

第 21 章

Linux 设备驱动的调试

本章导读

“工欲善其事，必先利其器”，为了方便进行 Linux 设备驱动的开发和调试，建立良好的开发环境很重要，还要使用必要的工具软件以及掌握常用的调试技巧等。

21.1 节讲解了 Linux 下调试器 GDB 的基本用法和技巧。

21.2 节讲解了 Linux 内核的调试方法。

21.3 ~ 21.10 节对 21.3 节的概述展开了讲解，内容有：Linux 内核调试用的 `printk()`、`BUG_ON()`、`WARN_ON()`、`/proc`、`Oops`、`strace`、`KGDB`，以及使用仿真器进行调试的方法。

21.11 节讲解了 Linux 应用程序的调试方法，驱动工程师往往需要编写用户空间的应用程序以对自身编写的驱动进行验证和测试，因此，掌握应用程序调试方法对驱动工程师而言也是必需的。

21.12 节讲解了 Linux 常用的一些稳定性、性能分析和调优工具。

21.1 GDB 调试器的用法

21.1.1 GDB 的基本用法

GDB 是 GNU 开源组织发布的一个强大的 UNIX 下的程序调试工具，GDB 主要可帮助工程师完成下面 4 个方面的功能。

- 启动程序，可以按照工程师自定义的要求运行程序。
- 让被调试的程序在工程师指定的断点处停住，断点可以是条件表达式。
- 当程序被停住时，可以检查此时程序中所发生的事，并追踪上文。
- 动态地改变程序的执行环境。

不管是调试 Linux 内核空间的驱动还是调试用户空间的应用程序，都必须掌握 GDB 的用法。而且，在调试内核和调试应用程序时使用的 GDB 命令是完全相同的，下面以代码清单 21.1 的应用程序为例演示 GDB 调试器的用法。

代码清单 21.1 GDB 调试器用法的演示程序

```

1 int add(int a, int b)
2 {
3     return a + b;
4 }
5
6 main()
7 {
8     int sum[10] =
9     {
10        0, 0, 0, 0, 0, 0, 0, 0, 0, 0
11    };
12     int i;
13
14     int array1[10] =
15     {
16        48, 56, 77, 33, 33, 11, 226, 544, 78, 90
17    };
18     int array2[10] =
19     {
20        85, 99, 66, 0x199, 393, 11, 1, 2, 3, 4
21    };
22
23     for (i = 0; i < 10; i++)
24     {
25         sum[i] = add(array1[i], array2[i]);
26     }
27 }
```

使用命令 `gcc-g gdb_example.c-o gdb_example` 编译上述程序，得到包含调试信息的二进制文件 `example`，执行 `gdb gdb_example` 命令进入调试状态，如下所示：

```

$ gdb gdb_example
GNU gdb (Ubuntu 7.7-0ubuntu3.1) 7.7
Copyright (C) 2014 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law. Type "show copying"
and "show warranty" for details.
This GDB was configured as "i686-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
<http://www.gnu.org/software/gdb/documentation/>.
For help, type "help".
Type "apropos word" to search for commands related to "word".
(gdb)
```

1. list 命令

在 GDB 中运行 list 命令（缩写 l）可以列出代码，list 的具体形式如下。

- list <linenum>，显示程序第 linenum 行周围的源程序，如下所示：

```
(gdb) list 15
10
11     int array1[10] =
12     {
13         48, 56, 77, 33, 33, 11, 226, 544, 78, 90
14     };
15     int array2[10] =
16     {
17         85, 99, 66, 0x199, 393, 11, 1, 2, 3, 4
18     };
19
```

- list <function>，显示函数名为 function 的函数的源程序，如下所示：

```
(gdb) list main
2
3     {
4         return a + b;
5     }
6     main()
7     {
8         int sum[10];
9         int i;
10
11     int array1[10] =
```

- list，显示当前行后面的源程序。
- list -，显示当前行前面的源程序。

下面演示了使用 GDB 中的 run（缩写为 r）、break（缩写为 b）、next（缩写为 n）命令控制程序的运行，并使用 print（缩写为 p）命令打印程序中的变量 sum 的过程：

```
(gdb) break add
Breakpoint 1 at 0x80482f7: file gdb_example.c, line 3.
(gdb) run
Starting program: /driver_study/gdb_example

Breakpoint 1, add (a=48, b=85) at gdb_example.c:3
warning: Source file is more recent than executable.

3     return a + b;
(gdb) next
4
(gdb) next
main () at gdb_example.c:23
```

```

23      for (i = 0; i < 10; i++)
(gdb) next
25      sum[i] = add(array1[i], array2[i]);
(gdb) print sum
$1 = {133, 0, 0, 0, 0, 0, 0, 0, 0, 0}

```

2. run 命令

在 GDB 中，运行程序使用 run 命令。在程序运行前，我们可以设置如下 4 方面的工作环境。

(1) 程序运行参数

用 set args 可指定运行时参数，如 set args 10 20 30 40 50；用 show args 命令可以查看设置好的运行参数。

(2) 运行环境

用 path<dir> 可设定程序的运行路径；用 how paths 可查看程序的运行路径；用 set environment varname[=value] 可设置环境变量，如 set env USER=baohua；用 show environment[varname] 则可查看环境变量。

(3) 工作目录

cd<dir> 相当于 shell 的 cd 命令，pwd 可显示当前所在的目录。

(4) 程序的输入输出

info terminal 用于显示程序用到的终端的模式；在 GDB 中也可以使用重定向控制程序输出，如 run>outfile；用 tty 命令可以指定输入输出的终端设备，如 tty/dev/ttyS1。

3. break 命令

在 GDB 中用 break 命令来设置断点，设置断点的方法如下。

(1) break<function>

在进入指定函数时停住，在 C++ 中可以使用 class::function 或 function(type, type) 格式来指定函数名。

(2) break<linenum>

在指定行号停住。

(3) break+offset/break-offset

在当前行号的前面或后面的 offset 行停住，offset 为自然数。

(4) break filename:linenum

在源文件 filename 的 linenum 行处停住。

(5) break filename:function

在源文件 filename 的 function 函数的入口处停住。

(6) break*address

在程序运行的内存地址处停住。

(7) break

break 命令没有参数时，表示在下一条指令处停住。

(8) break…if<condition>

…可以是上述的 break<linenum>、break+offset/break-offset 中的参数, condition 表示条件, 在条件成立时停住。比如在循环体中, 可以设置 break if i=100, 表示当 i 为 100 时停住程序。

查看断点时, 可使用 info 命令, 如 info breakpoints[n]、info break[n] (n 表示断点号)。

4. 单步命令

在调试过程中, next 命令用于单步执行, 类似于 VC++ 中的 step over。next 的单步不会进入函数的内部, 与 next 对应的 step (缩写为 s) 命令则在单步执行一个函数时, 进入其内部, 类似于 VC++ 中的 step into。下面演示了 step 命令的执行情况, 在第 23 行的 add() 函数调用处执行 step 会进入其内部的 return a+b; 语句:

```
(gdb) break 25
Breakpoint 1 at 0x8048362: file gdb_example.c, line 25.
(gdb) run
Starting program: /driver_study/gdb_example

Breakpoint 1, main () at gdb_example.c:25
25      sum[i] = add(array1[i], array2[i]);
(gdb) step
add (a=48, b=85) at gdb_example.c:3
3      return a + b;
```

单步执行的更复杂用法如下。

(1) step<count>

单步跟踪, 如果有函数调用, 则进入该函数 (进入函数的前提是, 此函数被编译有 debug 信息)。step 后面不加 count 表示一条条地执行, 加 count 表示执行后面的 count 条指令, 然后再停住。

(2) next<count>

单步跟踪, 如果有函数调用, 它不会进入该函数。同理, next 后面不加 count 表示一条条地执行, 加 count 表示执行后面的 count 条指令, 然后再停住。

(3) set step-mode

set step-mode on 用于打开 step-mode 模式, 这样, 在进行单步跟踪 (运行 step 指令) 时, 若跨越某没有调试信息的函数, 程序的执行则会在该函数的第一条指令处停住, 而不会跳过整个函数。这样我们可以查看该函数的机器指令。

(4) finish

运行程序, 直到当前函数完成返回, 并打印函数返回时的堆栈地址、返回值及参数值等信息。

(5) until (缩写为 u)

一直在循环体内执行单步而退不出来是一件令人烦恼的事情, 用 until 命令可以运行程序直到退出循环体。

(6) stepi (缩写为 si) 和 nexti (缩写为 ni)

stepi 和 nexti 用于单步跟踪一条机器指令。比如，一条 C 程序代码有可能由数条机器指令完成，stepi 和 nexti 可以单步执行机器指令，相反，step 和 next 是 C 语言级别的命令。

另外，运行 display/i \$pc 命令后，单步跟踪会在打出程序代码的同时打出机器指令，即汇编代码。

5. continue 命令

当程序被停住后，可以使用 continue 命令（缩写为 c，fg 命令同 continue 命令）恢复程序的运行直到程序结束，或到达下一个断点，命令格式为：

```
continue [ignore-count]
c [ignore-count]
fg [ignore-count]
```

ignore-count 表示忽略其后多少次断点。

假设我们设置了函数断点 add()，并观察 i，则在 continue 过程中，每次遇到 add() 函数或 i 发生变化，程序就会停住，如下所示：

```
(gdb) continue
Continuing.
Hardware watchpoint 3: i

Old value = 2
New value = 3
0x0804838d in main () at gdb_example.c:23
23      for (i = 0; i < 10; i++)
(gdb) continue
Continuing.

Breakpoint 1, main () at gdb_example.c:25
25      sum[i] = add(array1[i], array2[i]);
(gdb) continue
Continuing.
Hardware watchpoint 3: i

Old value = 3
New value = 4
0x0804838d in main () at gdb_example.c:23
23      for (i = 0; i < 10; i++)
```

6. print 命令

在调试程序时，当程序被停住时，可以使用 print 命令（缩写为 p），或是同义命令 inspect 来查看当前程序的运行数据。print 命令的格式如下：

```
print <expr>
print /<f> <expr>
```

<expr> 是表达式，也是被调试的程序中的表达式，<f> 是输出的格式，比如，如果要

把表达式按十六进制的格式输出，那么就是/x。在表达式中，有几种 GDB 所支持的操作符，它们可以用在任何一种语言中，@ 是一个和数组有关的操作符，:: 指定一个在文件或是函数中的变量，{<type>}<addr> 表示一个指向内存地址 <addr> 的类型为 type 的对象。

下面演示了查看 sum[] 数组的值的过程：

```
(gdb) print sum
$2 = {133, 155, 0, 0, 0, 0, 0, 0, 0, 0}
(gdb) next

Breakpoint 1, main () at gdb_example.c:25
25      sum[i] = add(array1[i], array2[i]);
(gdb) next
23      for (i = 0; i < 10; i++)
(gdb) print sum
$3 = {133, 155, 143, 0, 0, 0, 0, 0, 0, 0}
```

当需要查看一段连续内存空间的值时，可以使用 GDB 的 @ 操作符，@ 的左边是第一个内存地址，@ 的右边则是想查看内存的长度。例如如下动态申请的内存：

```
int *array = (int *) malloc (len * sizeof (int));
```

在 GDB 调试过程中这样显示这个动态数组的值：

```
p *array@len
```

print 的输出格式如下。

- x：按十六进制格式显示变量。
- d：按十进制格式显示变量。
- u：按十六进制格式显示无符号整型。
- o：按八进制格式显示变量。
- t：按二进制格式显示变量。
- a：按十六进制格式显示变量。
- c：按字符格式显示变量。
- f：按浮点数格式显示变量。

我们可用 display 命令设置一些自动显示的变量，当程序停住时，或是单步跟踪时，这些变量会自动显示。

如果要修改变量，如 x 的值，可使用如下命令：

```
print x=4
```

当用 GDB 的 print 查看程序运行时数据时，每一个 print 都会被 GDB 记录下来。GDB 会以 \$1, \$2, \$3…这样的方式为每一个 print 命令编号。我们可以使用这个编号访问以前的表达式，如 \$1。

7. watch 命令

watch一般用来观察某个表达式（变量也是一种表达式）的值是否有了变化，如果有变化，马上停止程序运行。我们有如下几种方法来设置观察点。

watch<expr>：为表达式（变量）expr设置一个观察点。一旦表达式值有变化时，马上停止程序运行。

rwatch<expr>：当表达式（变量）expr被读时，停止程序运行。

awatch<expr>：当表达式（变量）的值被读或被写时，停止程序运行。

info watchpoints：列出当前所设置的所有观察点。

下面演示了观察i并在连续运行next时一旦发现i变化，i值就会显示出来的过程：

```
(gdb) watch i
Hardware watchpoint 3: i
(gdb) next
23      for (i = 0; i < 10; i++)
(gdb) next
Hardware watchpoint 3: i

Old value = 0
New value = 1
0x0804838d in main () at gdb_example.c:23
23      for (i = 0; i < 10; i++)
(gdb) next

Breakpoint 1, main () at gdb_example.c:25
25      sum[i] = add(array1[i], array2[i]);
(gdb) next
23      for (i = 0; i < 10; i++)
(gdb) next
Hardware watchpoint 3: i

Old value = 1
New value = 2
0x0804838d in main () at gdb_example.c:23
23      for (i = 0; i < 10; i++)
```

8. examine 命令

我们可以使用examine命令（缩写为x）来查看内存地址中的值。examine命令的语法如下所示：

```
x/<n/f/u> <addr>
```

<addr>表示一个内存地址。“x/”后的n、f、u都是可选的参数，n是一个正整数，表示显示内存的长度，也就是说从当前地址向后显示几个地址的内容；f表示显示的格式，如果地址所指的是字符串，那么格式可以是s，如果地址是指令地址，那么格式可以是i；u表示从当前地址往后请求的字节数，如果不指定的话，GDB默认的是4字节。u参数可以被一些

字符代替：b 表示单字节，h 表示双字节，w 表示四字节，g 表示八字节。当我们指定了字节长度后，GDB 会从指定的内存地址开始，读写指定字节，并把其当作一个值取出来。n、f、u 这 3 个参数可以一起使用，例如命令 x/3uh 0x54320 表示从内存地址 0x54320 开始以双字节为 1 个单位 (h)、16 进制方式 (u) 显示 3 个单位 (3) 的内存。

9. set 命令

examine 命令用于查看内存，而 set 命令用于修改内存。它的命令格式是“set* 有类型的指针 =value”。

比如，下列程序，在用 gdb 运行起来后，通过 Ctrl+C 停住。

```
main()
{
    void *p = malloc(16);
    while(1);
}
```

我们可以在运行中用如下命令来修改 p 指向的内存。

```
(gdb) set *(unsigned char *)p='h'
(gdb) set *(unsigned char *) (p+1)='e'
(gdb) set *(unsigned char *) (p+2)='l'
(gdb) set *(unsigned char *) (p+3)='l'
(gdb) set *(unsigned char *) (p+4)='o'
```

看看结果：

```
(gdb) x/s p
0x804b008:      "hello"
```

也可以直接使用地址常数：

```
(gdb) p p
$2 = (void *) 0x804b008
(gdb) set *(unsigned char *) 0x804b008='w'
(gdb) set *(unsigned char *) 0x804b009='o'
(gdb) set *(unsigned char *) 0x804b00a='r'
(gdb) set *(unsigned char *) 0x804b00b='l'
(gdb) set *(unsigned char *) 0x804b00c='d'
(gdb) x/s 0x804b008
0x804b008:      "world"
```

10. jump 命令

一般来说，被调试程序会按照程序代码的运行顺序依次执行，但是 GDB 也提供了乱序执行的功能，也就是说，GDB 可以修改程序的执行顺序，从而让程序随意跳跃。这个功能可以由 GDB 的 jump 命令 jump<linespec> 来指定下一条语句的运行点。<linespec> 可以是文件的行号，可以是 file:line 格式，也可以是 +num 这种偏移量格式，表示下一条运行语句从哪里开始。

```
jump <address>
```

这里的<address>是代码行的内存地址。

注意：jump命令不会改变当前程序栈中的内容，如果使用jump从一个函数跳转到另一个函数，当跳转到的函数运行完返回，进行出栈操作时必然会发生错误，这可能会导致意想不到的结果，因此最好只用jump在同一个函数中进行跳转。

11. signal 命令

使用singal命令，可以产生一个信号量给被调试的程序，如中断信号Ctrl+C。于是，可以在程序运行的任意位置处设置断点，并在该断点处用GDB产生一个信号量，这种精确地在某处产生信号的方法非常有利于程序的调试。

signal命令的语法是signal<signal>，UNIX的系统信号量通常为1~15，因此<signal>的取值也在这个范围内。

12. return 命令

如果在函数中设置了调试断点，在断点后还有语句没有执行完，这时候我们可以使用return命令强制函数忽略还没有执行的语句并返回。

```
return
return <expression>
```

上述return命令用于取消当前函数的执行，并立即返回，如果指定了<expression>，那么该表达式的值会被作为函数的返回值。

13. call 命令

call命令用于强制调用某函数：

```
call <expr>
```

表达式可以是函数，以此达到强制调用函数的目的，它会显示函数的返回值（如果函数返回值不是void）。比如在下列程序执行while(1)的时候：

```
main()
{
    void *p = malloc(16);
    while(1);
}
```

我们强制要求其执行strcpy()和printf()：

```
(gdb) call strcpy(p, "hello world")
$3 = 134524936
(gdb) call printf("%s\n", p)
hello world
$4 = 12
```

14. info 命令

info命令可以用来在调试时查看寄存器、断点、观察点和信号等信息。要查看寄存器的

值，可以使用如下命令：

```
info registers (查看除了浮点寄存器以外的寄存器)
info all-registers (查看所有寄存器，包括浮点寄存器)
info registers <regname ...> (查看所指定的寄存器)
```

要查看断点信息，可以使用如下命令：

```
info break
```

要列出当前所设置的所有观察点，可使用如下命令：

```
info watchpoints
```

要查看有哪些信号正在被 GDB 检测，可使用如下命令：

```
info signals
info handle
```

也可以使用 info line 命令来查看源代码在内存中的地址。info line 后面可以跟行号、函数名、文件名：行号、文件名：函数名等多种形式，例如用下面的命令会打印出所指定的源码在运行时的内存地址：

```
info line tst.c:func
```

15. disassemble

disassemble 命令用于反汇编，可用它来查看当前执行时的源代码的机器码，实际上只是把目前内存中的指令冲刷出来。下面的示例用于查看函数 func 的汇编代码：

```
(gdb) disassemble func
Dump of assembler code for function func:
0x8048450 <func>:    push    %ebp
0x8048451 <func+1>:   mov     %esp,%ebp
0x8048453 <func+3>:   sub    $0x10,%esp
0x8048456 <func+6>:   movl   $0x0,0xfffffff(%ebp)
...
End of assembler dump.
```

21.1.2 DDD 图形界面调试工具

GDB 本身是一种命令行调试工具，但是通过 DDD (Data Display Debugger，见 <http://www.gnu.org/software/ddd/>) 可以被图形界面化。DDD 可以作为 GDB、DBX、WDB、Ladebug、JDB、XDB、Perl Debugger 或 Python Debugger 的可视化图形前端，其特有的图形数据显示功能 (Graphical Data Display) 可以把数据结构按照图形的方式显示出来。

DDD 最初源于 1990 年 Andreas Zeller 编写的 VSL 结构化语言，后来经过一些程序员的努力，演化成今天的模样。DDD 的功能非常强大，可以调试用 C/C++、Ada、Fortran、Pascal、Modula-2 和 Modula-3 编写的程序；能以超文本方式浏览源代码；能够进行断点设置、

回溯调试和历史记录；具有程序在终端运行的仿真窗口，具备在远程主机上进行调试的能力；能够显示各种数据结构之间的关系，并将数据结构以图形形式显示；具有 GDB/DBX/XDB 的命令行界面，包括完整的文本编辑、历史纪录、搜寻引擎等。

DDD 的主界面如图 21.1 所示，它和 Visual Studio 等集成开发环境非常相近，而且 DDD 包含了 Visual Studio 所不包含的部分功能。

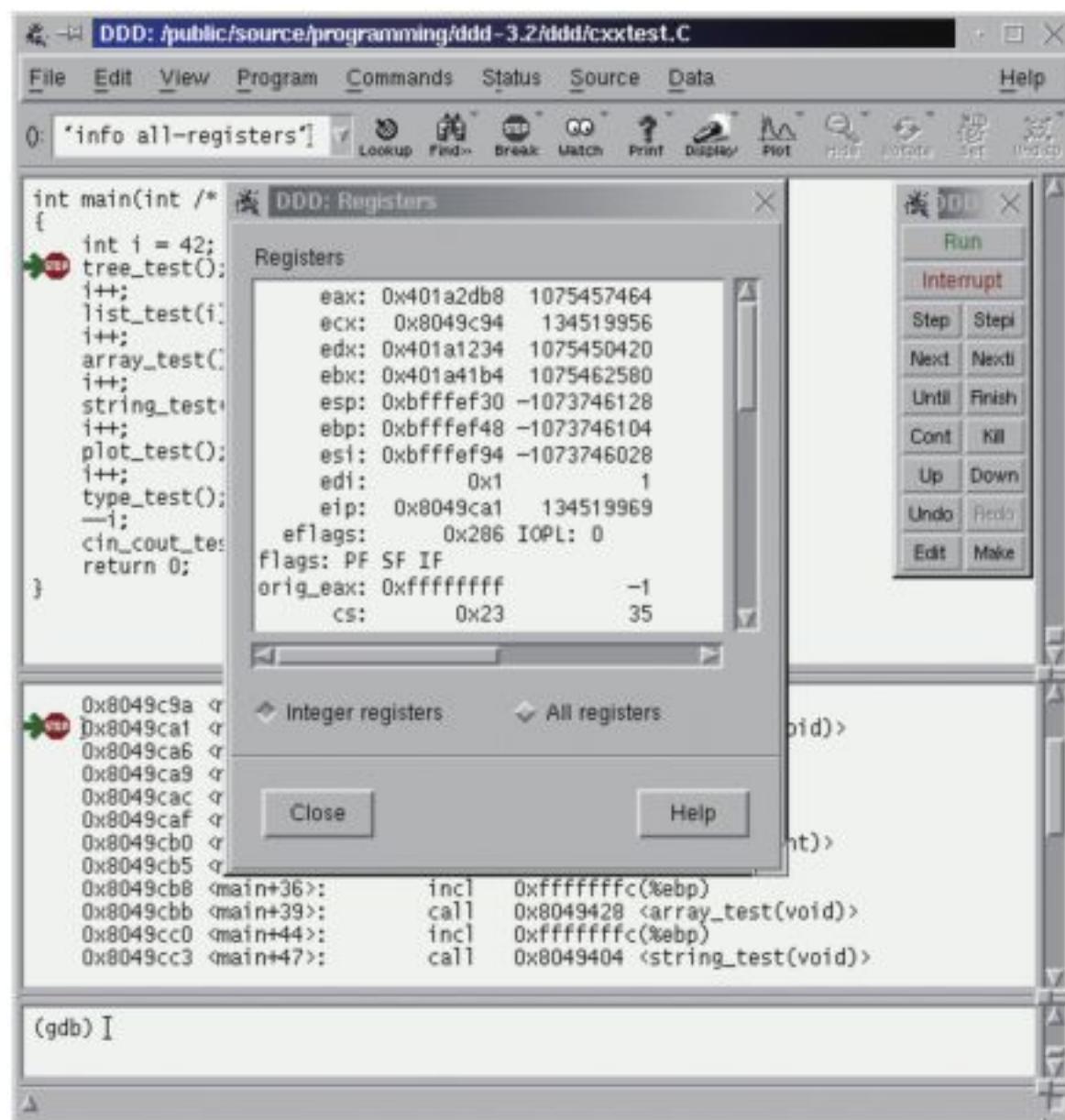


图 21.1 DDD 的主界面

在设计 DDD 的时候，设计人员决定把它与 GDB 之间的耦合度尽量降低。因为像 GDB 这样的开源软件，更新的速度比商业软件快，所以为了使 GDB 的变化不会影响到 DDD，在 DDD 中，GDB 是作为独立的进程运行的，通过命令行接口与 DDD 进行交互。

图 21.2 显示了用户、DDD、GDB 和被调试进程之间的关系，DDD 和 GDB 之间的所有通信都是异步进行的。在 DDD 中发出的 GDB 命令都会与一个回调函数相连，放入命令队列中。这个回调函数在合适的时间会处理 GDB 的输出。例如，如果用户手动输入一条 GDB 的

命令，DDD 就会把这条命令与显示 GDB 输出的一个回调函数连起来。一旦 GDB 命令完成，就会触发回调函数，GDB 的输出就会显示在 DDD 的命令窗口中。

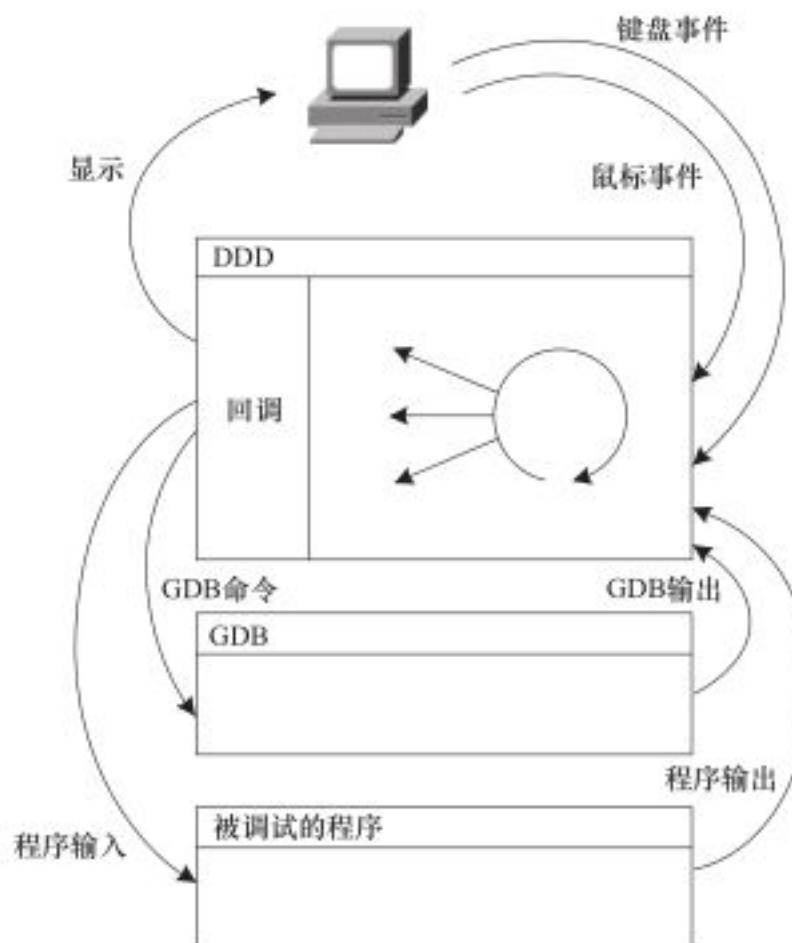


图 21.2 DDD 运行机理

DDD 在事件循环时等待用户输入和 GDB 输出，同时等着 GDB 进入等待输入状态。当 GDB 可用时，下一条命令就会从命令队列中取出，送给 GDB。GDB 到达的输出由上次命令的回调函数过程来处理。这种异步机制避免了 DDD 在等待 GDB 输出时发生阻塞现象，到达的事件可以在任何时间得到处理。

不可否认的是，DDD 和 GDB 的分离使得 DDD 的运行速度相对来说比较慢，但是这种方法带来了灵活性和兼容性的好处。例如，用户可以把 GDB 调试器换成其他调试器，如 DBX 等。另外，GDB 和 DDD 的分离使得用户可以在不同的机器上分别运行 GDB 和 DDD。

在 DDD 中，可以直接在底部的控制台中输入 GDB 命令，也可以通过菜单和鼠标以图形方式触发 GDB 命令的运行，使用方法甚为简单，因此这里不再赘述。

DDD 不仅可用于调试 PC 上的应用程序，也可调试目标板子，方法是用如下命令启动 DDD（通过 `-debugger` 选项指定一个针对 ARM 的 GDB）：

```
ddd --debugger arm-linux-gnueabihf-gdb <要调试的程序>
```

除了 DDD 以外，在 Linux 环境下，也可以使用广受欢迎的 Eclipse 来编写代码并进行

调试。安装 Eclipse IDE for C/C++Developer 后，在 Eclipse 中，可以设置 Using GDB(DSF) Manual Remote Debugging Launcher 以及 ARM 的 GDB 等，如图 21.3 所示。

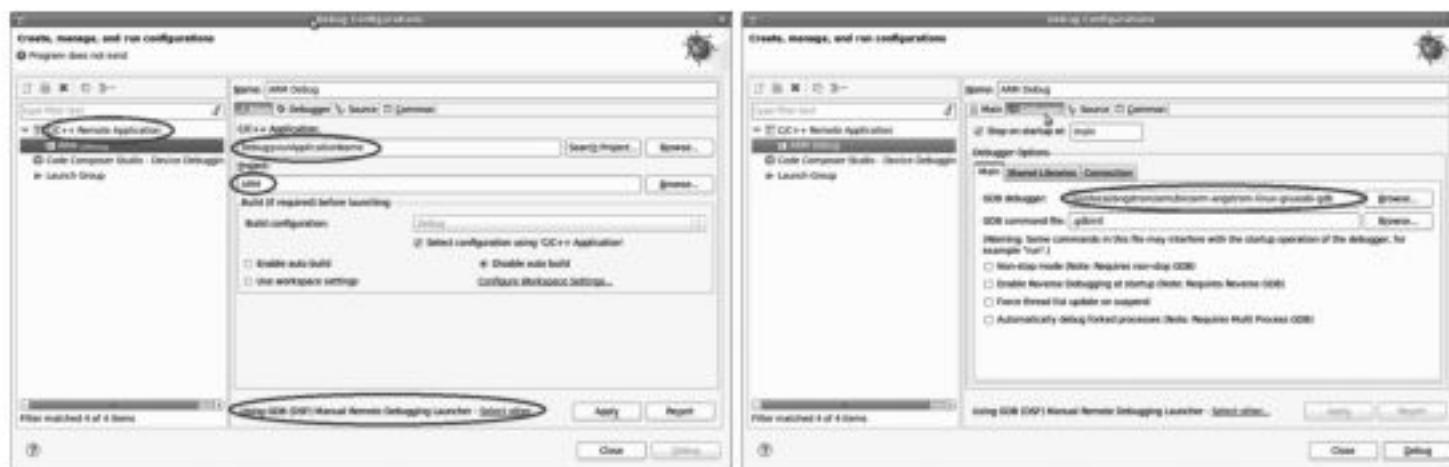


图 21.3 在 Eclipse 中设置 Remote 调试模式和 GDB

21.2 Linux 内核调试

在嵌入式系统中，由于目标机资源有限，因此往往在主机上先编译好程序，再在目标机上运行。用户所有的开发工作都在主机开发环境下完成，包括编码、编译、连接、下载和调试等。目标机和主机通过串口、以太网、仿真器或其他通信手段通信，主机用这些接口控制目标机，调试目标机上的程序。

调试嵌入式 Linux 内核的方法如下。

1) 目标机“插桩”，如打上 KGDB 补丁，这样主机上的 GDB 可与目标机的 KGDB 通过串口或网口通信。

2) 使用仿真器，仿真器可直接连接目标机的 JTAG/BDM，这样主机的 GDB 就可以通过与仿真器的通信来控制目标机。

3) 在目标板上通过 printk()、Oops、strace 等软件方法进行“观察”调试，这些方法不具备查看和修改数据结构、断点、单步等功能。

21.4 ~ 21.7 节将对这些调试方法进行一一讲解。

不管是目标机“插桩”还是使用仿真器连接目标机 JTAG/SWD/BDM，在主机上，调试工具一般都采用 GDB。

GDB 可以直接把 Linux 内核当成一个整体来调试，这个过程实际上可以被 QEMU 模拟出来。进入本书配套 Ubuntu 的 /home/baohua/develop/linux/extr 目录下，修改 run-nolcd.sh 的脚本，将其从

```
qemu-system-arm -nographic -sd vexpress.img -M vexpress-a9 -m 512M -kernel zImage -dtb vexpress-v2p-ca9.dtb -smp 4 -append "init=/linuxrc root=/dev/mmcblk0p1 rw rootwait earlyprintk console=ttyAMA0" 2>/dev/null
```

改为：

```
qemu-system-arm -s -S -nographic -sd vexpress.img -M vexpress-a9 -m 512M -kernel zImage -dtb vexpress-v2p-ca9.dtb -smp 4 -append "init=/linuxrc root=/dev/mmcblk0p1 rw rootwait earlyprintk console=ttyAMA0" 2>/dev/null
```

即添加 -s-S 选项，则会使嵌入式 ARM Linux 系统等待 GDB 远程连入。在终端 1 运行新的 ./run-nolcd.sh，这样嵌入式 ARM Linux 的模拟平台在 1234 端口侦听。开一个新的终端 2，进入 /home/baohua/develop/linux/，执行如下代码：

```
baohua@baohua-VirtualBox:~/develop/linux$ arm-linux-gnueabihf-gdb ./vmlinux
GNU gdb (crosstool-NG linaro-1.13.1-4.8-2013.05 - Linaro GCC 2013.05) 7.6-2013.05
Copyright (C) 2013 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law. Type "show copying"
and "show warranty" for details.
This GDB was configured as "--host=i686-build_pc-linux-gnu --target=arm-linux-gnueabihf".
For bug reporting instructions, please see:
<https://bugs.launchpad.net/gcc-linaro>...
Reading symbols from /home/baohua/develop/linux/vmlinux...done.
(gdb)
```

接下来我们远程连接 127.0.0.1:1234

```
(gdb) target remote 127.0.0.1:1234
Remote debugging using 127.0.0.1:1234
0x60000000 in ?? ()
```

设置一个断点到 start_kernel()。

```
(gdb) b start_kernel
Breakpoint 1 at 0x805fd8ac: file init/main.c, line 490.
```

继续运行：

```
(gdb) c
Continuing.

Breakpoint 1, start_kernel () at init/main.c:490
490  {
(gdb)
```

断点停在了内核启动过程中的 start_kernel() 函数，这个时候我们按下 Ctrl+X, A 键，可以看到代码，如图 21.4 所示。

进一步，可以看看 jiffies 值之类的：

```
(gdb) p jiffies
$1 = 775612
(gdb) c
```

```

Continuing.
^C
Program received signal SIGINT, Interrupt.
cpu_v7_do_idle () at arch/arm/mm/proc-v7.S:74
74      ret lr
(gdb) p jiffies
$2 = 775687
(gdb)

```

```

init/main.c
481         page_ext_init_flatmem();
482         mem_init();
483         knem_cache_init();
484         percpu_init_late();
485         pgtable_init();
486         vmalloc_init();
487     }
488
489     asmlinkage __visible void __init start_kernel(void)
B+> 490     {
491         char *command_line;
492         char *after_dashes;
493
494         /*
495          * Need to run as early as possible, to initialize the
496          * lockdep hash:
497          */
498         lockdep_init();
499         set_task_stack_end_magic(&init_task);
500         smp_setup_processor_id();
501         debug_objects_early_init();

remote Thread 1 In: start kernel
(gdb)

```

图 21.4 GDB 调试内核

尽管采用“插桩”和仿真器结合 GDB 的方式可以查看和修改数据结构、断点、单步等，而 `printf()` 这种最原始的方法却应用得更广泛。

`printf()` 这种方法很原始，但是一般可以解决工程中 95% 以上的问题。因此具体何时打印，以及打印什么东西，需要工程师逐步建立敏锐的嗅觉。加深对内核的认知，深入理解自己正在调试的模块，这才是快速解决问题的“王道”。工具只是一个辅助手段，无法代替工程师的思维。

工程师不能抱着得过且过的心态，也不能总是一知半解地进行低水平的重复建设。求知欲望对工程师技术水平的提升有着最关键的作用。

21.3 内核打印信息——`printf()`

在 Linux 中，内核打印语句 `printf()` 会将内核信息输出到内核信息缓冲区中，内核缓冲区是在 `kernel/printk.c` 中通过如下语句静态定义的：

```
static char __log_buf[__LOG_BUF_LEN] __aligned(LOG_ALIGN);
```

内核信息缓冲区是一个环形缓冲区 (Ring Buffer)，因此，如果塞入的消息过多，则就会将之前的消息冲刷掉。

printk() 定义了 8 个消息级别，分为级别 0 ~ 7，级别越低（数值越大），消息越不重要，第 0 级是紧急事件级，第 7 级是调试级，代码清单 21.2 所示为 printk() 的级别定义。

代码清单 21.2 printk() 的级别定义

```

1 #define KERN_EMERG "<0>"          /* 紧急事件，一般是系统崩溃之前提示的消息 */
2 #define KERN_ALERT "<1>"           /* 必须立即采取行动 */
3 #define KERN_CRIT "<2>"            /* 临界状态，通常涉及严重的硬件或软件操作失败 */
4 #define KERN_ERR  "<3>"             /* 用于报告错误状态，设备驱动程序会
5                                经常使用 KERN_ERR 来报告来自硬件的问题 */
6 #define KERN_WARNING "<4>"           /* 对可能出现问题的情况进行警告，
7                                这类情况通常不会对系统造成严重的问题 */
8 #define KERN_NOTICE  "<5>"            /* 有必要进行提示的正常情形，
9                                许多与安全相关的状况用这个级别进行汇报 */
10 #define KERN_INFO   "<6>"             /* 内核提示性信息，很多驱动程序
11                                在启动的时候，用这个级别打印出它们找到的硬件信息 */
12 #define KERN_DEBUG  "<7>"             /* 用于调试信息 */

```

通过 /proc/sys/kernel/printk 文件可以调节 printk() 的输出等级，该文件有 4 个数字值，如下所示。

- 控制台（一般是串口）日志级别：当前的打印级别，优先级高于该值的消息将被打印至控制台。
- 默认的消息日志级别：将用该优先级来打印没有优先级前缀的消息，也就是在直接写 printk(“xxx”) 而不带打印级别的情况下，会使用该打印级别。
- 最低的控制台日志级别：控制台日志级别可被设置的最小值（一般都是 1）。
- 默认的控制台日志级别：控制台日志级别的默认值。

如在 Ubuntu PC 上，/proc/sys/kernel/printk 的值一般如下：

```
$ cat /proc/sys/kernel/printk
4 4 1 7
```

而我们通过如下命令可以使得 Linux 内核的任何 printk() 都从控制台输出：

```
# echo 8 > /proc/sys/kernel/printk
```

在默认情况下，DEBUG 级别的消息不会从控制台输出，我们可以通过在 bootargs 中设置 ignore_loglevel 来忽略打印级别，以保证所有消息都被打印到控制台。在系统启动后，用户还可以通过写 /sys/module/printk/parameters/ignore_loglevel 文件动态来设置是否忽略打印级别。

要注意的是，/proc/sys/kernel/printk 并不控制内核消息进入 __log_buf 的门槛，因此无论消息级别是多少，都会进入 __log_buf 中，但是最终只有高于当前打印级别的内核消息才会从控制台打印。

用户可以通过 dmesg 命令查看内核打印缓冲区，而如果使用 dmesg-c 命令，则不仅会显示 __log_buf，还会清除该缓冲区的内容。也可以使用 cat/proc/kmsg 命令来显示内核信息。/proc/kmsg 是一个“永无休止的文件”，因此，cat/proc/kmsg 的进程只能通过“Ctrl+C”或 kill 终止。

在设备驱动中，经常需要输出调试或系统信息，尽管可以直接采用 printk(“<7>debug info…\n”) 方式的 printk() 语句输出，但是通常可以使用封装了 printk() 的更高级的宏，如 pr_debug()、dev_debug() 等。代码清单 21.3 所示为 pr_debug() 和 pr_info() 的定义。

代码清单 21.3 可替代 printk() 的宏 pr_debug() 和 pr_info() 的定义

```

1 #ifdef DEBUG
2 #define pr_debug(fmt,arg...) \
3     printk(KERN_DEBUG fmt,##arg)
4 #else
5 static inline int __attribute__((format (printf, 1, 2))) pr_debug(const char * fmt, ...)
6 {
7     return 0;
8 }
9 #endif
10
11 #define pr_info(fmt,arg...) \
12     printk(KERN_INFO fmt,##arg)

```

使用 pr_xxx() 族 API 的好处是，可以在文件最开头通过 pr_fmt() 定义一个打印格式，比如在 kernel/watchdog.c 的最开头通过如下定义可以保证之后 watchdog.c 调用的所有 pr_xxx() 打印的消息都自动带有“NMI watchdog：“的前缀。

```

#define pr_fmt(fmt) "NMI watchdog: " fmt

#include <linux/mm.h>
#include <linux/cpu.h>
#include <linux/nmi.h>
...

```

代码清单 21.4 所示为 dev_dbg()、dev_err()、dev_info() 等的定义，使用 dev_xxx() 族 API 打印的时候，设备名称会被自动加到打印消息的前头。

代码清单 21.4 包含设备信息的可替代 printk() 的宏

```

1 #define dev_printk(level, dev, format, arg...) \
2     printk(level "%s %s: " format , dev_driver_string(dev) , (dev)->bus_id , ## arg)
3
4 #ifdef DEBUG
5 #define dev_dbg(dev, format, arg...) \
6     dev_printk(KERN_DEBUG , dev , format , ## arg)
7 #else
8 #define dev_dbg(dev, format, arg...) do { (void)(dev); } while (0)

```

```

9 #endif
10
11 #define dev_err(dev, format, arg...)          \
12     dev_printk(KERN_ERR, dev, format, ##arg)
13 #define dev_info(dev, format, arg...)          \
14     dev_printk(KERN_INFO, dev, format, ##arg)
15 #define dev_warn(dev, format, arg...)           \
16     dev_printk(KERN_WARNING, dev, format, ##arg)
17 #define dev_notice(dev, format, arg...)          \
18     dev_printk(KERN_NOTICE, dev, format, ##arg)

```

在打印信息时，如果想输出 printk() 调用所在的函数名，可以使用 `_func_`；如果想输出其所在代码的行号，可以使用 `_LINE_`；想输出源代码文件名，可以使用 `_FILE_`。例如 drivers/block/sx8.c 中的：

```

#endif CARM_NDEBUG
#define assert(expr)
#else
#define assert(expr) \
    if(unlikely(!(expr))) { \
        printk(KERN_ERR "Assertion failed! %s,%s,%s,line=%d\n", \
            #expr, __FILE__, __func__, __LINE__); \
    }
#endif

```

21.4 DEBUG_LL 和 EARLY_PRINTRK

DEBUG_LL 对应内核的 Kernel low-level debugging 功能，EARLY_PRINTRK 则对应内核中一个早期的控制台。为了在内核的 drivers/tty/serial 下的控制台驱动初始化之前支持打印，可以选择 DEBUG_LL 和 EARLY_PRINTRK 这两个配置选项。另外，也需要在 bootargs 中设置 earlyprintk 的选项。

对于 LDD3_vexpress 而言，没有 DEBUG_LL 和 EARLY_PRINTRK 的时候，我们看到的内核最早的打印是：

```

Booting Linux on physical CPU 0x0
Initializing cgroup subsys cpuset
Linux version ...

```

如果我们使能 DEBUG_LL 和 EARLY_PRINTRK，选择如图 21.5 所示的“Use PL011 UART0 at 0x10009000(V2P-CA9 core tile)”这个低级别调试口，并在 bootargs 中设置 earlyprintk，则我们看到了更早的打印信息：

```
Uncompressing Linux... done, booting the kernel.
```

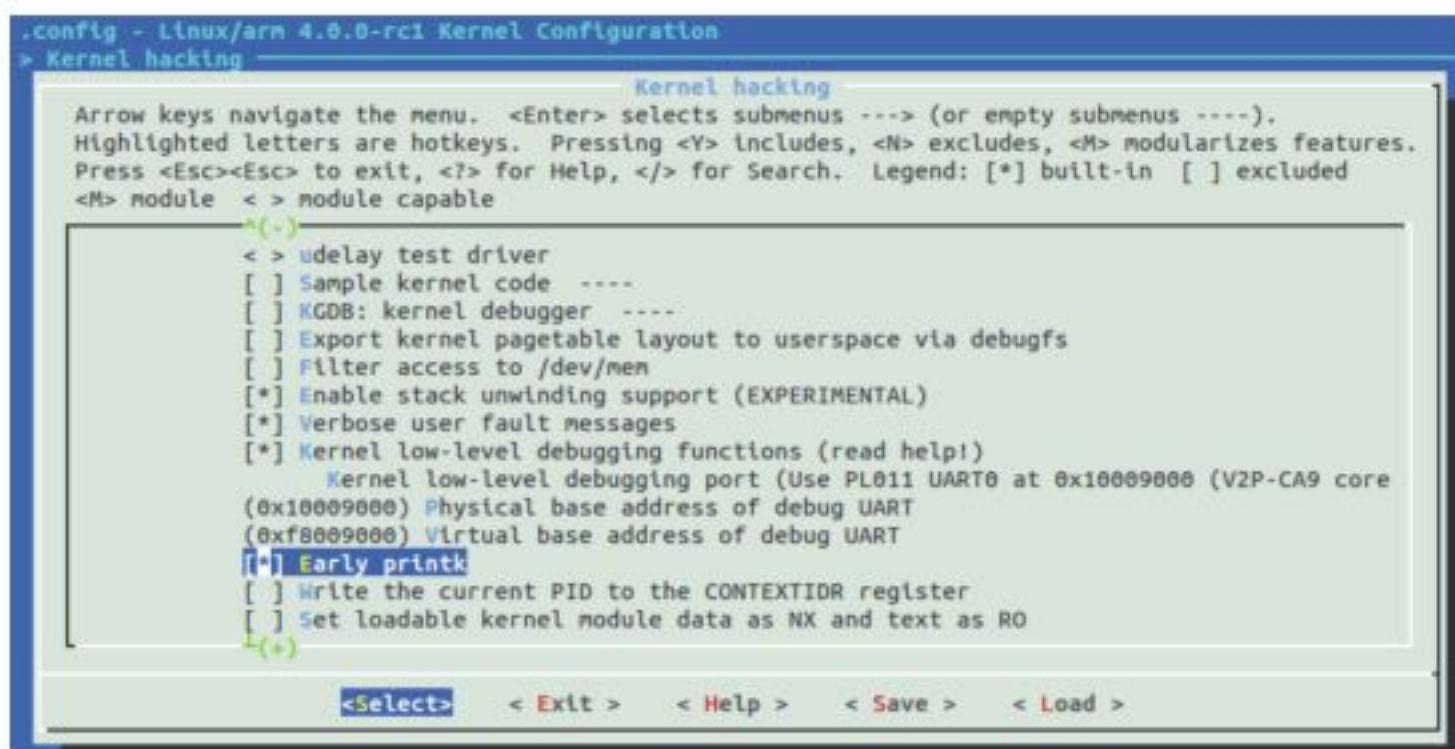


图 21.5 选择低级别调试 UART

21.5 使用“/proc”

在 Linux 系统中，“/proc”文件系统十分有用，它被内核用于向用户导出信息。“/proc”文件系统是一个虚拟文件系统，通过它可以在 Linux 内核空间和用户空间之间进行通信。在 /proc 文件系统中，我们可以将对虚拟文件的读写作为与内核中实体进行通信的一种手段，与普通文件不同的是，这些虚拟文件的内容都是动态创建的。

“/proc”下的绝大多数文件是只读的，以显示内核信息为主。但是“/proc”下的文件也并不是完全只读的，若节点可写，还可用于一定的控制或配置目的，例如前面介绍的写 /proc/sys/kernel/printk 可以改变 printk() 的打印级别。

Linux 系统的许多命令本身都是通过分析“/proc”下的文件来完成的，如 ps、top、uptime 和 free 等。例如，free 命令通过分析 /proc/meminfo 文件得到可用内存信息，下面显示了对应的 meminfo 文件和 free 命令的结果。

1. meminfo 文件

```
[root@localhost proc]# cat meminfo
MemTotal:      29516 kB
MemFree:       1472 kB
Buffers:        4096 kB
Cached:        12648 kB
SwapCached:      0 kB
Active:        14208 kB
```

```

Inactive:          8844 kB
HighTotal:         0 kB
HighFree:          0 kB
LowTotal:          29516 kB
LowFree:           1472 kB
SwapTotal:         265064 kB
SwapFree:          265064 kB
Dirty:             20 kB
Writeback:          0 kB
Mapped:            10052 kB
Slab:              3864 kB
CommitLimit:       279820 kB
Committed_AS:      13760 kB
PageTables:        444 kB
VmallocTotal:      999416 kB
VmallocUsed:       560 kB
VmallocChunk:      998580 kB

```

2. free 命令

```
[root@localhost proc]# free
      total        used        free      shared      buffers      cached
Mem:    29516       28104       1412        0        4100       12700
  -/+ buffers/cache:   11304       18212
Swap:   265064          0      265064
```

在 Linux 3.9 以及之前的内核版本中，可用如下函数创建 “/proc” 节点：

```

struct proc_dir_entry *create_proc_entry(const char *name, mode_t mode,
                                         struct proc_dir_entry *parent);

struct proc_dir_entry *create_proc_read_entry(const char *name, mode_t mode,
                                             struct proc_dir_entry *base, read_proc_t *read_proc, void * data);

```

`create_proc_entry()` 函数用于创建 “/proc” 节点，而 `create_proc_read_entry()` 调用 `create_proc_entry()` 创建只读的 “/proc” 节点。参数 `name` 为 “/proc” 节点的名称，`parent/base` 为父目录的节点，如果为 `NULL`，则指 “/proc” 目录，`read_proc` 是 “/proc” 节点的读函数指针。当 `read()` 系统调用在 “/proc” 文件系统中执行时，它映像到一个数据产生函数，而不是一个数据获取函数。

下列函数用于创建 “/proc” 目录：

```
struct proc_dir_entry *proc_mkdir(const char *name, struct proc_dir_entry *parent);
```

结合 `create_proc_entry()` 和 `proc_mkdir()`，代码清单 21.5 中的程序可用于先在 `/proc` 下创建一个目录 `procfs_example`，而后在该目录下创建一个文件 `example_file`。

代码清单 21.5 `proc_mkdir()` 和 `create_proc_entry()` 函数使用范例

```

1 /* 创建 /proc 下的目录 */
2 example_dir = proc_mkdir("procfs_example", NULL);

```

```

3 if (example_dir == NULL) {
4     rv = -ENOMEM;
5     goto out;
6 }
7
8 example_dir->owner = THIS_MODULE;
9
10 /* 创建一个 /proc 文件 */
11 example_file = create_proc_entry("example_file", 0666, example_dir);
12 if (example_file == NULL) {
13     rv = -ENOMEM;
14     goto out;
15 }
16
17 example_file->owner = THIS_MODULE;
18 example_file->read_proc = example_file_read;
19 example_file->write_proc = example_file_write;

```

作为上述函数返回值的 proc_dir_entry 结构体包含了“/proc”节点的读函数指针 (read_proc_t*read_proc)、写函数指针 (write_proc_t*write_proc) 以及父节点、子节点信息等。

/proc 节点的读写函数的类型分别为：

```

typedef int (read_proc_t)(char *page, char **start, off_t off,
                         int count, int *eof, void *data);
typedef int (write_proc_t)(struct file *file, const char __user *buffer,
                         unsigned long count, void *data);

```

读函数中 page 指针指向用于写入数据的缓冲区，start 用于返回实际的数据并写到内存页的位置，eof 是用于返回读结束标志，offset 是读的偏移，count 是要读的数据长度。start 参数比较复杂，对于 /proc 只包含简单数据的情况，通常不需要在读函数中设置 *start，这意味着内核将认为数据保存在内存页偏移 0 的地方。

写函数与 file_operations 中的 write() 成员函数类似，需要一次从用户缓冲区到内存空间的复制过程。

在 Linux 系统中可用如下函数删除 /proc 节点：

```
void remove_proc_entry(const char *name, struct proc_dir_entry *parent);
```

在 Linux 系统中已经定义好的可使用的 /proc 节点宏包括：proc_root_fs (/proc)、proc_net (/proc/net)、proc_bus (/proc/bus)、proc_root_driver (/proc/driver) 等，proc_root_fs 实际上就是 NULL。

代码清单 21.6 所示为一个简单的“/proc”文件系统使用范例，这段代码在模块加载函数中创建 /proc/test_dir 目录，并在该目录中创建 /proc/test_dir/test_rw 文件节点，在模块卸载函数中撤销“/proc”节点，而 /proc/test_dir/test_rw 文件中只保存了一个 32 位的整数。

代码清单 21.6 /proc 文件系统使用范例

```
1 #include <linux/module.h>
2 #include <linux/kernel.h>
3 #include <linux/init.h>
4 #include <linux/proc_fs.h>
5
6 static unsigned int variable;
7 static struct proc_dir_entry *test_dir, *test_entry;
8
9 static int test_proc_read(char *buf, char **start, off_t off, int count,
10     int *eof, void *data)
11 {
12     unsigned int *ptr_var = data;
13     return sprintf(buf, "%u\n", *ptr_var);
14 }
15
16 static int test_proc_write(struct file *file, const char *buffer,
17     unsigned long count, void *data)
18 {
19     unsigned int *ptr_var = data;
20
21     *ptr_var = simple_strtoul(buffer, NULL, 10);
22
23     return count;
24 }
25
26 static __init int test_proc_init(void)
27 {
28     test_dir = proc_mkdir("test_dir", NULL);
29     if (test_dir) {
30         test_entry = create_proc_entry("test_rw", 0666, test_dir);
31         if (test_entry) {
32             test_entry->nlink = 1;
33             test_entry->data = &variable;
34             test_entry->read_proc = test_proc_read;
35             test_entry->write_proc = test_proc_write;
36             return 0;
37         }
38     }
39
40     return -ENOMEM;
41 }
42 module_init(test_proc_init);
43
44 static __exit void test_proc_cleanup(void)
45 {
46     remove_proc_entry("test_rw", test_dir);
47     remove_proc_entry("test_dir", NULL);
48 }
49 module_exit(test_proc_cleanup);
```

```

50
51 MODULE_AUTHOR("Barry Song <baohua@kernel.org>");
52 MODULE_DESCRIPTION("proc exmaple");
53 MODULE_LICENSE("GPL v2");

```

上述代码第 21 行调用的 simple_strtoul() 用于将用户输入的字符串转换为无符号长整数，第 3 个参数 10 意味着转化方式是十进制。

编译上述简单的 proc.c 为 proc.ko，运行 insmod proc.ko 加载该模块后，“/proc”目录下将多出一个目录 test_dir，该目录下包含一个 test_rw，ls-l 的结果如下：

```

$ ls -l /proc/test_dir/test_rw
-rw-rw-rw- 1 root root 0 Aug 16 20:45 /proc/test_dir/test_rw
测试 /proc/test_dir/test_rw 的读写：
$ cat /proc/test_dir/test_rw
0
$ echo 111 > /proc/test_dir/test_rw
$ cat /proc/test_dir/test_rw

```

说明我们上一步执行的写操作是正确的。

在 Linux 3.10 及以后的版本中，“/proc”的内核 API 和实现架构变更较大，create_proc_entry()、create_proc_read_entry() 之类的 API 都被删除了，取而代之的是直接使用 proc_create()、proc_create_data() API。同时，也不再存在 read_proc()、write_proc() 之类的针对 proc_dir_entry 的成员函数了，而是直接把 file_operations 结构体的指针传入 proc_create() 或者 proc_create_data() 函数中，其原型为：

```

static inline struct proc_dir_entry *proc_create(
    const char *name, umode_t mode, struct proc_dir_entry *parent,
    const struct file_operations *proc_fops);

struct proc_dir_entry *proc_create_data(
    const char *name, umode_t mode, struct proc_dir_entry *parent,
    const struct file_operations *proc_fops, void *data);

```

我们把代码清单 21.6 的范例改造为同时支持 Linux 3.10 以前的内核和 Linux 3.10 以后的内核。改造结果如代码清单 21.7 所示。#if LINUX_VERSION_CODE<KERNEL_VERSION(3, 10, 0) 中的部分是旧版本的代码，与 21.6 相同，所以省略了。

代码清单 21.7 支持 Linux 3.10 以后内核的 /proc 文件系统使用范例

```

1 #include <linux/module.h>
2 #include <linux/kernel.h>
3 #include <linux/init.h>
4 #include <linux/version.h>
5 #include <linux/proc_fs.h>
6 #include <linux/seq_file.h>
7

```

```
8 static unsigned int variable;
9 static struct proc_dir_entry *test_dir, *test_entry;
10
11 #if LINUX_VERSION_CODE < KERNEL_VERSION(3, 10, 0)
12 ...
13#else
14 static int test_proc_show(struct seq_file *seq, void *v)
15 {
16     unsigned int *ptr_var = seq->private;
17     seq_printf(seq, "%u\n", *ptr_var);
18     return 0;
19 }
20
21 static ssize_t test_proc_write(struct file *file, const char __user *buffer,
22     size_t count, loff_t *ppos)
23 {
24     struct seq_file *seq = file->private_data;
25     unsigned int *ptr_var = seq->private;
26
27     *ptr_var = simple_strtoul(buffer, NULL, 10);
28     return count;
29 }
30
31 static int test_proc_open(struct inode *inode, struct file *file)
32 {
33     return single_open(file, test_proc_show, PDE_DATA(inode));
34 }
35
36 static const struct file_operations test_proc_fops =
37 {
38     .owner = THIS_MODULE,
39     .open = test_proc_open,
40     .read = seq_read,
41     .write = test_proc_write,
42     .llseek = seq_llseek,
43     .release = single_release,
44 };
45 #endif
46
47 static __init int test_proc_init(void)
48 {
49     test_dir = proc_mkdir("test_dir", NULL);
50     if (test_dir) {
51 #if LINUX_VERSION_CODE < KERNEL_VERSION(3, 10, 0)
52         ...
53#else
54         test_entry = proc_create_data("test_rw", 0666, test_dir, &test_proc_fops, &variable);
55     if (test_entry)
56         return 0;
57#endif
58 }
```

```

58 }
59
60 return -ENOMEM;
61 }
62 module_init(test_proc_init);
63
64 static __exit void test_proc_cleanup(void)
65 {
66 remove_proc_entry("test_rw", test_dir);
67 remove_proc_entry("test_dir", NULL);
68 }
69 module_exit(test_proc_cleanup);

```

21.6 Oops

当内核出现类似用户空间的 Segmentation Fault 时（例如内核访问一个并不存在的虚拟地址），Oops 会被打印到控制台和写入内核 log 缓冲区。

我们在 globalmem.c 的 globalmem_read() 函数中加上下面一行代码：

```

    ) else {
        *ppos += count;
        ret = count;
        *(unsigned int *)0 = 1; /* a kernel panic */
        printk(KERN_INFO "read %u bytes(s) from %lu\n", count, p);
    }

```

假设这个字符设备对应的设备节点是 /dev/globalmem，通过 cat/dev/globalmem 命令读设备文件，将得到如下 Oops 信息：

```

# cat /dev/globalmem
Unable to handle kernel NULL pointer dereference at virtual address 00000000
pgd = 9ec08000
[00000000] *pgd=7f733831, *pte=00000000, *pte=00000000
Internal error: Oops: 817 [#1] SMP ARM
Modules linked in: globalmem
CPU: 0 PID: 609 Comm: cat Not tainted 3.16.0+ #13
task: 9f7d8000 ti: 9f722000 task.ti: 9f722000
PC is at globalmem_read+0xbc/0xcc [globalmem]
LR is at 0x0
PC : [<7f000200>]    lr : [<00000000>]    psr: 00000013
sp : 9f723f30    ip : 00000000    fp : 00000000
r10: 9f414000    r9 : 00000000    r8 : 00001000
r7 : 00000000    r6 : 00001000    r5 : 00001000    r4 : 00000000
r3 : 00000001    r2 : 00000000    r1 : 00001000    r0 : 7f0003cc
Flags: nzcv  IRQs on  FIQs on  Mode SVC_32  ISA ARM  Segment user
Control: 10c53c7d  Table: 7ec08059  DAC: 00000015
Process cat (pid: 609, stack limit = 0x9f722240)

```

```

Stack: (0x9f723f30 to 0x9f724000)
3f20:                               7ed5ff91 9f723f80 00000000 9f79ab40
3f40: 00001000 7ed5eb18 9f723f80 00000000 00000000 800cb114 00000020 9f722000
3f60: 9f5e4628 9f79ab40 9f79ab40 00001000 7ed5eb18 00000000 00000000 800cb2ec
3f80: 00001000 00000000 9f7168c0 00001000 7ed5eb18 00000003 00000003 8000e4e4
3fa0: 9f722000 8000e360 00001000 7ed5eb18 00000003 7ed5eb18 00001000 0000002f
3fc0: 00001000 7ed5eb18 00000003 00000003 7ed5eb18 00000001 00000003 00000000
3fe0: 0015c23c 7ed5eb00 0000f718 00008d8c 60000010 00000003 00000000 00000000
[<7f000200>] (globalmem_read [globalmem]) from [<800cb114>] (vfs_read+0x98/0x13c)
[<800cb114>] (vfs_read) from [<800cb2ec>] (SyS_read+0x44/0x84)
[<800cb2ec>] (SyS_read) from [<8000e360>] (ret_fast_syscall+0x0/0x30)
Code: e1a05008 e2a77000 e1c360f0 e3a03001 (e58c3000)
---[ end trace 5a36d6470da50d02 ]---
Segmentation fault

```

上述Oops的第一行给出了“原因”，即访问了NULL pointer。Oops中的PC is at globalmem_read+0xbc/0xcc这一行代码也比较关键，给出了“事发现场”，即globalmem_read()函数偏移0xbc字节的指令处。

通过反汇编globalmem.o可以寻找到globalmem_read()函数开头位置偏移0xbc的指令，反汇编方法如下：

```
drivers/char/globalmem$ arm-linux-gnueabihf-objdump -d -S globalmem.o
```

对应的反汇编代码如下，global_read()开始于0x144，偏移0xbc的位置为0x200：

```

static ssize_t globalmem_read(struct file *filp, char __user *buf, size_t size,
                             loff_t *ppos)
{
    144:   e92d45f0      push   {r4, r5, r6, r7, r8, sl, lr}
    148:   e24dd00c      sub    sp, sp, #12
    unsigned long p = *ppos;
    14c:   e5934000      ldr    r4, [r3]
    ...
    *ppos += count;
    1f4:   e2a77000      adc    r7, r7, #0
    1f8:   e1c360f0      strd   r6, [r3]
    ret = count;
    *(unsigned int *)0 = 1; /* a kernel panic */
    1fc:   e3a03001      mov    r3, #1
    200:   e58c3000      str    r3, [ip]
    printk(KERN_INFO "read %u bytes(s) from %lu\n", count, p);
    204:   ...

    return ret;
}

```

“str r3,[ip]”是引起Oops的指令。这里仅仅给出了一个例子，工程实践中的“事发现场”并不全那么容易找到，但方法都是类似的。

21.7 BUG_ON() 和 WARN_ON()

内核中有许多地方调用类似BUG()的语句，它非常像一个内核运行时的断言，意味着本来不该执行到BUG()这条语句，一旦执行即抛出Oops。BUG()的定义为：

```
#define BUG() do { \
    printk("BUG: failure at %s:%d/%s()!\n", __FILE__, __LINE__, __func__); \
    panic("BUG!"); \
} while (0)
```

其中的panic()定义在kernel/panic.c中，会导致内核崩溃，并打印Oops。比如arch/arm/kernel/dma.c中的enable_dma()函数：

```
void enable_dma (unsigned int chan)
{
    dma_t *dma = dma_channel(chan);

    if (!dma->lock)
        goto free_dma;

    if (dma->active == 0) {
        dma->active = 1;
        dma->d_ops->enable(chan, dma);
    }
    return;

free_dma:
    printk(KERN_ERR "dma%d: trying to enable free DMA\n", chan);
    BUG();
}
```

上述代码的含义是，如果在dma->lock不成立的情况下，驱动直接调用了enable_dma()，实际上意味着内核的一个bug。

BUG()还有一个变体叫BUG_ON()，它的内部会引用BUG()，形式为：

```
#define BUG_ON(condition) do { if (unlikely(condition)) BUG(); } while (0)
```

对于BUG_ON()而言，只有当括号内的条件成立的时候，才抛出Oops。比如drivers/char/random.c中的类似代码：

```
static void push_to_pool(struct work_struct *work)
{
    struct entropy_store *r = container_of(work, struct entropy_store,
                                             push_work);
    BUG_ON(!r);
    _xfer_secondary_pool(r, random_read_wakeup_bits/8);
    trace_push_to_pool(r->name, r->entropy_count >> ENTROPY_SHIFT,
                       r->pull->entropy_count >> ENTROPY_SHIFT);
}
```

除了 BUG_ON() 外，内核有个稍微弱一些 WARN_ON()，在括号中的条件成立的时候，内核会抛出栈回溯，但是不会 panic()，这通常用于内核抛出一个警告，暗示某种不太合理的事情发生了。如在 kernel/locking/mutex-debug.c 中的 debug_mutex_unlock() 函数发现 mutex_unlock() 的调用者和 mutex_lock() 的调用者不是同一个线程的时候或者 mutex 的 owner 为空的时候，会抛出警告信息：

```
void debug_mutex_unlock(struct mutex *lock)
{
    if (likely(debug_locks)) {
        DEBUG_LOCKS_WARN_ON(lock->magic != lock);

        if (!lock->owner)
            DEBUG_LOCKS_WARN_ON(!lock->owner);
        else
            DEBUG_LOCKS_WARN_ON(lock->owner != current);

        DEBUG_LOCKS_WARN_ON(!lock->wait_list.prev && !lock->wait_list.next);
        mutex_clear_owner(lock);
    }
}
```

有时候，WARN_ON() 也可以作为一个调试技巧。比如，我们进到内核某个函数后，不知道这个函数是怎么一级一级被调用进来的，那可以在该函数中加入一个 WARN_ON(1)。

21.8 strace

在 Linux 系统中，strace 是一种相当有效的跟踪工具，它的主要特点是可以被用来监视系统调用。我们不仅可以用 strace 调试一个新开始的程序，也可以调试一个已经在运行的程序（这意味着把 strace 绑定到一个已有的 PID 上）。对于第 6 章的 globalmem 字符设备文件，以 strace 方式运行如代码清单 21.8 所示的用户空间应用程序 globalmem_test，运行的结果如下：

```
execve("./globalmem_test", ["../globalmem_test"], /* 24 vars */) = 0
...
open("/dev/globalmem", O_RDWR)           = 3      /* 打开的 /dev/globalmem 的 fd 是 3 */
ioctl(3, FIBMAP, 0)                     = 0
read(3, 0xbff17920, 200)                = -1 ENXIO (No such device or address) /* 读取失败 */
fstat64(1, {st_mode=S_IFCHR|0620, st_rdev=makedev(136, 0), ...}) = 0
mmap2(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0xb7f04000
write(1, "-1 bytes read from globalmem\n", 29-1 bytes read from globalmem
) = 29                                /* 向标准输出设备 (fd 为 1) 写入 printf 中的字符串 */
write(3, "This is a test of globalmem", 27) = 27
```

```

write(1, "27 bytes written into globalmem\n", 3227 bytes written into globalmem
) = 32
...

```

输出的每一行对应一次 Linux 系统调用，其格式为“左边 = 右边”，等号左边是系统调用的函数名及其参数，右边是该调用的返回值。

代码清单 21.8 用户空间应用程序 globalmem_test

```

1 #include ...
2
3 #define MEM_CLEAR 0x1
4 main()
5 {
6     int fd, num, pos;
7     char wr_ch[200] = "This is a test of globalmem";
8     char rd_ch[200];
9     /* 打开 /dev/globalmem */
10    fd = open("/dev/globalmem", O_RDWR, S_IRUSR | S_IWUSR);
11    if (fd != -1) /* 清除 globalmem */
12        if(ioctl(fd, MEM_CLEAR, 0) < 0)
13            printf("ioctl command failed\n");
14    /* 读 globalmem */
15    num = read(fd, rd_ch, 200);
16    printf("%d bytes read from globalmem\n", num);
17
18    /* 写 globalmem */
19    num = write(fd, wr_ch, strlen(wr_ch));
20    printf("%d bytes written into globalmem\n", num);
21    ...
22    close(fd);
23 }
24 }

```

使用 strace 虽然无法直接追踪到设备驱动中的函数，但是足以帮助工程师进行推演，如从 open(“/dev/globalmem”，O_RDWR)=3 的返回结果知道 /dev/globalmem 的 fd 为 3，之后对 fd 为 3 的文件进行 read()、write() 和 ioctl() 系统调用，最终会使 globalmem 里 file_operations 中的相应函数被调用，通过系统调用的结果就可以知道驱动中 globalmem_read()、globalmem_write() 和 globalmem_ioctl() 的运行结果。

21.9 KGDB

Linux 直接提供了对 KGDB 的支持，KGDB 采用了典型的嵌入式系统“插桩”技巧，一般依赖于串口与调试主机通信。为了支持 KGDB，串口驱动应该实现纯粹的轮询收发单一字符的成员函数，以供 drivers/tty/serial/kgdboc.c 调用，譬如 drivers/tty/serial/8250/8250_core.c 中的：

```

static struct uart_ops serial8250_pops = {
...
#endif CONFIG_CONSOLE_POLL
    .poll_get_char = serial8250_get_poll_char,
    .poll_put_char = serial8250_put_poll_char,
#endif
};

```

在编译内核时，运行 make ARCH=arm menuconfig 时需选择关于 KGDB 的编译项目，如图 21.6 所示。

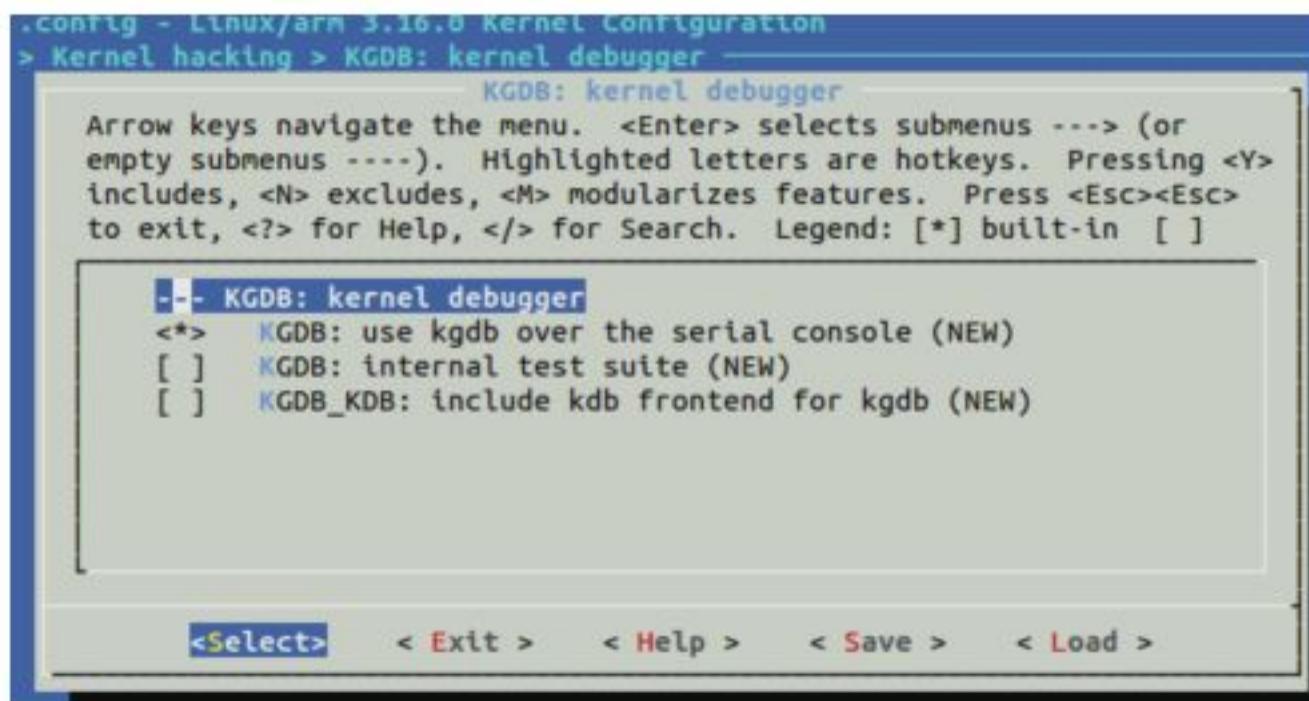


图 21.6 KGDB 编译选项配置

对于目标板而言，需要在 bootargs 中设置与 KGDB 对应的串口等信息，如 kgdboc=ttyS0,115200 kgdbcon。

如果想一开机内核就直接进入等待 GDB 连接的调试状态，可以在 bootargs 中设置 kgdbwait，kgdbwait 的含义是启动时就等待主机的 GDB 连接。而若想在内核启动后进入 GDB 调试模式，可运行 echo g>/proc/sysrq_trigger 命令给内核传入一个键值是 g 的 magic_sysrq。

在调试 PC 上，依次运行如下命令就可以启动调试并连接至目标机（假设串口在 PC 上对应的设备节点是 /dev/ttys0）：

```

# arm-eabi-gdb ./vmlinux
(gdb) set remotebaud 115200
(gdb) target remote /dev/ttys0                                // 连接目标机
(gdb)

```

之后，在主机上，我们可以使用 GDB 像调试应用程序一样调试使能了 KGDB 的目标机上的内核。

21.10 使用仿真器调试内核

在 ARM Linux 领域，目前比较主流的是采用 ARM DS-5 Development Studio 方案。ARM DS-5 是一个针对基于 Linux 的系统和裸机嵌入式系统的专业软件开发解决方案，它涵盖了开发的所有阶段，从启动代码、内核移植直到应用程序调试、分析。如图 21.7 所示，它使用了 DSTREAM 高性能仿真器（ARM 已经停止更新 RVI-RVT2 仿真器），在 Eclipse 内包含了 DS-5 和 DSTREAM 的开发插件。

调试主机一般通过网线与 DSTREAM 仿真器连接，而仿真器则连接与电路板类似的 JTAG 接口，之后用 DS-5 调试器进行调试。DS-5 图形化调试器提供了全面和直观的调试图，非常易于调试 Linux 和裸机程序，易于查看代码，进行栈回溯，查看内存、寄存器、表达式、变量，分析内核线程，设置断点。

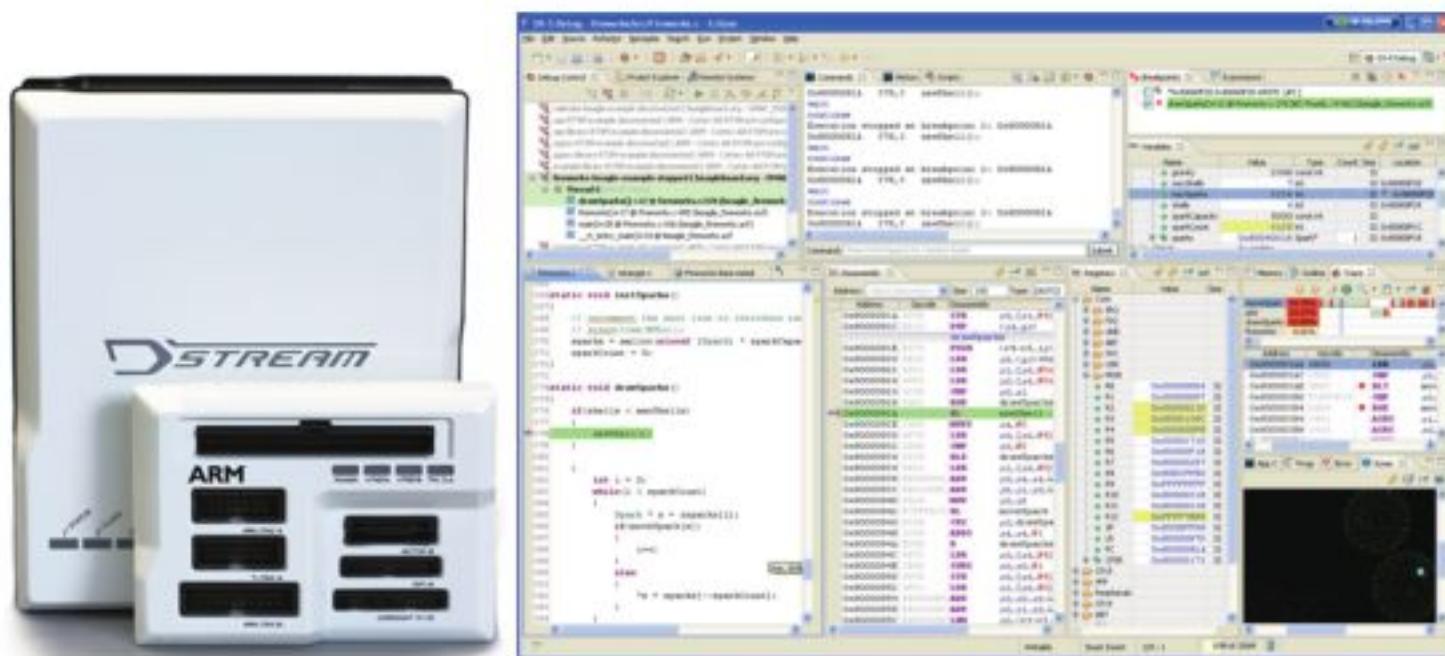


图 21.7 DSTREAM 仿真器和 DS-5 开发环境

值得一提的是，DS-5 也提供了 Streamline Performance Analyzer。ARM Streamline 性能分析器（见图 21.8）为软件开发人员提供了一种用来分析和优化在 ARM926、ARM11 和 Cortex-A 系列平台上运行的 Linux 和 Android 系统的直观方法。使用 Streamline，Linux 内核中需包含一个 gator 模块，用户空间则需要使能 gatord 后台服务器程序。关于 Streamline 具体的操作方法可以查看《ARM® DS-5 Using ARM Streamline》。

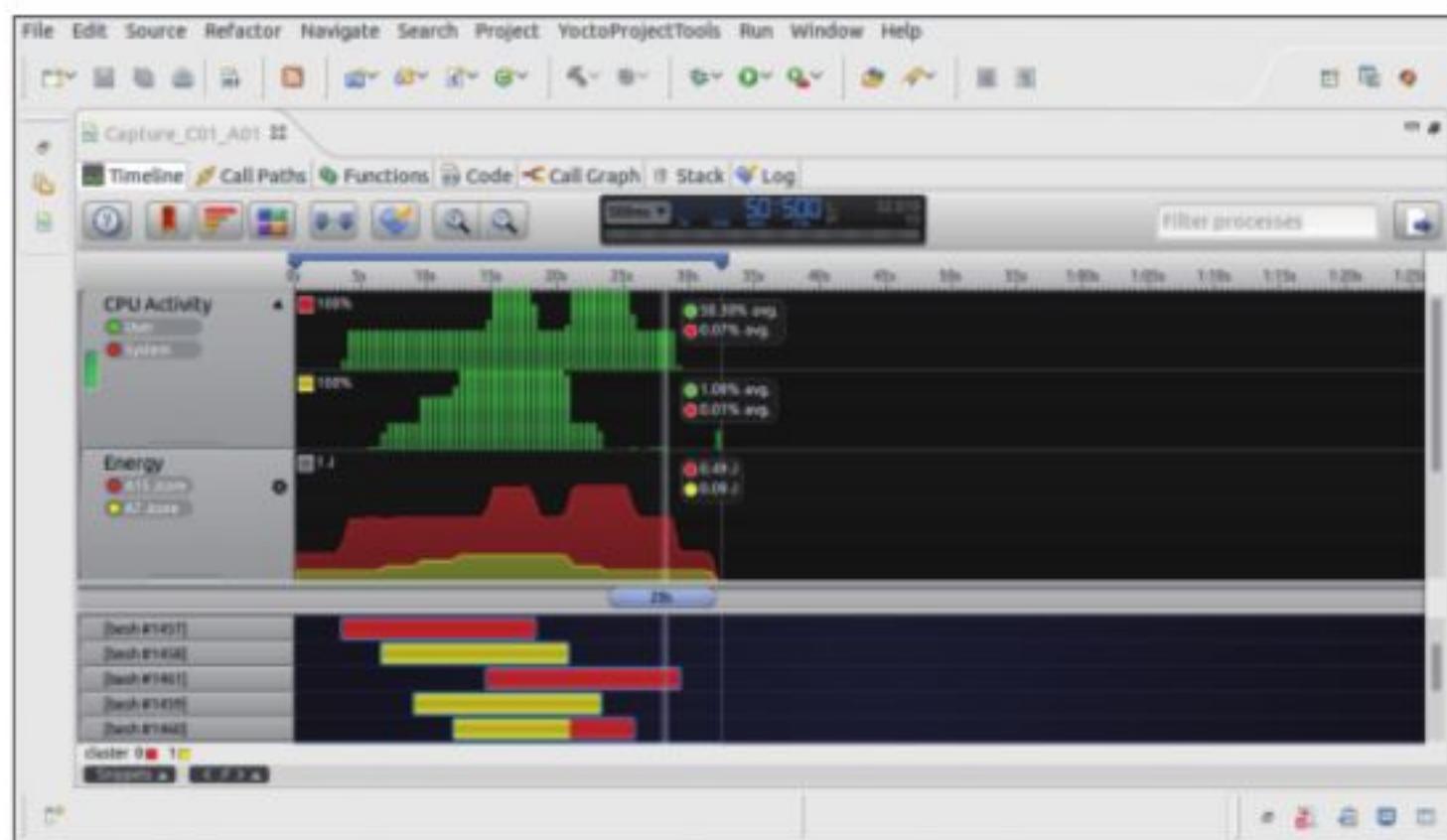


图 21.8 ARMStreamline 性能分析器

21.11 应用程序调试

在嵌入式系统中，为调试 Linux 应用程序，可在目标板上先运行 GDBServer，再让主机上的 GDB 与目标板上的 GDBServer 通过网口或串口通信。

1. 目标板

需要运行如下命令启动 GDBServer：

```
gdbserver <host_ip>:<port> <app>
```

<host_ip>:<port> 为主机的 IP 地址和端口，app 是可执行的应用程序名。

当然，也可以用系统中空闲的串口作为 GDB 调试器和 GDBServer 的底层通信手段，如：

```
gdbserver /dev/ttyS0 ./tdemo
```

2. 主机

需要先运行如下命令启动 GDB：

```
arm-eabi-gdb <app>
```

app 与 GDBServer 的 app 参数对应。

之后，运行如下命令就可以连接目标板：

```
target remote <target_ip>:<port>
```

<target_ip>:<port> 为目标机的 IP 地址和端口。

如果目标板上的 GDBServer 使用串口，则在宿主机上 GDB 也应该使用串口，如：

```
(gdb)target remote/dev/ttys1
```

之后，便可以使用 GDB 像调试本机上的程序一样调试目标机上的程序。

3. 通过 GDB server 和 ARM GDB 调试应用程序

在 ARM 开发板上放置 GDB server，便可以通过目标板与调试 PC 之间的以太网等调试。要调试的应用程序的源代码如下：

```
/*
 * gdb_example.c: program to show how to use arm-linux-gdb
 */

void increase_one(int *data)
{
    *data = *data + 1;
}

int main(int argc, char *argv[])
{
    int dat = 0;
    int *p = 0;
    increase_one(&dat);
    /* program will crash here */
    increase_one(p);
    return 0;
}
```

通过 debug 方式编译它：

```
arm-linux-gnueabi-gcc -g -o gdb_example gdb_example.c
```

将程序下载到目标板后，在目标板上运行：

```
# gdbserver 192.168.1.20:1234 gdb_example
Process gdb_example created; pid = 1096
Listening on port 1234
```

其中 192.168.1.20 为目标板的 IP，1234 为 GDBserver 的侦听端口。

如果目标机是 Android 系统，且没有以太网，可以尝试使用 adb forward 功能，比如 adb forward tcp:1234 tcp:1234 是把目标机 1234 端口与主机 1234 端口进行转发。

在主机上运行：

```
$ arm-eabi-gdb gdb_example
...
```

主机的 GDB 中运行如下命令以连接目标板：

```
(gdb) target remote 192.168.1.20:1234
Remote debugging using 192.168.1.20:1234
...
0x400007b0 in ?? ()
```

如果是 Android 的 adb forward，则上述 target remote 192.168.1.20:1234 中的 IP 地址可以去掉，因为它变成直接连接本机了，可直接写成 target remote: 1234。

运行如下命令将断点设置在 increase_one(&dat); 这一行：

```
(gdb) b gdb_example.c:16
Breakpoint 1 at 0x8390: file gdb_example.c, line 16.
```

通过 c 命令继续运行目标板上的程序，发生断点：

```
(gdb) c
Continuing.
...
Breakpoint 1, main (argc=1, argv=0xbead4eb4) at gdb_example.c:16
16 increase_one(&dat);
```

运行 n 命令执行完 increase_one(&dat); 再查看 dat 的值：

```
(gdb) n
19 increase_one(p); (gdb) p dat
$1 = 1
```

发现 dat 变成 1。继续运行 c 命令，由于即将访问空指针，gdb_example 将崩溃：

```
(gdb) c
Continuing.
Program received signal SIGSEGV, Segmentation fault.
0x00000834c in increase_one (data=0x0) at gdb_example.c:8
8 *data = *data + 1;
```

我们通过 bt 命令可以拿到 backtrace：

```
(gdb) bt
#0 0x00000834c in increase_one (data=0x0) at gdb_example.c:8
#1 0x0000083a4 in main (argc=1, argv=0xbead4eb4) at gdb_example.c:19
```

通过 info reg 命令可以查看当时的寄存器值：

```
(gdb) info reg
r0 0x0 0
r1 0xbead4eb4      3199028916
r2 0x1 1
r3 0x0 0
r4 0x4001e5e0      1073866208
r5 0x0 0
r6 0x826c 33388
r7 0x0 0
```

```
r8 0x0 0
r9 0x0 0
r10 0x40025000 1073893376
r11 0xbead4d44 3199028548
r12 0xbead4d48 3199028552
sp 0xbead4d30 0xbead4d30
lr 0x83a4 33700
pc 0x834c 0x834c <increase_one+24>
fps 0x0 0
cpsr 0x60000010 1610612752
```

21.12 Linux 性能监控与调优工具

除了保证程序的正确性以外，在项目开发中往往还关心性能和稳定性。这时候，我们往往要对内核、应用程序或整个系统进行性能优化。在性能优化中常用的手段如下。

1. 使用 top、vmstat、iostat、sysctl 等常用工具

`top` 命令用于显示处理器的活动状况。在缺省情况下，显示占用 CPU 最多的任务，并且每隔 5s 做一次刷新；`vmstat` 命令用于报告关于内核线程、虚拟内存、磁盘、陷阱和 CPU 活动的统计信息；`iostat` 命令用于分析各个磁盘的传输繁忙状况；`netstat` 是用来检测网络信息的工具；`sar` 用于收集、报告或者保存系统活动信息，其中，`sar` 用于显示数据，`sar1` 和 `sar2` 用于收集和保存数据。

`sysctl` 是一个可用于改变正在运行中的 Linux 系统的接口。用 `sysctl` 可以读取几百个以上的系统变量，例如用 `sysctl -a` 可读取所有变量。

`sysctl` 的实现原理是：所有的内核参数在 `/proc/sys` 中形成一个树状结构，`sysctl` 系统调用的内核函数是 `sys_sysctl`，匹配项目后，最后的读写在 `do_sysctl_strategy` 中完成，如

```
echo "1" > /proc/sys/net/ipv4/ip_forward
```

就等价于：

```
sysctl -w net.ipv4.ip_forward ="1"
```

2. 使用高级分析手段，如 OProfile、gprof

OProfile 可以帮助用户识别诸如模块的占用时间、循环的展开、高速缓存的使用率低、低效的类型转换和冗余操作、错误预测转移等问题。它收集有关处理器事件的信息，其中包括 TLB 的故障、停机、存储器访问以及缓存命中和未命中的指令的攫取数量。

OProfile 支持两种采样方式：基于事件的采样（Event Based）和基于时间的采样（Time Based）。基于事件的采样是 OProfile 只记录特定事件（比如 L2 缓存未命中）的发生次数，当达到用户设定的定值时 Oprofile 就记录一下（采一个样）。这种方式需要 CPU 内部有性能计数器（Performance Counter）。基于时间的采样是 OProfile 借助 OS 时钟中断的机制，在每个时钟中断，OProfile 都会记录一次（采一次样）。引入它的目的在于，提供对没有性能计数器的

CPU 的支持，其精度相对于基于事件的采样要低，因为要借助 OS 时钟中断的支持，对于禁用中断的代码，OProfile 不能对其进行分析。

OProfile 在 Linux 上分两部分，一个是内核模块 (oprofile.ko)，另一个是用户空间的守护进程 (oprofiled)。前者负责访问性能计数器或者注册基于时间采样的函数，并将采样值置于内核的缓冲区内。后者在后台运行，负责从内核空间收集数据，写入文件。其运行步骤如下。

- 1) 初始化 opcontrol --init
- 2) 配置 opcontrol --setup --event=...
- 3) 启动 opcontrol --start
- 4) 运行待分析的程序 xxx
- 5) 取出数据


```
opcontrol --dump
opcontrol --stop
```
- 6) 分析结果 opreport -l ./xxx

用 GNU gprof 可以打印出程序运行中各个函数消耗的时间，以帮助程序员找出众多函数中耗时最多的函数；还可产生程序运行时的函数调用关系，包括调用次数，以帮助程序员分析程序的运行流程。

GNU gprof 的实现原理：在编译和链接程序的时候（使用 -pg 编译和链接选项），gcc 在应用程序的每个函数中都加入名为 mcount(_mcount 或 __mcount，依赖于编译器或操作系统）的函数，也就是说应用程序里的每一个函数都会调用 mcount，而 mcount 会在内存中保存一张函数调用图，并通过函数调用堆栈的形式查找子函数和父函数的地址。这张调用图也保存了所有与函数相关的调用时间、调用次数等的所有信息。

GNU gprof 的基本用法如下。

- 1) 使用 -pg 编译和链接应用程序。
- 2) 执行应用程序并使它生成供 gprof 分析的数据。
- 3) 使用 gprof 程序分析应用程序生成的数据。

3. 进行内核跟踪，如 LTTng

LTTng (Linux Trace Toolkit-next generation，官方网站为 <http://lttng.org/>) 是一个用于跟踪系统详细运行状态和流程的工具，它可以跟踪记录系统中的特定事件。这些事件包括：系统调用的进入和退出；陷阱 / 中断 (Trap/Irq) 的进入和退出；进程调度事件；内核定时器；进程管理相关事件——创建、唤醒、信号处理等；文件系统相关事件——open/read/write/seek/ioctl 等；内存管理相关事件——内存分配 / 释放等；其他 IPC/ 套接字 / 网络等事件。而对于这些记录，我们可以通过图形的方式经由 lttv-gui 查看，如图 21.9 所示。

4. 使用 LTP 进行压力测试

LTP (Linux Test Project，官方网站为 <http://ltp.sourceforge.net/>) 是一个由 SGI 发起并由 IBM 负责维护的合作计划。它的目的是为开源社区提供测试套件来验证 Linux 的可靠性、健

壮性和稳定性。它通过压力测试来判断系统的稳定性和可靠性，在工程中我们可使用 LTP 测试套件对 Linux 操作系统进行超长时间的测试，它可进行文件系统压力测试、硬盘 I/O 测试、内存管理压力测试、IPC 压力测试、SCHED 测试、命令功能的验证测试、系统调用功能的验证测试等。

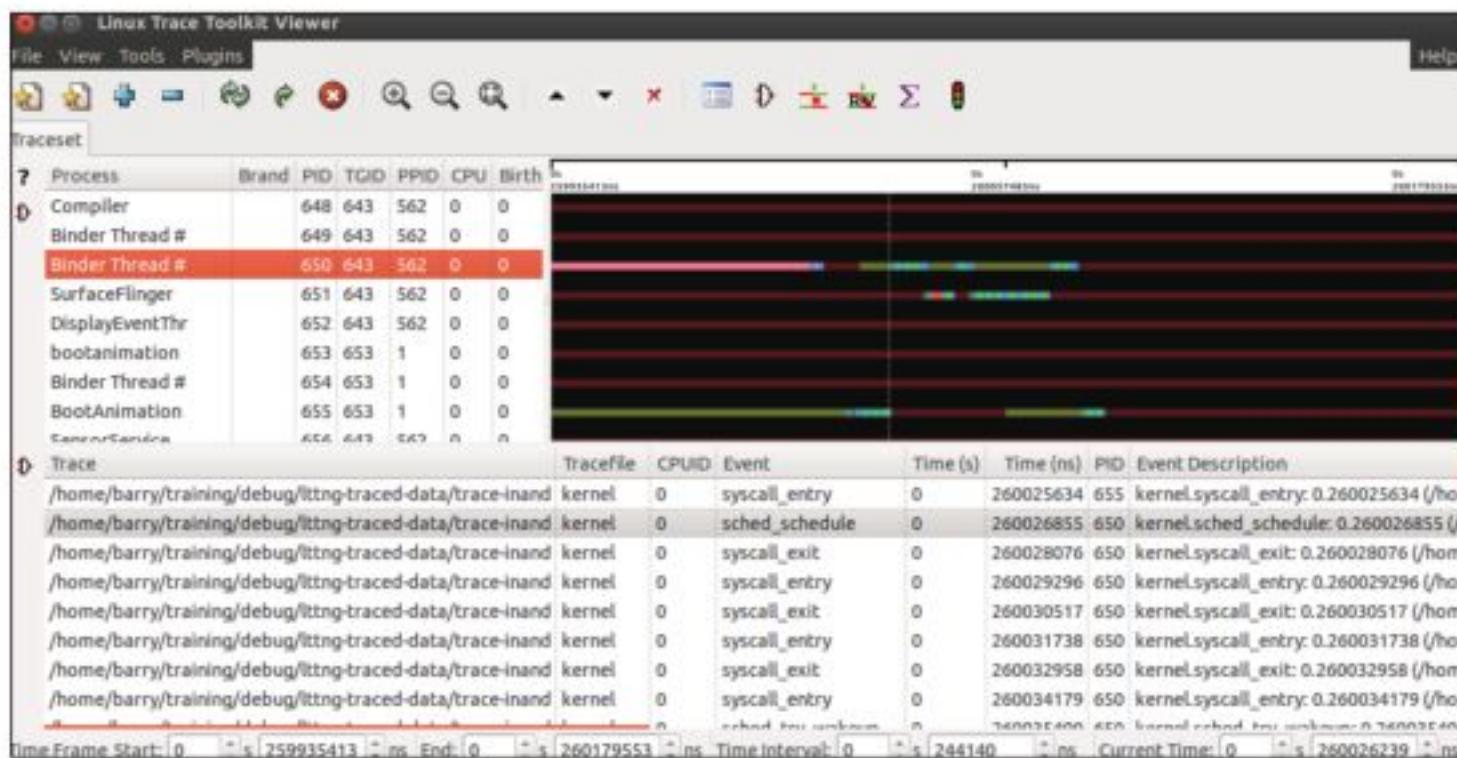


图 21.9 LTTng 形成的时序图

5. 使用 Benchmark 评估系统

可用于 Linux 的 Benchmark 的包括 lmbench、UnixBench、AIM9、Netperf、SSLperf、dbench、Bonnie、Bonnie++、Iozone、BYTEMARK 等，它们可用于评估操作系统、网络、I/O 子系统、CPU 等的性能，参考网址 <http://lbs.sourceforge.net/> 列出了许多 Benchmark 工具。

21.13 总结

Linux 程序的调试，尤其是内核的调试看起来比较复杂，没有类似于 VC++、Tornado 的 IDE 开发环境，最常用的调试手段依然是文本方式的 GDB。文本方式的 GDB 调试器功能异常强大，当我们使用习惯后，就会用得非常自然。

Linux 内核驱动的调试方法包括“插桩”、使用仿真器和借助 printk()、Oops、strace 等，在大多数情况下，原始的 printk() 仍然是最有效的手段。

除了本章介绍的方法外，在驱动的调试中很可能还会借助其他的硬件或软件调试工具，如调试 USB 驱动最好借助 USB 分析仪，用 USB 分析仪将可捕获 USB 通信中的数据包，如同网络中的 Sniffer 软件一样。

壮性和稳定性。它通过压力测试来判断系统的稳定性和可靠性，在工程中我们可使用 LTP 测试套件对 Linux 操作系统进行超长时间的测试，它可进行文件系统压力测试、硬盘 I/O 测试、内存管理压力测试、IPC 压力测试、SCHED 测试、命令功能的验证测试、系统调用功能的验证测试等。

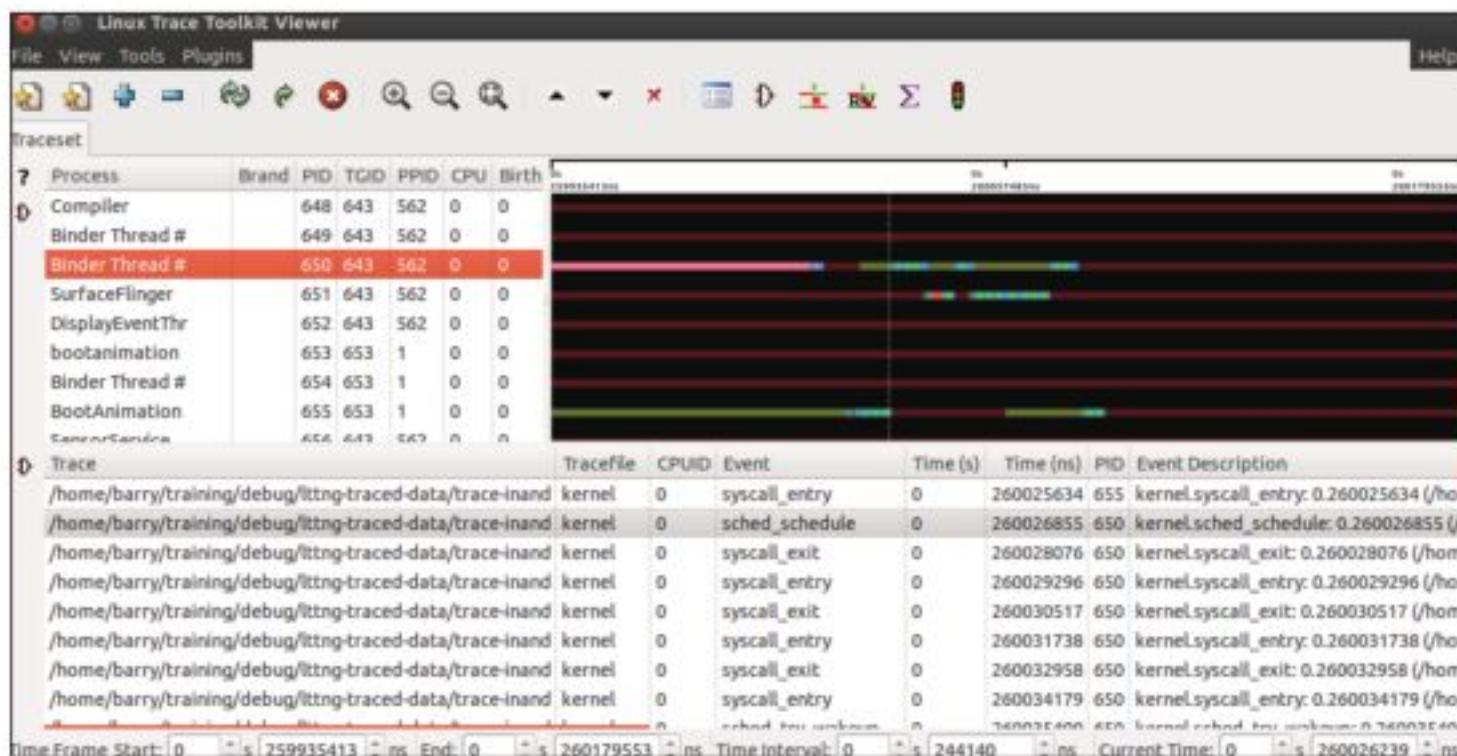


图 21.9 LTTng 形成的时序图

5. 使用 Benchmark 评估系统

可用于 Linux 的 Benchmark 的包括 lmbench、UnixBench、AIM9、Netperf、SSLperf、dbench、Bonnie、Bonnie++、Iozone、BYTEMARK 等，它们可用于评估操作系统、网络、I/O 子系统、CPU 等的性能，参考网址 <http://lbs.sourceforge.net/> 列出了许多 Benchmark 工具。

21.13 总结

Linux 程序的调试，尤其是内核的调试看起来比较复杂，没有类似于 VC++、Tornado 的 IDE 开发环境，最常用的调试手段依然是文本方式的 GDB。文本方式的 GDB 调试器功能异常强大，当我们使用习惯后，就会用得非常自然。

Linux 内核驱动的调试方法包括“插桩”、使用仿真器和借助 printk()、Oops、strace 等，在大多数情况下，原始的 printk() 仍然是最有效的手段。

除了本章介绍的方法外，在驱动的调试中很可能还会借助其他的硬件或软件调试工具，如调试 USB 驱动最好借助 USB 分析仪，用 USB 分析仪将可捕获 USB 通信中的数据包，如同网络中的 Sniffer 软件一样。

壮性和稳定性。它通过压力测试来判断系统的稳定性和可靠性，在工程中我们可使用 LTP 测试套件对 Linux 操作系统进行超长时间的测试，它可进行文件系统压力测试、硬盘 I/O 测试、内存管理压力测试、IPC 压力测试、SCHED 测试、命令功能的验证测试、系统调用功能的验证测试等。

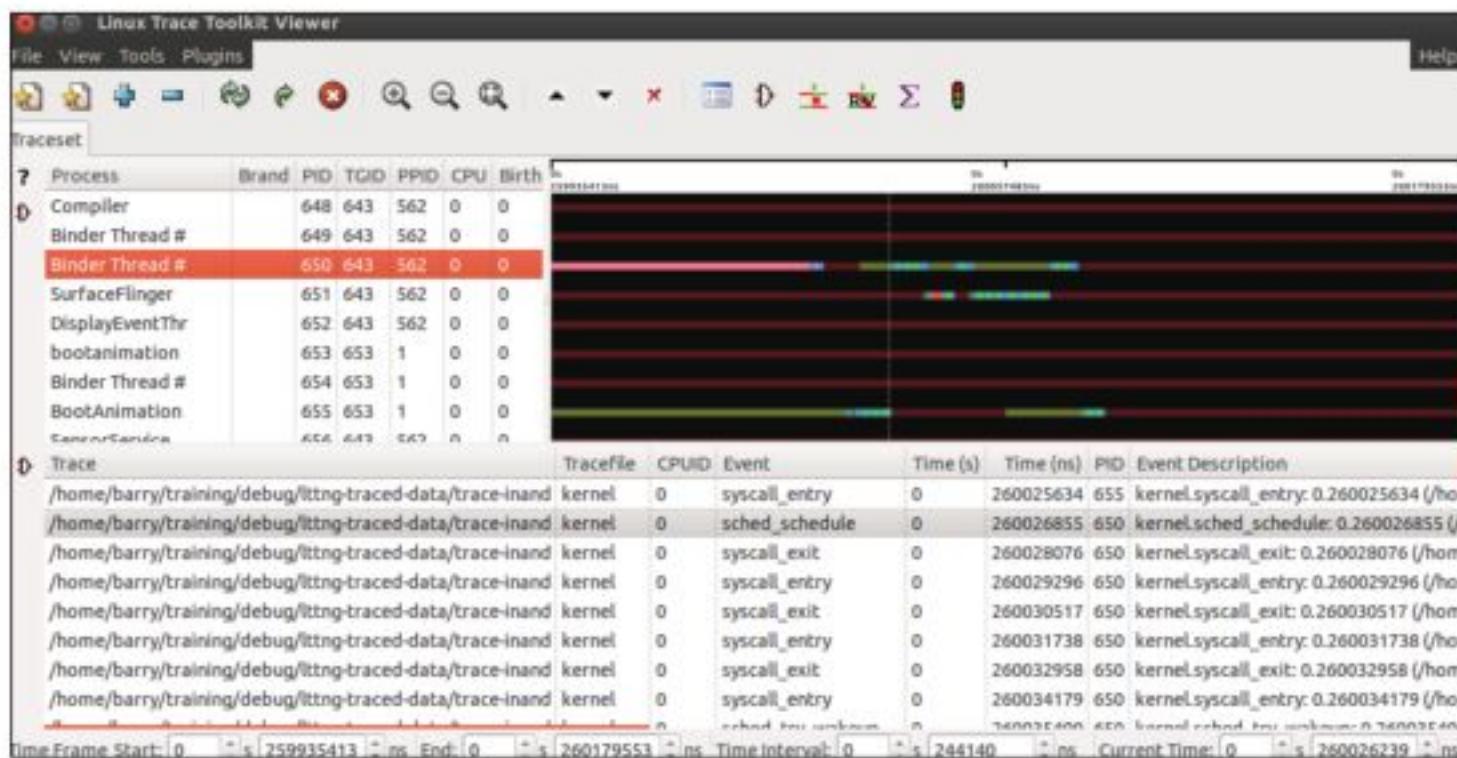


图 21.9 LTTng 形成的时序图

5. 使用 Benchmark 评估系统

可用于 Linux 的 Benchmark 的包括 lmbench、UnixBench、AIM9、Netperf、SSLperf、dbench、Bonnie、Bonnie++、Iozone、BYTEMARK 等，它们可用于评估操作系统、网络、I/O 子系统、CPU 等的性能，参考网址 <http://lbs.sourceforge.net/> 列出了许多 Benchmark 工具。

21.13 总结

Linux 程序的调试，尤其是内核的调试看起来比较复杂，没有类似于 VC++、Tornado 的 IDE 开发环境，最常用的调试手段依然是文本方式的 GDB。文本方式的 GDB 调试器功能异常强大，当我们使用习惯后，就会用得非常自然。

Linux 内核驱动的调试方法包括“插桩”、使用仿真器和借助 printk()、Oops、strace 等，在大多数情况下，原始的 printk() 仍然是最有效的手段。

除了本章介绍的方法外，在驱动的调试中很可能还会借助其他的硬件或软件调试工具，如调试 USB 驱动最好借助 USB 分析仪，用 USB 分析仪将可捕获 USB 通信中的数据包，如同网络中的 Sniffer 软件一样。

壮性和稳定性。它通过压力测试来判断系统的稳定性和可靠性，在工程中我们可使用 LTP 测试套件对 Linux 操作系统进行超长时间的测试，它可进行文件系统压力测试、硬盘 I/O 测试、内存管理压力测试、IPC 压力测试、SCHED 测试、命令功能的验证测试、系统调用功能的验证测试等。

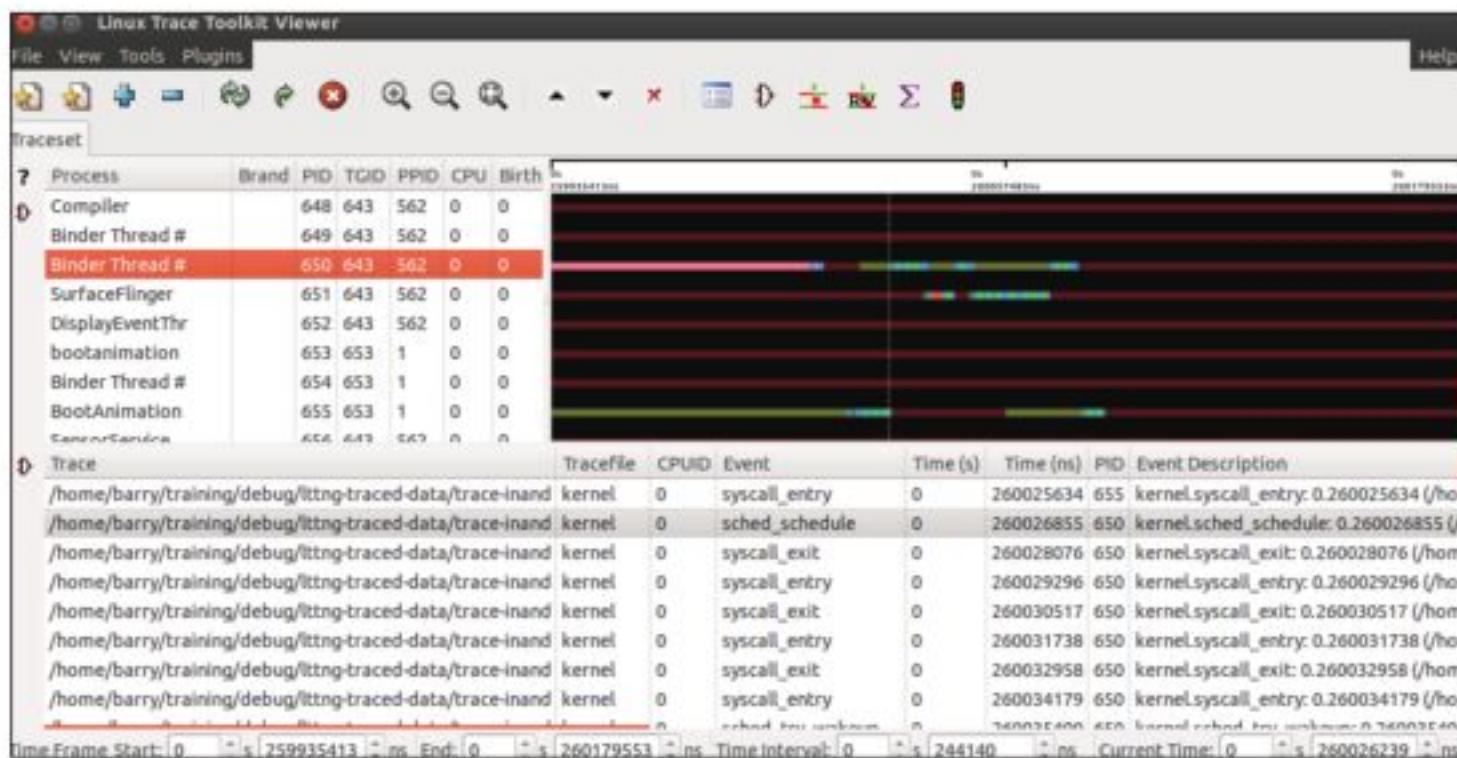


图 21.9 LTTng 形成的时序图

5. 使用 Benchmark 评估系统

可用于 Linux 的 Benchmark 的包括 lmbench、UnixBench、AIM9、Netperf、SSLperf、dbench、Bonnie、Bonnie++、Iozone、BYTEMARK 等，它们可用于评估操作系统、网络、I/O 子系统、CPU 等的性能，参考网址 <http://lbs.sourceforge.net/> 列出了许多 Benchmark 工具。

21.13 总结

Linux 程序的调试，尤其是内核的调试看起来比较复杂，没有类似于 VC++、Tornado 的 IDE 开发环境，最常用的调试手段依然是文本方式的 GDB。文本方式的 GDB 调试器功能异常强大，当我们使用习惯后，就会用得非常自然。

Linux 内核驱动的调试方法包括“插桩”、使用仿真器和借助 printk()、Oops、strace 等，在大多数情况下，原始的 printk() 仍然是最有效的手段。

除了本章介绍的方法外，在驱动的调试中很可能还会借助其他的硬件或软件调试工具，如调试 USB 驱动最好借助 USB 分析仪，用 USB 分析仪将可捕获 USB 通信中的数据包，如同网络中的 Sniffer 软件一样。

壮性和稳定性。它通过压力测试来判断系统的稳定性和可靠性，在工程中我们可使用 LTP 测试套件对 Linux 操作系统进行超长时间的测试，它可进行文件系统压力测试、硬盘 I/O 测试、内存管理压力测试、IPC 压力测试、SCHED 测试、命令功能的验证测试、系统调用功能的验证测试等。

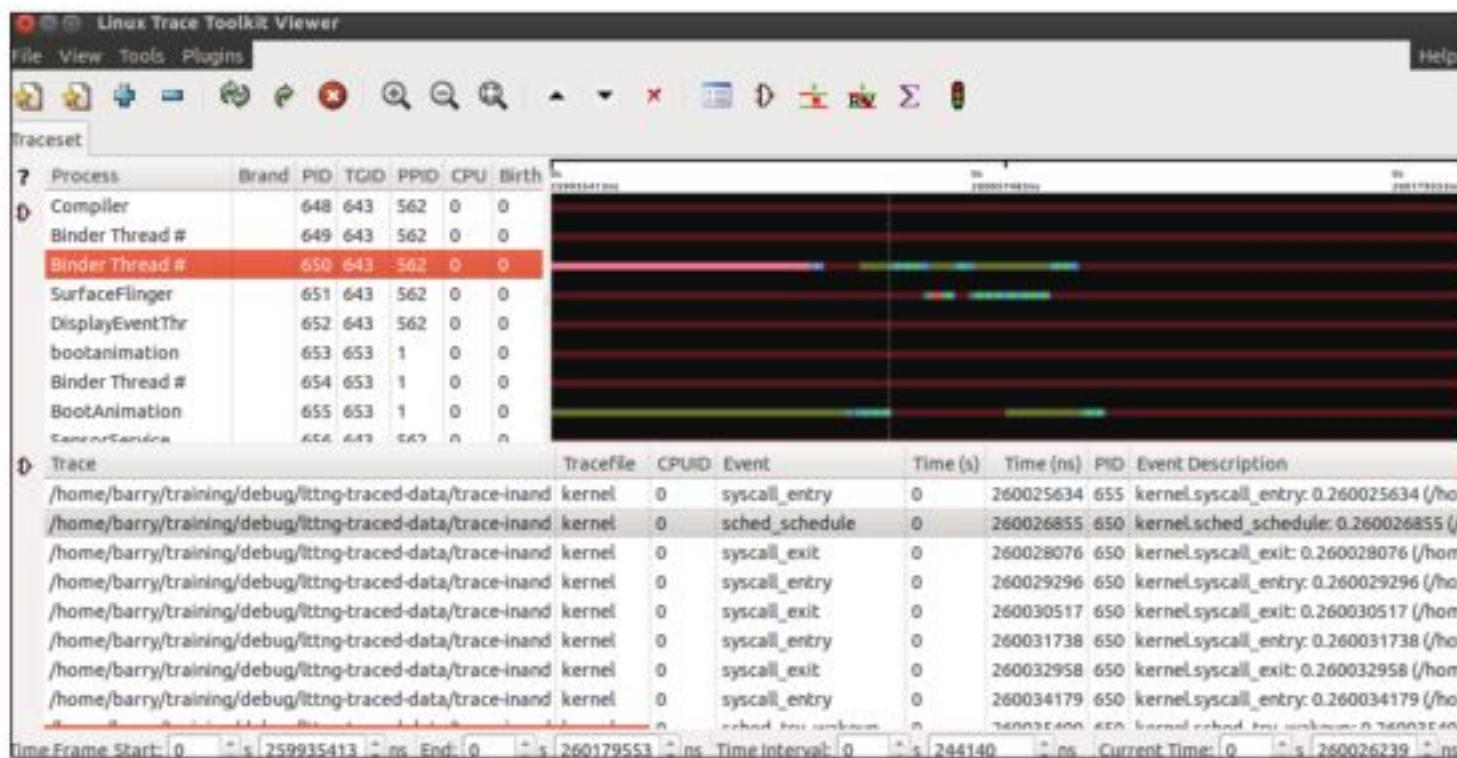


图 21.9 LTTng 形成的时序图

5. 使用 Benchmark 评估系统

可用于Linux的Benchmark的包括lmbench、UnixBench、AIM9、Netperf、SSLperf、dbench、Bonnie、Bonnie++、Iozone、BYTEMARK等，它们可用于评估操作系统、网络、I/O子系统、CPU等的性能，参考网址 <http://lbs.sourceforge.net/> 列出了许多Benchmark工具。

21.13 总结

Linux程序的调试，尤其是内核的调试看起来比较复杂，没有类似于VC++、Tornado的IDE开发环境，最常用的调试手段依然是文本方式的GDB。文本方式的GDB调试器功能异常强大，当我们使用习惯后，就会用得非常自然。

Linux内核驱动的调试方法包括“插桩”、使用仿真器和借助printf()、Oops、strace等，在大多数情况下，原始的printf()仍然是最有效的手段。

除了本章介绍的方法外，在驱动的调试中很可能还会借助其他的硬件或软件调试工具，如调试USB驱动最好借助USB分析仪，用USB分析仪将可捕获USB通信中的数据包，如同网络中的Sniffer软件一样。

壮性和稳定性。它通过压力测试来判断系统的稳定性和可靠性，在工程中我们可使用 LTP 测试套件对 Linux 操作系统进行超长时间的测试，它可进行文件系统压力测试、硬盘 I/O 测试、内存管理压力测试、IPC 压力测试、SCHED 测试、命令功能的验证测试、系统调用功能的验证测试等。

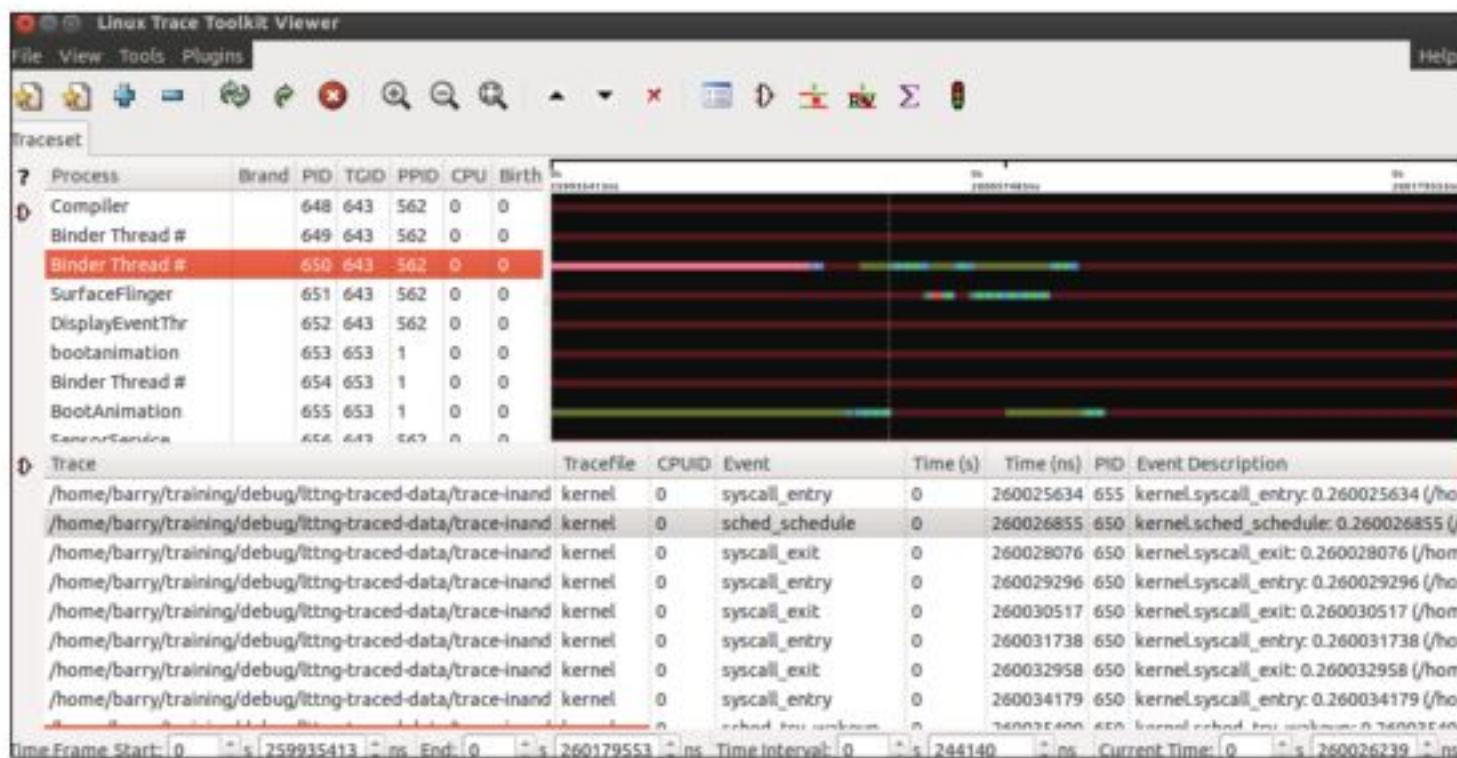


图 21.9 LTTng 形成的时序图

5. 使用 Benchmark 评估系统

可用于 Linux 的 Benchmark 的包括 lmbench、UnixBench、AIM9、Netperf、SSLperf、dbench、Bonnie、Bonnie++、Iozone、BYTEMARK 等，它们可用于评估操作系统、网络、I/O 子系统、CPU 等的性能，参考网址 <http://lbs.sourceforge.net/> 列出了许多 Benchmark 工具。

21.13 总结

Linux 程序的调试，尤其是内核的调试看起来比较复杂，没有类似于 VC++、Tornado 的 IDE 开发环境，最常用的调试手段依然是文本方式的 GDB。文本方式的 GDB 调试器功能异常强大，当我们使用习惯后，就会用得非常自然。

Linux 内核驱动的调试方法包括“插桩”、使用仿真器和借助 printk()、Oops、strace 等，在大多数情况下，原始的 printk() 仍然是最有效的手段。

除了本章介绍的方法外，在驱动的调试中很可能还会借助其他的硬件或软件调试工具，如调试 USB 驱动最好借助 USB 分析仪，用 USB 分析仪将可捕获 USB 通信中的数据包，如同网络中的 Sniffer 软件一样。

壮性和稳定性。它通过压力测试来判断系统的稳定性和可靠性，在工程中我们可使用 LTP 测试套件对 Linux 操作系统进行超长时间的测试，它可进行文件系统压力测试、硬盘 I/O 测试、内存管理压力测试、IPC 压力测试、SCHED 测试、命令功能的验证测试、系统调用功能的验证测试等。

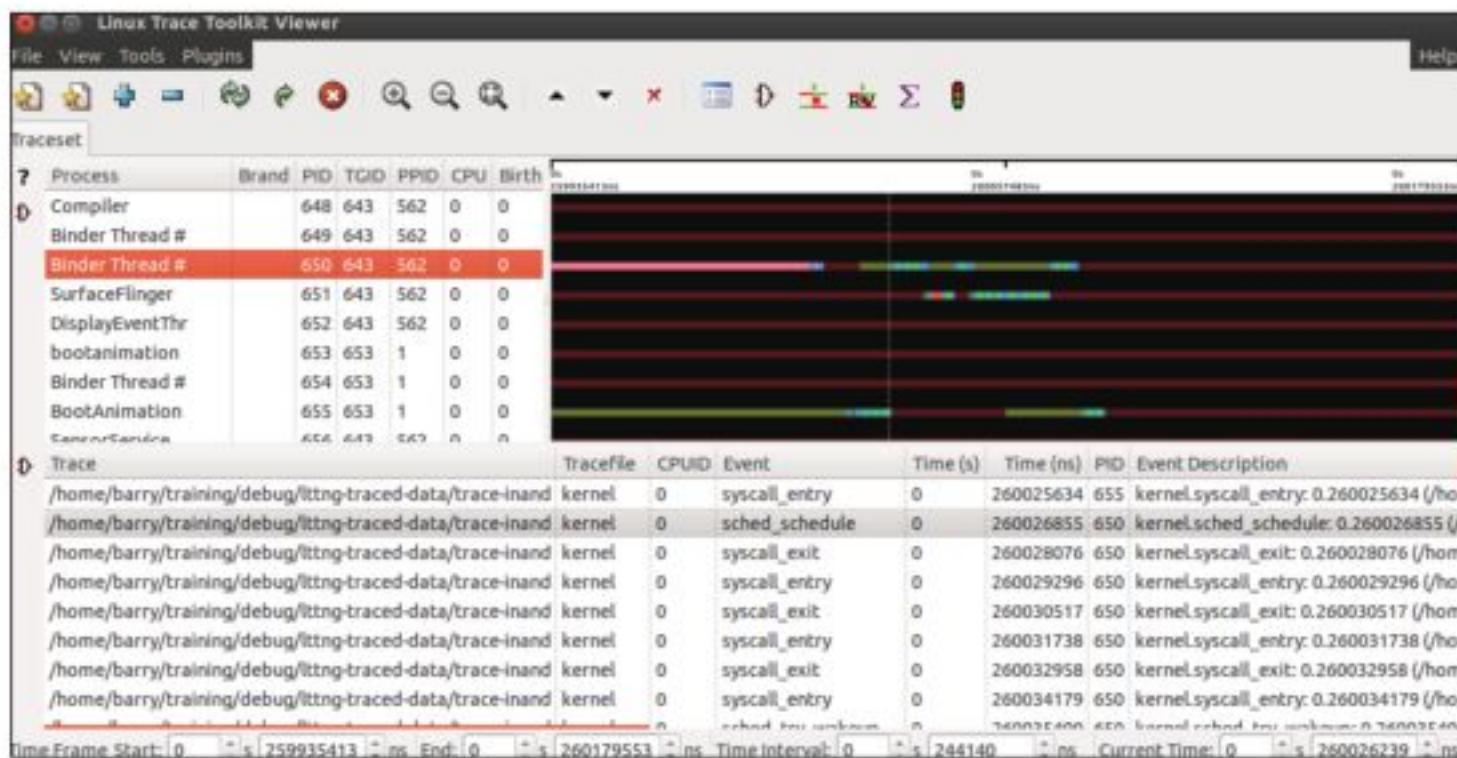


图 21.9 LTTng 形成的时序图

5. 使用 Benchmark 评估系统

可用于 Linux 的 Benchmark 的包括 lmbench、UnixBench、AIM9、Netperf、SSLperf、dbench、Bonnie、Bonnie++、Iozone、BYTEMARK 等，它们可用于评估操作系统、网络、I/O 子系统、CPU 等的性能，参考网址 <http://lbs.sourceforge.net/> 列出了许多 Benchmark 工具。

21.13 总结

Linux 程序的调试，尤其是内核的调试看起来比较复杂，没有类似于 VC++、Tornado 的 IDE 开发环境，最常用的调试手段依然是文本方式的 GDB。文本方式的 GDB 调试器功能异常强大，当我们使用习惯后，就会用得非常自然。

Linux 内核驱动的调试方法包括“插桩”、使用仿真器和借助 printk()、Oops、strace 等，在大多数情况下，原始的 printk() 仍然是最有效的手段。

除了本章介绍的方法外，在驱动的调试中很可能还会借助其他的硬件或软件调试工具，如调试 USB 驱动最好借助 USB 分析仪，用 USB 分析仪将可捕获 USB 通信中的数据包，如同网络中的 Sniffer 软件一样。

壮性和稳定性。它通过压力测试来判断系统的稳定性和可靠性，在工程中我们可使用 LTP 测试套件对 Linux 操作系统进行超长时间的测试，它可进行文件系统压力测试、硬盘 I/O 测试、内存管理压力测试、IPC 压力测试、SCHED 测试、命令功能的验证测试、系统调用功能的验证测试等。

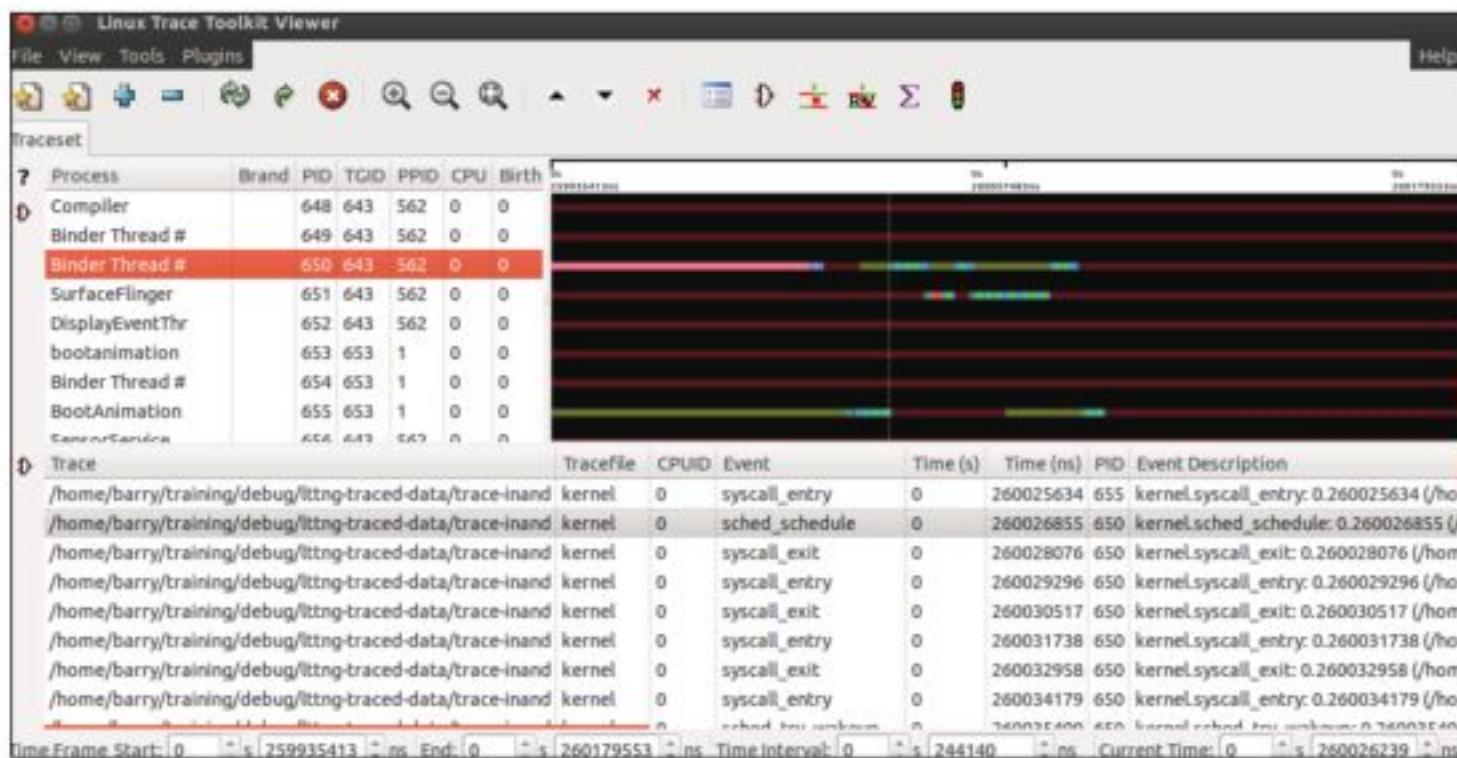


图 21.9 LTTng 形成的时序图

5. 使用 Benchmark 评估系统

可用于 Linux 的 Benchmark 的包括 lmbench、UnixBench、AIM9、Netperf、SSLperf、dbench、Bonnie、Bonnie++、Iozone、BYTEMARK 等，它们可用于评估操作系统、网络、I/O 子系统、CPU 等的性能，参考网址 <http://lbs.sourceforge.net/> 列出了许多 Benchmark 工具。

21.13 总结

Linux 程序的调试，尤其是内核的调试看起来比较复杂，没有类似于 VC++、Tornado 的 IDE 开发环境，最常用的调试手段依然是文本方式的 GDB。文本方式的 GDB 调试器功能异常强大，当我们使用习惯后，就会用得非常自然。

Linux 内核驱动的调试方法包括“插桩”、使用仿真器和借助 printk()、Oops、strace 等，在大多数情况下，原始的 printk() 仍然是最有效的手段。

除了本章介绍的方法外，在驱动的调试中很可能还会借助其他的硬件或软件调试工具，如调试 USB 驱动最好借助 USB 分析仪，用 USB 分析仪将可捕获 USB 通信中的数据包，如同网络中的 Sniffer 软件一样。