hw3 实验报告

一、实验名称

存储器层次结果和存储保护

二、实验环境

visual studio2017

ubuntu14.4 64bits+gcc4.8

三、实验内容

实验（一）基于 cache 的存储访问

1. 分别给出在 Windows 和 Linux 系统中的程序代码、执行结果。

windows 程序代码：

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

//#define M 10
//#define N 10000000

//#define M 10000
//#define N 10000

#define M 10000000
#define N 10

short a[M][N];

void assign_array_rows() {
    int i, j;
    for (i = 0; i < M; i++)
        for (j = 0; j < N; j++)
            a[i][j] = 0;
}

void assign_array_cols() {
    int i, j;
    for (j = 0; j < N; j++)
        for (i = 0; i < M; i++)
            a[i][j] = 0;
}
```

```
29
30  void main() {
31      clock_t r_start, r_stop, c_start, c_stop;
32      r_start = clock();
33      assign_array_rows();
34      r_stop = clock();
35      printf("程序段A循环历时：%f\n", (double)(r_stop - r_start) / CLOCKS_PER_SEC);
36
37      c_start = clock();
38      assign_array_cols();
39      c_stop = clock();
40      printf("程序段B循环历时：%f\n", (double)(c_stop - c_start) / CLOCKS_PER_SEC);
41      system("pause");
42  }
```

windows 执行结果：

M=10，N=10000000，全局变量

```
程序段A循环历时：0.390000
程序段B循环历时：0.254000
请按任意键继续. . .
```

M=10000，N=10000，全局变量

```
程序段A循环历时：0.305000
程序段B循环历时：0.974000
请按任意键继续. . .
```

M=10000000，N=10，全局变量

```
程序段A循环历时：0.288000
程序段B循环历时：0.290000
请按任意键继续. . .
```

linux 程序代码：

一开始用了 **time 函数**，发现精度不够

```
void main(){
        time_t r_start,r_stop,c_start,c_stop;
        r_start=time(NULL);
        assign_array_rows();
        r_stop=time(NULL);
        printf("程序段A循环历时:%ld\n",r_stop-r_start);

        c_start=time(NULL);
        assign_array_cols();
        c_stop=time(NULL);
        printf("程序段B循环历时:%ld\n",c_stop-c_start);
}
```

```
qjw@qjw-VirtualBox:~/hw3$ gcc mn.c -o mn
qjw@qjw-VirtualBox:~/hw3$ ./mn
程序段A循环历时:0
程序段B循环历时:1
```

改为使用 clock()函数

```
#include<stdio.h>
#include<time.h>
//#define M 10000
//#define N 10000

//#define M 10
//#define N 10000000

#define M 10000000
#define N 10

short a[M][N];

void assign_array_rows(){
        int i,j;
        for(i=0;i<M;i++)
                for(j=0;j<N;j++)
                        a[i][j]=0;
}

void assign_array_cols(){
        int i,j;
        for(j=0;j<N;j++)
                for(i=0;i<M;i++)
                        a[i][j]=0;
}
void main(){
         clock_t r_start,r_stop,c_start,c_stop;
         r_start=clock();
         assign_array_rows();
         r_stop=clock();
         printf("程序段A循环历时:%f\n",(double)(r_stop-r_start)/CLOCKS_PER_SEC);

         c_start=clock();
         assign_array_cols();
         c_stop=clock();
         printf("程序段B循环历时:%f\n",(double)(c_stop-c_start)/CLOCKS_PER_SEC);
}
```

linux 运行结果：

M=10，N=10000000，全局变量

```
qjw@qjw-VirtualBox:~/hw3$ gcc mn2.c -o mn2
qjw@qjw-VirtualBox:~/hw3$ ./mn2
程序段A循环历时:0.271502
程序段B循环历时:0.180571
```

M=10000，N=10000，全局变量

```
qjw@qjw-VirtualBox:~/hw3$ gcc mn2.c -o mn2
qjw@qjw-VirtualBox:~/hw3$ ./mn2
程序段A循环历时:0.321001
程序段B循环历时:0.983774
```

M=10000000，N=10，全局变量

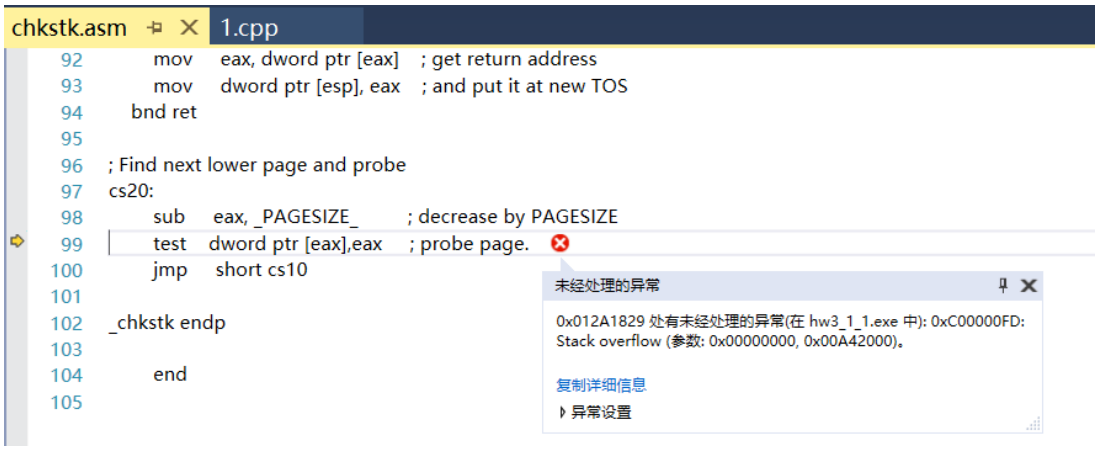
```
qjw@qjw-VirtualBox:~/hw3$ gcc mn2.c -o mn2
qjw@qjw-VirtualBox:~/hw3$ ./mn2
程序段A循环历时:0.308298
程序段B循环历时:0.255689
```

2. 修改程序使 a 作为非静态局部变量并分配在栈区，分别给出在 Windows 和 Linux 系统中的执行结果。(注：windows 和 linux 对栈分配大小有一定限制，需要手动修改)

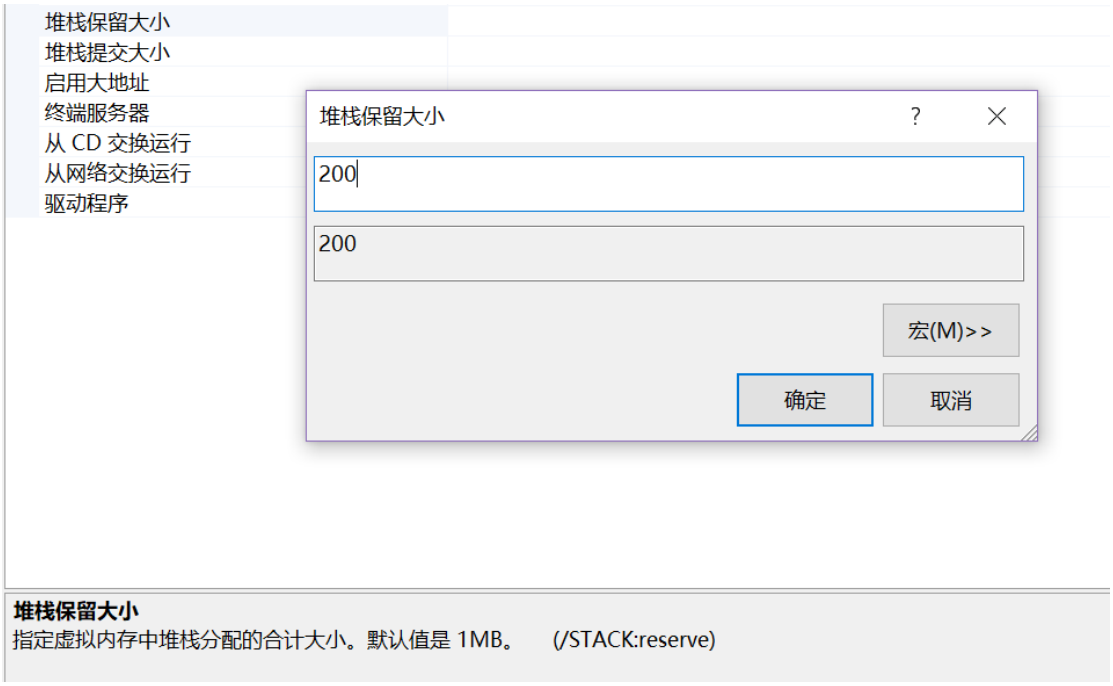
windows 程序代码：

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

#define M 10
#define N 10000000

//#define M 10000
//#define N 10000

//#define M 10000000
//#define N 10

void assign_array_rows() {
    short a[M][N];
    int i, j;
    for (i = 0; i < M; i++)
        for (j = 0; j < N; j++)
            a[i][j] = 0;
}

void assign_array_cols() {
    short a[M][N];
    int i, j;
    for (j = 0; j < N; j++)
        for (i = 0; i < M; i++)
            a[i][j] = 0;
}

void main() {
    clock_t r_start, r_stop, c_start, c_stop;
    r_start = clock();
    assign_array_rows();
    r_stop = clock();
    printf("程序段A循环历时：%f\n", (double)(r_stop - r_start) / CLOCKS_PER_SEC);

    c_start = clock();
    assign_array_cols();
    c_stop = clock();
    printf("程序段B循环历时：%f\n", (double)(c_stop - c_start) / CLOCKS_PER_SEC);
    system("pause");
}
```

windows 运行结果：



发生了溢出，因为 windows 对**栈分配大小**有一定限制，默认是 1MB，因为数组元素有 10^8 个，元素类型 short 占 2 字节，所以应该分配 200MB，超出了默认值，发生泄漏，需要进行手动修改



运行以后发现 overflow 问题依然存在！发现是因为之前以为默认的单位是 MB，所以写成了 200，这样看来应该是 B，改成 200000000，运行成功

windows 运行结果

M=10，N=10000000，栈区

```
程序段A循环历时：0.454000
程序段B循环历时：0.343000
请按任意键继续. . .
```

M=10000，N=10000，栈区

```
程序段A循环历时：0.520000
程序段B循环历时：0.829000
请按任意键继续. . .
```

M=10000000，N=10，栈区

```
程序段A循环历时：0.692000
程序段B循环历时：0.405000
请按任意键继续. . .
```

linux 程序代码：

```c
#include<stdio.h>
#include<time.h>
//#define M 10
//#define N 10000000

//#define M 10000
//#define N 10000

#define M 10000000
#define N 10

void assign_array_rows(){
        short a[M][N];
        int i,j;
        for(i=0;i<M;i++)
                for(j=0;j<N;j++)
                        a[i][j]=0;
}

void assign_array_cols(){
        short a[M][N];
        int i,j;
        for(j=0;j<N;j++)
                for(i=0;i<M;i++)
                        a[i][j]=0;
}

void main(){
        clock_t r_start,r_stop,c_start,c_stop;
        r_start=clock();
        assign_array_rows();
        r_stop=clock();
        printf("程序段A循环历时:%f\n",(double)(r_stop-r_start)/CLOCKS_PER_SEC);

        c_start=clock();
        assign_array_cols();
        c_stop=clock();
        printf("程序段B循环历时:%f\n",(double)(c_stop-c_start)/CLOCKS_PER_SEC);
}
```

linux 运行结果：

```
qjw@qjw-VirtualBox:~/hw3$ gcc mn3.c -o mn3
qjw@qjw-VirtualBox:~/hw3$ ./mn3
段错误 (核心已转储)
qjw@qjw-VirtualBox:~/hw3$ gcc mn3.c -o mn3
qjw@qjw-VirtualBox:~/hw3$ ./mn3
段错误 (核心已转储)
qjw@qjw-VirtualBox:~/hw3$ gcc mn3.c -o mn3
qjw@qjw-VirtualBox:~/hw3$ ./mn3
段错误 (核心已转储)
```

也需要**手动修改栈分配大小**

ulimit -s 查看得知默认分配 8192KB，修改为 200MB

```
qjw@qjw-VirtualBox:~/hw3$ ulimit -s
8192
qjw@qjw-VirtualBox:~/hw3$ ulimit -s 204800
qjw@qjw-VirtualBox:~/hw3$ ulimit -s
204800
```

linux 运行结果

M=10，N=1000000，栈区

```
qjw@qjw-VirtualBox:~/hw3$ ./mn3
程序段A循环历时:0.310512
程序段B循环历时:0.174441
```

M=10000，N=10000，栈区

```
qjw@qjw-VirtualBox:~/hw3$ ./mn3
程序段A循环历时:0.295889
程序段B循环历时:0.847727
```

M=1000000，N=10，栈区

```
qjw@qjw-VirtualBox:~/hw3$ ./mn3
程序段A循环历时:0.197699
程序段B循环历时:0.215537
```

3. 修改程序使 a 作为非静态局部变量并分配在堆区，分别给出在 Windows 和 Linux 系统中的执行结果。

windows 程序代码：

```c
#include<stdio.h>
#include<stdlib.h>
#include<malloc.h>
#include<time.h>

//#define M 10
//#define N 10000000

//#define M 10000
//#define N 10000

 #define M 10000000
 #define N 10

void assign_array_rows() {
    short *a=(short* )malloc(sizeof(short)*(M*N));
    int i, j;
    for (i = 0; i < M; i++)
        for (j = 0; j < N; j++)
            a[i*N + j] = 0;
}

void assign_array_cols() {
    short *a = (short*)malloc(sizeof(short)*(M*N));
    int i, j;
    for (j = 0; j < N; j++)
        for (i = 0; i < M; i++)
            a[i*N + j] = 0;
}

```

```
31  void main() {
32      clock_t r_start, r_stop, c_start, c_stop;
33      r_start = clock();
34      assign_array_rows();
35      r_stop = clock();
36      printf("程序段A循环历时：%f\n", (double)(r_stop - r_start) / CLOCKS_PER_SEC);
37
38      c_start = clock();
39      assign_array_cols();
40      c_stop = clock();
41      printf("程序段B循环历时：%f\n", (double)(c_stop - c_start) / CLOCKS_PER_SEC);
42      system("pause");
43  }
```

windows 运行结果：

M=10，N=10000000，非静态局部变量，分配在堆区

```
程序段A循环历时：0.472000
程序段B循环历时：0.389000
请按任意键继续. . .
```

M=10000，N=10000，非静态局部变量，分配在堆区

```
程序段A循环历时：0.464000
程序段B循环历时：0.999000
请按任意键继续. . .
```

M=10000000，N=10，非静态局部变量，分配在堆区

```
程序段A循环历时：0.357000
程序段B循环历时：0.509000
请按任意键继续. . .
```

linux 程序代码：

```
#include<stdio.h>
#include<time.h>
#include<malloc.h>
//#define M 10
//#define N 10000000

//#define M 10000
//#define N 10000

#define M 10000000
#define N 10

void assign_array_rows(){
        short *a=malloc(sizeof(short)*(M*N));
        int i,j;
        for(i=0;i<M;i++)
                for(j=0;j<N;j++)
                        a[i*N+j]=0;
}

void assign_array_cols(){
        short *a=malloc(sizeof(short)*(M*N));
        int i,j;
        for(j=0;j<N;j++)
                for(i=0;i<M;i++)
                        a[i*N+j]=0;
}
```

```c
void main(){
        clock_t r_start,r_stop,c_start,c_stop;
        r_start=clock();
        assign_array_rows();
        r_stop=clock();
        printf("程序段A循环历时:%f\n",(double)(r_stop-r_start)/CLOCKS_PER_SEC);

        c_start=clock();
        assign_array_cols();
        c_stop=clock();
        printf("程序段B循环历时:%f\n",(double)(c_stop-c_start)/CLOCKS_PER_SEC);
}
```

linux 运行结果：

M=10，N=10000000，非静态局部变量，分配在堆区

```
qjw@qjw-VirtualBox:~/hw3$ gcc mn4.c -o mn4
qjw@qjw-VirtualBox:~/hw3$ ./mn4
程序段A循环历时:0.324184
程序段B循环历时:0.482762
```

M=10000，N=10000，非静态局部变量，分配在堆区

```
qjw@qjw-VirtualBox:~/hw3$ gcc mn4.c -o mn4
qjw@qjw-VirtualBox:~/hw3$ ./mn4
程序段A循环历时:0.295919
程序段B循环历时:1.415286
```

M=10000000，N=10，非静态局部变量，分配在堆区

```
qjw@qjw-VirtualBox:~/hw3$ gcc mn4.c -o mn4
qjw@qjw-VirtualBox:~/hw3$ ./mn4
程序段A循环历时:0.214133
程序段B循环历时:0.378014
```

4. 对上述执行结果进行分析，说明局部数据块大小、数组访问顺序等和执行时间之间关系。

实验结果整理：

windows

程序运行时间	程序 A（先行后列访问数组）			程序 B（先列后行访问数组）		
	静态区	栈区	堆区	静态区	栈区	堆区
M=10 N=10000000	0.390000	0.454000	0.472000	0.254000	0.343000	0.389000
M=10000 N=10000	0.305000	0.520000	0.464000	0.974000	0.829000	0.999000
M=10000000 N=10	0.288000	0.692000	0.357000	0.290000	0.405000	0.509000

linux

程序运行时间	程序 A（先行后列访问数组）			程序 B（先列后行访问数组）		
	静态区	栈区	堆区	静态区	栈区	堆区
M=10 N=10000000	0.271502	0.310512	0.324184	0.180571	0.174441	0.482762
M=10000 N=10000	0.321001	0.295889	0.295919	0.982774	0.847727	1.415286
M=10000000 N=10	0.308298	0.197699	0.214123	0.255689	0.215537	0.378014

影响执行时间的因素：

- **局部数据块大小：**

为了更好地利用程序访问地空间局部性，通常把当前访问单元及附近单元作为一个贮存块一起调入 cache，理论上主存块越大，命中机会越大；但是如果过大则会造成读一个块的时间变多，缺失损失变大，所以应该选择适中的主存块大小。

- **数组访问顺序：**

因为数组在存储器中是按行存储，程序 A 对数组的访问顺序（先行后列）与存放顺序是一致的，所以 A 的空间局部性较好，cache 命中率较高，访存时间较短。但是当 M 和 N 相差较大时，二维数组接近一维数组，先行或者先列区别不明显，出现表中程序 B 比程序 A 时间短的现象。

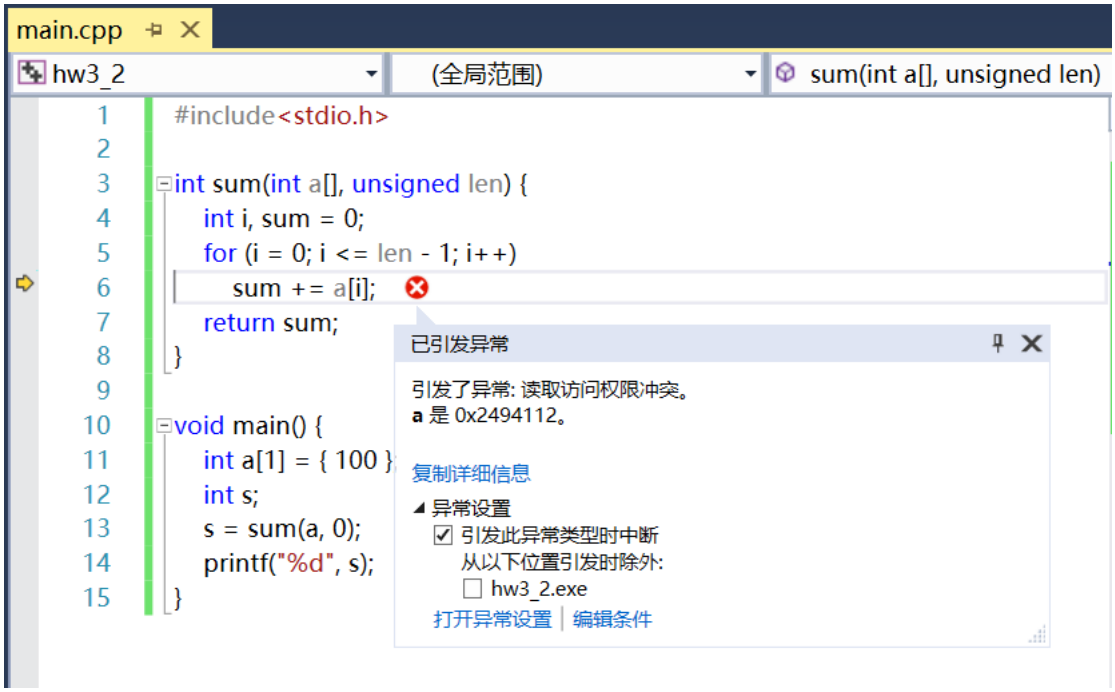
实验（二）存储保护

1. 使用 len=0 调用 sum 函数，分别给出在 Windows 和 Linux 系统中的执行结果。

windows 程序代码：

```
1   #include<stdio.h>
2
3   int sum(int a[], unsigned len) {
4       int i, sum = 0;
5       for (i = 0; i <= len - 1; i++)
6           sum += a[i];
7       return sum;
8   }
9
10  void main() {
11      int a[1] = { 100 };
12      int s;
13      s = sum(a, 0);
14      printf("%d", s);
15  }
```

windows 运行结果：



linux 程序代码：

```
#include<stdio.h>

int sum(int a[],unsigned len){
        int i,sum=0;
        for(i=0;i<=len-1;i++)
                sum+=a[i];
        return sum;
}

void main(){
        int a[1]={100};
        int s;
        s=sum(a,0);
        printf("%d",s);
}
```

linux 运行结果：

```
qjw@qjw-VirtualBox:~/hw3$ gcc main.c -o main
qjw@qjw-VirtualBox:~/hw3$ ./main
段错误（核心已转储）
```

2. 在 Linux 系统中，用 gdb 调试工具，设置正确的断点、显示通用寄存器的内容等手段，确定发生异常的指令，并指出发生的是什么异常，以及发生访问违例的存储单元地址。

使用 gdb 调试工具，直接运行查看

```
(gdb) run
Starting program: /home/qjw/hw3/main2

Program received signal SIGSEGV, Segmentation fault.
0x000000000040055c in sum (a=0x7fffffffdf50, len=0) at main.c:6
6                       sum+=a[i];
```

收到操作系统的 SIGSEGV 信号，意味着我们试图去访问一段非法的内存，后面提示异常发生在 main.c 的第 6 行，eip 指向 0x40055c。我们通过 objdump 指令得到反汇编代码

```
000000000040052d <sum>:
  40052d:       55                      push   %rbp
  40052e:       48 89 e5                mov    %rsp,%rbp
  400531:       48 89 7d e8             mov    %rdi,-0x18(%rbp)//a[]
  400535:       89 75 e4                mov    %esi,-0x1c(%rbp)//len
  400538:       c7 45 fc 00 00 00 00    movl   $0x0,-0x4(%rbp)//sum
  40053f:       c7 45 f8 00 00 00 00    movl   $0x0,-0x8(%rbp)//i
  400546:       eb 1d                   jmp    400565 <sum+0x38>
  400548:       8b 45 f8                mov    -0x8(%rbp),%eax
  40054b:       48 98                   cltq
  40054d:       48 8d 14 85 00 00 00    lea    0x0(,%rax,4),%rdx//i*4
  400554:       00
  400555:       48 8b 45 e8             mov    -0x18(%rbp),%rax//a
  400559:       48 01 d0                add    %rdx,%rax//%rax存放的是a[i]的地址（&a[i]=a+i*4）
  40055c:       8b 00                   mov    (%rax),%eax//把a[i]的值放入eax
  40055e:       01 45 fc                add    %eax,-0x4(%rbp)//sum+=a[i]
  400561:       83 45 f8 01             addl   $0x1,-0x8(%rbp)//i++
  400565:       8b 45 f8                mov    -0x8(%rbp),%eax
  400568:       8b 55 e4                mov    -0x1c(%rbp),%edx
  40056b:       83 ea 01                sub    $0x1,%edx//len-1
  40056e:       39 d0                   cmp    %edx,%eax//比较i和len-1
  400570:       76 d6                   jbe    400548 <sum+0x1b>//如果<=跳转回去
  400572:       8b 45 fc                mov    -0x4(%rbp),%eax
  400575:       5d                      pop    %rbp
  400576:       c3                      retq
```

发生异常的指令是：0x40055c: mov (%rax),%eax，即把%rax 中存储地址存储的数据传入%eax 时发生异常，联系之前第二章学习数据的机器级表示时的分析，len 的定义是 unsigned 类型，所以 i 和 len-1 是按照无符号数比较的，当 len=0 时，len-1 表示为 32 个 1，任何一个无符号数都要比 32 个 1 小，所以循环条件一直满足，循环不断执行，最终导致访问越界而发生存储器访问异常。我们可以查看存储 a[i]地址的%rax 的值，a[i]，以及此时的 i 的值

```
(gdb) info register rax
rax            0x7fffffff000    140737488351232
```

```
(gdb) x/s 0x7fffffff000
0x7fffffff000: <error: Cannot access memory at address 0x7fffffff000>
```

```
(gdb) print i
$1 = 1068
(gdb) print a[i]
Cannot access memory at address 0x7fffffff000
```

3. 从 CPU 检测到异常到屏幕中出现“Segment fault”的整个过程中，CPU 和操作系统各做了哪些事情?

CPU 产生一个访问异常，进程立即被暂停，然后操作系统注册在终端描述表中的异常处理程序被执行，然后这个程序发现进程确实访问越界了，就会发送信号给出问题的进程，如果该进程没有注册信号处理函数，就调用默认的信号处理函数，大部分信号的默认处理函数都是中止这个进程。

四、实验体会

首先通过运行程序 A 和 B，比较两个 for 循环的执行时间长短，相差有时甚至达到 2-3 倍，体会到了程序访问局部性对带有 cache 的计算机系统性能的影响。实验一也练习了比 time()精度更高的计时函数 clock()。

实验二这个例子第一节课在体会系统思维的时候就提到了，第二章学习数据类型的时候为比较有符号数和无符号数的时候也进行了越界原因的分析，到第六章就是对访问异常的处理。学习了用 gdb 调试器找异常的方法。