MINIX 3. Отладка

Цилюрик О.И.

< olej@front.ru >

Редакция 1.05

от 02.06.2010

Оглавление

Аннотация

Ниже рассматриваются средства отладки при разработке программ в MINIX 3, и все сопутствующие вопросы: динамический контроль утечки памяти, профилирование, измерение производительности. Относительно каждого средства освещены, по возможности: использование и, где это необходимо, его настройка. Относительно сторонних продуктов, представленных программными пакетами, дополнительно рассматривается их установка.

Введение

Все показанные в тексте протоколы выполнения команд сохранены прямым копированием с экрана терминала (так же как и графические скриншоты); все действия, описываемые в тексте, могут быть повторно вами воспроизведены.

Версии системы

Версии MINIX3 в очень большой мере «волатильны» - разработчики часто вносят существенные изменения, даже не считая должным отражать их даже в MAN страницах. Примеры этого часть описания отрабатывалась на стабильной версии 3.1.6:

```
$ version
```

```
3.1.6, SVN revision 6084, generated Thu Feb 4 19:19:56 GMT 2010
```

В используемой вами версии могут быть, порой, довольно существенные отличия, но основные принципы при этом сохраняются.

Иногда в тексте будет идти речь о Linux, на это будет указано особо. При чём здесь Linux? MINIX система совместима с POSIX, и большинство инструментов, о которых идёт речь в тексте, перенесены из проекта GNU. Рассмотрение аналогий Linux часто много даёт для понимания возможностей инструментов в MINIX.

Обозначения в тексте

В самом тексте, все примеры команд (скопированные с терминала), а также все листинги программных примеров, будут показываться моноширинным шрифтом. Кроме того, в большинстве случаев, пользовательский ввод в записи команды (ввод команды, ввод ответа на запрос программы) будет показан жирным шрифтом, а ответный вывод от системы — обычным. Короткие цитаты из различных источников информации будут показываться *курсивом*. Перед листингами программных примеров будет записываться имя файла кода, оно также будет выделяться курсивом, чтобы не спутываться с последующим содержимым файла.

Средства MINIX

Достаточно трудно разграничить, что из рассматриваемых инструментов относить к собственным средствам MINIX (уже присутствующим в системе без ваших на то усилий: поиск, портирование, установка...), а что — к инструментам от сторонних производителей. Это связано, существенным образом, с тем, что в MINIX уже портировано большое число известных GNU пакетов, и они представлены в качестве дополнительных пакетов на дистрибутивном CD MINIX (например, профайлер `gprof`). Дальше принято следующее разделение: всё, что может быть установлено с дистрибутивного CD MINIX описывается в этом разделе как средства MINIX; то, что требует дополнительного поиска и установки описывается в следующем разделе, как продукт сторонних производителей.

Отладчик mdb

Используем простейшую программу вычисления факториала, которую очень любят при рассмотрении отладчиков:

fact.c :

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

unsigned long factorial( int value ) {
   printf( "%d\n", value );
   if( 1 == value ) return 1;
   return value * factorial( value - 1 );
}

int main( int argc, char* argv[] ) {
   register unsigned long result;
   if( argc != 2 ) {
       printf( "usage: %s <NUMBER>\n", argv[ 0 ] );
       return 1;
   }
   result = factorial( atoi( argv[ 1 ] ) );
   printf( "%ld\n", result );
   return 0;
}
```

Поскольку в MINIX присутствует 2 системы программирования C (ACK `cc` и GNU `gcc`), и в отладке нуждаются

обе, то соберём файл сборки, для компиляции двух образцов отлаживаемых программ:

Makefile :

```
GCC=gcc -g -Wall
CC=cc -g -Wall
LIST=fact_gcc fact_cc
all: $(LIST)
fact_cc:       fact.c
        $(CC) -o $@  $<
fact_gcc:  fact.c
        $(GCC) -o $@  $<
clean:
        rm -f $(LIST)
```

В результате:

```
# ls -l fact_*
-rwxr-xr-x 1 root  operator   22828 May 29 20:02 fact_cc
-rwxr-xr-x 1 root  operator  144802 May 29 20:02 fact_gcc
```

Примечание: отметим разницу в размерах исполнимых программ, созданных `cc` и `gcc`.

Загружаем программы в отладчик `mdb` и рассматриваем таблицы символов:

```
# mdb fact_cc
Sorting 222 MINIX symbols ....
* r 5
Process stopped.
* e T
begtext  T 00000000
crtso    T 00000000
__minix_ T 00001000
__minix_ T 00001011
_factori T 00002000
_main    T 00002032
_atoi    T 00002084
_exit    T 000020B5
...
# mdb ./fact_gcc
Sorting 5741 GNU symbols ....
* r 5
Process stopped.
* e T
00000000 T   begtext
00000000 T   crtso
00001000 T   __minix_mainjump
00001011 T   __minix_unmapzero
00002038 T   _factorial
00002082 T   _main
```

```
...
```

Первое, что бросается в глаза, что таблицы символов имеют принципиально разный формат, но `mdb` успешно справляется с обоими. Второе, это то, что таблица символов программы, скомпилированной `gcc`, больше чем в 20 раз больше, чем скомпилированной `cc` (это, возможно, и объясняет в значительной мере отмеченную ранее разницу в размерах). Возьмём на заметку адреса точки входа в функцию `factorial()` - этот адрес в 2-х вариантах различается; нам этот адрес понадобится вскоре для установки точки останова.

Начальный запуск процесса на отладку выполняем командой `r` (точнее это не запуск, а загрузка на выполнение, потому, что процесс тут же останавливается на начальной точке выполнения `main()`). Устанавливаем точку останова для `gcc` варианта:

```
# mdb ./fact_gcc
Sorting 5741 GNU symbols ....
* r 5
Process stopped.
* 0x2038 b
* B
 1: _factorial  (0x2038)    -
* c 1
Breakpoint hit.
_factorial:
0000:2038  55              push    ebp
...
```

Начало выполнения и последующие продолжения после прохождения точек останова добиваемся командой `c`.

Примечание: фокус здесь, и везде далее, когда мы будем вводить отладчику адреса, состоит в том (и этим часто вводит в заблуждение), что информационная команда `e`, которая выводит всю адресную информацию, выводит её в шестнадцатеричном виде не указывая это явно (`02038`), а мы, в вводе адреса в команде, должны обязательно указывать это явно (`0x2038`); в противном случае адрес будет воспринят как десятичное значение!

Теперь то же самое, но для варианта `cc`, показано прохождение всех уровней рекурсивного вызова:

```
# mdb fact_cc
Sorting 222 MINIX symbols ....
* r 5
Process stopped.
* 0x2000 b
* B
 1: _factori    (0x2000)    -
* c
Breakpoint hit.
_factori:
0000:2000  55              push    ebp
* c
5
Breakpoint hit.
_factori:
0000:2000  55              push    ebp
```

```
* c
4
Breakpoint hit.
_factori:
0000:2000  55              push    ebp
* c
3
Breakpoint hit.
_factori:
0000:2000  55              push    ebp
* c
2
Breakpoint hit.
_factori:
0000:2000  55              push    ebp
* c
1
120
child exited with status 0
*
```

Получение детальной информации по mdb

В MAN по `mdb` информации очень мало... А никакой иной информации по этой утилите нет вообще. Грубую справку по `mdb` можно получить, запустив её без параметров (или, что то же самое, воспользовавшись командой ?):

```
# mdb
Help for mdb. For more details, type 'command ?'
!#      - Shell escape / Set Variable or register
Tt      - Current call / Backtrace all
/nsf    - Display for n size s with format f
Xx [n]  - Disasm / & display reg for n instructions
Rr a    - Run / with arguments a
Cc [n]  - Continue with current signal / no signal n times
Ii [n]  - Single step with / no signal for n instructions
Mm t n  - Trace until / Stop when modified t type for n instructions
k       - Kill traced process
Bb      - Display / Set Break-pt
Dd      - Delete all / one break-points
P       - Toggle Paging
Ll name - Log to file name / and to standard output
Vv      - Version info / Toggle debug flag
e [t]   - List symbols for type t
y       - Print segment mappings
s [n]   - Dump stack for n words
```

```
?       - Help - this screen
@ file  - Execute commands from file
Qq      - Quit / and kill traced process
Usage: mdb -x debug-level [-Ll]logfile exec-file core-file @command-file
       mdb [-fc] file
```

Но этого явно недостаточно для продуктивной работы.

К счастью, `mdb` имеет детальную систему подсказок, которую можно вызывать прямо во время сессии отладки, не нарушая эту сессию (к счастью потому, что помнить всё это множество синтаксических деталей невозможно). Подсказка вызывается при работе `mdb` в виде:

`* <command> ?`

- где `<command>` — это односимвольный идентификатор интересующей нас команды отладчика.

Некоторые самые полезные (и неординарные) подсказки я приведу здесь для ознакомления:

```
* b ?
<address> b [commands]  - Set Break-pt at address
                          commands will be executed by mdb at break-pt
* l ?
l name  - Log to file name and standard output
l      - Reset output to standard output
          Defaults to none
* / ?
<address> /nsf  - Display for n items of size s with format f from address
          n defaults to 1
          s defaults to size of int
             can be b for byte h for short l for long
          f defaults to d for decimal
             can be x X o d D c s or u as in printf
             y treat value as address
             i disasm
* # ?
# <address> cs value    - Set Variable(s) at address to value
                            for c count and size s
                            b for byte h for short or l for long
                            Count or size must be specified
# $xx value             - Set register $xx to value
* x ?
<address> x [n] offset  - Disasm & display registers for n instructions
                          Starting at address+offset
* X ?
<address> X [n] [offset] - Disasm for n instructions
                           Starting at address+offset
* M ?
<address> M t n - Trace until
```

```
                <address> is modified t type for n instructions
                n defaults to 1
                b for byte h for short l for long defaults to size of int
* m ?
<address> m t n - Stop when
                <address> is modified t type for n instructions
                n defaults to 1
                b for byte h for short l for long defaults to size of int
* d ?
<address> d     - Delete one break-point at address
```

Работа с символьными именами

Вернёмся к назначению точек останова. Выполним команду:

```
* _factori
0x00002000
```

Такая команда (просто ввести имя, известное `mdb` из таблицы символов) возвращает адрес размещения этого имени. Таким образом, мы можем назначить точку останова проще, чем делали это раньше (префикс T: указывает, что имя относится к сегменту кода — text — программы):

```
* T:_factori b
* B
 1: _factori    (0x2000)    -
* c
Breakpoint hit.
_factori:
0000:2000  55              push    ebp
```

Можно поступить и ещё проще:

```
* _factori b
```

Потому, что сегмент кода предполагается по умолчанию.

Таким образом, в любом месте команды, где подсказка называет `<address>`, может стоять **имя** из таблицы имён, возможно с уточняющим префиксом сегмента.

Переменные программы

До сих пор мы работали с точками останова. Теперь хотелось бы диагностировать текущие значения переменных программы и, возможно, их оперативное изменение. Изменим незначительно код тестовой задачи:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int gl_int = 321;
char* gl_str = "test string";
/* дальше всё остаётся неизменным */
...
```

Мы добавили 2 фиктивных глобальных переменных в сегменте данных (D:), инициализированные значениями.

Выполняем процесс в отладчике:

```
# mdb fact_cc
Sorting 226 MINIX symbols ....
* r 5
Process stopped.
* _gl_int
0x0000119c
* _gl_str
0x000011a0
* e D
__minix_ D 00001000
endrom   D 00001198
__penvir D 00001198
begdata  D 00001198
_gl_int  D 0000119C
_gl_str  D 000011A0
...
* e T
begtext  T 00000000
crtso    T 00000000
__minix_ T 00001000
__minix_ T 00001011
_factori T 00002000
_main    T 00002032
...
* / ?
<address> /nsf  - Display for n items of size s with format f from address
          n defaults to 1
          s defaults to size of int
             can be b for byte h for short l for long
          f defaults to d for decimal
             can be x X o d D c s or u as in printf
             y treat value as address
             i disasm
* D:0x0000119C /
0x0000119c:      321
* D:0x000011A0 /
0x000011a0:     4100
* D:0x000011A0 /1bs
0x00001004: test string
```

Это и есть в точности те переменные, которые мы инициировали.

Вот то же самое, но здесь вместо адресов переменных мы используем их имена, как об этом говорилось ранее:

```
* D:_gl_int /
```

```
0x0000119c:      321
* _gl_int /
0x0000119c:      321
* D:_gl_str /1bs
0x00001004: test string
* _gl_str /1bs
0x00001004: test string
```

Команда dumpcore

Утилита `dumpcore`, как говорит на неё справочная MAN страница, предназначен для генерации файл дампа памяти для **исполняющегося** процесса, основываясь на PID процесса. Результирующий файл дампа будет сохранён под именем `core` в текущем рабочем каталоге. Какие либо предыдущие файлы с этим именем будут уничтожены.

Получим дамп для какого-то из выполняющихся в системе процессов:

```
# ps -ax | grep in.ftpd
  230   ?  0:00 in.ftpd
# time dumpcore 230
       4.38 real    0.21 user    4.11 sys
# ls -l core
-rw------- 1 root  operator  140228 May 30 14:50 core
```

Примечание: показано выполнение операции на медленном процессоре 300Mhz, время операции достаточно значительно. Ещё одной особенностью утилиты есть то, то ей нужно достаточно много (100Mb или более) свободной памяти, в противном случае вы получаете сообщение подобное следующему:

```
# dumpcore 125
ptrace(T_GETRANGE): Not enough core
```

Результирующий файл дампа может быть использован, например, для анализа с отладчиком `mdb` (мы предварительно переименовываем файл, чтобы убедиться, что `mdb` не требует для этой цели фиксированного имени):

```
# mv ./core cr1
# mdb -c cr1
From core file:
T              0     f000      3c4
D              0    10000     f3c4
S       7fffd000  80000000    1f3c4

WARNING: don't know pid from core; using proc nr for pid.
Proc =    225
           Virtual     Physical    Length
           address     address
T:     0x00000000   0x19000000      61440 (0x0000f000)
D:     0x00000000   0x1900f000      65536 (0x00010000)
S:     0x7ffff000   0x9900e000 1727991808 (0x66ff1000)

PC  = 0x0000d089      _start+D089
0000:D089  5B              pop     ebx
```

*

Системный профайлер

В системе MINIX 3 изначально имеется системный профайлер `profile`. Информацию нём я привожу крайне сжато, поскольку: а). это мощный инструмент анализа производительности в системе, который требует для использования весьма объёмного изучения, и б). этот инструмен требуется не очень широкому кругу разработчиков. Информация по утилите доступна, более того, она указывает при каких конфигурационных параметрах, возможно, следует перекомпилировать ядро для использования этого средства:

```
# man profile
NAME
     profile - MINIX system profiling control command
SYNOPSIS
     Statistical profiling:
     profile start [ -m memsize ] [ -o outfile ] [ -f frequency ]
     profile stop
     sprofalyze.pl

     Call profiling:
     profile get [ -m memsize ] [ -o outfile ]
     profile reset
     cprofalyze.pl

DESCRIPTION
     This command controls  MINIX  system  profiling.  There  is  support  for
     statistical  profiling, which causes a CMOS interrupt to regularly sample
     the position of the program  counter,  and  call  profiling,  which  uses
     compiler hooks to measure the time and frequency of call paths.

     For statistical profiling support, recompile the kernel with SPROFILE  in
     <minix/config.h>  set  to  1.  For  call profiling support, recompile the
     system with new libraries with CPROFILE <minix/config.h>  set  to  1  and
     environment variable CPROFILE set to -Rcem-p.
...
```

Посмотрим что записано в `/usr/include/minix/config.h` по умолчанию:

```
/* Enable or disable system profiling. */
#define SPROFILE        1    /* statistical profiling */
#define CPROFILE        0    /* call profiling */
```

Что для сбора статистики вполне достаточно.

Запустим профайлер:

```
# profile start -o 1.prof
Starting statistical profiling.
```

А через некоторое время завершим интервал профилирования:

```
# profile stop
Statistical profiling stopped.
```

```
Writing to 1.prof ... header 21 bytes, data 1320 bytes.
```

Будет сформирован файл `1.prof`. Для анализа файла предназначены Perl скрипты, упоминаемые в MAN странице: `sprofalyze.pl` и `cprofalyze.pl`.

Запустим:

```
# ./sprofalyze.pl 1.prof
```

После того, как скрипт запросит у вас перекомпилировать ядро, сервисы и драйверы (если это не делалось ранее, и сделать это предстоит только один раз) скрипт будет формировать отчёт профилирования по файлу. Ниже показан только краткий начальный фрагмент (полный отчёт занимает около 3-х страниц) очень детального отчёта:

```
================================================================================
Data file: 1.prof
================================================================================
  System process ticks:        110 (  1%)
    User process ticks:          4 (  0%)          Details of system process
       Idle time ticks:       9271 ( 99%)          samples, aggregated and
                         ----------  ----           per process, are below.
           Total ticks:       9385 (100%)
--------------------------------------------------------------------------------
Total system process time                                          110 samples
--------------------------------------------------------------------------------
  system __receiv ***********************************                    19.1%
  system _virtual *****************                                      9.1%
  system _lock_se ***************                                       8.2%
  dp8390 __sendre ***********                                           5.5%
  random __sendre *******                                               3.6%
  system _sys_tas *******                                               3.6%
     vfs __sendre *******                                               3.6%
...
```

Системный профайлер довольно бессмысленен для отработки простых пользовательских программ. Но он может оказаться незаменимым инструментом при отработке сложных программных проектов, когда необходимо определиться, какой из компонент этого проекта в какой степени нагружает системные службы, и какие именно.

GNU профайлер gprof

Присутствует в системе (после установки пакета GNU утилит):

```
# which gprof
/usr/gnu/bin/gprof
```

И даже на удивление полно документирован:

```
# gprof --help
Usage: gprof [-[abcDhilLsTvwxyz]] [-[ACeEfFJnNOpPqQZ][name]] [-I dirs]
        [-d[num]] [-k from/to] [-m min-count] [-t table-length]
        [--[no-]annotated-source[=name]] [--[no-]exec-counts[=name]]
...
# LANG=en; man gprof
GPROF(1)                 Minix Programmer's Manual                  GPROF(1)
```

```
NAME
     gprof - display call graph profile data
...
```

Но, похоже, что в сборке текущей версии `gprof` неработоспособен, при сборке приложения с профилированием:

```
# gcc -c ex1.c -p -o ex1.o
# gcc ex1.o -o ex1
ex1.o:ex1.o:(.text+0xc): undefined reference to `_mcount'
ex1.o:ex1.o:(.text+0x48): undefined reference to `_mcount'
ex1.o:ex1.o:(.text+0x8f): undefined reference to `_mcount'
collect2: ld returned 1 exit status
```

Получаем не разрешённую ссылку на имя mcount, которое обычно (в Linux) предоставляется библиотекой glib2, но библиотеки GNU сборки для MINIX его не предоставляют. Дополнительно убеждаемся в этом:

```
# objdump -t ex1.o
ex1.o:     file format a.out-i386-minix
SYMBOL TABLE:
00000000 g       .text 0000 00 05 _a
00000000         *UND* 0000 00 01 _mcount
0000003c g       .text 0000 00 05 _b
00000078 g       .text 0000 00 05 _main
00000000         *UND* 0000 00 01 ___main
00000000         *UND* 0000 00 01 _printf
00000000         *UND* 0000 00 01 _exit
00000000         *UND* 0000 00 01 _atoi
```

Сторонние отладочные инструменты

GNU отладчик gdb

На русскоязычном форуме MINIX 3: http://minix3.ru/cgi-bin/yabb2/YaBB.pl?num=1274934651/30#30 участник Ali сообщает о сборке GNU отладчика `gdb` :

Дистрибутив берётся здесь: http://www.cs.vu.nl/~jwlami/minix/gdb-6.3-minix.tar.bz2 , размер архива больше 13Mb, после разархивирования дерево кодов имеет объём свыше 360Mb, несколько больший объём свободного пространства нужно иметь в разделе MINIX для сборки.

Последовательность этой сборки:

```
# cd /usr/local/src
# wget http://www.cs.vu.nl/~jwlami/minix/gdb-6.3-minix.tar.bz2
# bunzip2 gdb-6.3-minix.tar.bz2
# tar -xvf gdb-6.3-minix.tar
# cd gdb-6.3
# ln -s /usr/local/lib/gcc/libncurses.a /usr/local/lib/libncurses.a
# make
```

Принципиальными для этой сборки есть те позиции, которые выделены жирным шрифтом, и то, что не делается `./configure`.

Библиотека dbug

Этот проект первоначально написан Фредом Фишем (Fred Fish), потом, судя по комментариям, в него вносили дополнения ещё несколько человек. Библиотека `dbug` является, как её именуют в публикациях, внутренним отладчиком — прикомпоновываясь к отлаживаемой программе она является фактически очень развитой версией `printf()`. Библиотека часто упоминается в публикациях, судя по всему, к версии `dbug-2.0.0.tar.gz` проект завершён; именно в этой версии он может быть взят с URL, приведенного в конце текста.

Если вы разархивируете этот архив, и выполните сборку (в Linux) как написано в файле `INSTALL` в составе архива... :

```
$ autoheader
$ aclocal
$ autoconf
$ automake
$ ./configure
$ make
$ ./example
```

- то из этого ничего хорошего не получится, сборка завершится ошибками (версии утилит устарели, или конфигурации?). Но, к счастью, это ничему не препятствует: в каталоге `./src` архива есть 2 файла на 50Kb :

```
$ ls -l dbug.*
-rw-rw-r-- 1 olej olej 50232 Май  1  2002 dbug.c
-rw-rw-r-- 1 olej olej  5351 Май  1  2002 dbug.h
```

- которые необходимы и достаточны для сборки, и больше ничего не надо, даже `Makefile`; показана компиляция в Linux:

```
$ gcc -c dbug.c -o dbug.o
$ ar -r libdbug.a dbug.o
ar: creating libdbug.a
$ ar -t libdbug.a
dbug.o
$ ls -l *dbug*
-rw-rw-r-- 1 olej olej 50232 Май  1  2002 dbug.c
-rw-rw-r-- 1 olej olej  5351 Май  1  2002 dbug.h
-rw-rw-r-- 1 olej olej  6372 Май  1  2002 dbug_long.h
-rw-rw-r-- 1 olej olej 10692 Май 29 22:23 dbug.o
-rw-rw-r-- 1 olej olej 11074 Май 29 22:28 libdbug.a
```

Переписываем программу факториала, который мы отлаживали ранее:

fact_dbug.c :

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "dbug.h"
```

```
unsigned long factorial( int value ) {
   DBUG_ENTER( "factorial" );
   DBUG_PRINT( "info", ( "Got argument: '%d'", value ) );
   if( 1 == value || 0 == value ) DBUG_RETURN( 1 );
   unsigned long res = value * factorial( value - 1 );
   DBUG_RETURN( res );
}

int main( int argc, char* argv[] ) {
   register unsigned long result;
   DBUG_ENTER( "main" );
   DBUG_PROCESS( argv[ 0 ] );
   int i;
   for( i = 1; i < argc && argv[ i ][ 0 ] == '-'; i++ ) {
       switch( argv[ i ][ 1 ] ) {
         case '#':
             DBUG_PUSH( &(argv[ i ][ 2 ]) );
       }
   }
   if( 1 != argc - i ) {
       printf( "usage: %s [-#{d|t|O}] <NUMBER>\n", argv[ 0 ] );
       DBUG_RETURN( 1 )
   }
   result = factorial( atoi( argv[ i ] ) );
   printf( "%ld\n", result );
   DBUG_RETURN( 0 )
}
```

Изменения очень небольшие и интуитивно понятные. Дальше компонуем нашу программу с `dbug.o`, или так:

```
gcc -g -Wall -o fact_dbug  fact_dbug.c dbug.o
...
```

или так:

```
gcc -g -Wall -o fact_dbug  fact_dbug.c libdbug.a
...
```

Результат выполнения:

```
$ ./fact_dbug -#t 5
| >factorial
| | >factorial
| | | >factorial
| | | | >factorial
| | | | | >factorial
| | | | | <factorial
| | | | <factorial
| | | <factorial
| | <factorial
```

```
| <factorial
120
<main
$ ./fact_dbug -#d 5
factorial: info: Got argument: '5'
factorial: info: Got argument: '4'
factorial: info: Got argument: '3'
factorial: info: Got argument: '2'
factorial: info: Got argument: '1'
120
```

Причём, отладочные макросы можно и не убирать из кода готового приложения, достаточно запускать приложение без ключей `-#...`:

```
$ ./fact_dbug 5
120
```

Теперь, убедившись что и как здесь работаем, проделываем то же в MINIX 3, причём для 2-х вариантов компиляторов: `gcc` и `cc`.

Вариант GNU `gcc` :

```
# uname -a
Minix minix 3 1.6 i686
# LAN=ua; gar -r libdbug.a dbug.o
gar: creating libdbug.a
# make
gcc -g -Wall -o example  example.c dbug.o
gcc -g -Wall -o example1  example.c  libdbug.a
gcc -g -Wall -o fact_dbug  fact_dbug.c dbug.o
# ./fact_dbug -#t 5
| >factorial
| | >factorial
| | | >factorial
| | | | >factorial
| | | | | >factorial
| | | | | <factorial
| | | | <factorial
| | | <factorial
| | <factorial
| <factorial
120
<main
# ./fact_dbug -#d 5
factorial: info: Got argument: '5'
factorial: info: Got argument: '4'
factorial: info: Got argument: '3'
```

```
factorial: info: Got argument: '2'
factorial: info: Got argument: '1'
120
```

Единственно, что в MINIX есть особенности с определениями типов в хедер файлах, поэтому в `dbug.h` приходится вписать (добавленное выделено жирным шрифтом):

```
#include <sys/types.h>
#define uint unsigned int
#define ulong unsigned long
```

В конечном итоге, для этого варианта подготовлен файл сборки:

```
Makefile_gcc :
CC=gcc -g -Wall
AR=gar
LIST=dbug fact example example1 fact_dbug
all:    $(LIST)
dbug:   dbug.h dbug.c
        $(CC) -c -o dbug.o dbug.c
        LAN=ua; $(AR) -r libdbug.a dbug.o
fact:   fact.c
        $(CC) -o $@  $<
example:        example.c
        $(CC) -o $@  $< dbug.o
example1:       example.c
        $(CC) -o $@  $<  libdbug.a
fact_dbug:      fact_dbug.c
        $(CC) -o $@  $< dbug.o
clean:
        rm -f *.o *.a $(LIST)
```

Вариант ACK `cc` :

Здесь пришлось поменять ещё одну строку в dbug.c (в районе 722 стр., это изменение не повлияло на сборку `gcc` и в Linux):

```
/*    uint * _slevel_, char ***_sframep_ __attribute__ ((unused)))  */
      uint * _slevel_, char ***_sframep_ )
```

Файл сборки:

Makefile_ack :

```
CC=cc -g -Wall
AR=ar
LIST=dbug fact example example1 fact_dbug
all:    $(LIST)
dbug:   dbug.h dbug.c
        $(CC) -c -o dbug.o dbug.c
        $(AR) cr libdbug.a dbug.o
```

```
fact:   fact.c
        $(CC) -o $@  $<
example:        example.c
        $(CC) -o $@  $< dbug.o
example1:       example.c
        $(CC) -o $@  $<  libdbug.a
fact_dbug:      fact_dbug.c
        $(CC) -o $@  $< dbug.o
clean:
        rm -f *.o *.a $(LIST)
```

Полная пересборка приложений (и библиотеки):

```
# make -f Makefile_ack  clean
rm -f *.o *.a dbug fact example example1 fact_dbug
# make -f Makefile_ack
cc -g -Wall -c -o dbug.o dbug.c
ar cr libdbug.a dbug.o
cc -g -Wall -o fact  fact.c
cc -g -Wall -o example  example.c dbug.o
cc -g -Wall -o example1  example.c  libdbug.a
cc -g -Wall -o fact_dbug  fact_dbug.c dbug.o
```

Выполнение (`example` — это пример, лежащий в исходном архиве `dbug-2.0.0.tar.gz`, и ранее не показывавшийся):

```
# ./example
>sub1
| >sub2
| | info: Got argument: 'Hello world!'
Hello world!
| <sub2
| >sub2
| | info: Got argument: 'Hello earth!'
Hello earth!
| <sub2
| >sub2
| | info: Got argument: 'Hello programmer!'
Hello programmer!
| <sub2
<sub1
info: Returned value: 0
# ./fact_dbug -#t 5
| >factorial
| | >factorial
| | | >factorial
| | | | >factorial
```

```
| | | | | >factorial
| | | | | <factorial
| | | | <factorial
| | | <factorial
| | <factorial
| <factorial
120
<main
```

Дополнительные источники информации

MAN страницы MINIX

1.1. `man` страница по `mdb` (перевод, дополнения) :
http://wiki.minix3.ru/index.php/Команда_mdb_-_отладчик_MINIX

1.2. `man` страница по команде `dumpcore` (перевод, дополнения) :

http://wiki.minix3.ru/index.php/Команда_dumpcore_-_генерирование_дампа_исполняемого_процесса.

Домашние страницы инструментов отладки

Здесь приводятся URL, с которых можно свободно загрузить обсуждаемые инструменты (все они свободные), а также ссылки на краткую информацию по ним.

2.1. Библиотека `dbug` (подробно описаны ранее в тексте):

http://sourceforge.net/projects/dbug/files/

2.2. Пакет Electric Fence, является вставляемым заместителем для `malloc()` :

http://www.sisyphus.ru/ru/srpm/Sisyphus/ElectricFence

http://www.sisyphus.ru/ru/srpm/Sisyphus/ElectricFence/sources/0

http://perens.com/FreeSoftware/ElectricFence

http://ru.wikipedia.org/wiki/Electric_Fence

2.3. Библиотека `dmalloc`. Предоставляет большое число опций отладки `malloc()`. Автор Грей Ватсон (Gray Watson) :

http://ru.wikipedia.org/wiki/Dmalloc

http://sourceforge.net/projects/dmalloc/

2.4. Пакет Valgrind, для диагностики не только выделения памяти, но и множества сопутствующих проблем, лицензия GPL:

http://ru.wikipedia.org/wiki/Valgrind

http://valgrind.org/

http://alexott.net/ru/linux/valgrind/Valgrind.html - «Что такое valgrind и зачем он нужен».

2.5. Замещающая `malloc()` библиотека `ccmalloc`, не нуждается в особой компиляции, может использоваться совместно с C++. Такое впечатление, что проект угас к 2009г., и его домашняя страница ликвидирована, но вот здесь можно найти реализацию:

http://fdp.pp.ru/Ports/devel/ccmalloc/

2.6. Реализация `malloc()`Марка Мораеса (Mark Moraes) — старая но полнофункциональная реализация, предоставляющая возможности профилирования, трассировки и отладки:

http://www.moraes.net/

ftp://ftp.cs.toronto.edu/pub/moraes/malloc.tar.gz

2.7. Пакет `mpatrol` с большими возможностями настройки для отладки памяти и тестирования:

http://sourceforge.net/projects/mpatrol/files/mpatrol/1.4.8/mpatrol_1.4.8.tar.gz/download

2.8. Пакет `memwatch`, требует использования специального заголовочного файла и опций времени компилирования. Последние обновления проекта — 01.2010г. :

http://www.linkdata.se/sourcecode/memwatch

http://www.linkdata.se/downloads/sourcecode/memwatch/

2.9. Пакет `njamd` (Not Just Another Malloc Debuger). Проект, похоже, завершён 2001г., но его хвалят:

http://sourceforge.net/projects/njamd/files/

http://sourceforge.net/projects/njamd/files/NJAMD%20devel%20source/0.9.3pre2/njamd-0.9.3pre2.tar.gz/download

Статьи, посвящённые технике отладки

3.1. Крис Касперски «Помощники компилятора»

http://www.insidepro.com/kk/316r.shtml

3.2. Steve Best «Mastering Linux debugging techniques»

http://www.ibm.com/developerworks/linux/library/l-debug/

(ниже показана отдельная книга этого же автора).

3.3. «Поиск утечек памяти, и прочих ошибок в программах»

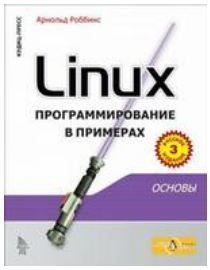
http://alexott.net/ru/linux/valgrind/

3.4. Мартин Ханифорд, «Ускорение кода при помощи GNU-профайлера»

http://www.interface.ru/home.asp?artId=3433

Книги,затрагивающие отладку

4.1. А. Роббинс, «Linux: программирование в примерах», 3-е издание, М.Кудиц-Пресс, 2008, ISBN 978-5-91136-056-6, 656 стр.



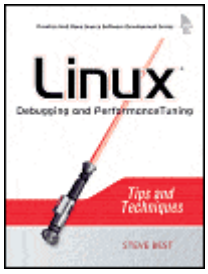
В этой книге целый раздел посвящён обзору инструментов отладки и обсуждению различных методик отладки.

Эту книгу можно свободно скачать в электронном виде:

http://lib.ololo.cc/gen/get?md5=74C00EA8295D1448654F677DFB1EF68E

http://lib.ololo.cc/booksearch?ask=%D0%A0%D0%BE%D0%B1%D0%B1%D0%B8%D0%BD%D1%81

4.2. Best S. «Linux Debugging and Performance Tuning: Tips and Techniques», 2005, 456 стр.



Эту книгу также можно свободно скачать в электронном виде:

http://lib.ololo.cc/booksearch?ask=Best+S

http://lib.ololo.cc/gen/get?md5=6C24D40B6000CE13D9D13E601F0F5769