Programowanie niskopoziomowe - laboratorium

Narzędzia do debugowania - Valgrind - konspekt

Arkadiusz Kasprzak
Wydział Fizyki i Informatyki Stosowanej
Informatyka Stosowana III rok

Data seminarium: 26.03.2019
Data laboratorium: 14.05.2019

Spis treści

Streszczenie

Tematem zajęć laboratoryjnych jest jedno z najpopularniejszych narzędzi do debugowania - *Valgrind*. Celem przygotowanych ćwiczeń jest zapoznanie studentów z jego wybranymi komponentami: *Memcheck*, *Massif* oraz *Cachegrind*. Ćwiczenia obejmują naukę sposobu użytkowania tych narzędzi oraz prawidłowej interpretacji zwracanych przez nie wyników.

1 Valgrind - wstęp

1.1 Czym jest Valgrind?

Valgrind jest **frameworkiem** do tworzenia narzędzi przeprowadzających dynamiczną analizę binarną kodu dzięki zastosowaniu techniki *DBI* (*Dynamic Binary Instrumentation*). Dokładny sposób działania tychże mechanizmów został omówiony na seminarium - nie zostanie on tutaj przedstawiony, gdyż jego znajomość nie jest konieczna do rozwiązania zadań laboratoryjnych.
Architekturę Valgrinda można przedstawić za pomocą poniższego równania:

$$Valgrind\ core\ +\ tool\ \ plug\text{-}in\ =\ Valgrind\ tool \tag{1}$$

1.2 Narzędzia

Z punktu widzenia laboratorium najważniejszymi narzędziami wchodzącymi w skład Valgrinda są:

- **Memcheck** - do wykrywania błędów związanych z pamięcią (np. wycieki)
- **Massif** - narzędzie do analizy *sterty* (*heap*)
- **Cachegrind** - narzędzia symulujące interakcję programu z pamięcią *cache*

Inne narzędzie to m.in. Callgrind, Hellgrind i DRD - jednak ze względu na ograniczony czas nie zostaną one użyte w trakcie laboratorium.

1.3 Valgrind - podstawowe użycie

Wywołanie Valgrinda z linii poleceń:

```
valgrind --tool=<tool_name> <program_name>
```

gdzie: <*tool_name*> to nazwa użytego narzędzia (np. memcheck), natomiast <*program_name*> to badany program. Dostarczone pliki Makefile pozwalają na skompilowanie programu z flagą -g, co jest konieczne do poprawnego użytkowania niektórych narzędzi Valgrinda.

2 Memcheck

2.1 Wstęp teoretyczny

Memcheck to narzędzie służące do wykrywania błędów związanych z zarządzaniem pamięcią. Wykrywa błędy takie jak:

- wycieki pamięci
- problemy ze stertą (heap), takie jak podwójne zwalnianie pamięci czy użycie niedopasowanych funkcji alokujących i zwalniających pamięć (np. alokacja za pomocą malloc i zwalnianie za pomocą delete).
- użycie niezainicjowanych wartości
- próba dostępu do pamięci, do której nie powinniśmy się odwoływać (np. zwolnionej lub niezaalokowanej)
- próba przekazania do funkcji typu strcpy() nakładających się obszarów pamięci

2.2 Podstawy użycia

Uruchomienia Memchecka:

```
valgrind <program_name>
```

lub

```
valgrind --tool=memcheck <program_name>
```

przy czym nie ma potrzeby podawania opcji `--tool=memcheck`, ponieważ Memcheck jest dla Valgrinda narzędziem domyślnym.

2.2.1 Przydatne opcje

Poniżej lista kilku opcji, które mogą okazać się przydatne w czasie laboratorium:

- `--leak-check=<no | summary | yes | full> [default: summary]` - określa szczegółowość wyjścia
- `--track-origin=<yes | no> [default: no]` - określa, czy Memcheck powinien śledzić pochodzenie niezainicjowanych zmiennych

- `--show-leak-kinds=<definite, indirect, possible, reachable | all | none > [default: definite, possible]` - pozwala wybrać typy wycieków pamięci, dla których zostanie wyświetlony komunikat

2.3 Makra - client requests

Valgrind posiada mechanizm tzw. Client Requests (żądania klienta) - są to makra, których podzbiór dostępny jest do użycia z poziomu kodu źródłowego badanego programu. Mechanizm ten pozwala na przekazanie prostych zapytań do rdzenia i aktualnie używanego narzędzia. W celu użycia makr należy dołączyć odpowiedni plik nagłówkowy:

- *valgrind/valgrind.h* - dla makr prowadzących interakcję z rdzeniem Valgrinda
- *valgrind/memcheck.h* - dla makr prowadzących interakcję z narzędziem, w tym przypadku z Memcheckiem.

Poniżej lista przydatnych makr rdzenia:

- RUNNING_ON_VALGRIND - zwraca 1 jeśli program został uruchomiony za pomocą Valgrinda. W przeciwnym razie zwraca wartość 0.
- VALGRIND_COUNT_ERRORS - zwraca liczbę znalezionych do tej pory błędów. Nie działa dla wszystkich narzędzi - działa np. dla Memchecka ale już np. dla Cachegrinda zawsze zwraca 0 (ponieważ Cachegrind nie wyszukuje błędów)
- VALGRIND_PRINTF - działa podobnie do printf() z języka C, ale wypisuje wiadomość tylko gdy uruchamiamy program za pomocą Valgrinda. Zwraca ilość wypisanych znaków.
- VALGRIND_PRINTF_BACKTRACE - wypisuje stack trace, czyli listę zagnieżdżonych funkcji. Można dodać swoją wiadomość, podobnie jak w przypadku poprzedniego makra.

Lista przydatnych makr dla Memchecka:

- VALGRIND_CHECK_VALUE_IS_DEFINED - sprawdza, czy Memcheck uważa podaną wartość (lvalue) za niezainicjowaną. Wypisuje wiadomość z błędem jeśli tak. Nie zwraca nic.

- VALGRIND_DO_LEAK_CHECK - wykonuje pełne sprawdzenie obecności wycieków pamięci w danym momencie. Użyteczne, ponieważ możemy sprawdzać co pewien czas, co może być pomocne w określeniu, gdzie pojawia się wyciek.
- VALGRIND_CHECK_MEM_IS_ADDRESSABLE (ptr, size) - sprawdza, czy do danego kawałka pamięci możemy się odwołać (czyli np. czy została zaalokowana). Jeśli nie, to wypisuje informację na ten temat.
- VALGRIND_CHECK_MEM_IS_DEFINED (ptr, size) - sprawdza, czy wartości w pamięci są zdefiniowane. Jeśli nie, to wypisuje informację na ten temat.

2.4 Tworzenie filtrów

Memcheck poza kodem aplikacji sprawdza również kod dołączonych bibliotek. Może to potencjalnie generować problemy - Memcheck będzie wyświetlał powiadomienia o błędach, na których istnienie możemy nie mieć wpływu, bo pochodzą z wyżej wspomnianych bibliotek. Rozwiązaniem tego problemu są tzw. **suppression files** - pliki pozwalające filtrować, jakie błędy mają być wyświetlane na wyjściu Memchecka. Filtry umieszcza się w pliku z rozszerzeniem .supp. Poniżej przykład filtru:

```
{
       Condition_1
       Memcheck:Cond
       fun:_Z3fun
       fun:main
}
```

gdzie:

- pierwsza linia to nazwa - może być jakakolwiek pozwalająca na identyfikację
- druga linia to nazwa narzędzia i typ filtru - jaki błąd jest blokowany
- opcjonalna trzecia linia (nie ma jej w tym przykładzie) - może zawierać dodatkowe informacje dla niektórych filtrów
- pozostałe linie - kontekst wystąpienia błędu, czyli łańcuch funkcji do niego prowadzących

Przydatne opcje:

- `valgrind --gen-suppressions=yes ./a.out` - wyświetla informacje o błędach związanych z pamięcią w sposób pozwalający łatwo tworzyć filtry (w formacie jak wyżej)
- `valgrind -v --suppressions=<plik.supp> ./a.out` - uruchamia Valgrinda z filtrami zdefiniowanymi w pliku <plik.supp>

2.5 Memcheck - zadania

2.5.1 Zadanie 1 - wykrywanie i poprawa błędów związanych z pamięcią za pomocą narzędzia Memcheck

Na pierwsze zadanie składają się trzy pliki: main.cpp, LinkedList.cpp oraz LinkedList.hpp. Program stanowi prostą implementację listy łączonej jednokierunkowej (linked list). Zaproponowana implementacja zawiera błędy związane z zarządzaniem pamięcią. Celem zadania jest poprawa programu w taki sposób, by błędy zostały wyeliminowane, a wyjście programu było zgodne z oczekiwanym (zostało ono podane na końcu pliku main.cpp). Aby wykonać zadanie należy posłużyć się narzędziem Memcheck. Dołączone pliki można modyfikować, natomiast proszę nie stosować rozwiązania polegającego na zastąpieniu funkcji main() samym wypisywaniem.

2.5.2 Zadanie 2 - Wykorzystanie client requests

Na drugie zadanie składa się jeden plik: main.cpp. Zadanie polega na wykorzystaniu poznanych żądań klienta (client requests) - czyli makr udostępnianych przez Valgrinda i jego narzędzia. Należy napisać 4 krótkie funkcje tak, by osiągnąć zadane wyjście programu. Pod funkcją main() znajdują się dwa przypadki możliwego wyjścia:

- jedno, gdy uruchamiamy program bez Valgrinda
- drugie, gdy uruchamiamy program z jego pomocą

Proszę doprowadzić do sytuacji, gdzie program będzie mógł dać oba z tych wyjść.

2.5.3 Zadanie 3 - Tworzenie suppression files

Zadanie trzecie składa się z jednego pliku: main.cpp. Program zawiera sporą ilość błędów związanych z zarządzaniem pamięcią. Celem jest napisanie pliku z odpowiednimi filtrami (tzw. suppression file) w taki sposób, by Memcheck nie wyświetlał błędów generowanych przez metody klasy ErrorMaker. Nie należy modyfikować pliku main.cpp.

3 Massif

3.1 Wstęp teoretyczny

Massif jest narzędziem do analizy sterty (heap) oraz opcjonalnie stosu.

3.2 Podstawowe użycie

Uruchomienie narzędzie Massif z linii poleceń:

```
valgrind --tool=massif <program>
```

Wywołanie to produkuje plik z wynikami, który następnie można przeanalizować za pomocą ms_print:

```
ms_print <plik_out>
```

Przydatne opcje:

- `--heap=<yes | no>` - włącza profilowanie sterty. Domyślnie yes.
- `--stack=<yes | no>` - włącza profilowanie stosu. Domyślnie no.
- `--massif-out-file=<nazwa>` - pozwala na zmianę domyślnej nazwy pliku wynikowego
- `--time-unit=<i | ms | B>` - pozwala wybrać jednostkę czasu używaną przy profilowaniu: i- wykonane instrukcje, ms - milisekundy, B - pamięc alokowana, dealokowana

3.3 Massif - zadania

3.3.1 Zadanie 4 - korzystanie z narzędzia Massif

Zadanie ma na celu przećwiczenie korzystania z narzędzie Massif. Należy przeprowadzić analizę działania dołączonego programu. W tym celu proszę wykonywać instrukcję zawarte w komentarzu w pliku main.cpp.

4 Cachegrind

4.1 Wstęp teoretyczny

Cachegrind jest narzędziem pozwalającym symulować interakcję programu z pamięcią cache.

4.2 Podstawowe użycie

Wywołanie Cachegrind z linii poleceń:

```
valgrind --tool=cachegrind <program>
```

Gdzie <program> to plik wykonywalny. który chcemy zbadać. Plik taki powinien być efektem kompilacji z flagą -g. Powyższe wywołanie tworzy plik o nazwie cachegrind.out.<pid>, gdzie <pid> to ID procesu. Aby uzyskać dane w formacie łatwym do analizy, należy użyć **cg_annotate** na pliku zwróconym przez Cachegrind. Przydatna może być opcja:

```
--auto=<no | yes>
```

Pozwala ona wyświetlać zawartość plików źródłowych w trakcie analizy.

4.3 Cachegrind - zadania

4.3.1 Zadanie 5 - Analiza programu za pomocą narzędzia Cachegrind pod kątem cache-miss

Celem zadania jest przećwiczenie korzystania z narzędzia Cachegrind. Zadanie składa się z dwóch części. Pierwsza z nich polega na analizie programu znajdującego się w katalogu *Tablica_struktur*. Program ten jest nieoptymalny ze względu na korzystanie z pamięci cache - podatny jest na tzw. cache miss. Są to sytuacje, gdy procesor musi pobrać dane z pamięci RAM, gdyż nie znajdują się one

w cache. Program należy poddać analizie za pomocą narzędzia Cachegrind, lokalizując linie kodu najmocniej wpływające na optymalność. Następnie należy przejść do katalogu *Struktura_tablic* gdzie w pliku main.cpp znajduje się druga część zadania. Polega ona na zmianie sposobu przechowywania danych - zamiast tablicy struktur program ma korzystać ze struktury tablic. Po napisaniu kodu program należy przetestować analogicznie do pierwszego i porównać oba podejścia.