**Лабораторна робота №**

**Профілювання програм**

**Профілювання** - збір характеристик роботи програми з метою зрозуміти наскільки ефективно працює програма і які зміни на яких ділянках програми варто робити для подальшої оптимізації.

Профілювання - важливий аспект в процесі розробки ПЗ. За допомогою профілювання можна визначити ті частини в програмному коді, виконання яких займає багато часу, і які, відповідно, вимагають переробки. Це допомагає прискорити процес виконання програми, що апріорі добре. Також, крім виявлення повільно працюючих частин коду профілювання може допомогти в зборі інших статистичних даних, які допоможуть виявити і виправити неочевидні на перший погляд помилки. Існує ручний і інструментальний підхід до профілювання програм.

Ручний спосіб - розстановка вимірів часу вручну.

Інструментальні засоби аналізу програми універсальні і дозволяють збирати велику інформацію про поведінку програми. Проектувальники ПО потребують таких інструментальних засобів, щоб оцінити, наскільки якісно виконана робота. Програмістам вони потрібні, щоб проаналізувати їх програми і ідентифікувати критичні ділянки програми. Обидва способи впливають на поведінку програми.

Недоліками ручного способу є:

* додавання додаткового коду в програму, що призводить до нових помилок і необхідності оптимізації коду;
* аналіз результату в текстовому вигляді;
* великий обсяг вихідних файлів;
* відсутність дерева викликів

Профілювальник - інструмент аналізу продуктивності програми. Збір інформації про поведінку програми здійснюється під час її виконання. Інструментальних засобів профілювання багато (gprof, Oprofile, CodeAnalyst, VTune, Valgring) і для збору інформації про роботу програми вони можуть використовувати різні підходи.

Мабуть найбільш поширеним підходом є семплювання. Це підхід, при якому через певні проміжки часу профілювальник перевіряє, де знаходиться програма і фіксує це становище ототожнюючи його з вихідним кодом.

Gprof - один з простих профілювальник, вільно доступних розробникам ПО. Програма веде підрахунок часу, яке програма провела в тій чи іншій функції, накладає інформацію на граф викликів і результат пише в файл звіту (gmon.out)

gprof може профілювати додатки, написані на мовах C, C ++, Pascal і Fortran 77. Наведені тут приклади використовують C.

**Лістинг 1. Приклад додатку, що виконується довго**

#include <stdio.h>

int a(void) {

int i=0,g=0;

while(i++<100000)

{

g+=i;

}

return g;

}

int b(void) {

int i=0,g=0;

while(i++<400000)

{

g+=i;

}

return g;

}

int main(int argc, char\*\* argv)

{

int iterations;

if(argc != 2)

{

printf("Usage %s <No of Iterations>\n", argv[0]);

exit(-1);

}

else

iterations = atoi(argv[1]);

printf("No of iterations = %d\n", iterations);

while(iterations--)

{

a();

b();

}

}

Як видно з цього коду, цей дуже простий додаток містить дві функції, a і b, обидві з яких виконують тривалі цикли, витрачаючи процесорний час. Функція main просто містить цикл, в якому викликає по черзі ці функції. У другій функції, b, цикл працює в чотири рази довше, ніж в a, тому при профілювання коду ми очікуємо, що 20% часу програма буде витрачати на функцію a, і 80% на b. Дозволимо профілювання і подивимося, чи підтверджуються ці очікування.

Для вирішення профілювання додаємо параметр -pg при виклику компілятора gcc. Виконуємо компіляція наступним чином:

gcc example1.c -pg -o example1 -O2 -lc

Після компонування додатки запускаємо його звичайним способом:

./example1 50000

Після завершення роботи програми ви повинні побачити файл gmon.out, створений в поточному каталозі.

*Як працювати з результатами?*

Перш за все, перегляньте "простий профіль" (flat profile), який ви отримуєте при виконанні команди gprof з наступними параметрами: сам виконуваний файл і файл gmon.out:

gprof example1 gmon.out -p

Отримали наступні результати:

##### Лістинг 2. Простий профіль

Flat profile:

Each sample counts as 0.01 seconds.

% cumulative self self total

time seconds seconds calls ms/call ms/call name

80.24 63.85 63.85 50000 1.28 1.28 b

20.26 79.97 16.12 50000 0.32 0.32 a

З отриманої інформації можна побачити, що, як і очікувалося, приблизно в чотири рази більше часу витрачається на функцію b, ніж на функцію a. Реальні цифри не дуже корисні; вони можуть бути дуже неточними через помилки округлення.

Багато виклики функцій (наприклад printf) не включені в результат. Це відбувається тому, що вони знаходяться в бібліотеці часу виконання C (libc.so), яка (в даному випадку) була відкомпільована з параметром -pgі, отже, інформація профілювання не збирається ні для якої функції з цієї бібліотеки.

«Граф викликів» (call graph) можна отримати наступним чином:

gprof example1 gmon.out -q

Отримали наступні результати:

**Лістинг 3. Граф викликів**

Call graph (explanation follows)

granularity: each sample hit covers 2 byte(s) for 0.01% of 79.97 seconds

index % time self children called name

<spontaneous>

[1] 100.0 0.00 79.97 main [1]

63.85 0.00 50000/50000 b [2]

16.12 0.00 50000/50000 a [3]

-----------------------------------------------

63.85 0.00 50000/50000 main [1]

[2] 79.8 63.85 0.00 50000 b [2]

-----------------------------------------------

16.12 0.00 50000/50000 main [1]

[3] 20.2 16.12 0.00 50000 a [3]

-----------------------------------------------

Можна отримати лістинг "анотований вихідний код" (annotated source), в якому виводиться вихідний код програми з відмітками про кількість викликів кожної функції.

Для використання цієї можливості відкомпілюємо вихідний код з дозволеною налагоджування, для того щоб вихідний код був поміщений в виконуваний файл:

gcc example1.c -g -pg -o example1 -O2 -lc

Знову виконуємо додаток:

./example1 50000

Команда gprof:

gprof example1 gmon.out -A

Результат :

**Лістинг 4. Анотований вихідний код**

\*\*\* File /home/martynh/profarticle/example1.c:

#include <stdio.h>

50000 -> int a(void) {

int i=0,g=0;

while(i++<100000)

{

g+=i;

}

return g;

}

50000 -> int b(void) {

int i=0,g=0;

while(i++<400000)

{

g+=i;

}

return g;

}

int main(int argc, char\*\* argv)

##### -> {

int iterations;

if(argc != 2)

{

printf("Usage %s <No of Iterations>\n", argv[0]);

exit(-1);

}

else

iterations = atoi(argv[1]);

printf("No of iterations = %d\n", iterations);

while(iterations--)

{

a();

b();

}

}

Top 10 Lines:

Line Count

3 50000

11 50000

Execution Summary:

3 Executable lines in this file

3 Lines executed

100.00 Percent of the file executed

100000 Total number of line executions

33333.33 Average executions per line

Підтримка профілювання додається компілятором, тому, якщо ви хочете отримати інформацію для профілювання із будь-якої вашої бібліотеки, включаючи бібліотеку C (libc.a), ви повинні теж відкомпілювати її з параметром -pg. На щастя, багато дистрибутивів поставляються з вже відкомпільованими для підтримки профілювання версіями бібліотек C (libc\_p.a).

У використовуваному дистрибутиві, gentoo, необхідно додати "profile" в прапори USE і ще раз виконати збірку glibc. Після цього має бути видно, що створилася бібліотека /usr/lib/libc\_p.a. Для тих дистрибутивів, які не поставляються з бібліотекою libc\_p в стандартному варіанті, треба перевірити, чи можна встановити її окремо, або завантажити вихідний код glibc і скомпонувати її самостійно.

Після отримання файлу libc\_p.a можна просто повторно скомпонувати приклад наступним чином:

gcc example1.c -g -pg -o example1 -O2 -lc\_p

Якщо ви знову запустіть додаток і отримаєте простий профіль або граф викликів, то повинні побачити безліч функцій часу виконання C, включаючи printf (жодна з яких не є важливою для нашої тестової програми).

*Час користувача або час ядра.*

Тепер, знаючи як використовувати gprof, ми можемо дуже просто і ефективно виконувати профілювання ваших додатків для аналізу та усунення слабких місць в продуктивності.

Однак зараз ми повинні звернути увагу на одне з найважливіших обмежень gprof: під час виконання додатка він профілює тільки час режиму користувача. Зазвичай додатки витрачають певну кількість часу на призначений для користувача код і деяку кількість на "системний код", наприклад, на системні виклики ядра.

Якщо ви трохи зміните лістинг 1, то зможете більш чітко побачити цю проблему:

**Лістинг 5. Додавання системних викликів у лістинг 1**

#include <stdio.h>

int a(void) {

sleep(1);

return 0;

}

int b(void) {

sleep(4);

return 0;

}

int main(int argc, char\*\* argv)

{

int iterations;

if(argc != 2)

{

printf("Usage %s <No of Iterations>\n", argv[0]);

exit(-1);

}

else

iterations = atoi(argv[1]);

printf("No of iterations = %d\n", iterations);

while(iterations--)

{

a();

b();

}

}

Код змінений таким чином, що замість виконання циклів функції a і b викликають функцію бібліотеки часу виконання sleep для затримки роботи на 1 і 4 секунди відповідно.

Відкомпілюємо це додаток:

gcc example2.c -g -pg -o example2 -O2 -lc\_p

і виконаємо його з 30-ю ітераціями:

./example2 30

Отримаємо наступний простий профіль:

**Лістинг 6. Простий профіль, що показує системні виклики**

Flat profile:

Each sample counts as 0.01 seconds.

no time accumulated

% cumulative self self total

time seconds seconds calls Ts/call Ts/call name

0.00 0.00 0.00 120 0.00 0.00 sigprocmask

0.00 0.00 0.00 61 0.00 0.00 \_\_libc\_sigaction

0.00 0.00 0.00 61 0.00 0.00 sigaction

0.00 0.00 0.00 60 0.00 0.00 nanosleep

0.00 0.00 0.00 60 0.00 0.00 sleep

0.00 0.00 0.00 30 0.00 0.00 a

0.00 0.00 0.00 30 0.00 0.00 b

0.00 0.00 0.00 21 0.00 0.00 \_IO\_file\_overflow

0.00 0.00 0.00 3 0.00 0.00 \_IO\_new\_file\_xsputn

0.00 0.00 0.00 2 0.00 0.00 \_IO\_new\_do\_write

0.00 0.00 0.00 2 0.00 0.00 \_\_find\_specmb

0.00 0.00 0.00 2 0.00 0.00 \_\_guard\_setup

0.00 0.00 0.00 1 0.00 0.00 \_IO\_default\_xsputn

0.00 0.00 0.00 1 0.00 0.00 \_IO\_doallocbuf

0.00 0.00 0.00 1 0.00 0.00 \_IO\_file\_doallocate

0.00 0.00 0.00 1 0.00 0.00 \_IO\_file\_stat

0.00 0.00 0.00 1 0.00 0.00 \_IO\_file\_write

0.00 0.00 0.00 1 0.00 0.00 \_IO\_setb

0.00 0.00 0.00 1 0.00 0.00 \_\_\_\_strtol\_l\_internal

0.00 0.00 0.00 1 0.00 0.00 \_\_\_fxstat64

0.00 0.00 0.00 1 0.00 0.00 \_\_cxa\_atexit

0.00 0.00 0.00 1 0.00 0.00 \_\_errno\_location

0.00 0.00 0.00 1 0.00 0.00 \_\_new\_exitfn

0.00 0.00 0.00 1 0.00 0.00 \_\_strtol\_internal

0.00 0.00 0.00 1 0.00 0.00 \_itoa\_word

0.00 0.00 0.00 1 0.00 0.00 \_mcleanup

0.00 0.00 0.00 1 0.00 0.00 atexit

0.00 0.00 0.00 1 0.00 0.00 atoi

0.00 0.00 0.00 1 0.00 0.00 exit

0.00 0.00 0.00 1 0.00 0.00 flockfile

0.00 0.00 0.00 1 0.00 0.00 funlockfile

0.00 0.00 0.00 1 0.00 0.00 main

0.00 0.00 0.00 1 0.00 0.00 mmap

0.00 0.00 0.00 1 0.00 0.00 moncontrol

0.00 0.00 0.00 1 0.00 0.00 new\_do\_write

0.00 0.00 0.00 1 0.00 0.00 printf

0.00 0.00 0.00 1 0.00 0.00 setitimer

0.00 0.00 0.00 1 0.00 0.00 vfprintf

0.00 0.00 0.00 1 0.00 0.00 write

Аналізуючи цей результат, ми можемо побачити, що хоча профайлер і зареєстрував правильно кількість викликів кожної функції, час роботи цих функцій (і, в кінцевому рахунку, всіх функцій) все одно 0.00. Причина цього полягає в тому, що функція sleep насправді робить виклик в простір ядра для затримки додатку, потім перериває виконання і чекає до тих пір, поки ядро ​​НЕ продовжить роботу додатка. Оскільки сумарний час виконання функції в просторі користувача дуже малий у порівнянні з часом очікування в просторі ядра, цей час округлюється до нуля. Програма gprof працює таким чином, що вимірювання ведуться фіксованими інтервалами під час виконання програми. Отже, коли програма не виконується, ніяких вимірів не проводиться.

У цьому є як плюси, так і мінуси. З одного боку, важко оптимізувати додатки, що витрачають велику частину часу в просторі ядра, або працюють повільно через зовнішні чинники, такі як перезавантаження підсистеми вводу / виводу операційної системи. З іншого боку, це означає, що профілювання не схильне до виміру всіх інших подій, що відбуваються в системі (наприклад, якщо інший користувач займає багато часу CPU).

У загальному випадку гарною оцінкою корисності застосування gprof для оптимізації програми є його запуск з командою time. Ця команда вимірює тривалість роботи програми і час, проведений в просторі користувача і ядра.

Запустимо програму, наведену в лістингу 2, в такий спосіб:

time ./example2 30

Отримаємо наступний результат:

**Лістинг 7. Результат команди time**

No of iterations = 30

real 2m30.295s

user 0m0.000s

sys 0m0.004s

З цього видно, що на виконання програми в просторі користувача час майже не витрачається, тому gprof не буде корисним в даному випадку.

**Завдання на роботу:**

1) Проаналізувати програму, заміряти час її виконання, подивитися, де є простір для поліпшення.

2) Виходячи з описаних в теорії механізмів організації пам'яті внести необхідні зміни в структуру коду.

3) Заміряти новий час виконання, при наявності очевидних поліпшень, переходити до п.4, якщо немає - до п.1

4) Зафіксувати нові показники і пояснити, за рахунок чого було досягнуто прискорення.

**Варіанти:**

Варіанти знаходяться в архіві «Профілізація програм: завдання»

**Звіт повинен містити:**

1. Лістинг програми в початковому стані.

2. Виміряні дані про час її роботи.

3. Список проведених поліпшень.

4. Лістинг виправленої програми.

5. Виміряні дані про час роботи виправленої програми.

6. Аналіз отриманих результатів.