Лекция 27. Ошибки и отладка программ на языке Си. Часть 2.

Ревун Артем Леонидович
ст. преп. кафедры вычислительных систем

Курс «Программирование», весенний семестр, 2024
Сибирский государственный университет телекоммуникаций и информатики (г. Новосибирск)

Содержание

- Определение отладки и отладки программ
- Классификация ошибок
- Знакомство с инструменты отладки на Си: printf debugging, GNU Debugger (GDB), Valgrind (Memcheck), strace/ltrace.

Повторение (1)

Основные инструменты отладки на Си:

- **printf debugging** — неформальный, но эффективный способ диагностики.
- **GNU Debugger (GDB)** — основной инструмент отладки на СИ
- Valgrind (Memcheck) — обнаруживает утечки памяти, чтение неинициализированной памяти
- strace/ltrace — трассировка системных вызовов и библиотечных функций

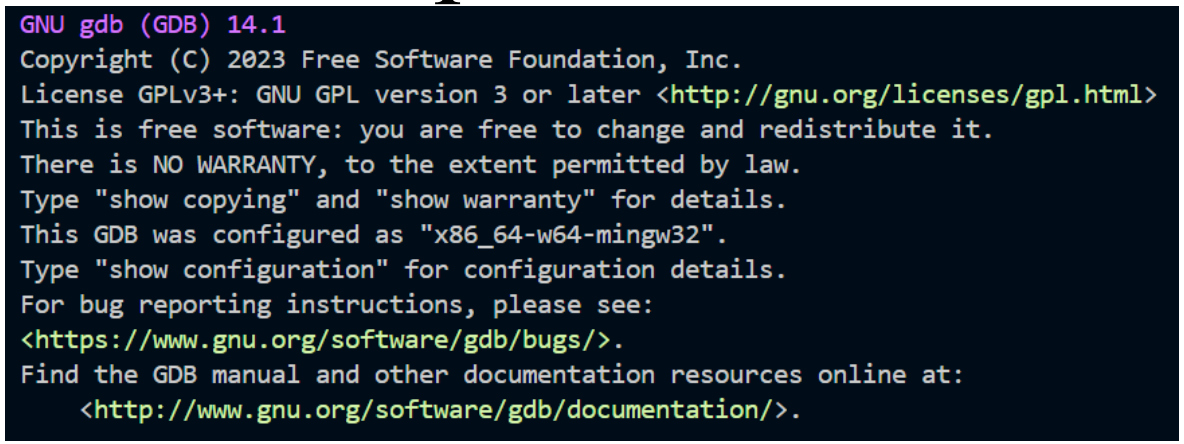
Повторение (2)

printf debugging

Это подход к диагностике программы, при котором вы вручную добавляете вызовы **printf()** (или аналогичные функции вывода) в ключевые точки кода, чтобы **наблюдать** за состоянием переменных, порядком выполнения, а также **для поиска источника ошибок**.

Повторение (3)

GNU Debugger (GDB) — это мощный инструмент отладки с открытым исходным кодом , разработанный проектом GNU. Он позволяет анализировать поведение программы на низком уровне , что особенно важно при системном программировании.



Повторение (4)

Понимание типов ошибок — позволяет находить ошибки, определять их местонахождение и принимать действия к их исправлениям:

- Синтаксические ошибки
- Семантические ошибки
- Логические ошибки
- Ошибки времени выполнения
- Ошибки работы с памятью

Ошибки работы с памятью (1)

- Утечки памяти (memory leaks)
- Использование освобождённой памяти (use-after-free)
- Чтение неинициализированной памяти (uninit read)
- Переполнение буфера (buffer overflow)
- Двойное освобождение (double free)
- Доступ за границами массива
- Разыменование NULL-указателя

Ошибки работы с памятью (2)

```
void leak_memory() {
  int *ptr = (int *)malloc(10 *
                           sizeof(int));
  if (ptr == NULL) {
    fprintf(stderr, "Ошибка выделения
памяти\n");
    return;} ptr[0] = 42;
  printf("Значение: %d\n", ptr[0]); }
```

Ошибки работы с памятью (3)

```
int use_after_free() {
    int *p = malloc(sizeof(int));
    *p = 42;
    free(p);
    return *p;
}
```

Ошибки работы с памятью (4)

```
void uninitialized_memory() {
  char *buffer = malloc(64);
  if (buffer == NULL){
fprintf(stderr,"ALOCERR\n");  return; }
 for (int i = 0; i < 64; ++i) {
   if (buffer[i] >= 32 && buffer[i] <= 126)
     printf("%c", buffer[i]);
   else printf("."); }
 printf("\n"); free(buffer);}
```

Ошибки работы с памятью (5)

```
void buffer_overflow() {
    char buf[10];
    strcpy(buf,
   "Это_слишком_длинная_строка");
 }
```

Ошибки работы с памятью (6)

```
void double_free() {
    int *p = malloc(sizeof(int));
    *p = 42;
    free(p);
    free(p);
}
```

Ошибки работы с памятью (7)

```
void out_of_bounds() {
  int arr[5];
  arr[10] = 42;
}
void out_of_bounds() {
  int arr[5]; arr[10] = 42;
}
```

Ошибки работы с памятью (8)

```
void pointer_dereference() {
    int *p = NULL; *p = 10
}
void head_corruptio() {
    int *p = malloc(10);
    p[-1] = 0xDEADBEEF;
}
```

Инструментальное ПО Valgrind (1)

Valgrind — это мощный инструмент для анализа программ на языке C (и других), который помогает находить ошибки, связанные с работой с памятью. Поддерживает несколько инструментов:

- **Memcheck** — основной, для проверки работы с памятью
- Callgrind — профилирование производительности
- Helgrind / DRD — анализ многопоточных программ

Инструментальное ПО Valgrind (2)

Код выполняется виртуальной среде, где подменяет выполнение каждой инструкции, чтобы отслеживать:

- Как выделяется и освобождается память
- Какие данные используются
- Где происходит обращение к "мусорным" или неинициализированным данным

Он динамически анализирует поведение программы во время выполнения.

Инструментальное ПО Valgrind (3)

Установка:

```
sudo apt update
sudo apt install valgrind
```

или `apt-get install valgrind`

Arch Linux: `pacman -S valgrind`

macOS (через Homebrew): `brew install valgrind`

Windows: через WSL или Cygwin (ограниченная)

Инструментальное ПО Valgrind (4)

`valgrind --tool=memcheck ./myprogram`

Включает подробную проверку утечек памяти .

Флаг: `--leak-check=<comm1>`

`no` или `none` — не проверять утечки.

`summary` — показать количество утечек.

`yes` — показывает места, где память была выделена (стандартная детализация).

`full` — показать все найденные утечки полностью , включая трассировку стека для каждой из них.

Инструментальное ПО Valgrind (5)

Определяет, какие типы утечек будут отображаться

флаг: `--show-leak-kinds=<comm1>`

`definite` – точно потерянная память.

`indirect` – память, на которую нет прямых указателей, но есть косвенные.

`possible` – потенциальные утечки.

`reachable` – память, которая доступна, но не освобождена к концу программы.

`all` – показать все виды утечек.

Инструментальное ПО Valgrind (6)

Включает трассировку источников использования неинициализированных данных

флаг: `--track-origins=yes`

Если программа читает неинициализированную память , **Valgrind** попытается найти в какой момент эта неопределённость появилась впервые в коде. Позволяет определить происхождение этой памяти, а не только лишь место в котором происходит ошибка.

Инструментальное ПО Valgrind (7)

```
int main() {
  memory_leak(); //Утечка памяти
  use_after_free(); //Use-after-free
  uninit_read(); // Чтение
неинициализированной памяти
  buffer_overflow(); //Переполнение
буфера
} ./buggy_prog
```

Инструментальное ПО Valgrind (8)

```
valgrind  --tool=memcheck \
          --leak-check=full \
          --show-leak-kinds=all \
          --track-origins=yes \
          ./my_program
```

Инструментальное ПО Valgrind (9)

```
==50680==    HEAP SUMMARY:
==50680==    in use at exit: 400 bytes in 1 blocks
==50680==    total heap usage: 4 allocs, 3 frees,
                                   4,514 bytes allocated
==50680==    400 bytes in 1 blocks are definitely
                             lost in loss record 1 of 1
==50680==    at 0x483C79B: malloc
                             (vg_replace_malloc.c:380)
==50680==    by 0x109186: memory_leak_example
                                    (baggy_prog.c:6)
==50680==    by 0x10929C: main (baggy_prog.c:28)
```

Инструментальное ПО Valgrind (10)

```
==50680== LEAK SUMMARY:
==50680== definitely lost: 400 bytes in 1 blocks
==50680== indirectly lost: 0 bytes in 0 blocks
==50680== possibly lost: 0 bytes in 0 blocks
==50680== still reachable: 0 bytes in 0 blocks
==50680== suppressed: 0 bytes in 0 blocks
```

Инструментальное ПО Valgrind (11)

```
==50680== For lists of detected and suppressed errors, rerun with: -s
==50680== ERROR SUMMARY: 33 errors from 10 contexts (suppressed: 0 from 0)
Аварийный останов
```

Инструментальное ПО Valgrind (12)

- Обнаруживает множество проблем с памятью
- Не работает напрямую в Windows
- Не ловит все виды ошибок (например, логические)
- Замедляет выполнение программы в разы
- Прост в использовании
- Подходит для тестирования

Инструментальное ПО Valgrind (13)

Callgrind	Профилирование производительности
Massif	Анализ потребления памяти
Helgrind / DRD	Поиск гонок данных в многопоточных программах
SGCheck	Быстрая, но менее точная проверка доступа к массивам

Инструментальное ПО strace (1)

strace (system call trace) — это утилита командной строки в Unix-подобных системах (Linux, BSD и др.), которая перехватывает и записывает все системные вызовы , сделанные процессом, а также все сигналы , полученные этим процессом.

Инструментальное ПО strace (2)

`strace` – это мощный системный анализатор , который позволяет отслеживать системные вызовы и сигналы , которые выполняет программа в процессе своей работы.

Полезен при отладке, анализе поведения программ, поиске ошибок, а также для понимания того, как взаимодействует программа с ядром ОС.

Инструментальное ПО strace (3)

Системные вызовы - (syscalls) способ, которым программа запрашивает выполнение задач у ядра операционной системы.

Открытие файла: open()

Чтение из файла или сокета: read()

Запись в файл: write()

Создание нового процесса: fork(), execve()

Работа с сетью: socket(), connect(), sendto(), recvfrom()

Инструментальное ПО strace (4)

```c
#include <stdio.h>
#include <unistd.h>
int main() {
    printf("Hello, world!\n");
    sleep(2);
    return 0;
}
```

Инструментальное ПО strace (5)

```
execve("./hello", ["./hello"], 0x7fff5a1b9010 /* 93 vars */)
= 0
brk(NULL)                                       = 0x55d1b68f4000
access("/etc/ld.so.preload", R_OK)              = -1 ENOENT (No such
file or directory)
...
write(1, "Hello, world!\n", 13)                 = 13
sleep(2)                                        = 0
exit_group(0)                                   = ?
+++ exited with 0 +++
```

Инструментальное ПО strace (6)

Получение статистики

```
strace -c ./hello
```

Отслеживание только определённых вызовов

```
strace -e trace=read,write ./hello
```

Если у вас уже запущена программа, например, её PID = 1234

```
strace -p 1234
```

Инструментальное ПО strace (7)

Вызов open(), read(), write() будет искусственно прерываться с ошибками — полезно для тестирования обработки ошибок.

```
strace -f -o debug.log -e
fault=open,read,write ./my_program
```

Инструментальное ПО ltrace (1)

ltrace (library trace) — это инструмент в Unix-подобных системах, который перехватывает и выводит вызовы динамических библиотечных функций , которые использует программа во время выполнения. Он также может показывать возвращаемые значения этих функций .

Инструментальное ПО ltrace (2)

```c
#include <stdio.h>
#include <string.h>
int main() {
    char dest[100];
    strcpy(dest, "Hello");
    printf("Length: %zu\n", strlen(dest));
    return 0;}
```

Инструментальное ПО ltrace (3)

```
ltrace ./example
__libc_start_main(0x400526, 1, 0x7ffc85a3e9d8, 0x4005f0
<unfinished ...>
strcpy(0x7ffc85a3e8b0, "Hello")
= "Hello"
strlen("Hello")
= 5
printf("Length: %zu\n", 5Length: 5
)                                  = 9
__libc_start_main returned 0
exit(0)
```

Инструментальное ПО (1)

Критерий	STRACE	LTRACE
Что отслеживает	Системные вызовы	Библиотечные функции
Цель	Анализ взаимодействия с ядром	Анализ работы с библиотеками
Может показать	read,write,open,socket	strcpy,malloc,printf
Сложные случаи	Работает с fork/clone, сигналами	Работает с C++, деманглинг
Полезен для	Поиска IO-проблем, блокировок	Отладки пользовательского кода
Флаг -S	Только системные вызовы	Показывает системные вызовы

Спасибо за внимание

Ревун Артем Леонидович
ст. преп. кафедры вычислительных систем

Курс «Программирование», весенний семестр, 2024
Сибирский государственный университет телекоммуникаций и информатики (г. Новосибирск)