Optymalizacja programów Open-Source

Optymalizacja czasu uruchamiania programu

Krzysztof Lichota
lichota@mimuw.edu.pl

Wprowadzenie

Dlaczego czas uruchomienia programu jest ważny

- Użytkownik chce zacząć pracę z programem w danej chwili, bo chce coś zrobić, nie chce czekać długo na uruchomienie programu
- Zbyt długi czas uruchomienia programu obniża jego użyteczność – np. zanim uruchomi się przeglądarka WWW możemy zapomnieć co chcieliśmy sprawdzić w Internecie
- Jeśli program uruchamia się długo, użytkownik ma tendencję do unikania uruchamiania go i szuka zastępników

Co się dzieje w trakcie uruchomienia programu

1. Program macierzysty wykonuje fork() (tanie, bo Linux ma stosuje copy-on-write przy kopiowaniu pamięci procesu)
2. Proces potomny wykonuje exec(), co powoduje wymazanie pamięci procesu, wmapowanie nowego pliku wykonywalnego i skok do jego procedury wejścia
3. Uruchamiany jest dynamiczny linker, który dolinkowuje biblioteki dynamiczne i łączy je z programem (drogie)
4. Następuje skok do funkcji main().

Co się dzieje w trakcie uruchomienia programu (2)

1. Program wczytuje swoją konfigurację, o ile ją ma.
2. Program inicjalizuje swoje struktury potrzebne do działania.
3. Program doładowuje dynamiczne biblioteki np. na podstawie konfiguracji.
4. Program komunikuje się z demonami i usługami systemowymi.
5. Program tworzy swój interfejs graficzny.
6. Użytkownik może zacząć korzystać z programu.

Kosztowne operacje w trakcie uruchomienia programu

- Linkowanie – intensywnie wykorzystuje CPU i pamięć (w tym również doładowywanie bibliotek w trakcie działania za pomocą dlopen(), np. wtyczek)
- Wczytywanie wmapowanych plików on-demand, np. przy linkowaniu, na początku działania - losowe I/O w plikach
- Wczytywanie plików konfiguracyjnych i innych plików niezbędnych do działania (ikon, tłumaczeń, itp.) – losowe I/O po systemie plików, dużo operacji stat()

Dynamiczne linkowanie

Dynamiczne linkowanie

- W trakcie uruchomienia programu w formacie ELF uruchamiany jest dynamiczny linker, który łączy program z dynamicznymi bibliotekami (i dynamiczne biblioteki ze sobą) pod adresami, który zostały im przydzielone w przestrzeni adresowej
- Dynamiczne linkowanie wymaga poprawienia pewnych miejsc w przestrzeni adresowej procesu – poprawka zwiększa czas uruchomienia i liczbę stron pamięci używanych przez proces

GOT

- Odwołania do funkcji i symboli wewnątrz biblioteki nie są poprawiane – używane jest adresowanie względne (PIC – position independent code)
- Odwołania do kodu i danych z innej biblioteki idą przez GOT (Global Offset Table) – tablicę wskaźników do symboli używanych w innej bibliotece
- Program używa adresowania względem rejestru bazowego, który wskazuje na GOT
- GOT jest poprawiany w trakcie uruchamiania programu przez dynamiczny linker, jeśli adres biblioteki w pamięci się zmienił od czasu linkowania

PLT i lazy binding

- Ponieważ nie wszystkie symbole odwołujące się do kodu są używane od razu przez program, stosowane jest „leniwe rozwiązywanie symboli” - symbol jest wyliczany dopiero gdy nastąpi skok do danej funkcji
- Używana jest do tego tablica PLT (Process Linkage Table), która zawiera adres pod który należy skoczyć
- Początkowo PLT zawiera puste wpisy, które powodują rozwiązanie symbolu i poprawienie wpisu w tablicy tak, że kolejne skoki już są pod właściwy adres

PLT i lazy binding (2)

- Lazy binding skraca czas uruchomienia programu, ale kosztem późniejszych przestojów
- Można wyłączyć lazy binding ustawiając zmienną środowiskową LD_BIND_NOW=1

Łancuchy bibliotek

- Zgodnie z semantyką wyszukiwania symboli w plikach ELF tworzony jest łańcuch bibliotek, które mogą zawierać symbol
- Ten łańcuch jest określony przez ścieżki podane przy linkowaniu programu, ścieżki używane przez dynamiczny linker (/etc/ld.so.conf), zmienne środowiskowe LD_LIBRARY_PATH i LD_PRELOAD
- Daje to dużą elastyczność, ale sprawia, że wyszukiwanie symbolu jest kosztowne

Wyszukiwanie symbolu w bibliotece

- W każdym miejscu łańcucha symbol jest wyszukiwany po nazwie (potencjalnie bardzo długiej)
- Symbole są poustawiane w tablice haszujące – liczba symboli ma wpływ na czas sprawdzenia czy symbol jest zdefiniowany w danej bibliotece
- Wyszukanie symbolu składa się z wielu niepomyślnych wyszukań, bo np. symbol zdefiniowany w bibliotece libc może być najpierw wyszukiwany w bibliotece Qt, X, itd.
- Symbole często różnią się dopiero na ostatnich znakach (z powodu wymagań ABI C++)

Czas przetwarzania relokacji

- Relokacje dzielą się na nazwane i względne
- Względne to dodanie pewnego przesunięcia (np. adresu biblioteki) w określonym miejscu (stosunkowo tanie)
- Nazwane to wyszukanie nazwy symbolu w łańcuchu bibliotek i poprawienie miejsca (drogie)
- Czas dynamicznego linkowania to O(R+nr), gdzie:
  - R to liczba relokacji względnych
  - r to liczba relokacji po nazwie
  - n to liczba bibliotek dynamicznych

Linkowanie programów w C++

- Tablice funkcji wirtualnych dla klas w C++ zawierają wskaźniki do metod dziedziczonych z innych klas, potencjalnie z innych bibliotek, np. każdy element interfejsu w KDE dziedziczy pośrednio po klasie QWidget z biblioteki Qt
- Wskaźniki w tablicach funkcji wirtualnych są relokowane i musi to być wykonane przy uruchomieniu programu, ponieważ nie ma możliwości sprawdzenia, czy symbol będzie używany
- Dlatego programy w C++ mają dużo relokacji i od razu przy uruchomieniu

Podglądanie działań linkera

- LD_DEBUG=symbols <program> – pokazuje gdzie linker szuka poszczególnych symboli
- LD_DEBUG=statistics <program> – pokazuje statystyki dynamicznego linkera
- LD_DEBUG=help <program> – pokazuje listę opcji

Wnioski

- Żeby skrócić czas linkowania, należy:
  - zmniejszyć liczbę bibliotek i/lub
  - zmniejszyć liczbę symboli do przeszukania i/lub
  - zmniejszyć długość symboli i/lub
  - usunąć potrzebę relokacji i/lub
  - zamienić relokacje nazwane na względne

Ograniczanie widoczności symboli

- Domyślnie wszystkie funkcje i zmienne w C/C++ są eksportowane
- Ograniczając liczbę eksportowanych symboli zmniejszamy czas wyszukania symboli przy linkowaniu
- W standardzie C, żeby zablokować eksportowanie symbolu z pliku należy zadeklarować zmienną/funkcję jako „static” (Uwaga! W klasach C++ to słowo oznacza co innego)
- Od wersji gcc 4.0 istnieje możliwość domyślnego ukrywania symboli za pomocą opcji kompilatora -fvisibility=hidden

Ograniczanie widoczności symboli (2)

- Potrzebne symbole (funkcje, zmienne, klasy, metody) można uwidocznić za pomocą: __attribute__ ((visibility ("default")))
- Istnieje też możliwość zdefiniowania mapy eksportowanych symboli dla programu przy linkowaniu za pomocą opcji --version-script linkera

Inne sztuczki dotyczące linkowania

- prelink – wykonanie relokacji bibliotek do wyliczonych pozycji (unika w ogóle relokacji, nie działa dla dlopen())
- objprelink1/2 – zamiana relokacji nazwanych na względne i inne sztuczki
- kdeinit – w systemie istnieje program z dolinkowanymi bibliotekami, który przy uruchomieniu programu dostaje sygnał, żeby się sforkować i załadować program skompilowany jako biblioteka
- quickstartery – utrzymywanie gotowego do wykonania programu w pamięci w ukryciu (np. OpenOffice)

Zmiana działania programu w trakcie startu

Ogólne zasady optymalizacji startu

- Odwlec wszystko, co nie jest niezbędne do działania programu na jak najpóźniej
- Nie zatrzymywać wykonania programu w oczekiwaniu na jakieś zdarzenie (np. na nawiązanie połączenia sieciowego)
- Jeśli musimy zrobić coś kosztownego (np. sięgnąć do dysku), to starać się zrobić za jednym zamachem maksymalnie dużo, by potem to procentowało – np. wczytać od razu całą bibliotekę do pamięci

Odwlekanie

- Jeśli nasz program może obyć się przez jakiś czas bez pewnej czynności (np. jakiegoś połączenia sieciowego, sprawdzania pisowni w dokumencie), odkładamy ją na później albo wykonujemy w tle
- Jeśli nasz program nie może kontynuować bez danej funkcji, komunikujemy się z procesem w tle i czekamy na jego zakończenie
- Wady odwlekania: po uruchomieniu programu może on intensywnie wykonywać odwleczone czynności (np. uruchomienie Windows XP)

Odwlekanie (2)

- Przykład: okno otwierania pliku jest potrzebne dopiero gdy użytkownik kliknie przycisk otwierania pliku, więc można tę funkcję wydzielić do osobnej biblioteki i wczytywać ją dopiero, gdy będzie potrzebna
- Przykład (negatywny): blokowanie pisania w OpenOffice na czas wczytania modułu sprawdzania pisowni
- Przykład (hipotetyczny): wczytywanie w tle kanałów RSS

Caching

- Stosowane jeśli wygenerowanie początkowego stanu programu jest kosztowne
- Przy zamykaniu zapisujemy przetworzone dane
- Przy kolejnym uruchomieniu zamiast generować dane na nowo wczytujemy dane zapisane przy zamykaniu

Caching

- Przykłady:
  - prekompilowane moduły .pyc w Pythonie
  - sparsowane pliki w KDevelopie
  - generowanie grafiki z plików SVG w programach KDE
  - skalowanie tapety
  - baza utworów w Amaroku
  - (hipotetycznie) zapisywanie sparsowanej strony HTML w przeglądarce, wygenerowanego fraktala w programie do fraktali, itp.

Generowanie fragmentaryczne

- Jeśli program musi coś pokazać, generujemy tylko to, co konieczne do pierwszego wyświetlenia, resztę robimy w tle albo kiedy wymusi to działanie użytkownika
- Przykłady:
  - wczytywanie znaczników plików mp3 w WinAmpie
  - pokazywanie dużych obrazków (w każdym sensownym programie graficznym)
  - (hipotetycznie) odczytywanie szczegółowych informacji o plikach w katalogu w menedżerze plików

Prefetching

- Jeśli wiemy, że w trakcie uruchomienia programu będziemy potrzebować jakiegoś pliku (np. biblioteki), zlecamy wczytanie go z wyprzedzeniem i wszystkich potrzebnych części, żeby system mógł sobie to w miarę wolnych zasobów wykonać
- Optymalnie jest zlecić to z jak najniższym priorytetem, żeby nie opóźniało żądań wysyłanych na bieżąco
- Nie można przesadzić z ilością tak wczytywanych plików, żeby nie zarżnąć systemu i żeby wczytane zostały w potrzebnej nam kolejności

Prefetching (2)

- Przykład: ręczny prefetching bibliotek w OpenOffice.org 2.1
- Przykład: mój projekt na Google Summer of Code: http://code.google.com/p/prefetch/

Bibliografia

- http://www.die.net/doc/linux/man/man8/ld-linux.8.html
- http://www.iecc.com/linker/linker10.html i pozostałe z http://www.iecc.com/linker/
- http://people.redhat.com/drepper/dsohowto.pdf ?
- http://objprelink.sourceforge.net/objprelink.html
- http://people.redhat.com/jakub/prelink.pdf
- http://people.redhat.com/drepper/dsohowto.pdf
- http://go-oo.org/~michael/OOoStartup.pdf
- http://lwn.net/Articles/192624/
- http://web.archive.org/web/20060425103355/http://www.s