Homework3

陶静怡 18307130264

实验（一） 基于 cache 的存储访问

1.数组 a 分配在静态区

```
#include<stdio.h>
#include<stdlib.h>
#include<time.h>
#define M 1000
#define N 1000


void assign_array_rows(){
  int i, j;
  static short a[M][N];
  clock_t start1, end1;
  start1=clock();
  for(i=0;i<M;i++){
    for(j=0;j<N;j++){
      a[i][j]=0;
    }
  }
  end1=clock();
  printf("row first Run time: %lfs\n", (double)(end1-start1)/CLOCKS_PER_SEC);
}


void assign_array_cols(){
  int i, j;
  static short a[M][N];
  clock_t start2, end2;
  start2=clock();
  for(j=0;j<N;j++){
    for(i=0;i<M;i++){
      a[i][j]=0;
    }
  }
  end2=clock();
  printf("column first Run time: %lfs\n", (double)(end2-start2)/CLOCKS_PER_SEC);
}

int main(){
  assign_array_rows();
  assign_array_cols();


  return 0;
}
```

（1）M=1000，N=1000

```
tjy@tjy-virtual-machine:~$ ./b
row first Run time: 0.004880s
column first Run time: 0.006916s
```

（2）M=10，N=1000000

```
tjy@tjy-virtual-machine:~$ ./b
row first Run time: 0.315378s
column first Run time: 0.297215s
```

（3）M=1000000，N=10

```
tjy@tjy-virtual-machine:~$ gcc -o b b.c
tjy@tjy-virtual-machine:~$ ./b
row first Run time: 0.575680s
column first Run time: 0.881414s
```

数组 a 是在静态区，同时是按行优先存放的，根据执行时间，行优先访问所用的时间短于列优先，按行访问的空间局部性更好，cache 的命中率更高。但是在 M=10，N=1000000 时，由于行数少，例如第一次循环取 a[0][0]-a[9][0]所在的不同的 10 个块冲突少的话，第二次循环命中率会提高，所以导致时间上甚至比行优先快一点。但是可以看到在 M=1000000，N=10

时两者的差距是非常大的，所以总体来说行优先的执行速度快

2.数组 a 分配在栈区

```c
#include<stdio.h>
#include<stdlib.h>
#include<time.h>
#define M 1000000
#define N 10


void assign_array_rows(){
  int i, j;
  short a[M][N];
  clock_t start1, end1;
  start1=clock();
  for(i=0;i<M;i++){
    for(j=0;j<N;j++){
      a[i][j]=0;
    }
  }
  end1=clock();
  printf("row first Run time: %lfs\n", (double)(end1-start1)/CLOCKS_PER_SEC);
}


void assign_array_cols(){
  int i, j;
  short a[M][N];
  clock_t start2, end2;
  start2=clock();
  for(j=0;j<N;j++){
    for(i=0;i<M;i++){
      a[i][j]=0;
    }
  }
  end2=clock();
  printf("column first Run time: %lfs\n", (double)(end2-start2)/CLOCKS_PER_SEC);
}

int main(){
  assign_array_rows();
  assign_array_cols();


  return 0;
}
```

（1）M=1000000，N=10

运行以后发现出现段错误

```
tjy@tjy-virtual-machine:~$ ./b
Segmentation fault (core dumped)
```

（2）M=10，N=1000000

```
tjy@tjy-virtual-machine:~$ ./b
Segmentation fault (core dumped)
```

同样发现出错

（3）M=1000，N=1000

```
tjy@tjy-virtual-machine:~$ ./b
row first Run time: 0.005220s
column first Run time: 0.008588s
```

根据 M=1000，N=1000 的结果可以看到在数组 a 分配在栈区时行优先访问所用的时间更短，在栈区中同样是行优先存放的，所以空间局部性好，cache 命中率更高

而 segmentfault 的原因我认为是局部变量过大，大于了系统给的栈的大小

所以修改了一下数据，发现可以运行

M=100000，N=10

```
tjy@tjy-virtual-machine:~$ ./b
row first Run time: 0.005189s
column first Run time: 0.005177s
```

M=10，N=100000

```
tjy@tjy-virtual-machine:~$ ./b
row first Run time: 0.003587s
column first Run time: 0.002589s
```

这时行优先所用的时间大于列优先，是由于行数和列数不同，不能很好体现行优先的空间局部性好

3.数组 a 分配在堆区

```c
#include<stdio.h>
#include<stdlib.h>
#include<time.h>
#define M 10
#define N 1000000


void assign_array_rows(){
  int i, j;
  short **a;
  clock_t start1, end1;
  a=(short**)malloc(sizeof(int*)*M);
  for(i=0;i<M;i++)
    a[i]=(short*)malloc(sizeof(int)*N);
  start1=clock();
  for(i=0;i<M;i++){
    for(j=0;j<N;j++){
      a[i][j]=0;
    }
  }
  end1=clock();
  printf("row first Run time: %lfs\n", (double)(end1-start1)/CLOCKS_PER_SEC);
  for(i=0;i<M;i++){
    free(a[i]);
  }
  free(a);
}


void assign_array_cols(){
  int i, j;
  short **a;
  clock_t start2, end2;
  a=(short**)malloc(sizeof(int*)*M);
  for(i=0;i<M;i++)
    a[i]=(short*)malloc(sizeof(int)*N);
  start2=clock();
  for(j=0;j<N;j++){
    for(i=0;i<M;i++){
      a[i][j]=0;
    }
  }
  end2=clock();
  printf("column first Run time: %lfs\n", (double)(end2-start2)/CLOCKS_PER_SEC);
  for(i=0;i<M;i++){
    free(a[i]);
  }
  free(a);
}

int main(){
  assign_array_rows();
  assign_array_cols();
  return 0;
}
```

（1）M=10，N=1000000

```
tjy@tjy-virtual-machine:~$ ./a
row first Run time: 0.028499s
column first Run time: 0.025424s
```

（2）M=1000000，N=10

```
tjy@tjy-virtual-machine:~$ ./a
row first Run time: 0.017016s
column first Run time: 0.066522s
```

（3）M=1000，N=1000

```
tjy@tjy-virtual-machine:~$ ./a
row first Run time: 0.001699s
column first Run time: 0.006389s
```

可以看到总体上是行优先访问所用时间短

局部数据块大可以更好利用空间局部性，一次能把更多数据放入 cache 中，从而提高命中率。但是如果块太大，读取的时间就比较长，同时会让 cache 行数变少，替换的频率增加。

数组访问顺序：因为数组在存储器中是行优先，所以总体上行优先访问执行时间短，因为空间局部性好，cache 命中率高，所用的访存时间少。

实验（二） 存储保护

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

Linux 中无法运行，如图存储器访问异常

```
tjy@tjy-virtual-machine:~$ ./c
Segmentation fault (core dumped)
```

用 gdb 调试，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。

```
0804846b <sum>:
 804846b:       55                      push   %ebp
 804846c:       89 e5                   mov    %esp,%ebp
 804846e:       83 ec 10                sub    $0x10,%esp
 8048471:       c7 45 fc 00 00 00 00    movl   $0x0,-0x4(%ebp)
 8048478:       c7 45 f8 00 00 00 00    movl   $0x0,-0x8(%ebp)
 804847f:       eb 18                   jmp    8048499 <sum+0x2e>
 8048481:       8b 45 f8                mov    -0x8(%ebp),%eax
 8048484:       8d 14 85 00 00 00 00    lea    0x0(,%eax,4),%edx
 804848b:       8b 45 08                mov    0x8(%ebp),%eax
 804848e:       01 d0                   add    %edx,%eax
 8048490:       8b 00                   mov    (%eax),%eax
 8048492:       01 45 fc                add    %eax,-0x4(%ebp)
 8048495:       83 45 f8 01             addl   $0x1,-0x8(%ebp)
 8048499:       8b 45 0c                mov    0xc(%ebp),%eax
 804849c:       8d 50 ff                lea    -0x1(%eax),%edx
 804849f:       8b 45 f8                mov    -0x8(%ebp),%eax
 80484a2:       39 c2                   cmp    %eax,%edx
 80484a4:       73 db                   jae    8048481 <sum+0x16>
 80484a6:       8b 45 fc                mov    -0x4(%ebp),%eax
 80484a9:       c9                      leave
 80484aa:       c3                      ret
```

在 sum 函数处设置断点

```
(gdb) break sum
Breakpoint 1 at 0x8048471: file c.c, line 4.
```

发现满足了 i<=len-1 继续执行了

```
(gdb) si
0x0804847f      5               for(i=0;i<=len-1;i++){
(gdb) si
0x08048499      5               for(i=0;i<=len-1;i++){
(gdb) si
0x0804849c      5               for(i=0;i<=len-1;i++){
(gdb) si 4
6                       sum+=a[i];
```

Eax 中存放 i，edx 中存放 len-1 的值

```
(gdb) si
0x0804849f      5               for(i=0;i<=len-1;i++){
(gdb) info r
eax            0x0      0
ecx            0xffffcf30       -12496
edx            0xffffffff       -1
ebx            0x0      0
```

执行 com　%eax，%edx

即把 0xffffffff-0x0，sub=1， CF=0，ZF=0

所以满足 jae（0x80484a4 处）的转移条件：CF=0/ZF=1

既这里当成无符号数比较大小，0xffffffff>0x0

```
(gdb) si
0x080484a2      5               for(i=0;i<=len-1;i++){
```

所以跳转执行 sum=+a[i]

```
(gdb) si
0x080484a4      5               for(i=0;i<=len-1;i++){
(gdb) si
5                       sum+=a[i];
(gdb) si
0x08048484      6                       sum+=a[i];
```

```
(gdb) info r
eax            0x0      0
ecx            0xffffcf30       -12496
edx            0xffffcf54       -12460
ebx            0x0      0
esp            0xffffcf00       0xffffcf00
ebp            0xffffcf18       0xffffcf18
```

访问到了内核区

从 CPU 检测到异常到屏幕中出现“Segment fault”的整个过程中，首先 CPU 产生一个访问异常，进程立即被暂停，然后执行 OS 页故障处理程序，然后这个程序发现进程确实访问越界了，就会发送 SIGSEGV 信号给出客户进程

No.

Date . .

8. (1) ∵ 主存地址空间大小为 $1GB = 2^{30}B$.

∴ 地址 30 位.

∵ cache 有 $64kB / 128B = 512$ 行. $512 = 2^9$

∴ cache 行号 9 位.

∵ 每个主存块 128 B. $128 = 2^7$

∴ 块内地址 7 位.

∴ $30 - 9 - 7 = 14$.

∴ 高 14 位为主存标记.

中间 9 位为 cache 索引.

低 7 位为块内地址.

(2) ∵ 直接映射且直写. ∴ 没有控制位和修改位.

∴ 1 位有效位. 14 位标记位. 和 128 B 数据.

$512 \times (1 + 14 + 128 \times 8) = 531968$ bit

No.

Date

12. (1). ∵ x和y数组都是按存放顺序访问. ∴ 空间局部性好.

∵ 只访问一次 ∴ 无时间局部性

∵ 命中率高低与映射方式, 块大小, cache容量都有关

∴ 不能推断出命中率高低.

(2). cache共2行.

∵ 一个块16B. ∴ 4个元素占一个块.

数组x中 x[0]~x[3]在第4块

x[4]~x[7]在第5块.

数组y中. y[0]~y[3]在第6块

y[4]~y[7]在第7块.

∴ x[0]~x[3]和y[0]~y[3]映射到cache第1行

x[4]~x[7]和y[4]~y[7] ---2

即x[i]和y[i]互相淘汰. ∴ 每次都不命中. 命中率为0.

No.

Date . .

(3). cache 4行，共2组，每组2行。

一块 8B，∴ 2个元素占一个块。

x[0] ~ x[1] 第8块。

x[2] ~ x[3] 9

x[4] ~ x[5] 10

x[6] ~ x[7] 11.

y[0] ~ y[1] 12

y[2] ~ y[3] 13

y[4] ~ y[5] 14

y[6] ~ y[7] 15.

∵ 一组两行，∴ x[i]和y[i]可放入同一组中，每次一块放入2个元素，第2个命中。

∴ 命中率 50%.

(4) ∵ x数组占 48B.

4个元素占一个块。

∴ x[0] ~ x[3] 第4块。

x[4] ~ x[7] 5

x[8] ~ x[11] 6.

y[0] ~ y[3] 7

y[4] ~ y[7] 8.

∴ x[i]和y[i]不映射到同一行。

每调入一块，后3个元素命中。命中率 75%