02. Дополнительные инструменты разработчика в UNIX-подобных ОС

Егор Орлов

Курс: UNIX-DEV-SYS. Системное программирование в среде UNIX (Linux/FreeBSD). ВИШ СПбПУ, 2021

Содержание

1. Трассировка и отладка

1.1. Основные инструменты

- Отладчик **gdb**
- **ltrace** - трассировка библиотечных вызовов
- **strace**
- **ptrace**

1.2. Использование отладчика gdb

1.2.1. Установка отладчика gdb

- Debian, Ubuntu, ALTLinux

```
$ apt-get install gdb
```

1.2.2. Опции gcc для включения отладочной информации

- Для удобной работы с отладчиком необходимо, чтобы все модули программы были откомпилированы с параметрами **-ggdb** или **-g**
  - **-g** - включение в результирующие файлы отладочной информации - имена переменных и функций, номера строк исходных файлов и т.п. в формате, родном для ОС
  - **-ggdb** - добавление расширенной отладочной информацией, понятной только **gdb**

```
CFLAGS=-Wall -ggdb
TARGET=simple3
PREFIX=/usr/local

.PHONY: all clean install uninstall

all: $(TARGET)

simple3: simple3.c print.h libPrint.so
        cc $(CFLAGS) simple3.c -L. -lPrint -o $(TARGET)

libPrint.so: print.c print.h
        cc $(CFLAGS) print.c -shared -fPIC -o libPrint.so
```

1.2.3. Запуск программы в отладчике

- Запускаем программу в отладчике

```
$ gdb ./simple3
. . .
For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from ./simple3...
```

  Получаем общую информацию и приглашение

- Передача параметров запускаемой программе

```
$ gdb --args <prog> <p1> <p2> <p3>
```

  Передаем параметры при запуске оладчика

```
(gdb) run <p1> <p2> <p3>
```

Или передаем через команду **run**

1.2.4. Получение справки

```
(gdb) help
```

Общая справка

```
(gdb) help all
```

Список всех команд

```
(gdb) help <cmd>
```

Справка по конкретной команде

```
(gdb) apropos <word>
```

Список команд, связанных с данным ключевым словом

1.2.5. Основные команды отладчика

Команды	Описание
run	Запуск выполнения до ближайшей точки останова
start	Запуск с остановкой в начале функции main
list	Показывает на экране несколько строк программы до и после текущей
inspect	Посмотреть значение переменной, или включающее ее выражение
bt	Посмотреть текущее содержимое стека, т.е. последовательность вызов функций
frame	Позволяет ходить по фреймам bt, чтобы исследовать значения переменных и т.п. в посл-ти вызовов
step	Выполняет одну строчку с заходом в вызываемые функции
next	Выполняет одну строчку без захода в вызываемые функции
until	Выполнять до строки с указанным номером
call	Вызов функции по-имени
set var	Установка значения переменной
break	Задание точки останова
disable	Отключение точки останова
enable	Включение точки останова
ignore	Указание игнорировать точку останова
cont	Продолжение выполнения
quit	Завершение работы отладчика (Ctrl-D)

1.2.6. Работа с точками останова

- **Точки останова(breakpoints)** - являются наиболее удобным средством управления процессом отладки
- Указание точек

`(gdb) break <func>`

Указание точки останова на функции по ее имени

`(gdb) break <num>`

Указание точки останова номером строки в текущем файле

`(gdb) break <file>:<num>`

Указание точки останова в конкретном файле и строке

- Игнорирование точек

`(gdb) ignore <num> <count>`

Игнорировать точку останова с номером num count раз

- Использование точки останова при выполнении условия

`(gdb) cond <num> <cond>`

Остановиться в точке num, если выполнено условие cond

`(gdb) cond 3 var<6`

Остановиться в 3ьей точке, если значение переменной var меньше 6ти

1.2.7. Анализ аварийного завершения работы программы по core-файлу (segfault.c)

```c
int a (int *p) {
    int y = *p;
    return y;
}

int main (void) {
    int *p = 0; /* null pointer */
    return a (p);
}
```

Код программы с переходом по указателю на NULL

```
$ gcc -ggdb -o segfault segfault.c
```

Сборка программы.

- **Core**-файл - файл, содержащий сегмент данных и стека программы на момент ее аварийного завершения.

  При помощи отладчика можно проанализировать, при выполнении какой строки программы произошла ошибка, откуда и с какими параметрами была вызвана функция, содержащая эту ошибочную строку, какие значения были у переменных в этот момент и т.п.

```
$ gdb <prog> core
```

Запуск отладчика для программы с core-файлом. Программа должна указываться именно та, при сбое которой получен используемый core-файл.

- Доступные команды отладчика в режиме core-файла:
  - подразумевается, что выполнение программы аварийно завершено
  - можно пользовать команды, показывающие состояние программы: bt, list, inspect, frame
  - команды, управляющие выполнением - step, next, break, cont - использовать нельзя

1.2.8. Управление созданием core dump в Linux

- По умолчанию как правило отключены, т.к. представляют риски с т.з. безопасности

```
$ sysctl kernel.core_pattern
kernel.core_pattern = core
```

Имя создаваемого core-файла

```
$ ulimit -Sc
0
$ ulimit -Hc
unlimited
```

Ограничения процессов на размер создаваемых core-файлов (мягкое и жесткое). Мягкое ограничение может менять пользователь до уровня жесткого (системного). Значение 0 посути отключает создание core-файлов

```
$ ulimit -c unlimited
```

Установка значения мягкой границы, при условии, что позволяет жесткая.

```
$ vim /etc/security/limits.conf
*   soft  core unlimited
*   hard  core unlimited
```

Персистентная конфигурация. Для всех пользователей-групп установить мягкую и жесткую границу в бесконечность

1.2.9. Анализ процессов с безконечным циклом (infloop.c)

- Есть возможность присоединить отладчик к работающему процессу, в отношении которого имеются подозрения в переходе в бесконечный цикл

```
int fun (int arg) {
        while ( arg > 0 ) {
                arg = arg % 1000000;
                arg++;
        }
        return 1;
}

int main (void) {
        int arg = 1;
```

```
        return fun (arg);
}
```

Текст программы с бесконечным циклом

```
$ gcc -ggdb -o infloop infloop.c
```

Сборка

- Запускаем отладчик, указывая работающий зацикливающийся процесс и собранный образ исполняемого файла

```
$ gdb <prog> <pid>
```

При подключении отладчика выполнение программы останавливается. Продолжить выполнение можно при помощи команды cont. Снова остановить - Ctrl-C. Можно использовать точки останова и прочие возможности управление выполнением программы.

- Подключение к работающему процессу может требовать наличия привелегий для системного вызова **ptrace**, чего может не быть у обычного пользователя. Т.е. может потребоваться УЗ суперпользователя.

1.2.10. Отключение блокировки ptrace в Linux

- По соображеним безопасности трассировка запущенных процессов может быть запрещена за исключением процессов-потомков. В этом случае будет работать отладка процесса, запущенного из среды отладчика, но не будет возможность подсоединиться к процессу, запущенному отдельно без привелегий супервользователя

```
$ cat /proc/sys/kernel/yama/ptrace_scope
1
```

1 - Блокировка ptrace включена, 0 - выключена

```
$ sysctl kernel.yama.ptrace_scope=0
```

Выключение блокировок

- Выключение действует на все приложения системы, что снижает безопасность, т.к. какое-нибудь malware может цепляться к браузеру, перехватывать пароли и т.п.
- Можно выполнить включение отдельно для тех приложений, которым это может потребоваться (gdb, strace) путем установки соответствующей capability (привелегии) на исполняемый файл.

```
$ setcap cap_sys_ptrace+ep /usr/bin/gdb
$ getcap /usr/bin/gdb
/usr/bin/gdb = cap_sys_ptrace+ep
```

1.3. Трассировка системных вызовов - strace

1.3.1. Назначение трассировки

- Трассировка обычно помогает, когда нет возможности производить отладку или пересобрать приложение с включением отладочной информации

- **strace** позволяет посмотреть к каким системным вызовам и с какими аргументами обращается исследуемая программа

1.3.2. Установка

- Debian/Ubuntu/AltLinux

```
$ apt-get install strace
```

1.3.3. Запуск трассировки вызовов процесса

- Запуск трассировки программы

```
$ starace <prog>
```

- Вывод результата трассировки в файл

```
$ strace -o trace.log <prog>
```

- Указание в выводе идентификатора процесса - полезно для отладки многопроцессных приложений

```
$ strace -f <prog>
```

- Подключение к работающему процессу через механизм **ptrace**. Требует привелегии выполнять вызов ptrace

```
$ strace -p <pid>
```

1.3.4. Формат вывода strace

- В выводе **strace** присутствует
  - имя системного вызова
  - аргументы в скобках
  - результат после знака равенства (0 - OK, -1 - ошибка)

1.4. Трассировка библиотечных вызовов - ltrace

1.4.1. Возможности ltrace

- **ltrace** позволяет:
  - запускать процесс из исполняемого файла и наблюдать за его выполнением, а именно
  - отображать вызовы разделяемых (динамически линкуемых) библиотек, соврешаемых процессом
  - отображать сигналы, получаемые процессом
  - отображать производимые процессом системные вызовы

1.4.2. Установка

- Debian/Ubuntu/AltLinux

```
$ apt-get install ltrace
```

1.4.3. Трассировка вызовов разделяемых библиотек

```
$ ltrace <prog>
```

- Добавление времени в вывод

```
$ ltrace -t <prog>
```

-tt добавляет в вывод микросекунды. **-T** добавляет в вывод время, проведенное внутри каждого вызова.

- Отображение системных вызовов в добавление к библиотечным.

```
$ ltrace -S <prog>
```

2. Основы CMake

2.1. Назначение

- **CMake** - кроссплатформенная утилита для автоматизации сборки программного обеспечения из исходного кода.
  - сама сборкой не занимается, представляет собой **front-end** над такими инструментами сборки как **make**, и средами разработки, например VS, XCode
  - является современной, более производительной и простой в использовании заменой для широко применяемой в открытых проектах системы **autotools**
- **CMakeLists.txt** - файл с описанием параметров сборки, ее правил и целей, размещается в корне проекта.

  По итогам обработки этого файла в **сборочном каталоге** создаются необходимые конфигурационные файлы для сборки ПО выбранным инструментов сборки, например **make**
- https://cmake.org/
- Переход открытых проектов на CMake
  - https://lwn.net/Articles/188693/

2.2. Установка

- Debian/Ubuntu/AltLinux

```
$ apt-get install cmake
```

- Возможности

```
$ cmake --help
```

Внизу в разделе Generators можно посмотреть поддерживаемые варианты систем сборки (cmake back-ends)

2.3. Сборка простого проекта

- Готовим CMakeLists.txt

```
cmake_minimum_required(VERSION 3.0)
project(SimpleSample)
add_executable(simple simple.c)
```

- Создаем каталог для сбборки **out-of-source**

```
$ mkdir build
$ cd build
```

- При сборке указываем каталог с проектом, в корне которого должен лежать **CMakeLists.txt**

```
$ cmake ../
```

- Выполняем сборку из сборочного каталога на основании созданного **Makefile**

```
$ make
```

Сборочный каталог можно очищать без риска поломать проект. Если **CMakeLists.txt** был изменен, то вызов **make** автоматически запустит **cmake**. Если изменить расположение проекта, то нужно очистить временную директорию и запустить **cmake** вручную.

- Очистка каталога сборки

```
$ make clean
```

2.4. Сборка проекта с библиотекой

```
cmake_minimum_required(VERSION 3.0)
project(SimpleSample3)
set(SOURCE_EXE simple3.c)
set(SOURCE_LIB print.c)
add_library(print STATIC ${SOURCE_LIB})
add_executable(simple3 ${SOURCE_EXE})
target_link_libraries(simple3 print)
```

Файл для сборки с созданием статической библиотеки

```
add_library(print STATIC ${SOURCE_LIB})
```

Создаем статическую библиотеку с именем print из исходника, заданного ранее переменной. Для создания динамической библиотеки меняем **STATIC** на **SHARED**

```
add_executable(simple3 ${SOURCE_EXE})
```

Собираем объектный файл основной программы

```
target_link_libraries(simple3 print)
```

Выполняем линковку

- Создаем правила сборки в сборочном каталоге

```
$ cd build
$ cmake ../
```

- Выполняем сборку

```
$ make
$ make clean
```

2.5. Проект с подпроектами

- Делаем нашу библиотеку отдельным подпроектом

```
$ mkdir print/
$ mv print.* print/
```

- **CMakeLists.txt** для подпроекта **print**

```
cmake_minimum_required(VERSION 3.0)
project(print)
set(SOURCE_LIB print.c)
add_library(print SHARED ${SOURCE_LIB})
```

- Сборка подпроекта

```
$ mkdir build && cd build
$ cmake ../
$ make
```

- **CMakeLists.txt** головного проекта

```
cmake_minimum_required(VERSION 3.0)
project(SimpleSample3)
set(SOURCE_EXE simple3.c)
include_directories(print)
add_executable(simple3 ${SOURCE_EXE})
add_subdirectory(print)
target_link_libraries(simple3 print)
```

- Включение каталога подпроекта для поиска заголовочных файлов
  - опция **-I** компилятора gcc
  - теперь включение заголовочного файла можно делать без кавычек

  Эту команду можно вызывать несколько раз, заголовочные файлы будут искаться во всех указанных каталогах.

```
include_directories(print)
```

- Добавление подпроекта в указанном каталоге

```
add_subdirectory(print)
```