjalnie). Bit najbardziej przesunięcia jest podana albo w trybie natychmiastowym lub jest zawarty w rejestrze **%cl**.		
testl	I//R/M, R/M	

Wykonuje logiczne "and" obydwu operandów i pomija wyniki, ale ustawia odpowiednio flagi.		
xorl	I/R/M, R/M	
Wykonuje wykluczające "or" dwu operandów, i zachowuje wynik w drugim operandzie. Ustawia flagi przepełnienia i przenie		

Instrukcje Kontroli Przepływu

Te instrukcje mogą zmienić przepływ programu.

Tabela B-4. Instrukcje Kontroli Przepływu

Instrukcja	**Operandy**	**Z**
call	adres przeznaczenia	
Wkłada na stos coś co mogłoby być następną wartością dla **%eip**, i skacze do adresu przeznaczenia. Używana dla wywołań asteryskiem (gwiazdką) po której następuje rejestr dla pośredniego wywołania funkcji. Na przykład, **call *%eax** będzie wyw		
int	I	
Powoduje przerwanie o danym numerze. Jest to zwykle używane dla wywołań systemowych i innych interfejsów kernela.		
jcc	adres przeznaczenia	
Warunkowa gałąź. **cc** jest tym *kodem warunku*. Skacze do danego adresu jeśli kod warunku jest prawdą (ustawionym z poprz przeciwnym razie, przechodzi do następnej instrukcji. Kodami warunku są: - **[n]a[e]** - ponad (bez znaku większe niż). **n** może być dodane dla "nie" a **e** może być dodane dla "lub równe" - **[n]b[e]** - below (bez znaku mniejsze niż) - **[n]e** - równe - **[n]z** - zero - **[n]g[e]** - większe niż (porównanie ze znakiem) - **[n]l[e]** - mniej niż (porównanie ze znakiem) - **[n]c** - ustawiona flaga przeniesienia - **[n]o** - ustawiona flaga przepełnienia - **[n]p** - ustawiona flaga parzystości - **[n]s** - ustawiona flaga znaku - **ecxz - %ecx** wynosi zero		
jmp	adres przeznaczenia	
Skok bezwarunkowy. Po prostu ustawia **%eip** na adres przeznaczenia. Alternatywnie, adres przeznaczenia może być asterys przykład, **jmp *%eax** przeskoczy do adresu w **%eax**.		
ret		
Zdejmuje wartość ze stosu i wtedy ustawia **%eip** na tę wartość. Używana do powrotu z wywołań funkcji.		

Dyrektywy Asemblera

Są to instrukcje dla asemblera i linkera, zamiast instrukcji dla procesora. Używane są one aby pomóc asemblerowi poprawnie złożyć twój kod, i ułatwić jego użycie.

Tabela B-5. Dyrektywy Asemblera

Dyrektywa	**Operandy**
.ascii	ŁAŃCUCH ZNAKÓW W CUDZYSŁOWIE
Pobiera dany łańcuch znaków w cudzysłowie i zamienia go na dane bajtowe.	
.byte	WARTOŚCI
Pobiera listę wartości oddzielonych przecinkami i wprowadza je wprost do programu jako dane.	

.endr	
Kończy powtarzającą się sekcję zdefiniowaną przez **.rept**.	
.equ	ETYKIETA, WARTOŚĆ
Przypisuje danej etykiecie daną wartość. Wartość może być liczbą, znakiem, lub stałą ekspresją która ma wartość liczby lub z danej wartości.	
.globl	ETYKIETA
Ustawia daną etykietę jako globalną, oznaczającą, że może być używana z osobno skompilowanych plików obiektowych.	
.include	FILE
Umieszcza dany plik tak jakby był tam wpisany	
.lcomm	SYMBOL, ROZMIAR
Jest używana w sekcji **.bss** aby wyszczególnić obszar pamięci który powinien być alokowany kiedy program jest wykonywan ulokowany, i upewnia się, że jest on danej liczby bajtów długości.	
.long	WARTOŚCI
Pobiera sekwencję liczb oddzielonych przecinkami, i wprowadza te liczby jako 4-bajtowe słowa tam gdzie są one w program	
.rept	LICZNIK
Powtarza wszystko pomiędzy tą dyrektywą a dyrektywą **.endr** wyszczególnioną liczbę razy.	
.section	NAZWA SEKCJI
Włącza sekcję która ma działać. Zwykłe sekcje zawierają **.text** (dla kodu), **.data** (dla danych wbudowanych w program), i **.bs**	
.type	SYMBOL, @function
Mówi linkerowi, że dany symbol jest funkcją.	

Różnice w Innych Składniach i Terminologii

Składnia dla języka asemblerowego używanego w tej książce jest znana jako składnia *AT&T*. Jest to ta wspierana przez zestaw narzędzi GNU który standardowo przychodzi wraz z każdą dystrybucją Linuksa. Jednakże, oficjalna składnia dla języka asemblerowego x86 (znana jako składnia Intela) jest inna. Jest to ten sam język asemblerowy dla tej samej platformy, ale wygląda inaczej. Kilka różnic obejmuje:

- W składni Intela, operandy instrukcji są często odwrócone. Operand przeznaczenia jest wymieniony przed operandem źródła.
- W składni Intela, rejestry nie są poprzedzone znakiem procenta (**%**).
- W składni Intela, znak dolara (**$**) nie jest wymagany do wykonania adresowania trybu natychmiastowego. Zamiast tego, adresowaniu nie natychmiastowemu towarzyszy umieszczenie adresu w nawiasach kwadratowych (**[]**).
- W składni Intela, nazwa instrukcji nie zawiera rozmiaru danych przemieszczanych. Jeśli jest to znaczące, jest to oznaczane jako **BYTE**, **WORD**, lub **DWORD** natychmiast po nazwie instrukcji.
- Sposób w jaki reprezentowane są adresy pamięci w języku asemblerowym Intela jest znacznie różniący się (pokazane poniżej).
- Ponieważ linia procesorów x86 oryginalnie zaczęła się jako procesory 16-bitowe, większość literatury o procesorach x86 odnosi się do słów jako wartości 16-bitowych, i nazywa wartości 32-bitowe podwójnymi słowami. Jednakże, my używamy terminu "słowo" na odniesienie się do standardowego rozmiaru rejestru w procesorze, który jest 32 bitowy na procesorze x86. Składnia także utrzymuje tę konwencję nazywania - **DWORD** oznacza "podwójne słowo" w składni Intela i jest używane dla rejestrów standardowego rozmiaru, który my moglibyśmy nazwać po prostu "słowem".
- Język asemblerowy Intela ma zdolność adresowania pamięci jako para segment/przesunięcie. Nie wspominamy o tym ponieważ Linux nie udostępnia segmentowanej pamięci, i dlatego jest to nieadekwatne do normalnego programowania Linuksowego.

Występują też inne różnice, ale są one nieproporcjonalnie małe. Aby ukazać niektóre z nich, weź pod uwagę następującą instrukcję:

movl %eax, 8(%ebx, %edi, 4)

W składni Intela, mogłoby to być zapisane jako:

mov [8 + ebx + 1 * edi], eax

Odniesienie do pamięci jest trochę łatwiejsze do odczytania niż u jego odpowiednika AT&T ponieważ wyszczególnia dokładnie jak ten adres będzie obliczany. Jednakże, porządek operandów w składni Intela może być mylący.

Gdzie Wejść po Więcej Informacji

Intel posiada zestaw wszechstronnych przewodników do swoich procesorów. Są one dostępne na http://www.intel.com /design/pentium/manuals/. Zauważ, że wszystkie używają składni Intela, a nie składni AT&T. Najważniejszy z nich to *IA-32 Intel Architecture Software Developer's Manual* w swoich trzech tomach:
- Tom 1: System Programming Guide
(http://developer.intel.com/design/pentium4/manuals/245470.html)
- Tom 2: Instruction Set Reference
(http://developer.intel.com/design/pentium4/manuals/245471.html)
- Tom 3: System Programming Guide
(http://developer.intel.com/design/pentium4/manuals/245472.html)
Dodatkowo, możesz znaleźć wiele informacji w manualu dla asemblera GNU, dostępnym w sieci na http://www.gnu.org /software/binutils/manual/gas-2.9.1/as.html. Podobnie, manual dla linkera GNU jest dostępny w sieci na http://www.gnu.org/software/binutils/manual/ld-2.9.1/ld.html.

Dodatek C. Ważne Wywołania Systemowe

Tutaj są niektóre ważniejsze wywołania systemowe do użytku kiedy korzystamy z Linuksa. W większości przypadków, jednakże, najlepiej jest używać bibliotek funkcji raczej niż bezpośrednich wywołań systemowych, ponieważ wywołania systemowe były projektowane aby być minimalistycznymi podczas gdy biblioteki funkcji były zaprojektowane aby łatwiej się z nimi programowało. Więcej informacji o bibliotece Linux C, zobaczysz w manualu na http://www.gnu.org/software/libc/manual/
Pamiętaj, że **%eax** przechowuje numery wywołań systemowych, i że wartości powrotu i kody błędów także są przechowywane w **%eax**.

Tabela C-1. Ważne Wywołania Systemowe Linuksa

%eax	Nazwa	%ebx	%ecx	%edx
1	exit	wartość powrotu (int)		
3	read	deskryptor pliku	początek bufora	rozmiar bufora
4	write	deskryptor pliku	początek bufora	rozmiar bufora
5	open	nazwa pliku kończąca się znakiem null	lista opcji	tryb uprawnie
6	close	deskryptor pliku		

12	chdir	nazwa katalogu kończąca się znakiem null		
19	lseek	deskryptor pliku	przesunięcie	tryb
20	getpid			
39	mkdir	nazwa katalogu kończąca się znakiem null	tryb uprawnień	
40	rmdir	nazwa katalogu kończąca się znakiem null		
41	dup	deskryptor pliku		
42	pipe	tablica połączeń		
45	brk	nowe systemowe "break"		
54	ioctl	deskryptor pliku	żądanie	argumenty

Bardziej kompletna lista wywołań systemowych, wraz z dodatkowymi informacjami jest dostępna pod http://www.lxhp.in-berlin.de/lhpsyscal.html
Możesz także otrzymać więcej informacji o wywołaniu systemowym przez wpisanie **man 2 SYSCALLNAME** które zwróci informację o wywołaniu systemowym z sekcji 2 manuala UNIX. Jednakże, odnosi się to do użycia wywołania systemowego z języka programowania C, i może być lub może nie być pomocne.
Informacje jak wywołania systemowe są implementowane w Linuksie, zobacz sekcję "Linux Kernel 2.4 Internals" pod http://www.faqs.org/docs/kernel_2_4/lki-2.html#ss2.11

Dodatek D. Tabela Kodów ASCII

Aby używać tej tabeli, po prostu znajdź znak dla którego chcesz znać kod, i dodaj liczbę z lewej kolumny do górnego wiersza.

Tabela D-1. Tabela kodów ASCII w liczbach dziesiętnych

	+0	+1	+2	+3	+4	+5	+6	+7
0	NUL	SOH	STX	ETX	EOT	ENQ	ACK	BEL
8	BS	HT	LF	VT	FF	CR	SO	SI

16	DLE	DC1	DC2	DC3	DC4	NAK	SYN	ETB
24	CAN	EM	SUB	ESC	FS	GS	RS	US
32		!	"	#	$	%	&	'
40	(	)	*	+	,	-	.	/
48	0	1	2	3	4	5	6	7
56	8	9	:	;	<	=	>	?
64	@	A	B	C	D	E	F	G
72	H	I	J	K	L	M	N	O
80	P	Q	R	S	T	U	V	W
88	X	Y	Z	[	\	]	^	_
96	`	a	b	c	d	e	f	g
104	h	i	j	k	l	m	n	o
112	p	q	r	s	t	u	v	w
120	x	y	z	{	|	}	~	DEL

ASCII jest obecnie wypierany przez międzynarodowy standard znany jako Unicode, który pozwala wyświetlić jakikolwiek znak z jakiegokolwiek systemu zapisywania na świecie. Jak mogłeś zauważyć, ASCII ma tylko wsparcie dla znaków angielskich. Jednak, Unicode jest o wiele bardziej skomplikowany, ponieważ wymaga więcej niż jednego bajta do zakodowania pojedynczego znaku. Jest kilka różnych metod dla kodowania znaków Unicode. Najbardziej popularny jest UTF-8 i UTF-32. UTF-8 jest trochę kompatybilny wstecznie z ASCII (jest przechowywany tak samo dla znaków angielskich, ale rozszerza się na wiele bajtów dla znaków międzynarodowych). UTF-32 po prostu wymaga czterech bajtów dla każdego znaku raczej niż jednego. Windows używa UTF-16, który jest kodowaniem o zmiennej długości i wymaga co najmniej 2 bajtów na znak, więc nie jest kompatybilny wstecznie z ASCII.
Dobrym przewodnikiem na tematy międzynarodowe, czcionek, i Unicodu jest wspaniały Artykuł Joe Spolsky'iego, nazwany "The Absolute Minimum Every Software Developer Absolutely, Positively Must Know About Unicode and Character Sets (No Excuses!)", dostępny w sieci na
http://www.joelonsoftware.com/articles/Unicode.html

Dodatek E. Idiomy C w Języku Asemblerowym

Ten dodatek jest dla programistów C uczących się języka asemblerowego. Jest on pomyślany tak aby przekazać główną ideę jak konstrukcje C mogą być implementowane w języku asemblerowym.

Instrukcja if

W C, instrukcja **if** zawiera trzy części - warunek, gałąź prawdy, i gałąź fałszu. Jednakże, ponieważ język asemblerowy nie jest językiem blokowo strukturyzowanym, musisz trochę popracować aby zaimplementować bloko-podobną naturę C. Na przykład, zobacz następujący kod C:

```
if (a == b)
{
/* Tutaj Kod Gałęzi Prawdy */
}
else
```

```
{
/* Tutaj Kod Gałęzi Fałszu */
}

/* W Tym Miejscu, Spotykają się */
```

W języku asemblerowym, można to zapisać tak:

```
#Przenosimy a i b do rejestrów dla porównania
movl a, %eax
movl b, %ebx

#Porównanie
cmpl %eax, %ebx

#Jeśli Prawda, idziemy do gałęzi prawdy
je true_branch

false_branch: #Ta etykieta jest niepotrzebna, tutaj jest tylko dla dokumentacji
#Tutaj Kod Gałęzi Fałszu

#Skok do punktu spotkania
jmp reconverge


true_branch:
#Tutaj Kod Gałęzi Prawdy

reconverge:
#Obydwie gałęzie spotykają się w tym punkcie
```

Jak możesz zauważyć, ponieważ język asemblerowy jest liniowy, bloki muszą się wzajemnie przeskakiwać.
Spotkanie jest utrzymywane przez programistę, nie przez system.

Instrukcja **case** jest zapisywana tak jak sekwencja instrukcji **if**.

Wywoływanie Funkcji

Wywoływanie funkcji w języku asemblerowym po prostu wymaga odkładania argumentów funkcji na stosie w odwrotnej kolejności, i przeprowadzenia instrukcji **call**. Po wywołaniu, argumenty są wtedy zdejmowane z powrotem ze stosu. Na przykład, rozpatrz kod C:

```
printf ("The number is %d", 88);
```

W języku asemblerowym, mogłoby to być zapisane tak:

```
.section .data
text_string:
```

```
.ascii "The number is %d\0"

.section .text
pushl $88
pushl $text_string
call printf
popl %eax
popl %eax #%eax jest zmienną ukrytą, nic nie jest rzeczywiście robione z tą wartością.
#Możesz także bezpośrednio przywrócić %esp do prawidłowej lokalizacji.
```

Zmienne i Wskaźniki

Zmienne globalne i statyczne są deklarowane używając **.data** lub **.bss**. Zmienne lokalne są deklarowane przez rezerwowanie przestrzeni na stosie na początku funkcji. Ta przestrzeń jest oddawana na końcu funkcji. Interesująco, zmienne globalne są dostępne inaczej niż zmienne lokalne w języku asemblerowym. Zmienne globalne są dostępne używając adresowania bezpośredniego, podczas gdy zmienne lokalne są dostępne używając adresowania wskaźnika bazowego. Na przykład, rozpatrz następujący kod C:

```
int my_global_var;

int foo()
{
int my_local_var;

my_local_var = 1;
my_global_var = 2;

return 0;
}
```

Mogłoby to być zapisane w języku asemblerowym jako:

```
.section .data
.lcomm my_global_var, 4

.type foo, @function
foo:
pushl %ebp #Zachowuje stary wskaźnik bazowy
movl %esp, %ebp #Ustawia wskaźnik stosu na wskaźnik bazowy
subl $4, %esp #Robi miejsce dla my_local_var
.equ my_local_var, -4 #Można teraz użyć my_local_var do znalezienia zmiennej lokalnej

movl $1, my_local_var(%ebp)
movl $2, my_global_var

movl %ebp, %esp #Oczyszczenie funkcji i powrót
popl %ebp
ret
```

Co może nie być oczywiste jest to, że dostęp do zmiennych globalnych zabiera mniej cykli maszyny niż dostęp do zmiennych lokalnych. Jednakże, może to nie mieć znaczenia ponieważ stos bardziej prawdopodobnie jest w pamięci fizycznej (zamiast swapowania) niż zmienna globalna.
Zauważ także, że w języku programowania C, po tym jak kompilator załaduje wartość do rejestru, wartość ta będzie prawdopodobnie pozostawać w tym rejestrze dotąd aż ten rejestr będzie potrzebny na coś innego. Może to także przenosić rejestry. Na przykład, jeśli masz zmienną **foo**, może ona być początkowo na stosie, ale kompilator będzie ewentualnie przenosił ją do rejestrów dla procesowania. Jeśli nie ma wielu zmiennych w użyciu, wartość może po prostu pozostawać w tym rejestrze dotąd aż jest on znowu potrzebny. Inaczej mówiąc, kiedy ten rejestr jest potrzebny na coś innego, wartość, jeśli jest zmieniana, jest kopiowana z powrotem do związanej z nią lokalizacji pamięci. W C, możesz użyć słowa kluczowego **volatile** aby upewnić się, że wszystkie modyfikacje i odniesienia do zmiennych są zrobione do samej lokalizacji pamięci, raczej niż kopia rejestru z nich, na wypadek innych procesów, wątków lub sprzętu mogących modyfikować tę wartość podczas gdy twoja funkcja jest uruchomiona.

Pętle

Pętle pracują bardzo podobnie jak instrukcje **if** w języku asemblerowym - bloki są formowane przez przeskakiwanie tam i z powrotem. W C, pętla **while** zawiera ciało pętli, i test do stwierdzenia czy jest to moment na wyjście z pętli czy nie. Pętla **for** jest dokładnie taka sama, z opcjonalnymi sekcjami inicjalizacji i licznika inkrementacji. Mogą one być porzucone robiąc pętlę **while**.
W C, pętla **while** wygląda tak:

```
while(a < b)
{
/* Tutaj wykonywanie */
}
/* Zakończenie Pętli */
```

Może być to zapisane w języku asemblerowym tak:

```
loop_begin:
movl a, %eax
movl b, %ebx
cmpl %eax, %ebx
jge loop_end

loop_body:
#Tutaj wykonywanie

jmp loop_begin

loop_end:
#Zakończenie pętli
```

Język asemblerowy x86 także ma kilka bezpośrednich wsparć dla pętli. rejestr **%ecx** może być użyty jako licznik który *kończy* z zerem. Instrukcja **loop** będzie dekrementować **%ecx** i skakać do wskazanego adresu aż **%ecx** wyniesie zero. Na przykład, jeśli chciałbyś wykonać instrukcję 100 razy, w C mógłbyś zrobić tak:

```
for(i=0; i < 100; i++)
{
/* Tutaj wykonanie procesu */
}
```

W języku asemblerowym mogłoby to być zapisane tak:

```
loop_initialize:
movl $100, %ecx
loop_begin:
#
#Tutaj Wykonanie Procesu
#

#Dekrementuje %ecx i wykonuje pętlę jeśli %ecx nie jest zero
loop loop_begin

rest_of_program:
#Kontynuuje dotąd
```

Jedna rzecz warta odnotowania to, że instrukcja **loop** *wymaga odliczania wstecz do zera*. Jeśli potrzebujesz liczyć wprzód lub użycia innej liczby kończącej, powinieneś użyć formy pętli która nie zawiera instrukcji **loop**.
Dla rzeczywiście ścisłych pętli operacji łańcucha znaków, jest także instrukcja **rep**, ale pozostawiamy nauczenie się tego jako ćwiczenie dla czytelnika.

Struktury

Struktury są prostymi opisami bloków pamięci. Na przykład, w C możesz powiedzieć:

```
struct person {
char firstname[40];
char lastname[40];
int age;
};
```

To samo nie robi niczego, oprócz podania sposobów inteligentnego użycia 84 bajtów danych. Generalnie możesz zrobić to samo używając dyrektywy **.equ** w języku asemblerowym. W taki sposób:

```
.equ PERSON_SIZE, 84
.equ PERSON_FIRSTNAME_OFFSET, 0
.equ PERSON_LASTNAME_OFFSET, 40
.equ PERSON_AGE_OFFSET, 80
```

Kiedy deklarujesz zmienną tego typu, wszystko co robisz jest rezerwowanie 84 bajtów przestrzeni. Więc, jeśli masz to w C:

```
void foo()
{
struct person p;
```

```
/* Tutaj wykonanie */
}
```

W języku asemblerowym mógłbyś mieć:

```
foo:
#Standardowy początek nagłówka
pushl %ebp
movl %esp, %ebp

#Rezerwuje naszą zmienną lokalną
subl $PERSON_SIZE, %esp
#To jest przesunięcie zmiennej od %ebp
.equ P_VAR, 0 - PERSON_SIZE

#Tutaj Wykonanie

#Standardowe zakończenie funkcji
movl %ebp, %esp
popl %ebp
ret
```

Aby mieć dostęp do członków struktury, musisz użyć adresowania wskaźnika bazowego z przesunięciami zdefiniowanymi powyżej. Na przykład, w C mógłbyś ustawić wiek osoby tak:

```
p.age = 30;
```

W języku asemblerowym mogłoby wyglądać to tak:

```
movl $30, P_VAR + PERSON_AGE_OFFSET(%ebp)
```

Wskaźniki

Wskaźniki są bardzo proste. Pamiętaj, że wskaźniki są po prostu adresami pod którymi mieszczą się wartości. Zacznijmy od przyjrzenia się zmiennym globalnym. Na przykład:

```
int global_data = 30;
```

W języku asemblerowym, mogłoby to być:

```
.section .data
global_data:
.long 30
```

Pobierając adres tej danej w C:

```
a = &global_data;
```

Pobierając adres tej danej w języku asemblerowym:

```
movl $global_data, %eax
```

Widzisz, z językiem asemblerowym, prawie zawsze masz dostęp do pamięci poprzez wskaźniki. To jest właśnie adresowanie bezpośrednie. Aby dostać sam wskaźnik, musisz przejść do adresowania trybu natychmiastowego. Zmienne lokalne są trochę mniej skomplikowane, ale niewiele. Tutaj jest jak pobierasz adres zmiennej lokalnej w C:

```
void foo()
{
int a;
int *b;

a = 30;

b = &a;

*b = 44;
}
```

Ten sam kod w języku asemblerowym:

```
foo:
#Standardowe otwarcie
pushl %ebp
movl %esp, %ebp

#Rezerwowanie dwu słów pamięci
subl $8, $esp
.equ A_VAR, -4
.equ B_VAR, -8

#a = 30
movl $30, A_VAR(%ebp)

#b = &a
movl $A_VAR, B_VAR(%ebp)
addl %ebp, B_VAR(%ebp)

#*b = 30
movl B_VAR(%ebp), %eax
movl $30, (%eax)

#Standardowe zamknięcie
movl %ebp, %esp
popl %ebp
ret
```

Jak możesz zauważyć, aby pobrać adres zmiennej lokalnej, adres ten musi być obliczony w ten sam sposób w jaki komputer oblicza adresy w adresowaniu wskaźnika bazowego. Jest łatwiejszy sposób - procesor udostępnia instrukcję **leal**, która

oznacza "załaduj adres efektywny". To pozwala komputerowi obliczyć adres, i wtedy załadować go gdziekolwiek chcesz. Więc, moglibyśmy właśnie powiedzieć:

#b = &a
leal A_VAR(%ebp), %eax
movl %eax, B_VAR(%ebp)

Daje to taką samą liczbę wierszy, ale jest trochę przejrzystsze. Wtedy, aby użyć tej wartości, po prostu musisz przenieść ją do rejestru ogólnego przeznaczenia i użyć adresowania pośredniego, jak pokazano w powyższym przykładzie.

Otrzymywanie Pomocy w GCC

Jedną z miłych rzeczy w GCC jest jego zdolność do tłumaczenia kodu języka asemblerowego. Aby przekonwertować plik języka C do asemblera, możesz po prostu zrobić:

gcc -S file.c

Wynik będzie w **file.s**. Nie jest to najczytelniejszy wynik - większość nazw zmiennych została usunięta i zastąpiona albo numerycznymi lokalizacjami pamięci albo wskaźnikami do etykiet generowanych automatycznie. Aby zacząć, prawdopodobnie zechcesz wyłączyć optymalizacje przez **-O0** więc wynik języka asemblerowego będzie lepiej naśladował twój kod źródłowy.
Co jeszcze mógłbyś zauważyć to, że GCC rezerwuje więcej przestrzeni stosu dla zmiennych lokalnych niż my to robimy, i wtedy koniunkcja AND **%esp**. Jest to dla rozszerzenia pamięci i wydajności "cache" przez ujednolicenie zmiennych do podwójnego słowa.
Ostatecznie, na końcu funkcji, zwykle wykonujemy następujące instrukcje dla oczyszczenia stosu przed wydaniem instrukcji **ret**:

movl %ebp, %esp
popl %ebp

Jednakże, wyjście GCC będzie zwykle zawierało instrukcję **leave**. Ta instrukcja jest po prostu kombinacją tych dwóch powyższych instrukcji. Nie używamy **leave** w tym tekście ponieważ chcemy być klarowni co dokładnie się dzieje na poziomie procesora.
Zachęcam cię do wzięcia programu C który napisałeś i skompilowania go do języka asemlerowego i prześledzenia logiki. Potem, dodaj optymalizacje i spróbuj ponownie. Zobacz jak kompilator re aranżuje twój program aby był bardziej optymalny, i spróbuj odgadnąć dlaczego zostały wybrane te aranżacje i instrukcje.

Dodatek F. Używanie Debuggera GDB

Do chwili w której czytasz ten dodatek, będziesz miał prawdopodobnie napisany co najmniej jeden program z błędem. W języku asemblerowym, nawet niewielkie błędy zwykle powodują zatrzymanie całego programu z błędem segmentacji. W większości języków programowania, możesz po prostu wypisać wartości w twoich zmiennych, i użyć tego wypisu do odszukania gdzie popełniłeś błąd. W języku asemblerowym, wywoływanie funkcji wyjścia nie jest takie proste. Dlatego, aby umożliwić zdeterminowanie źródła błędów, musisz użyć *źródłowego debuggera*.

Debugger jest programem który pomaga odszukać błędy poprzez przeprowadzanie programu jeden krok na raz, pozwalając sprawdzić zawartość pamięci i rejestrów przez cały proces. *Debugger źródłowy* jest debuggerem który pozwala związać operację debugowania bezpośrednio z kodem źródłowym programu. Oznacza to, że debugger pozwala na wgląd w kod źródłowy jaki zapisałeś - kompletnie z symbolami, etykietami i komentarzami.
Debugger któremu się będziemy przyglądać to GDB - Debugger GNU. Aplikacja ta jest obecna niemal we wszystkich dystrybucjach GNU/Linux. Może ona debugować programy w wielorakich językach programowania, włączając w to język asemblerowy.

Przykładowa Sesja Debugowania

Najlepszym sposobem na wytłumaczenie jak działa debugger jest użycie go. Program do którego będziemy używać debuggera jest program **maximum** używany w Rozdziale 3. Powiedzmy, że zrobiłeś ten program perfekcyjnie, oprócz tego, że pominąłeś wiersz:

incl %edi

Kiedy uruchomisz ten program, wpada on właśnie w pętlę nieskończoną - nigdy nie wychodzi. Aby odkryć przyczynę, potrzebujesz uruchomić ten program pod GDB. Jednakże, aby to zrobić, musisz mieć asembler zawierający informacje debugujące w wykonaniu. Wszystko co powinieneś zrobić aby to umożliwić jest dodanie opcji **--gstabs** do komendy **as**. Więc, mógłbyś zasemblować program tak:

as --gstabs maximum.s -o maximum.o

Linkowanie mogłoby być takie samo jak normalnie. "stabs" jest formatem debugowania używanym przez GDB. Teraz, aby uruchomić ten program pod debuggerem, mógłbyś wpisać **gdb ./maximum**. Upewnij się, że pliki źródłowe są w bieżącym katalogu. Wynik powinien wyglądać podobnie do tego:

GNU gdb Red Hat Linux (5.2.1-4)
Copyright 2002 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public
License,and you are welcome to change it and/or
distribute copies of it under certain conditions. Type
"show copyright" to see the conditions. There is
absolutely no warrantyfor GDB. Type "show warranty"
for details.
This GDB was configured as "i386-redhat-linux"...
(gdb)

Zależnie od tego którą wersję GDB uruchamiasz, ten wypis może się różnić nieznacznie. W tym momencie, program jest załadowany, ale jeszcze nie uruchomiony. Debugger czeka na twoją komendę. Aby uruchomić program, wpisz tylko **run**. Nie będzie powrotu, ponieważ program jest uruchomiony w pętli nieskończonej. Aby zatrzymać program, naciśnij control-c. Ekran będzie wtedy wyświetlać to:

Starting program: /home/johnnyb/maximum

Program received signal SIGINT, Interrupt.
start_loop () at maximum.s:34
34 movl data_item(,%edi,4), %eax
Current language: auto; currently asm

(gdb)

Mówi to, że program był przerwany przez sygnał SIGINT (od twojego control-c), i był wewnątrz sekcji z etykietą **start_loop**, i wykonywał wiersz 34 kiedy został zatrzymany. Podaje kod który zamierza wykonać. Zależnie kiedy dokładnie nacisnąłeś control-c, może się zatrzymać na różnym wierszu lub na różnej instrukcji niż w podanym przykładzie. Jednym z najlepszych sposobów znalezienia błędów w programie jest podążanie za przepływem programu aby zobaczyć gdzie rozgałęzia się on nieprawidłowo. Aby podążać za przepływem programu, powtarzaj wpisywanie **stepi** (dla "krok instrukcji"), co spowoduje, że komputer będzie wykonywał jedną instrukcję na raz. Jeśli zrobisz tak kilka razy, wyświetli się coś takiego:

(gdb) stepi
35 cmpl %ebx, %eax
(gdb) stepi
36 jle start_loop
(gdb) stepi
32 cmpl $0, %eax
(gdb) stepi
33 je loop_exit
(gdb) stepi
34 movl data_item(,%edi,4), %eax
(gdb) stepi
35 cmpl %ebx, %eax
(gdb) stepi
36 jle start_loop
(gdb) stepi
32 cmpl $0, %eax

Jak możesz powiedzieć, wykonuje pętlę. Generalnie, jest to dobrze, ponieważ napisaliśmy go aby wykonywał pętlę. Jednakże, problemem jest to, że *nigdy się nie zatrzymuje*. Dlatego, aby znaleźć co jest problemem, spójrzmy na punkt w naszym kodzie gdzie powinniśmy opuszczać pętlę:

cmpl $0, %eax
je loop_exit

Zasadniczo, sprawdza to czy **%eax** osiąga zero. Jeśli tak, powinien opuścić pętlę. Tutaj jest kilka rzeczy do sprawdzenia. Przede wszystkim, mogłeś nie wpisać całej tej części. Nie jest to niezwykłe dla programistów, że zapominają włączyć sposób na wyjście z pętli. Jednakże, nie jest to powodem tutaj. Po drugie, powinieneś się upewnić, że **loop_exit** jest rzeczywiście poza pętlą. Jeśli postawimy etykietę w złym miejscu, dziwne rzeczy mogłyby się zdarzyć. Jednakże, znowu, nie jest to ten powód.
Żaden z tych potencjalnych problemów nie jest winowajcą. Więc, następną opcją jest, że może **%eax** ma złą wartość. Są dwa sposoby aby sprawdzić zawartość rejestru w GDB. pierwszym jest komenda **info register**. Wyświetli to zawartość wszystkich rejestrów w liczbach heksadecymalnych. Jednakże, jesteśmy zainteresowani tylko w **%eax** w tym punkcie. Aby wyświetlić tylko **%eax** możemy zrobić **print/ $eax** aby wyświetlić wynik w liczbach heksadecymalnych, lub wykonać **print/d $eax** aby wyświetlić wynik w liczbach dziesiętnych. Zauważ, że w GDB, rejestry są poprzedzone znakami dolara raczej niż znakami procenta. Twój ekran powinien pokazać to:

(gdb) print/d $eax
$1 = 3
(gdb)

Oznacza to, że wynik twojego pierwszego zapytania wynosi 3. Każdemu zapytaniu które zadasz będzie przypisana liczba poprzedzona znakiem dolara. Teraz, jeśli spojrzysz wstecz do kodu, odkryjesz, że 3 jest pierwszą liczbą z listy liczb do przeszukania. Jeśli przejdziesz pętlę kilka razy więcej, odkryjesz, że w każdej iteracji **%eax** ma liczbę 3. Nie jest to co powinno się dziać. **%eax** powinno przejść do następnej wartości na liście za każdą iteracją.
No dobrze, teraz wiemy, że **%eax** jest ładowana z tą samą wartością wciąż od nowa. Poszukajmy skąd **%eax** jest ładowana. Ten wiersz kodu to:

movl data_item(,%edi,4), %eax

Więc, krocz aż do tego wiersza kodu który jest gotowy do wykonania. Teraz, ten kod zależy od dwu wartości - **data_item** i **%edi. data_item** jest symbolem, i dlatego stałą. Dobrą ideą jest sprawdzenie twojego kodu źródłowego aby upewnić się, że etykieta jest przy właściwej danej, ale w naszym przypadku jest. Dlatego, musimy przyjrzeć się **%edi**. Więc, potrzebujemy wypisania go. Będzie to wyglądać tak:

(gdb) print/d $edi
$2 = 0
(gdb)

Oznacza to, że **%edi** jest ustawione na zero, to jest dlatego podtrzymuje ładowanie pierwszego elementu tablicy. Powinno to skłonić cię do zadania sobie dwu pytań - co jest celem **%edi**, i jak powinna się zmieniać jego wartość? Aby odpowiedzieć na pierwsze pytanie, potrzebujemy tylko spojrzeć do komentarzy. **%edi** przechowuje bieżący indeks **data_item**. Ponieważ nasze przeszukiwanie jest sekwencyjnym przeszukiwaniem przez listę liczb w **data_item**, mogłoby być sensowne, że **%edi** powinno być inkrementowane za każdą iteracją pętli.
Przeszukując ten kod, nie ma żadnego kodu który zmieniałby **%edi** w ogóle. Dlatego, powinniśmy dodać wiersz do inkrementacji **%edi** na początku każdej iteracji pętli. Jest to dokładnie ten wiersz który wyrzuciliśmy na początku.
Asemblacja, linkowanie, i uruchamianie tego programu znowu pokaże, że teraz działa on poprawnie.
To ćwiczenie umożliwiło niewielki wgląd w użycie GDB aby pomóc znaleźć błędy w twoich programach.

Punkty Przerwań i Inne Własności GDB

Program który otworzyliśmy w ostatniej sekcji miał pętlę nieskończoną, i mógł być łatwo zatrzymany przez użycie control-c. Inne programy mogą po prostu wyjść lub zakończyć się z błędami. W tych przypadkach, control-c nie pomaga, ponieważ do chwili gdy naciśniesz control-c, program jest już zakończony. Aby to rozwiązać, potrzebujesz ustawić *breakpoints* (punkty przerwań). *Breakpoint* jest miejscem w kodzie które zaznaczyłeś aby wskazać debuggerowi, że powinien zatrzymać program kiedy osiągnie to miejsce.
Aby ustawić *breakpoints* musisz je ustawić przed uruchomieniem programu. Przed przeprowadzeniem komendy **run**, możesz ustawić *breakpoints* używając komendy **break**. Na przykład, aby przerwać na wierszu 27, przeprowadź komendę **break 27**. Wtedy, kiedy program przekracza wiersz 27, zatrzyma się, i wypisze bieżący wiersz i instrukcję. Możesz wtedy przejść krok po kroku program od tego punktu i sprawdzić rejestry i pamięć. Aby przejrzeć wiersze i numery wierszy swojego programu, po prostu możesz użyć komendy **l**. Wyświetli to twój program z numerami wierszy po ekranie za jednym razem.
Kiedy rozpatrujesz funkcje, możesz także przerwać na nazwach funkcji. Na przykład, w programie factorial z Rozdziału 4, moglibyśmy ustawić *breakpoint* dla funkcji factorial przez wpisanie **break factorial**. Spowoduje to, że debugger przerwie natychmiast po wywołaniu funkcji i ustawieniu funkcji (pomija odkładanie **%ebp** i kopiowanie **%esp**).
Kiedy przechodzisz poprzez kod, często nie musisz przechodzić każdej instrukcji każdej funkcji. Dobrze przetestowane funkcje są zwykle stratą czasu aby je przechodzić oprócz rzadkich sytuacji. Dlatego, jeśli użyjesz komendy **nexti** zamiast komendy **stepi**, GDB zaczeka do skompletowania funkcji zanim pójdzie dalej. W przeciwnym razie, z **stepi**, GDB mógłby

wprowadzać cię do każdej instrukcji wewnątrz każdej wywołanej funkcji.

Uwaga

Jednym problemem który ma GDB jest przetwarzanie przerwań. Często GDB zgubi instrukcję która następuje bezpośrednio po przerwaniu. Instrukcja ta jest w rzeczywistości wykonana, ale GDB nie przechodzi do niej. Nie powinno to być problemem - tylko miej świadomość, że może się to zdarzyć.