Uniwersytet w Białymstoku

Wydział Matematyczno-Fizyczny

Instytut Matematyki

Michał Motyczko

Uruchomienie bazy danych Firebird na platformie Solaris

Praca dyplomowa napisana
pod kierunkiem
dr. Mariusza Żynela

Białystok 2005

Składam serdeczne podziękowania
dr. Mariuszowi Żynelowi
za wszelką pomoc oraz cenne rady
udzielone podczas pisania tej pracy.

Michał Motyczko

Spis treści

Wstęp

Firebird, obok MySQl i PostgreSQL, jest jedną z najbardziej popularnych i najczęściej stosowanych relacyjnych baz danych dostępnych jako OpenSource. Ponieważ wywodzi się ona z komercyjnego produktu znanego jako Interbase firmy Borland, jest bardziej dojrzała i posiada wiele użytecznych funkcji w stosunku do swoich konkurentów.

Baza MySQL jest pisana na platformie Sparc pod Solaris przez co nie ma problemu z dostępnością jej najnowszych wersji na Solaris w tym także na platformę x86. Podobnie nie ma większych problemów z uzyskaniem najnowszych wersji PostgreSQL na Solaris. Inaczej jest z bazą Firebird. W momencie rozpoczęcia tej pracy nie było dostępnej wersji binarnej bazy Firebird na Solaris x86 i nic nie wskazywało na to, że taka wersja wkrótce powstanie. Dzieje się tak dlatego, że Solaris w wersji na platformę x86 nie jest zbyt popularny.

Celem tej pracy było skompilowanie i uruchomienie bazy Firebird na Solaris 10 x86 przy pomocy kompilatora Sun Studio 10. Pierwszym krokiem przy realizacji tego zadania było zapoznanie się z pakietem zawierającym kod źródłowy oraz z narzędziami niezbędnymi podczas kompilacji. Do samego końca pracy z Firebirdem mieliśmy problemy ze strasznie zagmatwanymi regułami kompilacji, rozrzuconymi w wielu plikach `Makefile`. Bardzo ciężko było ustalić właściwe miejsce dla opcji kompilatora.

Do kompilacji bazy Firebird większość programistów używa kompilatora `gcc`. Przy małych programach nie ma praktycznie różnic pomiędzy `gcc` a Sun Studio, problemy zaczynają się przy złożonych projektach takich jak Firebird. Dlatego też następnym krokiem było dostosowanie kodu źródłowego Firebirda do wymogów kompilatora Sun Studio 10. Do podstawowych problemów podczas kompilacji należy zaliczyć niepoprawne typy zmiennych oraz niewłaściwe linkowanie zewnętrzne bibliotek C++ w programach C.

Po skompilowaniu bazy Firebird następnym etapem było przygotowanie odpowiedniego pakietu instalacyjnego. Należało tutaj uwzględnić standard instalacji oprogramowania obowiązujący w Solaris, a także hierarchię katalogów w jakich instalowane są poszczególne elementy pakietu. Aby spełnić ten drugi wymóg, musieliśmy dokonać pewnych modyfikacji w kodzie źródłowym.

Rozdział 1

Co to jest Firebird?

W sierpniu 2000 roku, firma Borland Software wypuściła wersję beta bazy danych InterBase 6.0 jako otwarty kod (ang. open source). Po udostępnieniu kodu źródłowego tej bazy danych użytkownicy założyli grupę i uruchomili projekt który nazwali Firebird.
Firebird 1.0 to praktycznie InterBase 6.0 wypuszczony przez Borlanda z mnóstwem poprawionych błędów i kilkoma udoskonaleniami, więc aby dowiedzieć się czym jest Firebird trzeba najpierw poznać InterBase 6.0.
InterBase jest relacyjną bazą danych z otwartym kodem, która uruchamia się na Linux i Windows. Jest to ta sama komercyjna baza danych której firmy: Motorola, Nokia, Boeing i Boston Stock Exchange używały przez wiele lat. InterBase oferuje doskonałą konkurencyjność, wysoką wydajność i potężny język oparty o składowane procedury (ang. stored procedures) i wyzwalacze (ang. triggers).
W przeciwieństwie do InterBase 6 i kolejnych wersji, wydanych przez firmę Borland, uruchamiających się tylko na Linux, Windows i Solaris, Firebird z powodzeniem uruchamia się na innych platformach takich jak: MacOS X, FreeBSD i OpenBSD, HP-UX i AIX. Obecnie Firebird regularnie wypuszcza kolejne wersje do Linuxa i386, Win32, Solaris Sparc, FreeBSD, MacOS X i HP-UX.
Ale Firebird przewyższa InterBase na wiele innych sposobów, nie tylko w liczbie platform na których można go uruchomić. Przede wszystkim Firebird skupia się na tym w jaki sposob błędy zostały usunięte. Jak w każdym podobnym projekcie cały proces rozwoju kodu źródłowego jest jawny dla każdego, więc można dowiedzieć się, w którym miejscu znajdował się błąd, o którym zawiadomiono, kiedy i w jaki sposób został poprawiony oraz ściągnąć poprawiony produkt tak szybko jak to możliwe.

1.1 Baza danych InterBase 6.0

Interbase (lub "IB") jest relacyjną bazą danych, która posiada niektóre wymagania ANSI-SQL-92.
Interbase zajmuje mało miejsca na dysku, więc może być używany przez aplikację bez zabierania wolnego miejsca na dyskach użytkowników. IB używa także kilka megabajtów RAMu w normalnych warunkach, więc nie musisz posiadać serwera najnowszej generacji aby uruchomić Interbase.
Firebird to po prostu Interbase z udostępnionym kodem źródłowym.
Instytucje używające IB:
- Motorola
- Nokia
- MCI
- Northern Telecom
- Bear Steams
- The Money Store
- The US Army
- NASA
- Boeing
oraz wiele innych na całym świecie.

Specyfikacja bazy danych InterBase 6.0:

- Maksymalny rozmiar bazy danych: 32 TB używając wielu plików; największa baza danych w InterBase ma 200GB.
- Maksymalny rozmiar jednego pliku: 4GB na większości platform; 2GB na niektórych platformach.
- Maksymalna liczba tabel: 64K tablic.
- Maksymalny rozmiar jednej tabeli: 32TB.
- Maksymalna liczba wierszy w tabeli: 4GB wierszy.
- Maksymalny rozmiar wiersza: 64K.
- Maksymalna liczba kolumn w tabeli: Zależy od użytych typów (np.: 16384 całkowitych (4 bajty) wartości na wiersz.)
- Maksymalna liczba indeksów w tabeli: 64KB indeksów.
- Maksymalna liczba indeksów w bazie danych: 4GB indeksów.

Typy danych w InterBase 6.0:

Nazwa	Rozmiar	Zakres/Precyzja	Opis
varchar(n)	n znaków	1 do 32767 bajtów	Łańcuch znaków o zmiennej długości n
Smallint	16 bitów	-2^{15} do 2^{15-1}	Oznaczany jako krótki
Integer	32 bity	-2^{31} do 2^{31-1}	Oznaczany jako długi
Float	32 bity	$3,4 * 10^{-38}$ do $3,4 * 10^{38}$	7 cyfr precyzji
Double Precision	64 bity	$1,7 * 10^{-308}$ do $1,7 * 10^{308}$	15 cyfr precyzji
Timestamp	64 bity	1 stycznia 100 do 28 lutego 32768	Zawiera czas i datę
Date	32 bity	1 stycznia 100 do 29 lutego 32768	Zawiera datę
Time	32 bity	0:00 do 23:59.9999	Zawiera czas
Blob	<32GB		Przechowuje dane zmiennego nieokreślonego rozmiaru
Numeric (precyzja, skala)	Zmienny (16, 32 lub 64)		Przykład: Numeric(10,3) trzyma liczby w podanym formacie: ppppppp.sss
Decimal (precyzja, skala)	Zmienny (16, 32 lub 64)		Przykład: Decimal(10,3) trzyma liczby w podanym formacie: ppppppp.sss

1.2 Firebird ClassicServer czy SuperServer?

Firebird udostępniany jest w dwóch wersjach: ClassicServer i SuperServer. Poniżej porównane są obie wersje.

ClassicServer	SuperServer
Całkowicie kompatybilny z Linux, na Windows ciągle trochę eksperymentalny	Całkowicie kompatybilny z Linux i Windows
Tworzy oddzielny proces dla każdego połączenia, każdy z własnym cache. Im mniej połączeń nawiązanych tym mniej zasobów używanych.	Jeden proces z oddzielnym wątkiem dla każdego połączenia. Dzieli zasoby cache. Bardziej wydajny im więcej równoczesnych połączeń.
Zezwala na szybki, bezpośredni dostęp wejścia/wyjścia do plików bazy danych dla lokalnych połączeń (tylko w Linux).	Lokalne połączenia muszą być zrobione w stylu sieciowym łącząc się do lokalnego hosta. Na Windows możliwe są bezpośrednie połączenia lokalne, ale nie są tak szybkie jak „Classic" na Linux i są także mniej bezpieczne.
Windows: częściowo zaimplementowane zarządzanie serwisowe, wspierające zadanie takie jak kopia zapasowa/przywrócenie kopii, zamknięcie bazy danych itp. przez sieć. Pozostałe zadania musza być uruchamiane lokalnie używając narzędzi (małe oddzielne pliki wykonywalne) dołączone do Firebird'a. Linux: Pełne zarządzanie serwisowe	Pełne zarządzanie serwisowe (na Windows i Linux) pozwala uruchomić zadania (tworzenie/przywracanie kopii zapasowej, zamknięcie bazy danych, statystyki, itp.) programowo. Można połączyć się z menedżerem z sieci i zdalnie uruchamiać te zadania.
Wsparcie dla wielu procesorów. Ilość połączeń nie ma wpływu na wydajność każdego z nich.	Nie ma wsparcia dla wielu procesorów. Na wieloprocesorowym systemie Windows, wydajność może spadać dramatycznie kiedy system przełącza procesy między procesorami. Aby temu zapobiec należy ustawić parametr `CpuAffinityMask` w pliku konfiguracyjnym `firebird.conf`

Jak widać żadna z dwóch wersji nie jest lepsza od drugiej pod każdym względem. W podsumowaniu można napisać aby:

- Na systemy z rodziny Windows wybierać SuperServer.
- Na systemy typu Unix wybierać jedną z dwóch wersji w zależności od potrzeb.

Rozdział 2

Narzędzia kompilacji

2.1 `automake` i `autoconf`

Narzędzia `autoconf` i `automake` są w dzisiejszych czasach bardzo powszechnie używane w projektach programistycznych. Są one używane przez niemal każdy projekt typu Open Source.
`Autoconf` jest narzędziem dzięki któremu programy mogą być przenoszone między różnymi systemami operacyjnymi i ich odmianami. Najbardziej widocznym dla użytkownika wynikiem działania tego programu jest wynikowy skrypt `configure`, z którym prawdopodobnie każdy zdążył się zetknąć. Wykonuje on szereg testów które mają na celu sprawdzić czy w systemie są zainstalowane wymagane przez program biblioteki / pliki nagłówkowe / funkcje, ustalić gdzie one się znajdują, wyłapać subtelne różnice między tym co oferują poszczególne platformy itp. Istnieje możliwość reagowania na znalezione nieprawidłowości, na przykład poprzez dołączanie definicji funkcji jeśli standardowa biblioteka jej nie dostarcza. Jednakże w większości przypadków jeśli stwierdzony zostanie brak jakiejś funkcjonalności (biblioteki, pliku nagłówkowego), będziemy chcieli aby nastąpiło przerwanie przetwarzania ze wskazaniem przyczyny. W przypadku pomyślnego wykonania wszystkich testów skrypt ten generuje piliki `Makefile` w całej strukturze katalogów danego programu używając do tego `automake'a`.
`Automake` jest narzędziem do generowania plików `Makefile` z poprawnie ustalonymi zależnościami i o ustandaryzowanej strukturze. Wygenerowane pliki `Makefile` posiadają wszystko co porządny `Makefile` powinien posiadać.
Połączenie tych dwóch narzędzi pozwala na to aby utrzymywać jedno drzewo programu które można kompilować na różnych systemach operacyjnych (Linux, Solaris, DOS, Windows, itd.). W każdym przypadku kompilacja będzie się składała z wydania dwóch komend:

```
$> configure
$> make
```

mimo iż poszczególne platformy różnią się choćby nazwą kompilatora i strukturą katalogów (ścieżkami dostępu do poszczególnych narzędzi systemowych).

2.2 Program MAKE

Do kompilacji dużych programów z kodem źródłowym zamieszczonym w wielu plikach (jak jest i w przypadku FireBirda) służy program `make`. Program ten pozwala w łatwy sposób zarządzać kompilacją programów. Po zmianie fragmentu kodu źródłowego programu, przekompiluje tylko te fragmenty dużych programów, które tego wymagają. Program `make` jest standardową częścią systemów Unix. Najpopularniejszą implementacją jest `gmake` (GNU make). Dla systemów Windows `make` jest dostępny wraz z niektórymi pakietami programistycznymi.
Program `make` może być użyty do kompilacji programów w dowolnym z języków których kompilatory można używać przy pomocy linii poleceń shella, a także przy dowolnych innych zastosowaniach, w których wymagane jest odświeżanie plików przy zmianie innych, np.: tworzenie dokumentacji w różnych formatach (np.: `html, txt, ps, pdf`) mając dane w innym (np.: `xml`) i dysponując programami konwertującymi.
Aby przygotować program `make` do współpracy, musimy stworzyć plik `Makefile`, który opisuje relacje między plikami programu i udostępnia polecenia pozwalające je odświeżać.
Przykład: programy w języku C linkowane są z plikami obiektów, które z kolei są kompilacją plików źródłowych. Szczegóły poniżej.
Pliki `Makefile`: zaleca się stosowanie nazwy plików: `Makefile` lub `makefile`, są to nazwy domyślne. Jeśli chcemy zastosować własną nazwę należy użyć polecenia:

```
make -f nazwa_pliku
```

W pliku `Makefile` zapisane są tzw. reguły (rules), które mówią kiedy i jak odświeżać pliki. Przykład:

```
foo.o : foo.c defs.h # określenie zależności
cc -c -g foo.c # polecenie do wykonania
```

Plik obiektu `foo.o` jest tworzony z plików `foo.c` i `defs.h`. Program `make` skompiluje plik `foo.c` poleceniem `cc -c -g foo.c`, ale tylko wtedy, gdy się okaże, że plik `foo.o` nie istnieje, lub jest starszy niż plik `defs.h` lub `foo.c`.
Ogólna składnia reguły jest następująca:

```
CEL : WYMAGANIA
POLECENIE
...
```

lub równoważna:

```
CEL : WYMAGANIA ; POLECENIE
POLECENIE
...
```

CEL (target) może oznaczać nazwę plików lub pliki oddzielone spacjami, których sposób aktualizacji opisuje reguła. CEL może także być tylko pojęciem występującym w WYMAGANIACH innej reguły. Używając `make` w wersji `gmake` (GNU) zaleca się wtedy wymienienie nazwy tego pojęcia po dyrektywie .PHONY.
WYMAGANIA (prerequisities) to nazwa(y) pliku(ów) oznaczająca(e) pliki których zmiana spowoduje odświeżenie plików CEL lub nazwy reguły CEL, która nie tworzy pliku.
Program `make` napotykając regułę sprawdza, czy istnieje(ą) plik(i) wymieniony(e) w polu WYMAGANIA, jeśli któryś z plików nie istnieje, to `make` szuka reguły tworzenia tego pliku (reguły, której CELem jest taki plik) i ją wykonuje. Po wykonaniu wszystkich potrzebnych reguł, `make` sprawdza czy istnieją pliki wymienione w polu CEL bieżącej reguły. Jeśli tak, to porównywane są daty modyfikacji plików z istniejącymi plikami wymienionymi w polu WYMAGANIA bieżącej reguły. Jeśli dowolny z plików pola CEL nie istnieje, lub plik(i) są nieaktualny(e) to wykonywane są POLECENIA.
POLECENIE (poprzedzone tabulacją lub średnikiem gdy znajduje się w pierwszej linii reguły) oznacza polecenie shella `sh`, jakie ma wykonać `make` aby uzyskać CEL.
Wywołując `make` bez argumentów wykonana zostanie pierwsza reguła, oraz wszystkie reguły od których ona zależy. W pliku `Makefile` mogą jednak znajdować się reguły nie powiązane w żaden sposób z tą pierwszą, domyślną. Jeżeli chcemy aby wykonana została konkretna reguła (oraz wszystkie te, od której ona zależy) wywołujemy:

```
make nazwa_reguły
```

Pattern rules umożliwiają tworzenie reguł w oderwaniu od konkretnej nazwy pliku, a biorący pod uwagę całą grupę plików, np.:

```
%.o : %.c %.h
```

Oznacza, że reguła zostanie uruchomiona, gdy make napotka potrzebę wykonania reguły dla dowolnego pliku obiektów `.o`. W miejscu znaku % zostanie wstawiona nazwa pliku obiektów bez rozszerzenia. Dalej wykonanie poleceń przebiega normalnie.
Znalezienie pliku w którymś z określonych katalogów nie pozwala jeszcze odpowiednio skonstruować poleceń potrzebnych do osiągnięcia celu. Do tego używane są tzw. zmienne automatyczne, np.:

- $^ - cała lista plików wymaganych z pełnymi ścieżkami,
- $(^D) - lista katalogowych części ścieżek plików wymaganych,
- $(^F) - lista wymaganych plików bez ścieżki,
- $< - pierwszy element na liście plików wymaganych z pełną ścieżką,
- $(<D) - katalogowa część pierwszego elementu na liście plików wymaganych,
- $(<F) - pierwszy element na liście plików wymaganych bez ścieżki dostępu,
- $@ - nazwa celu z pełną ścieżką dostępu,
- $(@D) - katalogowa część nazwy celu,
- $(@F) - nazwa pliku celu bez ścieżki dostępu.

Inne często używane zmienne:

- $(CC) zawiera nazwę kompilatora C, najczęściej gcc,
- $(MAKE) zawiera nazwę programu `make` wraz z parametrami, z jakimi make został uruchomiony, zmienna przydatna przy rekurencyjnych wywołaniach `make`,
- $(CFLAGS) zawiera systemowe ustawienia dotyczące kompilowania (flagi kompilatora), co pozwala użytkownikowi samemu decydować, jak kompilować programy nie ingerując w kod pliku `Makefile`.

Definicja własnych zmiennych:

```
ZMIENNA = WARTOSĆ
```

np.:

```
objects = foo.o bar.o car.o
prog : prog.c $ (objects)
$(CC) $< -l $(objects) -o $@
```

Fałszywy cel (phony). Często nie chcemy, aby CEL był plikiem, a jedynie pewnym pojęciem. Możemy to uzyskać jeśli POLECENIE jest tak skonstruowane, że na pewno nie stworzy pliku opisanego przez pole CEL. Dla pewności (może się zdarzyć, że nazwa CELu pokryje się z nazwą pliku stworzonego w inny sposób). Aby `make` (dotyczy tylko wersji `gmake`) wiedział, że ma nie sprawdzać istnienia pliku, ani nie porównywać czasów modyfikacji, tylko zawsze wykonał POLECENIA reguły, stosujemy dyrektywę:

```
.PHONY = CEL
```

CEL to przecinkami pooddzielane nazwy celów, które nie są plikami.
Ważniejsze opcje programu `make`:

- `-d` - tryb debugowy, make podaje maksymalnie dużo informacji o interpretowanym pliku,
- `-e` - daje zmiennym środowiskowym pierwszeństwo nad zmiennymi plików `Makefile`,
- `-f nazwa_ pliku` - wskazuje konkretny plik do interpretowania,
- `-k` - kontynuuj pomimo napotkania błędów,
- `-n` - drukuj komendy które byłyby wykonane, ale nie wykonuj ich,
- `-W nazwa_ pliku` - udawaj, że plik jest zmodyfikowany,

Rozdział 3

Problemy z kompilacją

Przy kompilacji kodu źródłowego bazy danych Firebird przy pomocy kompilatora Sun Studio 10 natkniemy się na pewne problemy. Problemy te dotyczą linkowania, konwersji typów zmiennych oraz otrzymujemy ostrzeżenie iż na platformie Solaris kompilując kod jako Superserver trzeba użyć bibliotek SFIO zamiast standardowych (kompilator GNU (`gcc/g++`) nie zgłasza błędów związanych ani z linkowaniem, ani z konwersją typów, problem dotyczący SFIO jest także zgłaszany przy kompilacji za pomocą tego kompilatora). W dalszej części opisane sa poszczególne problemy oraz podany jest sposób ich rozwiązania.

3.1 Extern „C”

Powszechnie stosowane jest wołanie funkcji bibliotek C z programu C++. Działa to bardzo dobrze dopóki programiści ograniczają się do korzystania ze standardowych plików nagłówkowych (ang. header files) i bibliotek w które zaopatrzone są systemy operacyjne. Ale początkujący programiści mogą natknąć się na kilka błędów linkowania wtedy gdy próbują wołać funkcje własnych bibliotek C z programu C++. Potencjalne powody niepowodzeń mogą być związane z nieznajomością specyfikacji linkowania i tego jak kompilatory C/C++ radzą sobie z symbolami podczas kompilacji. Standard C++ dostarcza mechanizm zwany specyfikacją linkowania (ang. linkage specification) dla mieszanego kodu napisanego w różnych językach programowania. Specyfikacja linkowania odnosi się do protokołu dla linkowanych funkcji lub procedur napisanych w różnych językach. Linkowanie (ang. linkage) jest terminem użytym przez standard C++ do opisania dostępności obiektów jednego pliku z drugiego lub nawet wewnątrz tego samego pliku. Istnieją trzy typy linkowania:

- brak linkowania (ang. no linkage),
- linkowanie wewnętrzne (ang. internal linkage),
- linkowanie zewnętrzne (ang. external linkage).

W przypadku argumentów i zmiennych funkcji zawsze mamy do czynienia z „no linkage" i w związku z tym mogą one być dostępne tylko wewnątrz tej funkcji. Czasami koniecznym jest aby deklarować funkcje i inne obiekty wewnątrz jednego pliku w taki sposób który pozwala na odnoszenie się do siebie nawzajem ale nie może być dostępne spoza tego pliku. Może to być zrobione za pomocą linkowania wewnętrznego. Symbole z linkowania wewnętrznego odnoszą się tylko do tych samych obiektów wewnątrz jednego pliku źródłowego. Poprzedzanie deklaracji słowem `static` zmienia linkowanie zewnętrznych obiektów z linkowania zewnętrznego na linkowanie wewnętrzne. Obiekty znajdujące się na najbardziej zewnętrznym poziomie są linkowane zewnętrznie. To jest domyślne likowanie dla funkcji i czegokolwiek zadeklarowanego poza funkcją. Wszystkie przypadki szczególnych nazw z linkowaniem zewnętrznym odnoszą się do tego samego obiektu w programie. Jeżeli dwie lub więcej deklaracji tego samego symbolu mają linkowanie zewnętrzne ale niekompatybilne typy (na przykład, różnica między deklaracją a definicją), wtedy program może albo „wykrzaczyć się" albo zachowywać się nieprzewidywalnie. Bardzo powszechnym jest korzystanie z funkcjonalności kodu napisanego w jednym języku programowania z kodu napisanego w innym języku. Trywialny przykład: programista C++ pracuje na standardowej bibliotece C (`libc`) aby posortować liczby całkowite za pomocą techniki `quick sort`. To działa ponieważ implementacja C dba o linkowanie za nas. Sami jednak musimy o to zadbać jeżeli używamy własnych bibliotek napisanych w C z programu C++. Inaczej kompilacja może być przerwana z powodu błędów linkowania spowodowanych przez nieznane symbole. Zwróćmy uwagę na poniższy przykład: Załóżmy że piszemy kod C++ i chcemy zawołać funkcję C z kodu C++. Poniżej jest kod do zawołania metody C:

```
bash-3.00$cat greet.h
char *greet();

bash-3.00$cat greet.c
#include "greet.h"

char *greet() {
       return ((char *) "Hello!");
}

bash-3.00$cc -G -o libgreet.so greet.c
```

Spróbujmy zawołać funkcję C `greet()` z programu C++.

```
bash-3.00$cat mixedcode.cpp
#include <iostream.h>

extern char *greet();
```

```
int main() {
        char *greeting = greet();
cout << greeting << "\n";
        return (0);
}
```

Słowo `extern` deklaruje zmienną lub funkcję i określa że ma ona linkowanie zewnętrzne. Jej nazwa jest widzialna z plików innych niż tylko te w których została zdefiniowana.

```
bash-3.00$CC -lgreet mixedcode.cpp
Undefined                       first referenced
 symbol                            in file
char*greet()                    mixedcode.o
ld: fatal: Symbol referencing errors. No output written to a.out
```

Mimo że kod C++ jest linkowany z dynamiczną biblioteka, `libgreet.so`, która zawiera implementację dla `greet()`, linkowanie nie powiodło się z powodu niezdefiniowanego symbolu. Co poszło źle? Powodem błędu linkowania jest to że typowy kompilator C++ przekształca (koduje, ang. mangles) nazwy funkcji aby funkcja mogła być przeładowana. Symbol `greet` jest zmieniony na coś innego w zależności od algorytmu użytego w kompilatorze. W związku z tym plik programu nie posiada symbolu `greet` w tabeli symboli (ang. symbol table). Tabela symboli pliku `mixedcode.o` to potwierdza. Zobaczmy tabele symboli obu plików: `libgreet.so` i `mixedcode.o`:

```
bash-3.00$elfdump -s libgreet.so

Symbol Table Section:  .symtab
index    value       size     type bind oth ver shndx       name
...
[1]  0x00000000 0x00000000  FILE LOCL  D    0 ABS         libgreet.so
...
[37]  0x00000268 0x00000004  OBJT GLOB  D    0 .rodata     _lib_version
[38]  0x000102f3 0x00000000  OBJT GLOB  D    0 .data1      _edata
[39]  0x00000228 0x00000028  FUNC GLOB  D    0 .text       greet
[40]  0x0001026c 0x00000000  OBJT GLOB  D    0 .dynamic    _DYNAMIC

bash-3.00$elfdump -s mixedcode.o

Symbol Table Section:  .symtab
index    value       size     type bind oth ver shndx       name
[0]  0x00000000 0x00000000  NOTY LOCL  D    0 UNDEF
[1]  0x00000000 0x00000000  FILE LOCL  D    0 ABS         mixedcode.cpp
[2]  0x00000000 0x00000000  SECT LOCL  D    0 .rodata
[3]  0x00000000 0x00000000  FUNC GLOB  D    0 UNDEF
    __1cDstd2l6Frn0ANbasic_ostream4Ccn0ALchar_traits4Cc____pkc_2_
[4]  0x00000000 0x00000000  FUNC GLOB  D    0 UNDEF       __1cFgreet6F_pc_
```

```
[5]  0x00000000 0x00000000  NOTY GLOB  D    0 UNDEF       __1cDstdEcout_
[6]  0x00000010 0x00000050  FUNC GLOB  D    0 .text       main
[7]  0x00000000 0x00000000  NOTY GLOB  D    0 ABS         __fsr_init_value

bash-3.00$dem __1cFgreet6F_pc_

__1cFgreet6F_pc_ == char*greet()
```

Nazwa `char*greet()` została przekształcona na `__1cFgreet6F_pc_` przez kompilator C++ Sun Studio 10. Właśnie dlatego statyczny linker `ld` nie mógł dopasować symbolu w pliku programu. Jakie jest rozwiązanie tego problmu? Standard C++ dostarcza mechanizm zwany specyfikacja linkowania (ang. linkage specification), który pozwala na kompilowanie mieszanego kodu. Linkowanie między fragmentami kodu C++ i nie C++ jest nazywane linkowaniem językowym (ang. language linkage). Wszystkie typy funkcji, nazwy funkcji i nazwy zmiennych mają domyślne linkowanie języka C++. Linkowanie językowe można uzyskać używając następujących specyfikacji linkowania:

```
extern string-literal {
deklaracja_funkcji
deklaracja_funkcji
}
extern string-literal deklaracja_funkcji;
```

„`string-literal`" specyfikuje skojarzone linkowanie z konkretną funkcją, na przykład, C i C++. Każda implementacja C++ pozwala na linkowanie do funkcji napisanych w C (`"C"`) i linkowanie do C++ (`"C++"`).
Rozwiązaniem rozważanego problemu jest wyłączenie „manglowania" zewnętrznych funkcji, tak aby móc użyć funkjonalności zewnętrznych funkcji C z kodu C++, bez żadnych problemów. Możemy to uzyskać poprzez użycie linkowania do C. Następująca deklaracja funkcji `greet()` w pliku `mixedcode.cpp` powinna rozwiązać problem

```
extern "C" char *greet();
```

Ponieważ próbowaliśmy wołać kod C z programu C++, zostało użyte linkowanie C dla funkcji `greet()`. Dyrektywa linkowania `extern "C"` mówi kompilatorowi aby nie używał domyślnego „manglowania" nazw funkcji dla tej szczególnej i użył konwencji wołania C kiedy przesyła zewnętrzne informacje do linkera. Innymi słowy: linkowanie C wymusza na kompilatorze C++ aby zaakceptował konwencje C, które nie są takie same jak konwencje C++. Zmodyfikujmy więc plik `mixedcode.cpp` w następujący sposób i przekompilujmy go.

```
bash-3.00$cat mixedcode.cpp
#include <iostream.h>
```

```
extern "C" char *greet();

int main() {
        char *greeting = greet();
       cout << greeting << "\n";
        return (0);
}

bash-3.00$CC -lgreet mixedcode.cpp
bash-3.00$./a.out
Hello!
```

Zajrzyjmy do tablicy symboli pliku `mixedcode.o` jeszcze raz:

```
bash-3.00$CC -c -lgreet mixedcode.cpp
bash-3.00$elfdump -s mixedcode.o

Symbol Table Section:  .symtab
index    value       size     type bind oth ver shndx       name
[0]  0x00000000 0x00000000  NOTY LOCL  D    0 UNDEF
[1]  0x00000000 0x00000000  FILE LOCL  D    0 ABS         mixedcode.cpp
[2]  0x00000000 0x00000000  SECT LOCL  D    0 .rodata
[3]  0x00000000 0x00000000  FUNC GLOB  D    0 UNDEF
    __1cDstd2l6Frn0ANbasic_ostream4Ccn0ALchar_traits4Cc____pkc_2_
[4]  0x00000000 0x00000000  FUNC GLOB  D    0 UNDEF       greet
[5]  0x00000000 0x00000000  NOTY GLOB  D    0 UNDEF       __1cDstdEcout_
[6]  0x00000010 0x00000050  FUNC GLOB  D    0 .text       main
[7]  0x00000000 0x00000000  NOTY GLOB  D    0 ABS         __fsr_init_value
```

Jak oczekiwaliśmy nazwa funkcji `greet()` nie została przekształcona przez kompilator C++ i w związku z tym linker mógł odnaleźć symbol i utworzyć plik wykonywalny. Podstawowe informacje związane z mieszanym kodem programowania:

- Jeżeli miesza się kod C i C++ należy używać kompilatorów które są kompatybilne. Na przykład, muszą definiować podstawowe typy jak `int`, `float` lub `pointer` w ten sam sposób.
- Kiedy miesza się kod należy unikać zmiany typów dla parametrów i zwracanych wartości (ang. return values).
- Nie należy przejmować się linkowaniem językowym kiedy używa się standardowych plików nagłówkowych, ponieważ większość kompilatorów C/C++ radzi sobie ze specyfikacjami linkowania w ich własnych plikach nagłówkowych które działają w obu kodach C i C++. Dlatego też większość istniejących bibliotek C może być wołana bez wyraźnego specyfikowania linkowania C.
- Należy zwracać uwagę na rozróżnianie wielkości znaków dla nazw funkcji w różnych językach.

- Funkcja zadeklarowana jako `extern "C"` nie może zostać przeładowana (ang. overloaded).
- Deklaracja `extern "C"` może być użyta tylko do funkcji globalnych.
- Deklaracja `extern "C"` musi być zawsze po ostatniej deklaracji `#include`.
- Jeżeli to możliwe należy używać dyrektywy linkowania ze wszystkimi fukcjami w pliku. Jest to użyteczne gdy chcemy używać bibliotek C w programie C++. Na przykład:

```
extern "C" {
#include "mylibrary.h"
}
```

- Kiedy programuje się pliki nagłówkowe aby były używane przez programy C i C++ należy użyć następującej konwencji z predefiniowanymi makrami:

```
#if defined __cplusplus
    extern "C" {
#endif

... /* body of header */

#if defined __cplusplus
    }
#endif
```

W kodzie źródłowym Firebirda natkniemy się na liczne problemy linkowania zewnętrznego. Rozwiązaniem jest dopisanie w odpowiednich plikach na początku, najbezpieczniej po ostatniej deklaracji `#include`, takich linii:

```
#if defined __cplusplus
    extern "C" {
#endif
```

oraz na końcu pliku linii:

```
#if defined __cplusplus
    }
#endif
```

3.2 Konwersja typów zmiennej

Zmianę typu danej (być może również połączoną ze zmianą jej wartości, co może być wywołane przez utratę dokładności) nazywa się konwersją. Językami programowania, pozwalającymi na konwersję typów są C/C++, Pascal, Java, Applescript i wiele innych obiektowo zorientowanych języków programowania. Konwersja typów niesie ze sobą ryzyko ponieważ kompilator sprawdza czy kod jest zapisany poprawnie a nie sprawdza czy ma sens. Dlatego też koniecznym jest aby znać jakie są typy danych i na jakie można je przekonwertować bez powodowania żadnych błędów. Na przykład możemy przekonwertować wskaźniki na liczby całkowite, znaki na liczby całkowite (zwracana jest wartość ASCII danego znaku) czy łańcuchy danych na inne typy łańcuchów (jak tablica znaków na łańcuch znaków).

W języku C mamy do czynienia z dwoma rodzajami konwersji:

- konwersje niejawne, których działanie wynika z normy języka i których nie zapisuje się w kodzie programu; ich działanie jest milczące i automatyczne;
- konwersje jawne, przeprowadzane na życzenie programisty; należy zapisać je jawnie w kodzie, wskazując, jaka dana ma być poddana konwersji oraz do jakiego typu; obowiązek podania tych informacji spada na programistę.

Konwersje niejawne wykonywane są w czasie działania programu według następujących, podanych poniżej reguł. Reguły te stosowane są w takiej właśnie kolejności aż do chwili, kiedy wszystkie dane użyte w wyrażeniu będą miały ten sam typ - to zastrzeżenie jest bardzo ważne! A oto te reguły zebrane razem:

- te dane, które są typu char lub short int, poddawane są konwersji do typu int;
- te dane, które są typu float, poddawane są konwersji do typu double;
- jeżeli w wyrażeniu występuje choć jedna dana typu double, to pozostałe dane zostaną poddane konwersji do typu double;
- jeżeli w wyrażeniu występuje choć jedna dana typu long int, to pozostałe dane zostaną poddane konwersji do typu long int;

Jeżeli z kontekstu, w jakim oblicza się wyrażenie, wynika, że ma mieć ono pewien typ inny, niż powstały w wyniku konwersji niejawnych, na koniec dokonuje się konwersji wyniku wyrażenia do tego właśnie typu.
Konwersje jawne natomiast programista zapisuje w programie sam, posługując się tzw. operatorem rzutowania typu (ang. typecast). Jest to operator jednoargumentowy, o wysokim priorytecie, równym priorytetowi jednoargumentowego operatora -, który zapisuje się następująco:

```
(typ)dana
```

gdzie `typ` jest nazwą typu lub opisem typu, do którego należy przeprowadzić konwersję. Np. w następującym fragmencie programu

```
float x;
double y;

y = (double) x;
```

wyraziliśmy życzenie, aby zmienna `x` typu float została poddana jawnej konwersji do typu double. Wiedząc już, jak działają konwersje niejawne (czasami zwane również automatycznymi), zinterpretujmy cały proces obliczania wyrażenia w poniższym fragmencie programu:

```
int Int;
char Char;
short Short;
float Float;

Int = Short + Char + Float;
```

Możemy przewidzieć, że będą miały miejsce następujące konwersje niejawne:

- najpierw promocje, w wyniku których będą miały miejsce następujące konwersje:

  ```
  (int)Short + (int)Char;
  ```

- suma Short i Char oraz Float zostaną poddane konwersji do double, czyli:

  ```
  (double)((int)Short + (int)Char) + (double)Float);
  ```

- obliczona suma końcowa będzie więc typu double; ponieważ z kontekstu, wynikającego z lewego argumentu operatora = widać, że typem spodziewanym jest int, zajdzie jeszcze jedna konwersja do typu int; stąd wynika, że ostateczna postać wyrażenia wygląda tak:

  ```
  (int)((double)((int)Short + (int)Char) + (double)Float));
  ```

Przy kompilacji Firebirda kompilator Sun Studio 10 zgłasza błędy związane z konwersją typów. Na przykład:

```
"../src/gpre/cmd.cpp", line 207: Error: Cannot cast from ref* to short.
```

Kompilator nie może w tym przypadku przekonwertować zmiennej typu wskaźnikowego na zmienną typu short int. Zajrzyjmy więc do pliku `cmd.cpp`. Okolice linii numer 207 wyglądają tak:

```
case ACT_drop_shadow:
        put_numeric(request, gds_dyn_delete_shadow,
                             (SSHORT) action->act_object);
        STUFF_END;
        break;
```

Wprowadźmy zmianę jak poniżej do tego fragmentu pliku:

```
case ACT_drop_shadow:
        put_numeric(request, gds_dyn_delete_shadow,
                             (SSHORT)(int) action->act_object);
        STUFF_END;
        break;
```

Teraz kompilator nie konwertuje zmiennej typu wskaźnikowego na zmienną typu short int. Najpierw konwertuje zmienną typu wskaźnikowego na zmienną typu int, a następnie na zmienną typu short int. Po takiej poprawce kompilator w tym miejscu nie zgłasza już błędu.

3.3 SFIO

Przy kompilacji kodu źródłowego bazy danych Firebird jako SuperServer na systemie Solaris konieczne jest użycie biblioteki SFIO (Safe/Fast Input/Output), przynajmniej takie są założenia autorów. Związane jest to z funkcją `fopen()` otwierającą pliki. Otóż używając tej funkcji możemy otworzyć maksymalnie do 256 plików. Ograniczenie to bierze się stąd, że numer otwartego pliku, tzw. deskryptor, jest pamiętany w zmiennej typu short int, na którą przypada 8 bitów. Tak naprawdę możemy otworzyć do 253 plików ponieważ trzy pierwsze deskryptory tj. 0, 1, 2 zajmują pliki `stdin`, `stdout` oraz `stderr`. Rozwiązaniem jest zastąpienie standardowej biblioteki `stdio.h` biblioteką SFIO, która między innymi deklaruje funkcję `fopen()` na nowo. SFIO rozwiązuje problem przechowując deskryptory plików w zmiennej typu int, czyli w zmiennej 32-bitowej.

Innym rozwiązaniem może być zamiana funkcji `fopen()` na funkcje `open()`. Funkcja `open()` ma trochę bardziej skomplikowany interfejs, ale deskryptory otwieranych plików pamiętane są jako liczby typu int. Wprawdzie system wprowadza ograniczenie, tzw. soft limit, na liczbę otwartych plików przez użytkownika, ale jeśli jest za niski możemy go łatwo zmienić za pomocą polecenia `ulimit`, do wartości tzw. hard limit. W poniższej tabeli przedstawione są wartości soft limit i hard limit.

Wersja systemu	Soft limit	Hard limit
Solaris 7	64	1024
Solaris 8	256	1024
Solaris 9	256	65536

Próba kompilacji Firebirda w wersji SuperServer na Solaris kończy się błędem. Błąd generowany jest makropoleceniem gdy nie jest zdefiniowana flaga SFIO. Flaga ta jest określona gdy kompilujemy Firebirda razem z biblioteką SFIO.

Przy pierwszych próbach kompilacji musieliśmy się cofnąć na początek, skompilować bibliotekę SFIO, której nie ma w Solaris, i skonfigurować kod od nowa dodając odpowiednie ścieżki do miejsca gdzie ta biblioteka została zainstalowana. Ostatecznie jednak usunęliśmy kontrolę z kodu Firebirda i skompilowaliśmy go bez użycia SFIO, gdyż funkcja `fopen()` używana jest sporadycznie, do otwarcia takich plików jak log systemowy. Nie powinniśmy więc przekroczyć limitu 253 plików. Tak skompilowana baza danych była testowana wymagającą aplikacją i nie stwierdziliśmy żadnych nieprawidłowości. Co więcej w momencie pisania pracy nasza wersja bazy Firebird pracuje w trzech różnych instytucjach, jako podstawowe oprogramowanie, bez żadnych problemów.

3.4 STLport

Biblioteka STLport jest implementacją standardowej biblioteki C++ zgodnej ze standardami ANSI/ISO. Kompilator `gcc` rozpowszechniany jest wraz z odpowiednikiem tej biblioteki, zwanym `stdc++`, który niestety nie spełnia tego standardu, spełnia jedynie częściowo jego starszą wersję (por. [2, 3]).

Podczas kompilacji Firebirda przy pomocy `gcc` biblioteka `stdc++` jest wystarczająca, natomiast przy kompilacji za pomocą Sun Studio 10 musieliśmy użyć biblioteki STLport. Kompilator Sun Studio 10 jest bliższy różnym obowiązującym standardom co do kompilatorów C i C++. Z tego względu zawiera bibliotekę STLport, natomiast `gcc` nie posiada tej biblioteki i domyślnie używa `stdc++`.

Biblioteka STLport jest dostępna całkowicie za darmo na wiele platform wraz z kodem źródłowym. Jej obecna wersja oparta jest o implementację SGI. Do jej najważniejszych cech należą:

- zaawansowane techniki i optymalizacje dla maksymalnej wydajności,
- bezpieczne wyjątki (ang. exception safety) i bezpieczne wątki (ang. thread safety),
- ważne rozszerzenia standardu: „hash tables”, „singly-linked list”, „rope”.

Rozdział 4

Pakiet instalacyjny

Standardowo baza Firebird instaluje się w jednym katalogu wskazanym przy konfiguracji źródeł, przed kompilacją. Nie jest to rozwiązanie efektywne gdyż w jednym katalogu mamy wymieszane cztery kategorie plików:

1. programy,
2. współdzielone biblioteki,
3. pliki tymczasowe i logi,
4. pliki baz danych.

Programy i biblioteki nie zmieniają się w trakcie używania, najwyżej w momencie uaktualnienia oprogramowania, natomiast logi i pliki baz danych podlegają ciągłym zmianom. Są przynajmniej dwa powody aby te dwa rodzaje plików od siebie rozdzielać. Po pierwsze, ze względu na bezpieczeństwo systemu programy i biblioteki instaluje się w systemach plików tylko do odczytu. Po drugie, pliki baz danych mogą szybko rosnąć i lepiej trzymać je w dostosowanych do tego systemach plików, oddzielnie od programów. Rozdzielenie poszczególnych kategorii plików ułatwi też tworzenie kopii zapasowych i aktualizacje programów. Nie chcielibyśmy stracić swoich danych przy instalacji nowszej wersji programów. Z powyższych powodów zdecydowaliśmy się rozdzielić wymienione wcześniej cztery kategorie plików w naszym pakiecie dystrybucyjnym.

Przy standardowej kompilacji obok programów bazy danych i podstawowych programów użytkowych kompilowane są dodatkowe programy. Mają one zastosowanie wyłącznie podczas modyfikacji samego Firebirda, są zbędne z punktu widzenia użytkownika końcowego. Zdecydowaliśmy że nie będziemy ich umieszczać w naszym pakiecie instalacyjnym, zostawiamy tylko niezbędne programy i biblioteki.

4.1 Dostosowanie kodu

Przyjęliśmy że programy i biblioteki bazy Firebird będą instalowane w katalogu `/opt/cfw/`, a dokładniej w odpowiednich jego podkatalogach:

`bin` – programy użytkowe, takie jak konsola,

`sbin` – programy bazy danych (demony),

`lib` – biblioteka klienta `fbclient`,

`lib/firebird` – biblioteki językowe i procedur,

`include/firebird` – pliki nagłówkowe do bibliotek,

`etc` – pliki konfiguracyjne,

`doc/firebird` – podstawowa dokumentacja,

`share/firebird` – pliki z tłumaczeniami komunikatów na różne języki.

Pliki baz danych, logi oraz pliki tymczasowe instalowane są w katalogu

`/var/cfw/firebird`

Takie rozmieszczenie plików wymagało przeróbek w plikach Makefile i kodzie źródłowym. W plikach Makefile możemy określić ścieżki do katalogów do których zostaną wkopiowane odpowiednie pliki podczas instalacji. Niestety jest to tylko fizyczne określenie położenia tych plików. Aby Firebird po skompilowaniu umiał je znaleźć należało dostosować kod źródłowy. W związku z tym wprowadziliśmy wiele zmian w różnych plikach, w miejscach gdzie były odwołania do bibliotek, plików baz danych, pliku konfiguracyjnego oraz plików tymczasowych.

4.2 Zarządzanie instalacją oprogramowania w Solaris

Nowoczesne systemy operacyjne dostarczają narzędzi do nadzoru i zarządzania instalacji i aktualizacji oprogramowania. W Solaris mamy zestaw programów narzędziowych, takich jak:

`pkgadd` – dodawanie i aktualizacja oprogramowania,

`pkgrm` – deinstalacja oprogramowania,

`pkginfo` – informacje na temat zainstalowanego oprogramowania,

`pkgchk` – weryfikacja poprawności instalacji,

`pkgproto` – tworzenie listy plików wchodzących w skład przygotowywanego pakietu instalacyjnego,

`pkgmk` – utworzenie pakietu instalacyjnego,

`pkgtrans` – przekształcanie formatu pakietu instalacyjnego.

Z jednej strony pozwalają one zarządzać pakietami oprogramowania, z drugiej dają możliwość tworzenia własnych pakietów. W Solaris są ściśle określone zasady budowy takiego pakietu. Pakiet (ang. package) jest tutaj najmniejszym zestawem plików jaki możemy zainstalować bądź odinstalować. Solaris przechowuje listę wszystkich zainstalowanych plików. Umożliwia to szybkie wykrycie konfliktu podczas instalacji, daje też możliwość dokładnej deinstalacji.

W skład pakietu oprogramowania wchodzą: plik z informacjami o pakiecie, lista plików wchodzących w skład pakietu, pliki do zainstalowania oraz skrypty (`preinstall`, `postinstall`, `preremove` i `postremove`) uruchamiane w trakcie instalacji i deinstalacji.

4.3 Przygotowanie pakietu

Baza Firebird może działać z uprawnieniami superużytkownika (root), ale bezpieczniej jest utworzyć użytkownika i uruchamiać tą bazę z jego uprawnieniami. Podyktowane jest to tym, że tworząc bazę danych w Firebirdzie podajemy pełną ścieżkę do jej pliku. Jako root Firebird mógłby w ten sposób zniszczyć ważne pliki systemowe. Zwykle zakłada się w systemie grupę i użytkownika o nazwie `firebird`, nie dając im żadnych specjalnych uprawnień.

Przygotowując pakiet z bazą Firebird chcemy aby po jego instalacji mieć w pełni działający serwer bazy danych, gotowy do uruchomienia zaraz po instalacji. W związku z tym przygotowaliśmy skrypt `preinstall`, który sprawdza obecność grupy i użytkownika `firebird` w systemie. Jeżeli ich nie ma zgłaszany jest odpowiedni komunikat sugerujący sposób dodania i instalacja jest przerywana. Zrezygnowaliśmy z dodawania odpowiedniej grupy i użytkownika poprzez skrypt, gdyż jest to zadanie administratora i powinien mieć on świadomość co się dzieje na administrowanym przez niego systemie. Poniżej przedstawiamy ten skrypt.

```
if [ -z "`grep \"^firebird:\" /etc/group`" ]
then
    echo
    echo "ERROR: group not found: firebird"
    echo
    echo "Add group firebird, eg. as follows:"
    echo
    echo "  groupadd firebird"
    exit 1
```

```
fi

if [ -z "`grep \"^firebird:\" /etc/passwd`" ]
then
    echo
    echo "ERROR: user not found: firebird"
    echo
    echo "Add user firebird, eg. as follows:"
    echo
    echo "  useradd -g firebird -c \"Firebird Server\" \\"
    echo " -d /var/cfw/firebird -s /bin/false firebird"
    exit 1
fi

exit 0
```

Począwszy od Solaris 10 serwisy uruchamiane są nie poprzez skrypty, jak to miało miejsce wcześniej, lecz za pośrednictwem wyspecjalizowanego oprogramowania (`restarter`). Dla zachowania kompatybilności stare skrypty startowe umieszczane w katalogu `/etc/init.d` są respektowane i zachowują się tak, jak w poprzednich wersjach. Ponieważ docelową platformą ma być Solaris 10 chcieliśmy aby nasze oprogramowanie było w pełni zgodne z tym systemem. Dlatego też zamiast skryptu startowego przygotowaliśmy tzw. manifest w XML, opisujący nowy serwis – serwer bazy danych Firebird. W takim manifeście określa się sposób w jaki dany serwis jest uruchamiany i zatrzymywany. Odpowiedni plik XML musi być zaimportowany do systemu. W tym celu przygotowaliśmy skrypt `postinstall`, który tym się zajmuje.

```
MANIFESTDIR=/opt/cfw/share/svc/manifest

for MANIFEST in /application/database/firebird
do
    SERVICE=svc:`echo $MANIFEST | sed s/-/:/`
    if svccfg import ${MANIFESTDIR}${MANIFEST}.xml >/dev/null
    then
        echo
        echo "New service imported: $SERVICE"
    else
        echo
        echo "Failed to import service: $SERVICE"
        exit 1
    fi
done

exit 0
```

Przed wyinstalowaniem naszego pakietu dobrze jest zatrzymać serwis bazy danych Firebird i usunąć z systemu jego opis. Zajmuje się tym napisany przez nas skrypt `preremove`.

```
for SERVICE in svc:/application/database/firebird
do
    svcadm disable -s $SERVICE >/dev/null
    if svccfg delete $SERVICE >/dev/null
    then
        echo
        echo "Deleted service: $SERVICE"
    else
        echo
        echo "Failed to delete service: $SERVICE"
    fi
done

exit 0
```

Następnym krokiem w przygotowaniu pakietu instalacyjnego było zebranie wszystkich plików, które mają być zainstalowane, w jednym katalogu i takie ich rozmieszczenie w podkatalogach jakie ma być po instalacji. Należało jeszcze przygotować opis samego pakietu i umieścić go w pliku `pkginfo`:

```
PKG=CFWfirebird
NAME=Firebird - SQL Database Server
ARCH=i386
VERSION=1.5.2
CATEGORY=application,server
DESC=Relational database offering many ANSI SQL-99 features.
VENDOR=Firebird Project
SOURCES=http://www.firebirdsql.org/
WEBSITE=http://www.firebirdsql.org/
BASEDIR=/opt/cfw
CLASSES=none
```

W celu uzyskania gotowego pakietu należało zawołać programy `pkgproto`, `pkgmk` i na koniec `pkgtrans`. W efekcie uzyskaliśmy pojedynczy plik zawierający nasz pakiet instalacyjny. Możemy go zainstalować przy pomocy `pkgadd`.

Podsumowanie

W trakcie prac nad kompilacją Firebird, jeden z programistów przygotował pakiet binarny ostatniej wersji tej bazy na Solaris. Pakiet ten z naszego punktu widzenia miał kika wad. Po pierwsze do jego przygotowania użyto kompilatora `gcc`, po drugie nie był dostosowany do wersji Solaris 10. Mieliśmy też zastrzeżenia co do rozmieszczenia w systemie poszczególnych plików wchodzących w skład pakietu.

Efektem tej pracy jest w pełni funkcjonalna baza Firebird na platformie Solaris 10 x86, dostępna w postaci standardowego pakietu instalacyjnego z uwzględnieniem najnowszych rozwiązań w systemie operacyjnym.

Bibliografia

[1] Strona domowa projektu Firebird:
`http://www.firebirdsql.org/`.

[2] Strona domowa projektu STLport:
`http://stlport.org`.

[3] Biblioteka STLport na Freshmeat:
`http://freshmeat.net/projects/stlport/`.

[4] Szczegóły dotyczące extern „C”:
`http://www.parashift.com/c++-faq-lite/mixing-c-and-cpp.html`

[5] Strona domowa projektu SFIO:
`http://www.research.att.com/sw/tools/sfio/`