刘子扬 2020K8009929043

作业1

1.1 一个C程序可以编译成目标文件或可执行文件。目标文件和可执行文件通常包含text、data、bss、rodata段，程序执行时也会用到堆（heap）和栈（stack）。

（1）请写一个C程序，使其包含data段和bss段，并在运行时包含堆的使用。请说明所写程序中哪些变量在data段、bss段和堆上。

（2）请了解readelf、objdump命令的使用，用这些命令查看（1）中所写程序的data和bss段，截图展示。

（3）请说明（1）中所写程序是否用到了栈。

```c
#include<stdio.h>
#include<stdlib.h>
int array[100*100]={1};
int main(){
    static int bss_1;
    char * str[100];
    void * now = (void *)malloc(sizeof(int));
    return 0;
}
```

本程序中，array变量在data段，bss_1变量在bss段，指针数组str位于栈上，now指针在栈上，在运行时分配给now的空间位于堆上。

首先执行size命令，可以观察到：

```
root@DESKTOP-JDVJ1RO:/mnt/c/Users/lzy/Desktop/大三/操作系统/作业1# size q1.o
   text    data     bss     dec     hex filename
    162   40000       4   40166    9ce6 q1.o
```

其次执行

其中text、data、bss段都分配空间给了一些变量。随后执行objdump -t q1.o指令：

```
q1.o:     file format elf64-x86-64

SYMBOL TABLE:
0000000000000000 l    df *ABS*  0000000000000000 q1.c
0000000000000000 l    d  .text  0000000000000000 .text
0000000000000000 l    d  .data  0000000000000000 .data
0000000000000000 l    d  .bss   0000000000000000 .bss
0000000000000000 l     O .bss   0000000000000004 bss_1.2833
0000000000000000 l    d  .debug_info    0000000000000000 .debug_info
0000000000000000 l    d  .debug_abbrev  0000000000000000 .debug_abbrev
0000000000000000 l    d  .debug_aranges 0000000000000000 .debug_aranges
0000000000000000 l    d  .debug_line    0000000000000000 .debug_line
0000000000000000 l    d  .debug_str     0000000000000000 .debug_str
0000000000000000 l    d  .note.GNU-stack        0000000000000000 .note.GNU-stack
0000000000000000 l    d  .note.gnu.property     0000000000000000 .note.gnu.property
0000000000000000 l    d  .eh_frame      0000000000000000 .eh_frame
0000000000000000 l    d  .comment       0000000000000000 .comment
0000000000000000 g     O .data  0000000000009c40 array
0000000000000000 g     F .text  000000000000004a main
```

可以观察到，bss_1变量是个静态局部变量，位于bss段。Array是已初始化的

全局变量，位于data段。

在这段代码中，指针数组str和now指针在栈上，因为其为局部变量，位于栈中，运行时才会向其分配内存。

1.2 Linux 下常见的3种系统调用方法包括有：

（1）通过glibc提供的库函数

（2）使用syscall函数直接调用相应的系统调用

（3）通过int 80指令（32位系统）或者syscall指令（64位系统）的内联汇编调用

系统环境：

```
root@DESKTOP-JDVJ1RO:/mnt/c/Users/lzy/Desktop/大三/操作系统/作业1# uname -a
Linux DESKTOP-JDVJ1RO 5.10.102.1-microsoft-standard-WSL2 #1 SMP Wed Mar 2 00:30:59 UTC 2022 x86_64 x86_64 x86_64 GNU/Linux
```

函数选择：

经过调研发现，gettimeofday和clock_gettime两种函数中clock_gettime更适合测量系统调用的时间开销，因为clock_gettime函数可以精准到纳秒级别，而gettimeofday只能达到微秒级别。

三种系统调用的实现：

通过glibc库原生提供的getpid()函数完成调用，代码如下：

```c
#include<stdio.h>
#include<sys/stat.h>
#include<unistd.h>
#include<time.h>
int main(){
    struct timespec t1 = {0, 0};
    struct timespec t2 = {0, 0};

    pid_t pid;

    clock_gettime(CLOCK_REALTIME, &t1);

    pid = getpid();

    clock_gettime(CLOCK_REALTIME, &t2);
    double duration=(t2.tv_sec - t1.tv_sec)*1000000000 + (t2.tv_nsec - t1.tv_nsec);
    printf("time = %.4lf ns\n",duration);

    printf("current process's pid(getpid):%d %d\n",pid,getpid());
    return 0;
}
```

执行结果：

刘子扬 2020K8009929043

```
root@DESKTOP-JDVJ1RO:/mnt/c/Users/lzy/Desktop/大三/操作系统/作业1# ./gblic
time = 2123.0000 ns
current process's pid(getpid):1666 1666
root@DESKTOP-JDVJ1RO:/mnt/c/Users/lzy/Desktop/大三/操作系统/作业1# ./gblic
time = 2177.0000 ns
current process's pid(getpid):1676 1676
root@DESKTOP-JDVJ1RO:/mnt/c/Users/lzy/Desktop/大三/操作系统/作业1# ./gblic
time = 2104.0000 ns
current process's pid(getpid):1680 1680
root@DESKTOP-JDVJ1RO:/mnt/c/Users/lzy/Desktop/大三/操作系统/作业1# ./gblic
time = 2287.0000 ns
current process's pid(getpid):1681 1681
root@DESKTOP-JDVJ1RO:/mnt/c/Users/lzy/Desktop/大三/操作系统/作业1# ./gblic
time = 632.0000 ns
current process's pid(getpid):1682 1682
```

通过syscall函数获取进程id代码如下：

```c
#include<stdio.h>
#include<sys/stat.h>
#include <sys/syscall.h>
#include <sys/types.h>
#include<unistd.h>
#include<time.h>
int main(){
    struct timespec t1 = {0, 0};
    struct timespec t2 = {0, 0};
    clock_gettime(CLOCK_REALTIME, &t1);
    pid_t pid;
    pid=syscall(SYS_gettid);
    clock_gettime(CLOCK_REALTIME, &t2);

    double duration=(t2.tv_sec - t1.tv_sec)*1000000000 + (t2.tv_nsec - t1.tv_nsec);
    printf("time = %.4lf ns\n",duration);

    printf("current process's pid(SYS_gettid):%d %d\n",pid,getpid());
    return 0;
}
```

执行结果：

```
root@DESKTOP-JDVJ1RO:/mnt/c/Users/lzy/Desktop/大三/操作系统/作业1# ./syscall
time = 507.0000 ns
current process's pid(SYS_gettid):1733 1733
root@DESKTOP-JDVJ1RO:/mnt/c/Users/lzy/Desktop/大三/操作系统/作业1# ./syscall
time = 685.0000 ns
current process's pid(SYS_gettid):1734 1734
root@DESKTOP-JDVJ1RO:/mnt/c/Users/lzy/Desktop/大三/操作系统/作业1# ./syscall
time = 2173.0000 ns
current process's pid(SYS_gettid):1735 1735
root@DESKTOP-JDVJ1RO:/mnt/c/Users/lzy/Desktop/大三/操作系统/作业1# ./syscall
time = 618.0000 ns
current process's pid(SYS_gettid):1736 1736
root@DESKTOP-JDVJ1RO:/mnt/c/Users/lzy/Desktop/大三/操作系统/作业1# ./syscall
time = 2482.0000 ns
current process's pid(SYS_gettid):1737 1737
```

刘子扬 2020K8009929043

通过汇编方式获得进程ID方式如下：

64位内联汇编

```c
#include<stdio.h>
#include<sys/stat.h>
#include<unistd.h>
#include<time.h>
int main(){
    struct timespec t1 = {0, 0};
    struct timespec t2 = {0, 0};
    clock_gettime(CLOCK_REALTIME, &t1);
    pid_t pid;
    asm volatile(
        "mov $39, %%eax\n\t"
        "syscall\n\t"
        :"=a"(pid)
                /*第一个冒号后的限定字符串用于描述指令中的“输出”操作数。
                  刮号中的 pid 将操作数与 C 语言的变量联系起来。
                  "a":寄存器 EAX。
                  "b":寄存器 EBX。
                  "c":寄存器 ECX。
                  "d":寄存器 EDX。
                */
    );

    clock_gettime(CLOCK_REALTIME, &t2);
    double duration=(t2.tv_sec - t1.tv_sec)*1000000000 + (t2.tv_nsec - t1.tv_nsec);
    printf("time = %.4lf ns\n",duration);

    printf("current process's pid(ASM):%d %d\n",pid,getpid());
    return 0;
}
```

32位内联汇编

```c
#include<stdio.h>
#include<sys/stat.h>
#include<unistd.h>
#include<time.h>
int main(){
    struct timespec t1 = {0, 0};
    struct timespec t2 = {0, 0};
    clock_gettime(CLOCK_REALTIME, &t1);
    pid_t pid;
    asm volatile(
        "mov $20, %%eax\n\t"
        "int $0x80\n\t"
        :"=a"(pid)
    );

    clock_gettime(CLOCK_REALTIME, &t2);
    double duration=(t2.tv_sec - t1.tv_sec)*1000000000 + (t2.tv_nsec - t1.tv_nsec);
    printf("time = %.4lf ns\n",duration);

    printf("current process's pid(ASM):%d %d\n",pid,getpid());
    return 0;
}
```

执行结果

```
root@DESKTOP-JDVJ1RO:/mnt/c/Users/lzy/Desktop/大三/操作系统/作业1# ./int80
time = 405.0000 ns
current process's pid(ASM-64):1857 1857
time = 424.0000 ns
current process's pid(ASM-32):1857 1857
root@DESKTOP-JDVJ1RO:/mnt/c/Users/lzy/Desktop/大三/操作系统/作业1# ./int80
time = 520.0000 ns
current process's pid(ASM-64):1882 1882
time = 601.0000 ns
current process's pid(ASM-32):1882 1882
root@DESKTOP-JDVJ1RO:/mnt/c/Users/lzy/Desktop/大三/操作系统/作业1# ./int80
time = 212.0000 ns
current process's pid(ASM-64):1883 1883
time = 376.0000 ns
current process's pid(ASM-32):1883 1883
root@DESKTOP-JDVJ1RO:/mnt/c/Users/lzy/Desktop/大三/操作系统/作业1# ./int80
time = 198.0000 ns
current process's pid(ASM-64):1893 1893
time = 399.0000 ns
current process's pid(ASM-32):1893 1893
root@DESKTOP-JDVJ1RO:/mnt/c/Users/lzy/Desktop/大三/操作系统/作业1# ./int80
time = 385.0000 ns
current process's pid(ASM-64):1894 1894
time = 461.0000 ns
current process's pid(ASM-32):1894 1894
```

运行时间的对比分析：

方式	时间（五次平均/ns）
gblic库函数调用	1864
Syscall系统调用	1293
64位内联汇编	344
32位内联汇编	452
编译时没有开任何优化开关	

分析：

三种方法的运行时间差异符合预期:

对于gblic库的库函数getpid()调研后发现其本质是一个封装好的系统调用,其实先和syscall的实现类似，那么其速度的确应该慢于 syscall。对于 32 位和 64 位内联汇编语言来讲，64 位汇编语言肯定是快于 32 位的，因为这是一台 64 位机，32 位 api 会被转为 64 位 api 再执行。总的来讲，汇编语言肯定要快于高层次的系统调用，所以三种方法都符合预期。