（p212 8）第一题

读如下代码，写出其执行结果，并描述其中的父子进程关系。

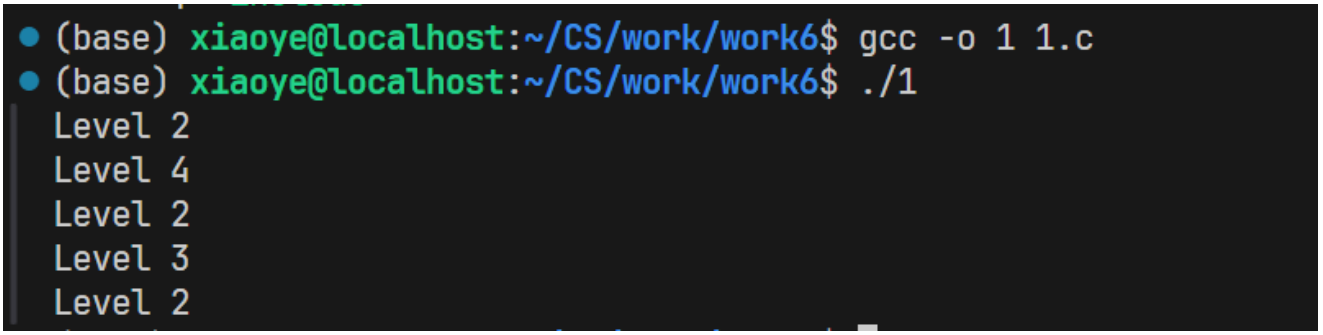
```
# include <stdio.h>

# include <sys/types.h>

# include <unistd.h>

int main(){

    pid_t pid1,pid2,pid3;

    pid1=0,pid2=0,pid3=0;

    pid1=fork();

    if (pid1==0)

    {

        pid2=fork();

        pid3=fork();

    }else{

        pid3=fork();

        if(pid3==0){

            pid2=fork();

        }

        if((pid1==0)&&(pid2==0)){

            printf("Level 1\n");

        }
```

```
        if(pid1!=0){
            printf("Level 2\n");
        }
        if(pid2!=0){
            printf("Level 3\n");
        }
        if(pid3!=0){
            printf("Level 4\n");
        }
        return 0;
    }
}
```

执行结果(先父后子的情况)

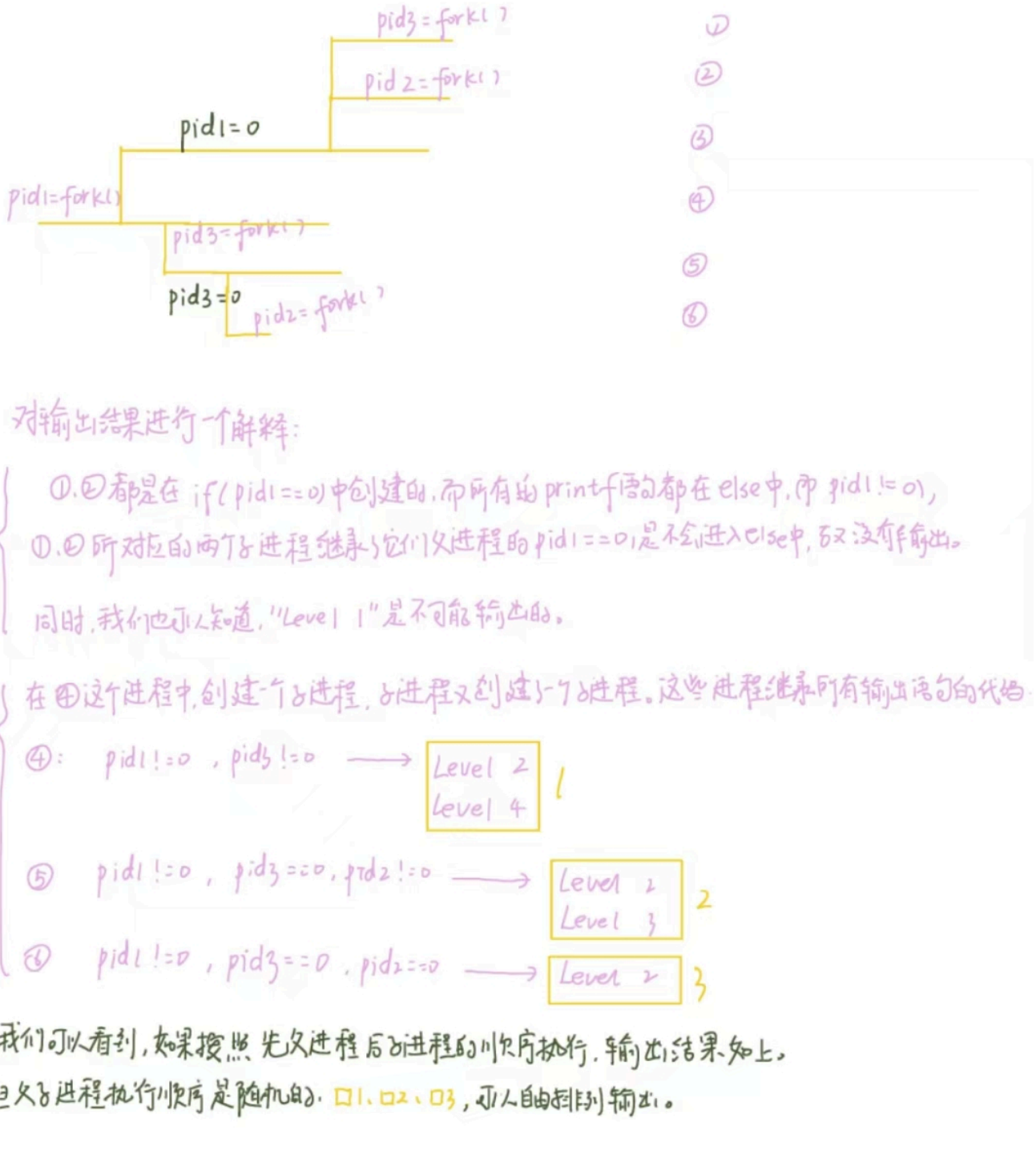
Level 2
Level 4
Level 2
Level 3
Level 2

验证：



父子进程关系

父进程创建子进程1（返回值保存在pid1)。在子进程1中创建两个平行的子进程2（返回值保存在pid2)和3（返回值保存在pid3)。
在父进程中又创建一个子进程（返回值保存在pid3)，在这个子进程中再次创建一个子进程（返回值保存在pid2)。



（p212 9）第二题

读如下代码，写出其执行结果，并解释其中wait函数的作用。

```
#include <sys/types.h>

#include <stdio.h>

#include <unistd.h>

#define SIZE 5

int nums[SIZE]={0,1,2,3,4};

int main(){

    int i;

    pid_t pid;

    pid=fork();

    if(pid==0){

        for(i=0;i<SIZE;i++){

            nums[i]*= -i;

            printf("CHILD: %d ",nums[i]);

        }



    }

    else if(pid>0){

        wait();

        for(i=0;i<SIZE;i++){

            printf("PARENT: %d ",nums[i]);

        }

    }
```

```
    return 0;

}
```

执行结果

CHILD: 0 CHILD: -1 CHILD: -4 CHILD: -9 CHILD: -16 PARENT: 0 PARENT: 1 PARENT: 2 PARENT: 3 PARENT: 4

wait函数的作用

`wait()` 函数的作用是 **让父进程阻塞并等待子进程结束**，同时回收子进程的资源。（它会等待任意一个子进程结束哦）
假如父进程先运行，调用wait()时，它会进入阻塞状态，直到子进程终止。这确保了父进程不会在子进程完成前继续执行后续代码。
子进程终止后，其占有的系统资源需要被父进程回收。wait()会自动清理这些资源，避免子进程成为僵尸进程。

（p214 13）第三题

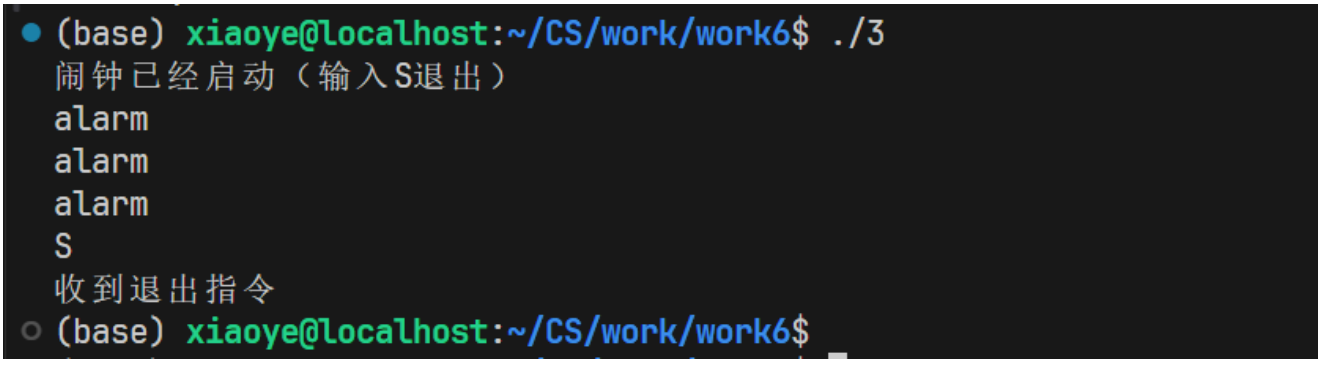
请使用信号写一个简单的闹钟程序：在当前时间的3s之后产SIGALRM信号，并且每隔3s，就进行alarm提示，直至用户输入“S”时为止。

```
#include <stdio.h>

#include <signal.h>

#include <unistd.h>

#include <fcntl.h>

#include <stdlib.h>



//信号处理函数

void alarm_handler(int signum) {
```

```
    printf("alarm\n");
    alarm(3);  //重置计时器
}


int main() {
    //信号处理
    if (signal(SIGALRM,alarm_handler) == -1) {
        perror("sigaction");
        exit(EXIT_FAILURE);
    }

    //先等待3s
    alarm(3);

    printf("闹钟已经启动（输入S退出）\n");
    //非阻塞式输入检测
    while (1) {
        //设置标准输入为非阻塞模式
        int flags = fcntl(STDIN_FILENO, F_GETFL, 0);
        fcntl(STDIN_FILENO, F_SETFL, flags | O_NONBLOCK);
```

```
        char c;
        if (read(STDIN_FILENO, &c, 1) == 1) {
            if (c == 'S') {
                printf("收到退出指令\n");
                break;
            }
        }
        usleep(100000);  //降低CPU占有率
    }

    return 0;
}
```

运行结果：



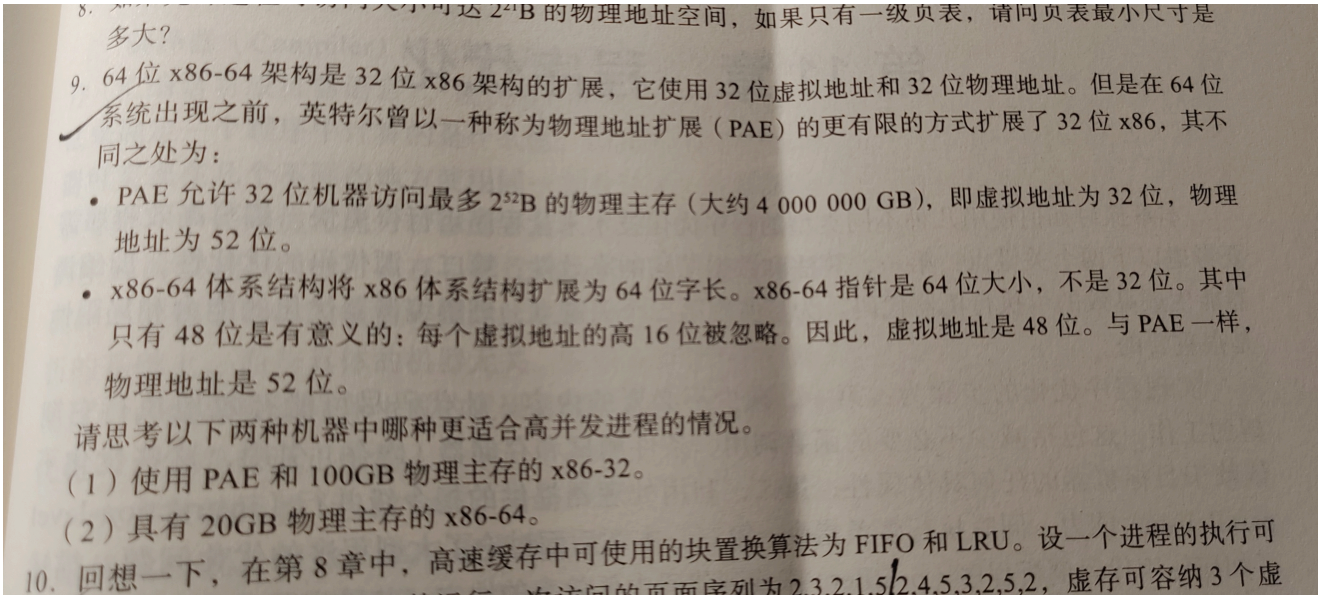
> **Note**
>
> 课本第208页介绍了alarm函数。这是一个专门为SLGALRM信号设定的函数。在指定的参数seconds设定的秒数后，将由内核向进程自身发送SIGALRM信号，即闹钟。

（p236 6）第四题

一个32位系统是一个包含若干虚页的虚存，页大小是2KB。如果虚拟地址为0x0030f40，请推断其包含多少虚页，以及该地址所在的虚页页号和页内偏移是多少？

- 因为这是32位系统，所以我们可以把地址分成32位。又因为页的大小是2KB，那么页内偏移应该占11位。剩下的21位就是虚页页号。
- 因为有21位来表示虚页页号，所以推断有2^{21}个虚页。
- 0x0030f40：0b 0000 0000 0000 0011 0000 1111 0100 0000
- 后11位：111 0100 0000
- 前21位：0000 0000 0000 0011 0000 1
- 那么该地址所在的虚页页号为： 97（0x61）
- 页内偏移为1856（0x740）

（p236 9）第五题



多大？

9. 64 位 x86-64 架构是 32 位 x86 架构的扩展，它使用 32 位虚拟地址和 32 位物理地址。但是在 64 位系统出现之前，英特尔曾以一种称为物理地址扩展（PAE）的更有限的方式扩展了 32 位 x86，其不同之处为：
   - PAE 允许 32 位机器访问最多 2^{52}B 的物理主存（大约 4 000 000 GB），即虚拟地址为 32 位，物理地址为 52 位。
   - x86-64 体系结构将 x86 体系结构扩展为 64 位字长。x86-64 指针是 64 位大小，不是 32 位。其中只有 48 位是有意义的：每个虚拟地址的高 16 位被忽略。因此，虚拟地址是 48 位。与 PAE 一样，物理地址是 52 位。

请思考以下两种机器中哪种更适合高并发进程的情况。

（1）使用 PAE 和 100GB 物理主存的 x86-32。

（2）具有 20GB 物理主存的 x86-64。

10. 回想一下，在第 8 章中，高速缓存中可使用的块置换算法为 FIFO 和 LRU。设一个进程的执行可

> **高并发进程的特点：**
>
> 通常需要创建大量进程或线程
> 每个进程可能需要独立的内存空间
> 需要频繁的上下文切换
> 可能涉及大量的内存映射和文件操作

更适合高并发进程的选择是：具有20GB物理内存的x86-64系统
原因如下：

- 从上下文切换开销的角度看：（86-64的寻址空间为2^{48},大于x86-32的寻址空间2^{32}）x86-64系统：尽管物理内存较小(20GB)，但虚拟地址空间充足，上下文切换时不需要频繁处理物理内存限制；x86-32+PAE系统：有限的虚拟地址空间导致频繁的物理内存交换，增加上下文切换开销。
- 从内存管理效率看：- x86-64的页表结构更高效，可以更好地组织和管理大量内存。**即使物理内存只有20GB，也能通过虚拟地址空间有效利用**。
- 从扩展性与未来兼容性看：x86-64架构设计面向未来，支持更大的内存扩展。现代软件生态系统主要针对64位架构优化。

总结：
尽管x86-32+PAE系统拥有100GB的物理内存，但其32位虚拟地址空间的限制(4GB/进程)严重制约了高并发环境下的性能。相比之下，x86-64系统提供的48位虚拟地址空间(~256TB)为每个进程提供了充足的独立内存空间，使系统能够更高效地管理和调度大量并发进程，减少内存争用和上下文切换开销，从而整体上提供更好的性能和可扩展性。

（p237 10）第六题

回想一下，在第8章中，高速缓存中可使用的块置换算法为FIFO和LRU。设一个进程的执行可能至少需要访问5页，其运行一次访问的页面序列为2,3,2,1,5,2,4,5,3,2,5,2，虚存可容纳3个虚页，请比较采用FIFO页面替换算法和采用LRU页面替换算法的缺页次数。

> **FIFO页面替换算法**
>
> 最早进入内存的页面最先被替换。算法维护一个队列结构，新页面加入时插入队尾，替换时从队头移除最老的页面。

> **LRU页面替换算法**
>
> 替换最近最久未被访问的页面。基于程序局部性原理，认为过去少用的页面未来被访问的概率更低。

FIFO

页面访问序列	虚存	缺页（Y/N)
2	{2}	Y
3	{2,3}	Y
2	{2,3}	N
1	{2,3,1}	Y
5	{5,3,1}	Y

//最早进入的是页2
//5替换2 以下替换同理

页面访问序列	虚存	缺页（Y/N)
2	{5,2,1}	Y
4	{5,2,4}	Y
5	{5,2,4}	N

页面访问序列	虚存	缺页（Y/N)
3	{3,2,4}	Y
2	{3,2,4}	N
5	{3,5,4}	Y
2	{3,5,2}	Y

缺页次数：9次。

LRU

页面访问序列	虚存	缺页（Y/N)
2	{2}	Y
3	{2,3}	Y
2	{2,3}	N
1	{2,3,1}	Y
5	{2,5,1}	Y

//最近最久未被访问的页面是页3
//5替换3 以下替换同理

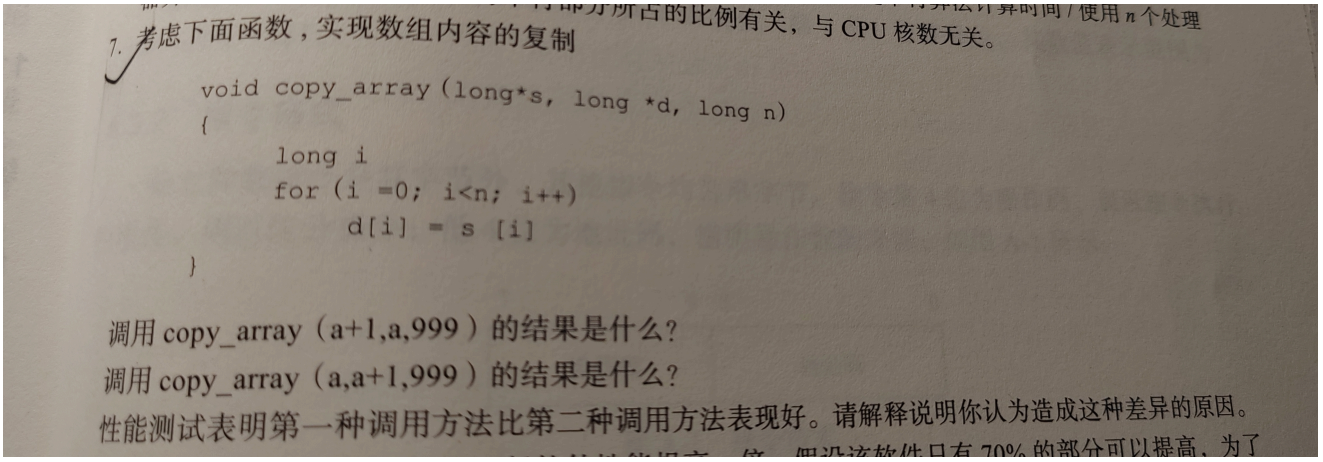
页面访问序列	虚存	缺页（Y/N)
2	{2,5,1}	N
4	{2,5,4}	Y
5	{2,5,4}	N
3	{3,5,4}	Y
2	{3,5,2}	Y
5	{3,5,2}	N
2	{3,5,2}	N

缺页次数：7次。

比较

- 采用LRU页面替换算法的缺页次数比采用FIFO页面替换算法的缺页次数少2次。
- 究其原因，FIFO未考虑页面使用频率，频繁替换活跃页面；LRU优先保留近期活跃页面，更适合具有时间局部性的访问模式。

（p271 7）第七题



7. 考虑下面函数，实现数组内容的复制

```
void copy_array(long*s, long *d, long n)
{
    long i
    for (i =0; i<n; i++)
        d[i] = s [i]
}
```

调用 copy_array（a+1,a,999）的结果是什么?

调用 copy_array（a,a+1,999）的结果是什么?

性能测试表明第一种调用方法比第二种调用方法表现好。请解释说明你认为造成这种差异的原因。

```
//s: 源数组指针
//d: 目标数组指针
//n: 要复制的元素个数
void copy_array(long *s, long *d, long n)
{
    long i;
    for (i = 0; i < n; i++)
        d[i] = s[i];
}
```

调用1: copy_array(a+1, a, 999)

在这个调用中：

- 源数组指针是 `a+1`，从数组 `a` 的第二个元素开始读取
- 目标数组指针是 `a`，从数组 `a` 的第一个元素开始写入
- 复制 999 个元素

结果：这将把 `a[1]` 到 `a[999]` 的内容复制到 `a[0]` 到 `a[998]`，相当于将整个数组向左移动一位。原数组的第一个元素被丢弃，最后一个元素保持不变。

调用2: copy_array(a, a+1, 999)

在这个调用中：

- 源数组指针是 `a`，从数组 `a` 的第一个元素开始读取
- 目标数组指针是 `a+1`，从数组 `a` 的第二个元素开始写入
- 复制 999 个元素

结果：这将把 `a[0]` 到 `a[998]` 的内容复制到 `a[1]` 到 `a[999]`，相当于将整个数组向右移动一个位置，相当于将整个数组向右移动一位。原数组的最后一个元素被丢弃，第一个元素保

持不变。

性能差异原因

第一种调用方法比第二种调用方法表现更好的主要原因有：

1. 内存访问模式

- **第一种调用**: 从高地址向低地址读(`a+1`), 然后写入连续的低地址(`a`)
- **第二种调用**: 从低地址向高地址读(`a`), 然后写入更高的地址(`a+1`)
  低地址读取时，预取器可能预测后续更高地址的访问。
  写入高地址时，若地址跳跃较小（如步长1），新数据可能覆盖之前预取的缓存组。

2. 缓存预取机制

现代CPU会根据访问模式预测并预取数据：

- 第一种调用遵循了CPU最擅长的预测模式：顺序读取然后向前写入（如：访问 `a[i]` 后预取 `a[i+1]` 所在的缓存行）
- 第二种调用会干扰缓存预取，因为写入位置在读取位置之后（写入方向与预取方向相反，导致预取数据无法被及时使用）。

3. 写缓冲区效率

- 第一种调用：写入操作发生在读取操作之后，不会干扰缓存预取
- 第二种调用：写入位置在后续读取位置之前，可能导致不必要的等待

4. 数据依赖问题

- 第一种调用方式避免了潜在的数据依赖问题
- 第二种调用方式可能会导致意外的内存覆盖，特别是在处理重叠区域时。
- 比如说：

```
arr = [1, 2, 3, 4, 5]
//我们希望通过复制操作将前4个元素，移动到后4个位置
copy_array(arr, arr+1, 4);
arr[1]=arr[0];//1
arr[2]=arr[1];//1
arr[3]=arr[2];//1
arr[4]=arr[3];//1
// 预期结果应为[1, 1, 2, 3, 4]，但实际得到全1的错误结果。
```