C语言内存

一、程序如何运行

1、内存当中只是存储指令和数据的，不能直接运行。想要计算，还需要寄存器（32位、64位）。虽然内存读取足够快了，但是还需要设置一个缓存、

2、CPU指令集

int a = 0X14,b = 0XAE,c;

c = a + b;

mov ptr[a],0X14

mov ptr[b],0XAE

mov eax,ptr[a]

add eax,ptr[b]

mov ptr[c],eax

3、虚拟地址：有两个全局变量：a,b，他们的内存地址在链接时就已经确定好了，但是如果程序占用了这块内存地址怎么办？出现了一个虚拟地址的概念，每一次通过映射机制映射的物理地址其实都不一样。

二、编译模式

1、32位编译模式

2^32 = 0X100000000 Bytes，即4GB，有效虚拟地址范围是 0 ~ 0XFFFFFFFF。

也就是说，对于32位的编译模式，不管实际物理内存有多大，程序能够访问的有效虚拟地址空间的范围就是0 ~ 0XFFFFFFFF，也即虚拟地址空间的大小是 4GB。换句话说，程序能够使用的最大内存为 4GB，跟物理内存没有关系。

如果程序需要的内存大于物理内存，或者内存中剩余的空间不足以容纳当前程序，那么操作系统会将内存中暂时用不到的一部分数据写入到磁盘，等需要的时候再读取回来。而我们的程序只管使用 4GB 的内存，不用关心硬件资源够不够。

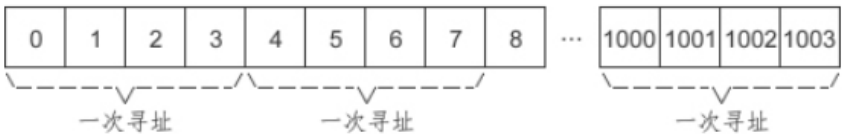
如果物理内存大于 4GB，例如目前很多PC机都配备了8GB的内存，那么程序也无能为力，它只能够使用其中的 4GB。

2、64位编译模式

2 ^ 64太大了，物理内存和CPU寻址都达不到，而且这么大的范围会增添地址转换的成本，所以会对其进行限制，只能使用低48位（6个字节），总的虚拟地址空间大小为2 ^ 64

三、C语言内存对齐

1、CPU通过地址总线来访问内存，32位机器可以一次处理4个字节的数据，64位机器可以一次处理8个字节的数据。32位机器只会对4的倍数的地址进行寻址：0 4 8 12……，而不会对1 3 5……



变量最好做到内存对齐：在一个步长里，而不是分散在两个步长里

2、结构体会内存对齐，变量也会内存对齐

int a;char m;int c

&a = DE3384

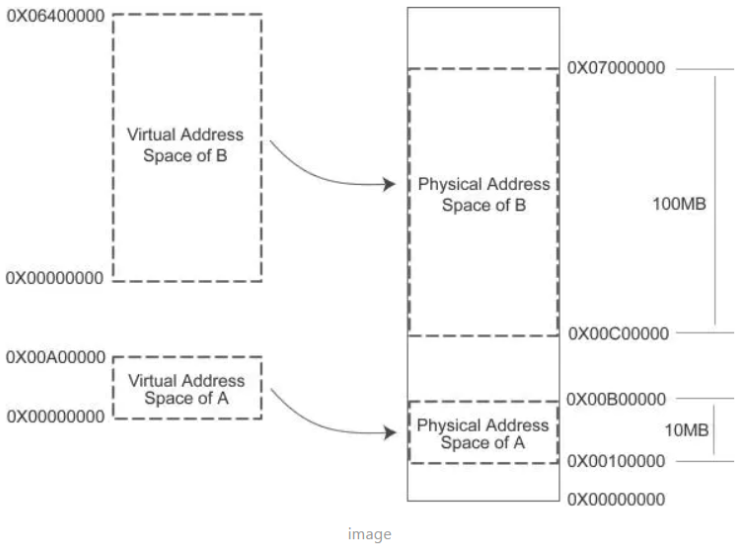
&m = DE338C

&c = DE3388

四、内存分页机制

1、映射

虚拟地址映射到物理地址上，不同的虚拟地址是隔离的



image

2、换入与换出

我们知道，当一个程序运行时，在某个时间段内，它只是频繁地用到了一小部分数据，也就是说，程序的很多数据其实在一个时间段内都不会被用到。

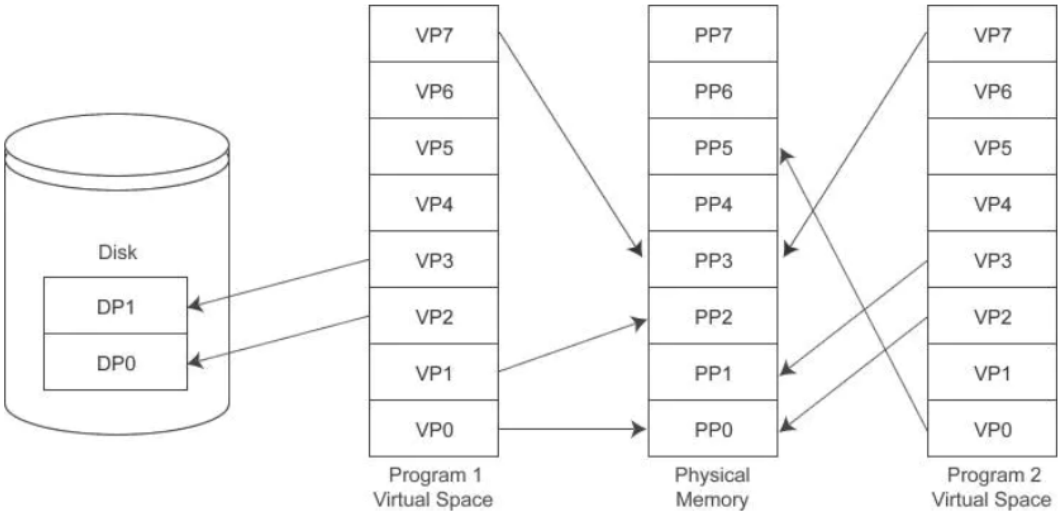
以整个程序为单位进行映射，不仅会将暂时用不到的数据从磁盘中读取到内存，也会将过多的数据一次性写入磁盘，这会严重降低程序的运行效率。

现代计算机都使用分页（Paging）的方式对虚拟地址空间和物理地址空间进行分割和映射，以减小换入换出的粒度，提高程序运行效率。

1）换入：当程序运行时，我们可以将需要的数据从磁盘置换到内存里

2）换出：当物理内存不够的时候，可以把物理页置换到磁盘里

3、分页



1）程序1的VP7 VP1 VP0缓存到了物理内存里，VP3 VP2还在磁盘里，VP6 VP5 VP4没有创建不占用磁盘空间

2）程序2的VP7和程序1的VP7都映射到了同一块物理内存页，他们实现了内存的共享

3）进程如果需要程序1的VP3 VP2，那么会发生页错误，然后操作系统接管线程，将这两个页从磁盘中读出来，然后与物理内存建立映射

五、页表

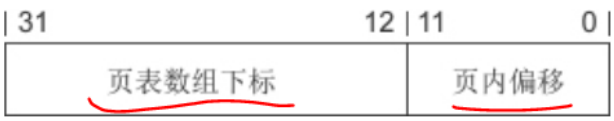
(13条消息) C语言内存篇 | 06-内存分页机制的实现（虚拟地址和物理地址的映射）_Systemcall驿站-CSDN博客_内存分页机制,完成虚拟地址的映射

1、一级页表

1）32位 - 虚拟地址空间4GB，一页为4KB，那么4GB / 4KB = 1M，页数为1M，可以开辟一个元素个数为1M的数组，数组每个元素的值为物理页面的编号，页表的一个元素大小为4个字节，页表大小4MB

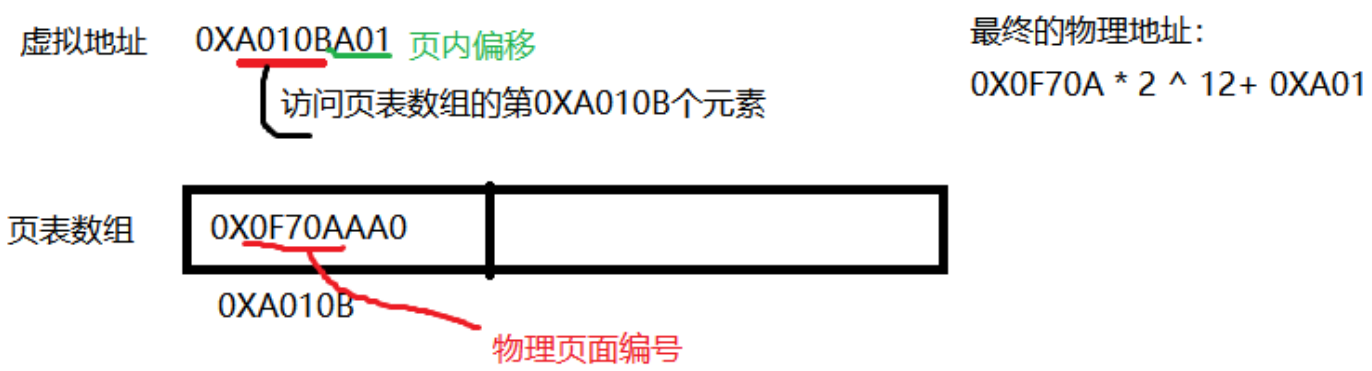
2）32位：高地址20位，2 ^ 20 = 物理页面的数量，低地址12位，2 ^ 12 = 4KB = 一页的大小

所以



3）物理页面编号为20位，但是一个页表数组元素大小位4个字节（32位），剩的12位表示当前页面的相关属性：是否有读写权限、是否已经分配物理内存、是否被换出到硬盘等

4）



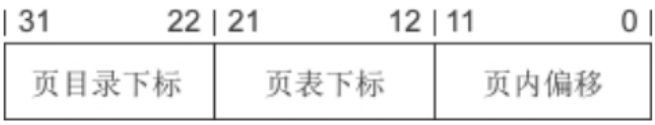
5）使用这种方案，不管程序占用多大的内存，都要为页表数组分配4M的内存空间（**页表数组也必须放在物理内存中**），因为虚拟地址空间中的高1G或2G是被系统占用的，必须保证较大的数组下标有效。

2、二级页表

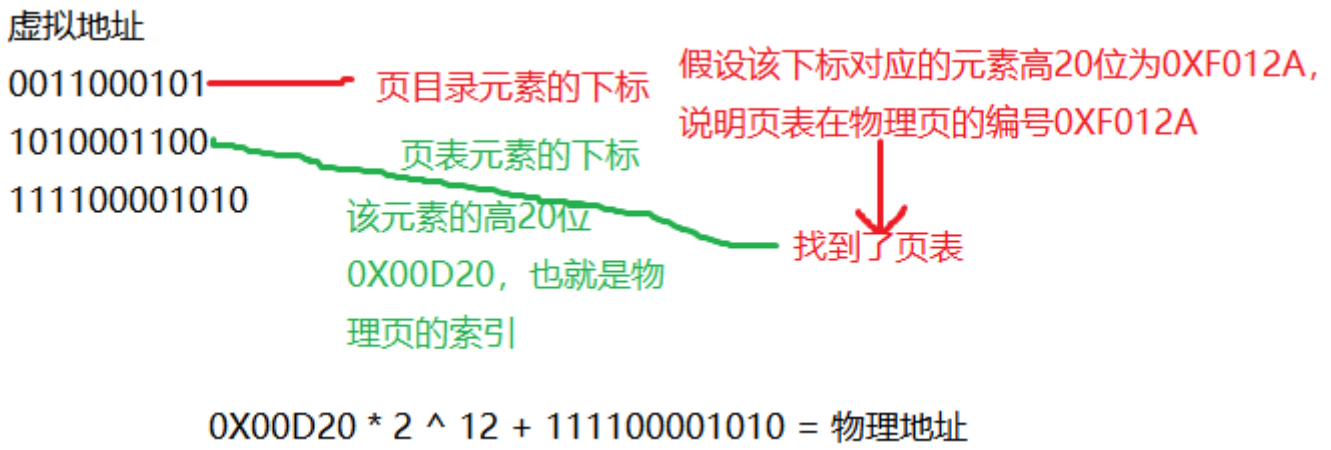
1）2 ^ 20 = 2 ^ 10 * 2 ^ 10个元素，所以可以拆分为二级页表，可以拆分出2 ^ 10 = 1024个页表，每一个页表2 ^ 10 = 1024 个元素，每个元素4个字节，一个页表4KB。这1024个小页表，可以存储在不同的物理页上。管理1024个小页表的，叫页目录，每一个元素对应一个小页表所在物理内存的编号。

2）只要使用一个指针来记住页目录的地址即可，等到进行地址转换时，可以根据这个指针找到页目录，再根据页目录找到页表，最后找到物理地址，前后共经过3次间接转换

3）



4）



5）采用这样的两级页表的一个明显优点是，如果程序占用的内存较少，分散的小页表的个数就会远远少于1024个，只会占用很少的一部分存储空间（远远小于4M）。

在极少数的情况下，程序占用的内存非常大，布满了4G的虚拟地址空间，这样小页表的数量可能接近甚至等于1024，再加上页目录占用的存储空间，总共是 1024 * 4KB + 4KB = 4MB+4KB，比上面使用一级页表的方案仅仅多出4KB的内存。这是可以容忍的，因为很少出现如此极端的情况。

也就是说，使用两级页表后，页表占用的内存空间不固定，它和程序本身占用的内存空间成正比，从整体上来看，会比使用一级页表占用的内存少得多。

六、MMU

1、可以实现映射

1）虚拟地址到物理地址的映射，如果由操作系统来实现，是会消耗性能的：多次转换加计算

2）MMU负责将虚拟地址映射到物理地址上，为了不让它多次访问内存，可以将页目录和（常用，缓存空间是有限制的）页表缓存到MMU上，不常用的页表再从物理内存中去加载到缓存，缓存的命中率可以达到90% - 通过硬件来提高效率

3）构建页表：

MMU不会构建页表，构建页表是操作系统的任务

CR3寄存器 - 页目录的物理地址

在程序加载和程序运行时，会更新页表，并把页目录的地址放到CR3寄存器里，在MMU把页表加载到缓存里时，通过CR3找到页目录，再找到页表。

切换程序时，每个程序都有自己的页表，这个时候改变CR3里的值，就可以找到合适的页表

2、对内存权限的控制

页表数组元素的低12位：决定了要不要映射到物理内存里、有没有执行权限和访问权限

如果有权限，就执行，没有权限，产生异常 - 段错误

七、Linux下的C语言的内存模型

1、内核空间和用户空间

内核空间不能被直接访问

2、内存布局

C++

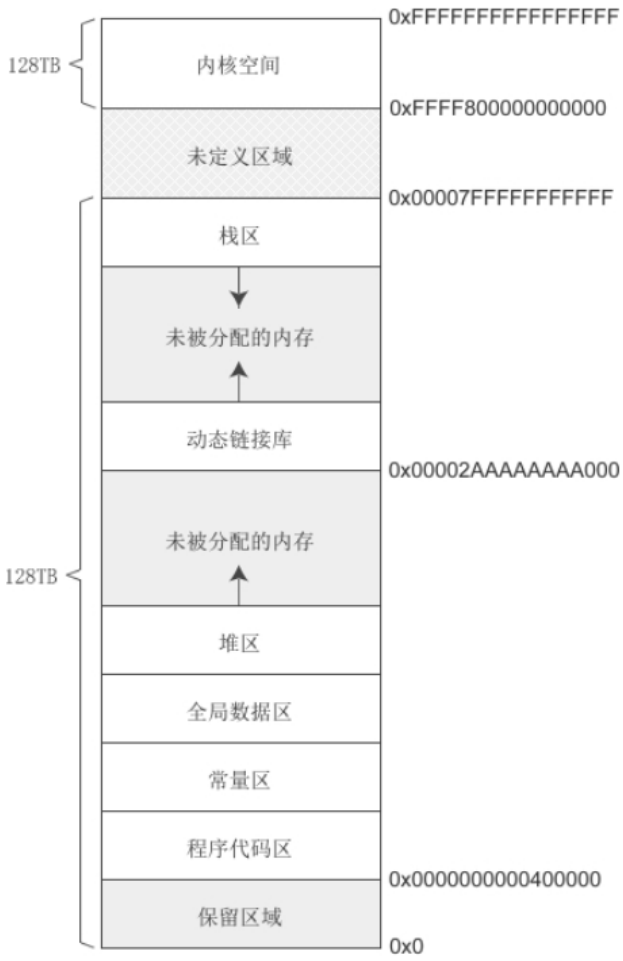
```
char *str1 = "abc";//字符串在常量区，str1在全局区
char *func() {
    char *str3 = "def";
    return str3;
}
int main() {
    char *str2 = "ghi";//str2在栈区
    char arr[20] = "jkl";//字符串和arr都在栈区
    char *str4 = func();
    printf("%s\n",str4);    //还会输出def，因为def在常量区，不会随着函数被销毁
    return 0;
}
```

全局变量在编译期就已经分配好了，默认为0

局部变量在函数调用时分配，初始值不确定

3、64位时

低128TB为用户空间，高128TB为内核空间



八、内核模式和用户模式

1、程序和进程

程序是存储在磁盘上的文件，把程序加载到内存里就是进程，所以一个程序可以创建很多个进程

2、内核模式和用户模式

1）内核空间存放的是操作系统内核代码和数据，是被所有程序共享的，在程序中修改内核空间中的数据是非常危险的行为，所以操作系统禁止用户程序直接访问内核空间。要想访问内核空间，必须借助操作系统提供的 API 函数（系统调用），让内核自己来访问。

2）内核模式：发生系统调用，会暂停用户代码，执行内核程序

用户模式：执行用户空间的代码时，其他程序不能访问，因为是私有的

3）内核主要是管理硬件，当进程想要进行与硬件相关的操作（输入输出、分配内存、相应鼠标等），需要进行系统调用，进入内核模式，执行完内核模式代码后，又回到了用户模式。

4）内核没有独立的地址空间（消耗巨大，切换进程会导致CPU里的数据失效、MMU页表缓存失效、寄存器进栈出栈），和进程的用户空间共享内存，这样效率高，只需要模式切换就可以了，仅仅需要寄存器进栈出栈

5）特权等级：内核空间 ring 0 - 可以直接访问所有资源，用户空间 ring 3

九、栈

ebp：栈底，esp：栈顶

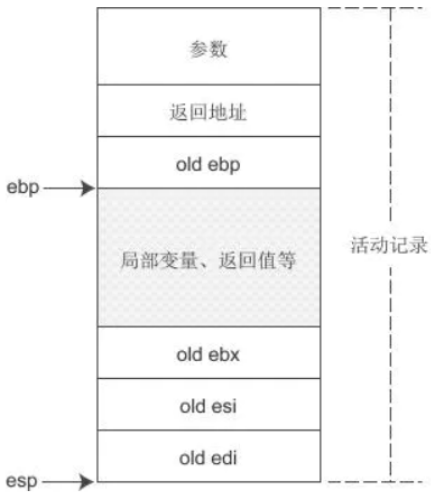
Linux GCC - 8M

1、一个函数在栈上到底是什么样的?

1）函数调用时的信息 - 栈帧

- 参数和局部变量
- 返回地址（该函数执行完毕之后，下一条语句的地址）
- 比较长的临时数据（如果比较短小 - 寄存器里）
- 寄存器的值，当函数调用结束时，恢复到之前的场景，继续执行上层函数

2）函数调用的实例



发生函数调用时的入栈顺序

- 实参 返回地址 ebp寄存器
- 分配一块足够大的内存，存储局部变量 返回值
- 寄存器的值

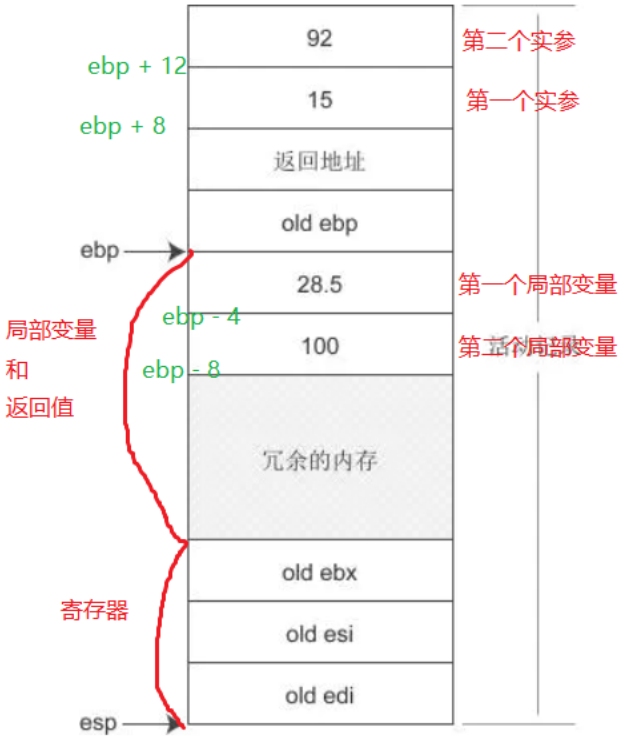
3）数据的定位

需要ebp来定位，因为esp总变

C++

```
void func(int a,int b) {
    float f = 28.5;
    int n = 100;
}
func(15,92);
```

假设变量之间挨着，第一个局部变量和old ebp挨着（实际上是有4字节的空白）



2、函数调用惯例

1）比如main函数调用B函数，B函数调用A函数，此时在调用函数时，必须要指明实参的入栈顺序，调用方和被调用方达成一致，是从左向右入栈，还是从右向左入栈

2）函数调用惯例可以决定入栈顺序 函数名在编译时的重命名方式 出栈是由调用方还是非调用方完成

3）默认是_cdecl，函数声明时int _cdecl max()，定义时int _cdecl max()

调用惯例	参数传递方式	参数出栈方式	名字修饰
cdecl	按照从右到左的顺序入栈	调用方	下划线+函数名， 如函数 max() 的修饰名为 _max
stdcall	按照从右到左的顺序入栈	函数本身 （被调用方）	下划线+函数名+@+参数的字节数， 如函数 int max(int m, int n) 的修饰名为 _max_@8
fastcall	将部分参数放入寄存器， 剩下的参数按照从右到左的顺序入栈	函数本身 （被调用方）	@+函数名+@+参数的字节数
pascal	按照从左到右的顺序入栈	函数本身 （被调用方）	较为复杂，这里不再展开讨论

3、实例分析函数进栈出栈

（C语言内存十五）用一个实例来深入剖析函数进栈出栈的过程 - Smah - 博客园 (cnblogs.com)

1）函数入栈

- 把实参 返回地址入栈，ebp不变，esp向下一个地址移动

- 把原来ebp寄存器的值压入栈中 - old ebp，esp的值赋值给ebp，这样ebp就指向了新的函数的栈底，切换了函数栈
- 为局部变量 返回值预留内存 - 也就是esp - 一个整数，因为内存其实已经分配好了
- 将ebx esi edi寄存器的值压入栈中，然后将局部变量 返回值放入预留好的内存里

2）函数出栈

- 将ebx esi edi寄存器的值出栈
- 将局部变量 返回值出栈，将ebp的位置赋值给esp，esp和ebp指向了同一个位置
- 将old esp出栈，并且把old esp赋值给ebp，ebp就指向了main栈帧的old esp位置
- 根据返回地址找到下一条指令的位置，将返回地址和实参都出栈，esp来到了main的栈顶

函数完全出栈，栈被还原到了一开始的情况

2）为什么要留出这么多的空白，岂不是浪费内存吗？这是因为我们使用Debug模式生成程序，留出多余的内存，方便加入调试信息；以Release模式生成程序时，内存将会变得更加紧凑，空白也被消除。

至此，func() 函数的活动记录就构造完成了。可以发现，在函数的实际调用过程中，形参是不存在的，不会占用内存空间，内存中只有实参，而且是在执行函数体代码之前、由调用方压入栈中的。

3）垃圾值：需要为函数的局部变量分配内存，将esp减去一个整数，这段空白内存的初始值是0XCCCCCCCC，是个垃圾值

4）局部变量和返回值不是被销毁的

局部变量和返回值出栈，只是esp的加法运算，没有被销毁，原来内存单元的数据只是被新的函数覆盖了

C++

```
int *p;
void func(int m,int n) {
    int a = 18,b = 100;
    p = &a;
}
int main() {
    int n;
    func(10,20);
    n = *p;         //n还等于18
    return 0;
}
```

4、栈溢出

1）char str[1024 * 1024 * 20 = {0}，超出了默认的1M

2）char str[10] = {0};gets(str);输入了特别多的字符串，他会把4字节空白内存、old esp、返回地址都覆盖了，有可能得到一个错误的返回地址

strcpy scanf都会导致栈溢出

5、修改栈空间大小

1）查看linux默认栈空间大小：ulimit -s，默认是2M~10M

2）修改栈空间大小：ulimie -s 51200

3）/etc/security/limits.conf也可以修改栈空间的大小

十、动态内存分配

1、malloc

(void *)malloc(int size)

int *p = (int *)malloc(4);其实不用强转，因为void *可以转换成任何类型的指针，所以是int *p = malloc(4);

p是静态分配的，但是p指向的内存空间是动态分配的

1）void *不能++，因为不知道指几个内存单元

2）内存泄漏：内存被使用完了，没有剩余的内存了

比如

C++

```
while(1) {
    int *p = (int *)malloc(1000);
}
```

程序把内存使用完了，接着使用磁盘内的虚拟内存，如果磁盘也使用完了，就会死机

3）free

free的是指针指向的内存单元，但是释放并不代表销毁，它只不过是把内存单元标为可用，操作系统还可以把这块内存分配给其他变量，这块内存空间的值会是一个非常小的负数

指针还是指向这块内存单元的，只不过内存单元被分配给了其他变量，所以如果在这块内存空间写入了新的值，但是还用指针写入变量，这种情况会覆盖新的值，是不被允许的，所以要把指针赋空

4）malloc实现的原理

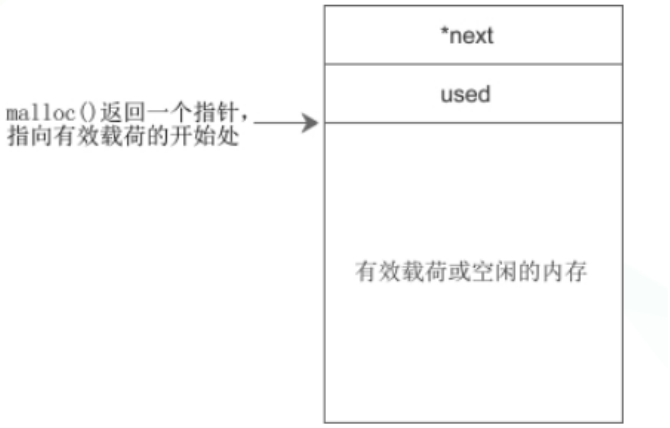
（C语言内存十八）malloc函数背后的实现原理——内存池 - Smah - 博客园 (cnblogs.com)

- 在程序运行过程中，堆内存从低地址向高地址连续分配，随着内存的释放，会出现不连续的空闲区域

图1：已分配内存和空闲内存相间出现

带阴影的方框是已被分配的内存，白色方框是空闲内存或已被释放的内存。程序需要内存时，malloc() 首先遍历空闲区域，看是否有大小合适的内存块，如果有，就分配，如果没有，就向操作系统申请（发生系统调用）。为了保证分配给程序的内存的连续性，malloc() 只会在一个空闲区域中分配，而不能将多个空闲区域联合起来。

- 内存块



可以加上一个prev

- 池化技术

  在计算机中，有很多使用“池”这种技术的地方，除了内存池，还有连接池、线程池、对象池等。以服务器上的线程池为例，它的主要思想是：先启动若干数量的线程，让它们处于睡眠状态，当接收到客户端的请求时，唤醒池中某个睡眠的线程，让它来处理客户端的请求，当处理完这个请求，线程又进入睡眠状态。

  所谓“池化技术”，就是程序先向系统申请过量的资源，然后自己管理，以备不时之需。之所以要申请过量的资源，是因为每次申请该资源都有较大的开销，不如提前申请好了，这样使用时就会变得非常快捷，大大提高程序运行效率。

十一、C语言变量的存储类别和生存期

1、存储类别

变量在哪个存储区，决定了生存期

2、auto

auto 是自动或默认的意思，很少用到，因为所有的变量默认就是 auto 的。也就是说，定义变量时加不加 auto 都一样，所以一般把它省略，不必多次一举。

auto int a = 100 与 int a = 100等效

3、static

作用域只在它定义的代码块

4、register

一般情况下，变量的值是存储在内存中的，CPU 每次使用数据都要从内存中读取。如果有一些变量使用非常频繁，从内存中读取就会消耗很多时间

C++

```
1  for(int i = 0;i < 1000;i++)
```

执行这段代码，CPU 为了获得 i，会读取 1000 次内存。

为了解决这个问题，可以将使用频繁的变量放在CPU的通用寄存器中，这样使用该变量时就不必访问内存，直接从寄存器中读取，大大提高程序的运行效率。

不过寄存器的数量是有限的，通常是把使用最频繁的变量定义为 register 的。

关于寄存器变量有以下事项需要注意：

1. 为寄存器变量分配寄存器是动态完成的，因此，只有局部变量和形式参数才能定义为寄存器变量。
1. 局部静态变量不能定义为寄存器变量，因为一个变量只能声明为一种存储类别。
2. 寄存器的长度一般和机器的字长一致，只有较短的类型如 int、char、short 等才适合定义为寄存器变量，诸如 double 等较大的类型，不推荐将其定义为寄存器类型。
3. CPU的寄存器数目有限，即使定义了寄存器变量，编译器可能并不真正为其分配寄存器，而是将其当做普通的auto变量来对待，为其分配栈内存。当然，有些优秀的编译器，能自动识别使用频繁的变量，如循环控制变量等，在有可用的寄存器时，即使没有使用 register 关键字，也自动为其分配寄存器，无须由程序员来指定。