# C++ Note v3



承接《C++ Note v3》本文档主要包含：cpu、SIMD、PCIe等内容

# 主板芯片组架构：

早先时候，计算机主板上有CPU、北桥（MCH）和南桥（ICH）这三个主要芯片。

之后由于FSB变成了系统效能的瓶颈，MCH被集成进CPU中。

CPU中MCH原来的部分，在桌面CPU中叫做System Agent（SA），在服务器CPU中叫做uncore（和内核core对应）。它基本还负责原来的功能，即内存管理和提供至少16个Lane的PCIe Root port来驱动显卡

ICH到PCH的转变相对很小，如果把PCH也整合进CPU，则为单芯片解决方案，即SoC（system on a chip）



Intel core i9-9900K：



芯片中间紫色部分是8个CPU核心（俗称8核i9），8个核心共享红色的L3 cache；左边是System Agent；右边蓝色的是“核显”Intel® UHD Graphics 630。

apple A12X即是SoC封装：



由上图可以看到，A12X系统一共8核CPU（4大Vortex，4小tempest）；神经网络处理器NPU在芯片中间；GPU在左上角；左下角有很多CPU之外的芯片，比如ISP(image signal Processor)，景深Depth Engine等。

# lambda捕获列表初始化

C++ 14支持lambda capture initializers

//按值捕获target，但是在Lambda内部的变量名叫做v

auto cnt =std::count\_if(books.begin(), books.end(), [v = target](const std::string& book) {return book.find(v) != std::string::npos;});

//按引用捕获target，但是在Lambda内部的名字叫做r

auto cnt =std::count\_if(books.begin(), books.end(), [&r = target](const std::string& book) {return book.find(r) != std::string::npos;});

Lambda捕获列表初始化最重要的一点是“支持Capture by Move”。在C++14之前，Lambda是不支持捕获一个Move-Only的对象的，比如：

std::unique\_ptr<int> uptr = std::make\_unique<int>(123);

auto callback = [uptr]() { // 编译错误，uptr is move-only

std::cout << \*uptr << std::endl;

};

通过捕获列表初始化，完成Move-Only对象的“Capture by Move”：

std::unique\_ptr<int> uptr = std::make\_unique<int>(123);

auto callback = [uptr = std::move(uptr)]() { //将uptr移动给Lambda表达式中的参数

std::cout << \*uptr << std::endl;

};

//... 将callback传给另一个线程

//return=>uptr是nullptr

# 堆：

heap本质上是用array或者vector实现的完全二叉树，二叉树的root节点代表整个heap的最大值（大根堆）或最小值（小根堆）

C++并没有将heap作为容器，而是作为算法放到<algorithm>中，默认是大根堆，可以通过指定比较算法构造小根堆。常用的API有以下几个：

* std::make\_heap(RandomIt first, RandomIt last, Compare comp): 在范围 [first, last) 中构造最大堆。
* std::push\_heap(RandomIt first, RandomIt last, Compare comp): 范围[first, last-1)已经是最大堆，将位于位置last-1的元素插入堆中。
* std::pop\_heap(RandomIt first, RandomIt last, Compare comp): 交换在位置 first 的值和在位置 last-1 的值，并令子范围 [first, last-1) 变为堆。这拥有从范围 [first, last) 所定义的堆移除首个元素的效果。

make\_heap之后的堆顶元素需要使用front()，push\_heap之后max元素在front()，pop\_heap之后[max元素](https://www.zhihu.com/search?q=max%E5%85%83%E7%B4%A0&search_source=Entity&hybrid_search_source=Entity&hybrid_search_extra=%7B%22sourceType%22%3A%22article%22%2C%22sourceId%22%3A%22452217326%22%7D)在back()

typedef std::vector<int> Heap;

Heap ints = {10,8,9,3,5,6,7,1,2,4};

std::make\_heap(ints.begin(), ints.end()); //front是10

std::pop\_heap(ints.begin(), ints.end()); //back是10，front是9

ints.pop\_back();

ints.push\_back(12);

std::push\_heap(ints.begin(), ints.end()); //front是12

# if(bool)汇编：

eax寄存器：累加器

MOVZX dst,src

将操作数src零扩展,再传给dst，操作数src空间必须小于操作数dst

MOVZX用0来扩展填充操作数dst的余下空间。MOVSX用操作数src的符号位扩展填充操作数A的余下空间，如果是负数则符号位为1，如果是正数则和MOVZX功能相同

AND dst,src

按位与，结果置于dst中。AND 指令总是清除溢出和进位标志位，并根据目标操作数的值来修改符号标志位、零标志位和奇偶标志位

TEST dst,src

逻辑与运算，两个操作数不会被改变，根据运算结果设置符号标志位、零标志位和奇偶标志位

test eax，100b；b后缀意为二进制

jnz \*\*\*\*\*\*；如果eax右数第三个位为1，jnz将会跳转

jnz或jne跳转的条件是ZF=0, ZF=0意味着ZF(零标志)没被置位，即按位与的结果非0

jz或je跳转的条件是ZF=1，即ZF(零标志)被置位



if(b)命中的条件是：字节b中有任一（或多）bit不为0

if(!b)命中的条件是：字节b中所有bit为0

**SIMD：**

SSE(Streaming SIMD Extensions， 流式SIMD扩展)指令集

SSE引入了16个新的128位寄存器(XMM0至XMM15)

SSE支持单个寄存器存储4个32位单精度浮点数

之后的SSE2则支持单个寄存器存储2个64位双精度浮点数，2个64位整数或4个32位整数或8个16位短整形

AVX（Advanced Vector Extensions）

AVX引入了16个256位寄存器(YMM0至YMM15)，AVX的256位寄存器和SSE的128位寄存器存在着相互重叠的关系(XMM寄存器为YMM寄存器的低位)，所以最好不要混用AVX与SSE指令集，否在会导致transition penalty(过渡处罚)

AVX与SSE支持的数据类型如下：



使用simd指令的方法之一是使用内置函数(intrinsics)

//使用SSE \_mm\_add\_ps 内置函数，一次执行8个单精度浮点数的加法

#include <xmmintrin.h>

int main()

{

\_\_m128 v0 = \_mm\_set\_ps(1.0f, 2.0f, 3.0f, 4.0f);

\_\_m128 v1 = \_mm\_set\_ps(1.0f, 2.0f, 3.0f, 4.0f);

\_\_m128 result = \_mm\_add\_ps(v0, v1);

}

SSE/AVX指令主要定义于以下一些头文件中：

immintrin.h头文件：定义了AVX指令, 支持同时操作8个单精度浮点数或4个双精度浮点数。

同时immintrin.h还包括了SSE~SSE4.2的头文件

SSE/AVX提供的数据类型和函数的命名规则如下：

* 数据类型通常以\_mxxx(T)的方式进行命名，其中xxx代表数据的位数，如SSE提供的\_\_m128为128位，AVX提供的\_\_m256为256位。T为类型，若为单精度浮点型则省略，若为整形则为i，如\_\_m128i，若为双精度浮点型则为d，如\_\_m256d。
* 操作浮点数的内置函数命名方式为：\_mm(xxx)\_name\_PT。 xxx为SIMD寄存器的位数，若为128m则省略，如\_mm\_addsub\_ps，若为\_256m则为256，如\_mm256\_add\_ps。 name为函数执行的操作的名字，如加法为\_mm\_add\_ps，减法为\_mm\_sub\_ps。 P代表的是对矢量(packed data vector)还是对标量(scalar)进行操作，如\_mm\_add\_ss是只对最低位的32位浮点数执行加法，而\_mm\_add\_ps则是对4个32位浮点数执行加法操作。 T代表浮点数的类型，若为s则为单精度浮点型，若为d则为双精度浮点，如\_mm\_add\_pd和\_mm\_add\_ps。
* 操作整形的内置函数命名方式为：\_mm(xxx)\_name\_epUY。xxx为SIMD寄存器的位数，若为128位则省略。 name为函数的名字。U为整数的类型，若为无符号类型则为u，否在为i，如\_mm\_adds\_epu16和\_mm\_adds\_epi16。Y为操作的数据类型的位数，如\_mm\_cvtpd\_pi32。

算术运算：

\_\_m128 \_mm\_dp\_ps(\_\_m128, \_\_m128, const int /\* mask \*/);

计算点积，但要考虑mask

First performs a SIMD multiplication of the lower four packed single-precision floating-point elements (float32 elements) from the first source vector m1 with corresponding elements in the second source vector m2.

Each of the four resulting single-precision elements is conditionally summed depending on the high four bits in the mask parameter.

The resulting summed value is broadcast to each of the lower 4 positions in the destination vector, if the corresponding lower bit of the mask is "1". If the corresponding lower bit of the mask is zero, the corresponding lower element in the destination vector is set to zero.

The process is then replicated with the high elements of the source vectors.

\_\_m128 v0 = \_mm\_set\_ps(4.0f, 3.0f, 2.0f, 1.0f);

\_\_m128 v1 = \_mm\_set\_ps(8.0f, 7.0f, 6.0f, 5.0f);//将4个32位浮点数按相反顺序赋值给\_\_m128中的4个浮点数

\_\_m128 r7 = \_mm\_dp\_ps(v0, v1, 0xff); //点乘，{70,70,70,70}

\_\_m128 r8 = \_mm\_dp\_ps(v0, v1, 0xf0); //点乘，{0,0,0,0}

\_\_m128 r9 = \_mm\_dp\_ps(v0, v1, 0xf4); //点乘，{0,0,70,0}

\_\_m128 r10 = \_mm\_dp\_ps(v0, v1, 0x0f); //点乘，{0,0,0,0}

\_\_m128 r11 = \_mm\_dp\_ps(v0, v1, 0x1f); //点乘，{5,5,5,5}

\_\_m128 r12 = \_mm\_dp\_ps(v0, v1, 0x3f); //点乘，{17,17,17,17}

\_\_m128 \_mm\_cmpeq\_ps(\_\_m128 \_A, \_\_m128 \_B);

逐分量比较4个浮点数是否相等，The result in the output vector will be 0xffffffff if the input elements were equal, or 0 otherwise

\_\_m128 v0 = \_mm\_set\_ps(4.0f, 3.0f, 2.0f, 1.0f);

\_\_m128 v1 = \_mm\_set\_ps(8.0f, 7.0f, 2.0f, 5.0f);

\_\_m128 r3 = \_mm\_cmpeq\_ps(v0, v1);//逐分量比较是否相等，{0,-nan,0,0}，即00 00 00 00 ff ff ff ff 00 00 00 00 00 00 00 00

SSE/AVX提供的逻辑运算操作包括：

* \_mm\_and\_pd对两个数据逐分量and
* \_mm\_andnot\_ps先对第一个数进行not，然后再对两个数据进行逐分量and
* \_mm\_or\_pd对两个数据逐分量or
* \_mm\_xor\_ps对两个数据逐分量xor

{

\_declspec(align(16)) uint8\_t f0[16] = { 0x00, 0x01, 0x03, 0x11 };

\_declspec(align(16)) uint8\_t f1[16] = { 0x01, 0x01, 0x02, 0x44 };

\_\_m128 vf0 = \_mm\_load\_ps((float\*)f0);

\_\_m128 vf1 = \_mm\_load\_ps((float\*)f1);

\_\_m128 r1 = \_mm\_andnot\_ps(vf0, vf1); //01 00 00 44 00 00 00 00 00 00 00 00 00 00 00 00

\_\_m128 r2 = \_mm\_xor\_ps(vf0, vf1); // 01 00 01 55 00 00 00 00 00 00 00 00 00 00 00 00

\_declspec(align(16)) uint8\_t d0[16] = { 0x00, 0x01, 0x03, 0x11 };

\_declspec(align(16)) uint8\_t d1[16] = { 0x01, 0x01, 0x02, 0x44 };

\_\_m128d vd0 = \_mm\_load\_pd((double\*)d0);

\_\_m128d vd1 = \_mm\_load\_pd((double\*)d1);

\_\_m128d r3 = \_mm\_and\_pd(vd0, vd1); //00 01 02 00 00 00 00 00 00 00 00 00 00 00 00 00

\_\_m128d r4 = \_mm\_or\_pd(vd0, vd1);//01 01 03 55 00 00 00 00 00 00 00 00 00 00 00 00

printf("0x%x\n", \*(unsigned\*)&vd0);

}

\_mm\_shuffle\_ps读取两个\_\_m128类型的数据a和b，并按照\_MM\_SHUFFLE提供的索引将返回的\_\_m128类型数据的低两位设置为a中按索引值取得到的对应值，将高两位设置为按索引值从b中取得到的对应值。索引值在0到3之间，分别以相反的顺序对应\_\_m128中的四个浮点数

\_declspec(align(16)) float p0[] = { 1, 2, 3, 4 };

\_declspec(align(16)) float p1[] = { 5, 6, 7, 8 };

\_\_m128 a = \_mm\_load\_ps(p0);

\_\_m128 b = \_mm\_load\_ps(p1);

\_\_m128 v0 = \_mm\_shuffle\_ps(a, b, \_MM\_SHUFFLE(1, 0, 3, 2)); // 3, 4, 5, 6

\_\_m128 v1 = \_mm\_shuffle\_ps(a, b, \_MM\_SHUFFLE(2, 2, 3, 3)); // 4, 4, 7, 7

\_\_m128 v2 = \_mm\_shuffle\_ps(a, b, \_MM\_SHUFFLE(1, 3, 2, 3)); //4, 3, 8, 6

**tcp吞吐量：**

tcp单流400Gbps吞吐需要每秒收发(400/8)\*1000\*1000\*1000/1460个数据包，大致34246575个，即每微秒处理34个数据包

对tcp，合适的MSS= MTU（最大传输单元）-IP报文头长度-tcp报文头长度=1500-20-20=1460

大致估算一下 25Gbps 场景下 Intel(R) Xeon(R) Platinum 8260 CPU @ 2.40GHz 的收包能力，关闭 GRO/LRO，下图三栏分别为 top/iperf -s -i 1/funclantency tcp\_v4\_rcv 的结果：



一款不错的 CPU 单核极限能力不到 10Gbps，发送端关闭 TSO/GSO，接收端 CPU 下降，但 tcp\_v4\_rcv 的执行能力已达上限，接收端打开 GRO/LRO，tcp\_v4\_rcv 开销变大，达到 1500us 以上。

**VDSO**

vsyscalls(virtual system calls)

VDSO (Virtual Dynamically Shared Objects)

VDSO是一个名称叫linux-vdso.so.1的.so文件，可以执行ldd `which bash`查看：

[root@localhost study]$ ldd `which bash`

linux-vdso.so.1 => (0x00007ffde4733000)

libtinfo.so.5 => /lib64/libtinfo.so.5 (0x00007f15ac77b000)

libdl.so.2 => /lib64/libdl.so.2 (0x00007f15ac577000)

libc.so.6 => /lib64/libc.so.6 (0x00007f15ac1a9000)

/lib64/ld-linux-x86-64.so.2 (0x00007f15ac9a5000)

Linux上，内核每加载一个ELF可执行程序时，都会在其进程地址空间中建立一个叫做vDSO mapping的内存区域

[root@localhost zcj]$ cat /proc/11716/maps

00400000-00402000 r-xp 00000000 fd:02 3358074883 /home/zcj/study/main

……

7fd66c253000-7fd66c254000 rw-p 00022000 fd:00 41952560 /usr/lib64/ld-2.17.so

7ffd01dfa000-7ffd01dfc000 r-xp 00000000 00:00 0 [vdso]

ffffffffff600000-ffffffffff601000 r-xp 00000000 00:00 0 [vsyscall]

VDSO加速某些系统调用函数的执行效率。以gettimeofday()为例，该系统调用返回的结果实际并不涉及任何数据安全问题，因为特权用户root和非特权用户都会获得相同的结果。因此与其费尽心力一定要通过陷入内核的方式来读取这些数据（昂贵的用户/内核态上下文切换开销），不如在内核与用户态之间建立一段共享内存区域（VDSO），内核定期“推送”最新值到该区域，调用gettimeofday()时并不真正执行系统调用，而是从该区域直接获取，相当于将系统调用改造成了函数调用。

VDSO支持的系统调用：gettimeofday()、time()、getcpu()、clock\_gettime()

gettimeofday()提供us级精度, clock\_gettime提供ns级精度

#include <sys/time.h>

int gettimeofday(struct timeval \*tv, struct timezone \*tz);

int settimeofday(const struct timeval \*tv, const struct timezone \*tz);

其中，tv参数是从Epoch算起的秒数和微秒数

struct timeval {

time\_t tv\_sec; /\* seconds \*/

suseconds\_t tv\_usec; /\* microseconds \*/

};

一般tz参数置0

函数返回0表示成功，返回-1失败

**TSC：**

x86 cpu提供TSC（time stamp counter）寄存器

用户态可直接用一条RDTSC指令读取TSC，性能消耗远小于借助VDSO的gettimeofday()、clock\_gettime()。

Intel cpu的TSC能力：

1. Variant TSC：TSC的增加受cpu频率变化的影响
2. Constant TSC：TSC以固定速率增加，不受cpu频率变化影响。但当cpu处于deep C-state状态时，TSC停止
3. Invariant TSC：TSC以固定速率增加，无论处于P-state、C-state还是T-state

Linux定义CPU feature bits描述cpu TSC能力：

1. X86\_FEATURE\_TSC：cpu支持TSC
2. X86\_FEATURE\_CONSTANT\_TSC：cpu支持constant TSC.
3. X86\_FEATURE\_NONSTOP\_TSC：与X86\_FEATURE\_CONSTANT\_TSC同时存在时表示cpu支持Invariant TSC

检查cpu tsc能力：cat /proc/cpuinfo | grep -E "constant\_tsc|nonstop\_tsc"

不同cpu的TSC如何保持同步：

1. 无同步机制。在早期SMP系统上，TSC并不会进行同步。这意味着如果程序在读取TSC后被OS切换到另一个核，再次读取TSC时可能出现“时间回退”现象。Variant TSC cpu和Constant TSC cpu存在此问题。
2. 同一主板上的多个CPU间同步。对支持Invariant TSC的cpu，多数SMP系统都会在cpu间同步TSC。在引导阶段，所有使用同一个RESET信号的CPU会被重置，它们的TSC以相同速率增加
3. 没有跨机柜、刀片或主板的同步机制。根据主板和计算机制造商的设计，不同主板上的CPU可能会连接到不同的时钟信号上，无法保证TSC同步

由以上分析可知，即使cpu支持Invariant TSC，SMP系统上，仍然无法保证TSC的可靠性。因此Linux内核执行boot time or runtime testing，如果在内核boot阶段通过了同步性测试，下面的文件会指定TSC作为时钟源：

$ cat /sys/devices/system/clocksource/clocksource0/current\_clocksource

tsc

如果要进行性能测试的代码非常短，最好使用LFENCE 或 RDTSCP 重新实现rdtsc函数。否则，CPU的乱序执行会对测量带来精度问题。CPU并不会等待前面的指令全部执行完再读取TSC值，后续指令同样有可能在读取操作前执行。如果需要在前面指令都执行完后再读取TSC值，开发者可以使用RDTSCP指令（如果CPU支持）或依次使用LFENCE、RDTSC两个指令。

**字符串分割**

查看进程内存泄漏时，pmap -p 34942和cat /proc/34942/maps配合使用

//注意，以下实现有一个缺陷，那就是相连的分隔符会被忽视，如"1,,3,4,5"相连的两个逗号会被当做一个

void split(const std::string& s, std::vector<std::string>& tokens, char delim = ',')

{

std::string::size\_type lastPos = s.find\_first\_not\_of(delim, 0);

std::string::size\_type pos = s.find\_first\_of(delim, lastPos);

while (pos != std::string::npos || lastPos != std::string::npos)

{

tokens.emplace\_back(s.substr(lastPos, pos - lastPos));

lastPos = s.find\_first\_not\_of(delim, pos);

pos = s.find\_first\_of(delim, lastPos);

}

}

**std::thread异常coredump导致调用堆栈丢失问题**

gcc 4.8.5

测试程序：

#include <iostream>

#include <thread>

#include <vector>

void thread\_func(){

std::cout << "thread\_func start ..." << std::endl;

std::vector<int> vec;

//vec.push\_back(1);

//vec.push\_back(2);

std::cout << vec.at(1) << std::endl;

}

int main (void){

std::thread th1(thread\_func);

th1.join();

return 0;

}

g++ -g -o main main.cpp -lpthread -std=c++11

[root@localhost study]$ ldd main

linux-vdso.so.1 => (0x00007ffc90fcf000)

libpthread.so.0 => /lib64/libpthread.so.0 (0x00007f2848112000)

libstdc++.so.6 => /lib64/libstdc++.so.6 (0x00007f2847e0b000)

libm.so.6 => /lib64/libm.so.6 (0x00007f2847b09000)

libgcc\_s.so.1 => /lib64/libgcc\_s.so.1 (0x00007f28478f3000)

libc.so.6 => /lib64/libc.so.6 (0x00007f2847525000)

/lib64/ld-linux-x86-64.so.2 (0x00007f284832e000)

gdb ./main

(gdb) bt

#0 0x00007ffff7008387 in raise () from /lib64/libc.so.6

#1 0x00007ffff7009a78 in abort () from /lib64/libc.so.6

#2 0x00007ffff79187d5 in \_\_gnu\_cxx::\_\_verbose\_terminate\_handler() () from /lib64/libstdc++.so.6

#3 0x00007ffff7916746 in ?? () from /lib64/libstdc++.so.6

#4 0x00007ffff7916773 in std::terminate() () from /lib64/libstdc++.so.6

#5 0x00007ffff796d105 in ?? () from /lib64/libstdc++.so.6

#6 0x00007ffff7bc6ea5 in start\_thread () from /lib64/libpthread.so.0

#7 0x00007ffff70d0b0d in clone () from /lib64/libc.so.6

解决方法：

1. 给线程函数thread\_func()加上noexcept声明

void thread\_func() noexcept {……}

(gdb) bt

#0 0x00007ffff7008387 in raise () from /lib64/libc.so.6

#1 0x00007ffff7009a78 in abort () from /lib64/libc.so.6

#2 0x00007ffff79187d5 in \_\_gnu\_cxx::\_\_verbose\_terminate\_handler() () from /lib64/libstdc++.so.6

#3 0x00007ffff7916746 in ?? () from /lib64/libstdc++.so.6

#4 0x00007ffff79156f9 in ?? () from /lib64/libstdc++.so.6

#5 0x00007ffff7916364 in \_\_gxx\_personality\_v0 () from /lib64/libstdc++.so.6

#6 0x00007ffff73af8a3 in ?? () from /lib64/libgcc\_s.so.1

#7 0x00007ffff73afc3b in \_Unwind\_RaiseException () from /lib64/libgcc\_s.so.1

#8 0x00007ffff7916986 in \_\_cxa\_throw () from /lib64/libstdc++.so.6

#9 0x00007ffff796b857 in std::\_\_throw\_out\_of\_range(char const\*) () from /lib64/libstdc++.so.6

#10 0x0000000000401595 in std::vector<int, std::allocator<int> >::\_M\_range\_check (this=0x7ffff6fd0e60, \_\_n=1)

at /usr/include/c++/4.8.2/bits/stl\_vector.h:794

#11 0x0000000000401313 in std::vector<int, std::allocator<int> >::at (this=0x7ffff6fd0e60, \_\_n=1)

at /usr/include/c++/4.8.2/bits/stl\_vector.h:812

#12 0x0000000000400fde in thread\_func () at main.cpp:9

#13 0x000000000040262f in std::\_Bind\_simple<void (\*())()>::\_M\_invoke<>(std::\_Index\_tuple<>) (this=0x606040)

at /usr/include/c++/4.8.2/functional:1732

#14 0x0000000000402589 in std::\_Bind\_simple<void (\*())()>::operator()() (this=0x606040)

at /usr/include/c++/4.8.2/functional:1720

#15 0x0000000000402522 in std::thread::\_Impl<std::\_Bind\_simple<void (\*())()> >::\_M\_run() (this=0x606028)

at /usr/include/c++/4.8.2/thread:115

#16 0x00007ffff796d070 in ?? () from /lib64/libstdc++.so.6

#17 0x00007ffff7bc6ea5 in start\_thread () from /lib64/libpthread.so.0

#18 0x00007ffff70d0b0d in clone () from /lib64/libc.so.6

1. gdb catch throw

(gdb)catch throw

(gdb)bt

#0 0x00007ffff7008387 in raise () from /lib64/libc.so.6

#1 0x00007ffff7009a78 in abort () from /lib64/libc.so.6

#2 0x00007ffff79187d5 in \_\_gnu\_cxx::\_\_verbose\_terminate\_handler() () from /lib64/libstdc++.so.6

#3 0x00007ffff7916746 in ?? () from /lib64/libstdc++.so.6

#4 0x00007ffff79156f9 in ?? () from /lib64/libstdc++.so.6

#5 0x00007ffff7916364 in \_\_gxx\_personality\_v0 () from /lib64/libstdc++.so.6

#6 0x00007ffff73af8a3 in ?? () from /lib64/libgcc\_s.so.1

#7 0x00007ffff73afc3b in \_Unwind\_RaiseException () from /lib64/libgcc\_s.so.1

#8 0x00007ffff7916986 in \_\_cxa\_throw () from /lib64/libstdc++.so.6

#9 0x00007ffff796b857 in std::\_\_throw\_out\_of\_range(char const\*) () from /lib64/libstdc++.so.6

#10 0x0000000000401595 in std::vector<int, std::allocator<int> >::\_M\_range\_check (this=0x7ffff6fd0e60, \_\_n=1)

at /usr/include/c++/4.8.2/bits/stl\_vector.h:794

#11 0x0000000000401313 in std::vector<int, std::allocator<int> >::at (this=0x7ffff6fd0e60, \_\_n=1)

at /usr/include/c++/4.8.2/bits/stl\_vector.h:812

#12 0x0000000000400fde in thread\_func () at main.cpp:9

#13 0x000000000040262f in std::\_Bind\_simple<void (\*())()>::\_M\_invoke<>(std::\_Index\_tuple<>) (this=0x606040)

at /usr/include/c++/4.8.2/functional:1732

#14 0x0000000000402589 in std::\_Bind\_simple<void (\*())()>::operator()() (this=0x606040)

at /usr/include/c++/4.8.2/functional:1720

#15 0x0000000000402522 in std::thread::\_Impl<std::\_Bind\_simple<void (\*())()> >::\_M\_run() (this=0x606028)

at /usr/include/c++/4.8.2/thread:115

#16 0x00007ffff796d070 in ?? () from /lib64/libstdc++.so.6

#17 0x00007ffff7bc6ea5 in start\_thread () from /lib64/libpthread.so.0

#18 0x00007ffff70d0b0d in clone () from /lib64/libc.so.6

1. 修改gcc源码bug

gcc-4.8.5/libstdc++-v3/src/c++11/thread.cc中

**namespace**

{

**extern** "C" **void\***

execute\_native\_thread\_routine(**void\*** \_\_p)

{

**thread::**\_Impl\_base**\*** \_\_t **=** **static\_cast<thread::**\_Impl\_base**\*>**(\_\_p);

**thread::**\_\_shared\_base\_type \_\_local;

\_\_local.swap(\_\_t**->**\_M\_this\_ptr);

**\_\_try**

{

\_\_t**->**\_M\_run();

}

\_\_catch(**const** \_\_cxxabiv1**::**\_\_forced\_unwind**&**)

{

\_\_throw\_exception\_again;

}

\_\_catch(...)

{

std**::**terminate();

}

**return** 0;

}

}

\_\_t->\_M\_run() 是运行线程的代码，当运行过程中触发了throw抛异常操作，且该异常类型为非 \_\_cxxabiv1::\_\_forced\_unwind 类型，则进入了std::terminate分支。由于异常捕获，这里调用堆栈信息已经丢失了了，所以期望看到的调用堆栈信息都没了

修改方法：

      /\*modify

      \_\_try

      {

        \_\_t->\_M\_run();

      }

       \_\_catch(const \_\_cxxabiv1::\_\_forced\_unwind&)

      {

        \_\_throw\_exception\_again;

      }

      \_\_catch(...)

      {

        std::terminate();

      }

      \*/

      \_\_t->\_M\_run();

[root@localhost study]$ export LD\_LIBRARY\_PATH=/home/zcj/gcccompile/gcc-4.8.5/install\_dir/lib64

[root@localhost study]$ vi main.cpp

[root@localhost study]$ g++ -g -o main main.cpp -lpthread -std=c++11

[root@localhost study]$ ldd main

linux-vdso.so.1 => (0x00007fff6051c000)

libpthread.so.0 => /lib64/libpthread.so.0 (0x00007f0f437d6000)

libstdc++.so.6 => /home/zcj/gcccompile/gcc-4.8.5/install\_dir/lib64/libstdc++.so.6 (0x00007f0f434d4000)

libm.so.6 => /lib64/libm.so.6 (0x00007f0f431d2000)

libgcc\_s.so.1 => /home/zcj/gcccompile/gcc-4.8.5/install\_dir/lib64/libgcc\_s.so.1 (0x00007f0f42fbc000)

libc.so.6 => /lib64/libc.so.6 (0x00007f0f42bee000)

/lib64/ld-linux-x86-64.so.2 (0x00007f0f439f2000)

(gdb) bt

#0 0x00007ffff700d387 in raise () from /lib64/libc.so.6

#1 0x00007ffff700ea78 in abort () from /lib64/libc.so.6

#2 0x00007ffff791c705 in \_\_gnu\_cxx::\_\_verbose\_terminate\_handler ()

at ../../.././libstdc++-v3/libsupc++/vterminate.cc:95

#3 0x00007ffff791a896 in \_\_cxxabiv1::\_\_terminate (handler=<optimized out>)

at ../../.././libstdc++-v3/libsupc++/eh\_terminate.cc:38

#4 0x00007ffff791a8c3 in std::terminate () at ../../.././libstdc++-v3/libsupc++/eh\_terminate.cc:48

#5 0x00007ffff791aae3 in \_\_cxxabiv1::\_\_cxa\_throw (obj=0x7ffff0000940,

tinfo=0x7ffff7ba31d0 <typeinfo for std::out\_of\_range>, dest=

0x7ffff792e930 <std::out\_of\_range::~out\_of\_range()>) at ../../.././libstdc++-v3/libsupc++/eh\_throw.cc:87

#6 0x00007ffff796bc17 in std::\_\_throw\_out\_of\_range (\_\_s=<optimized out>)

at ../../../.././libstdc++-v3/src/c++11/functexcept.cc:80

#7 0x00000000004015b7 in std::vector<int, std::allocator<int> >::\_M\_range\_check (this=0x7ffff6fd5e50, \_\_n=1)

at /usr/include/c++/4.8.2/bits/stl\_vector.h:794

#8 0x0000000000401335 in std::vector<int, std::allocator<int> >::at (this=0x7ffff6fd5e50, \_\_n=1)

at /usr/include/c++/4.8.2/bits/stl\_vector.h:812

#9 0x0000000000400fdf in thread\_func () at main.cpp:9

#10 0x0000000000402651 in std::\_Bind\_simple<void (\*())()>::\_M\_invoke<>(std::\_Index\_tuple<>) (this=0x606040)

at /usr/include/c++/4.8.2/functional:1732

#11 0x00000000004025ab in std::\_Bind\_simple<void (\*())()>::operator()() (this=0x606040)

at /usr/include/c++/4.8.2/functional:1720

#12 0x0000000000402544 in std::thread::\_Impl<std::\_Bind\_simple<void (\*())()> >::\_M\_run() (this=0x606028)

at /usr/include/c++/4.8.2/thread:115

#13 0x00007ffff796d260 in std::(anonymous namespace)::execute\_native\_thread\_routine (\_\_p=<optimized out>)

at ../../../.././libstdc++-v3/src/c++11/thread.cc:96

#14 0x00007ffff7bc6ea5 in start\_thread () from /lib64/libpthread.so.0

#15 0x00007ffff70d5b0d in clone () from /lib64/libc.so.6

**Linux权限管理：**

SUID（SetUID）：文件的一种特殊权限，当设置了SUID位的文件被执行时，该文件将以所有者的身份运行，即无论谁来执行这个文件，他都有文件所有者的特权。如，如果所有者是root的话，那么执行人就有root权限了，因此不要轻易设置该位。

只有文件所有者能为文件设置suid。bash的所有者是root，普通用户想通过给bash设置suid然后拿到root权限是不可能的，这个权限只有root能给。

chmod u+s temp：为temp文件添加suid标志

chmod u-s temp：移除temp文件的suid标志

设置完suid标志后，可以用ls-l来查看。如果有suid标志，则会在原来执行标志x的位置上显示。原来的执行标志x到哪里去了呢？系统规定，如果本来有执行权限，则显示为小写字母s，如果本来无执行权限则显示为大写字母S

-rw-r--r-- 1 root root 3633 Jan 6 02:26 runlog

-rwxr-xr-x 1 root root 1314 Jan 6 02:21 memcheck.sh

chmod u+s runlog

chmod u+s memcheck.sh

-rwSr--r-- 1 root root 3633 Jan 6 02:26 runlog

-rwsr-xr-x 1 root root 1314 Jan 6 02:21 memcheck.sh

su程序有suid权限：

ll `which su`

-rwsr-xr-x. 1 root root 32128 Aug 9 2019 /usr/bin/su

文件

$ ls -l /etc/hosts

-rw-r--r-- 1 root root 619 Oct 1 08:51 /etc/hosts

第一组rw-：  表示文件的拥有者对它的权限

第二组r--：  表示文件的所属组对它的权限

第三组r--：  表示其他用户对这个文件的权限

两个root：分别为文件的拥有者和所属的组

文件上的用户信息和权限信息是文件属性，是静态的，用来限制谁能对该文件执行什么操作

而对文件执行操作最终是由进程完成的，所以进程拥有的权限才是决定能否对文件执行操作的标准。

EUID、RUID、EGID、RGID都是进程的属性

EUID(Effective UID)，有效用户ID；

RUID(Real UID)，实际用户ID

EGID有效组ID；

RGID实际组ID

进程在执行某些涉及权限的操作时，内核将使用进程的【有效用户/组ID】作为凭证来判断是否有权限执行对应操作。例如，rm命令删除文件a.txt时，内核将使用rm进程的EUID、EGID和a.txt文件的用户ID、组ID做比较，进而再比较对应的权限位，从而判断rm进程是否有权限删除a.txt

既然涉及到权限的操作都以有效ID作为依据，那实际ID有什么作用？RUID和RGID用于确定进程所属的用户和组，主要用于判断是否有权限向进程发送信号：对于非特权用户，如果发送信号的进程A的RUID和目标进程B的RUID相同，则进程A可以发送信号。由于子进程会继承父进程的实际ID，所以父子进程的RUID相等，父子进程可互相发信号

$ useradd -m abc //创建用户abc

$ passwd abc //设置用户abc的密码

$ cat /etc/passwd //查看系统中用户列表

$ su abc //切换到用户abc

$ whoami //查看当前用户

$ id //查看当前用户id

uid=1002(abc) gid=1002(abc) groups=1002(abc)

id root //查看root用户id

uid=0(root) gid=0(root) groups=0(root)

$ userdel abc //删除用户abc

用户执行程序时，默认将当前用户或指定的用户(如sudo方式)设置为该进程的EUID和实际用户RUID

进程内部可通过代码修改进程的EUID。例如，以root身份运行的程序，初始时其有EUID为0(即root)，具有特权，之后在程序内部修改其EUID为nobody，修改后该进程将失去大量权限。

为程序文件设置suid，则该程序在执行时，其EUID被设置为程序文件所有者的UID

举个例子，分析用户user1执行/bin/app删除a.txt文件时的身份设置过程（只考虑用户不考虑组）：

1.获取/bin/app权限信息，判断user1是否有权限执行该程序，若有权，则exec加载成功。

2.如果app文件未设置SUID，则设置进程的有效用户ID和实际用户ID都为user1。

3.如果app文件设置了SUID，则设置进程的有效用户ID为app文件的用户ID。

4.app程序内部可能也会通过系统调用修改进程的有效用户ID。

5.app进程执行删除a.txt文件代码时，内核将比较此时app进程的有效用户ID和a.txt文件的用户ID，从而判断权限位

举个例子，分析登录界面的安全性：

登录界面本身可能需要拿到用户头像之类的个性化信息，所以它必须具有root身份，否则就无权翻看所有用户的profile。

但这样一来，当登录界面本身存在问题时，一旦被人抓住漏洞，就被拿到了root权限。这是非常危险的。

怎么办呢？

Linux还有个seteuid调用，这个调用可以拿来放弃root权限。

登录界面初始时有root权限，遍取所有用户的profile；但拿到profile之后，就没必要保留root权限了——调用个seteuid，放弃root身份，以访客身份提供登陆服务！

除非黑客在读取用户profile时成功攻击登录界面，否则拿不到任何好处；但读取profile时登录界面不对外提供服务，对外提供服务时，又已经放弃了root身份——无懈可击！

**网络并发模型**

消费队列数据的推拉方式：

工作线程消费队列数据的方式，分为『推』和『拉』两种模型。

推模型，即若队列无数据，则阻塞休眠，当生产者写入数据到队列后，则会唤醒休眠的工作线程来处理。

拉模型，即不需要条件变量，工作线程内死循环，不停地轮询查找取出队列数据。

两种模型各有利弊，主要要看实际业务场景，如果是IO密集型的，比如并发度特别高，以至于几乎总能取到数据，那么就采用拉模型。

条件变量的伪唤醒：

1. pthread的条件变量等待pthread\_cond\_wait是使用阻塞的系统调用实现的（比如Linux上的futex），这些阻塞的系统调用在进程被信号中断后，通常会中止阻塞、直接返回EINTR错误。即没有调notify但是被唤醒。
2. 有对应的唤醒，但条件不成立。由于线程调度，被条件变量唤醒的线程在本线程内真正执行「加锁并返回」前，另一个线程抢先插了进来，完整地进行了一套「拿锁、改条件、还锁」的操作。
3. wait、wait\_for和wait\_until，都有带判断式的重载版本，可避免伪唤醒的危害

单线程IO多路复用：



Polling API泛指select/poll/epoll/kqueue这种IO多路复用API

让Polling API监控服务端socket的状态，然后开始死循循环，循环过程中主要有三种逻辑分支：

1. 服务端socket的状态变为可读，则表示有客户端发起连接，此时就调用accept建立连接，得到一个客户端fd。将其加入到Polling API的监控集合，并标记其为可读。
2. 客户端fd的状态变为可读，则调用read/recv从fd读取数据，然后执行业务逻辑，处理完，再将其加入到Polling API的监控集合，并标记其为可写。
3. 客户端fd的状态变为可写，则调用write/send将数据发送给客户端

多线程/多进程IO：

主线程创建多个子线程，然后每个子线程内部开启死循环，listen、accept、read、send

借助SO\_REUSEPORT，不同socket可以绑定到相同ip:port上，同时Linux内核提供负载均衡功能

半同步半异步IO：

Half-Sync/Half-Async（HSHA）：异步IO层+队列+同步处理层



一般是一个IO线程和多个工作线程。IO线程负责异步地从客户端fd获取客户端的请求数据，而工作线程则是并发的对该数据进行处理

异步IO，调read后立即返回不阻塞，有数据到来时，内核自动将数据拷贝到用户缓冲区，然后发送signal通知用户或执行预设的回调函数

从IO线程到工作线程如何交换数据呢？借助一个单生产者多消费者队列。

处理完后返回数据如何发送？实现方式可以有很多，比如处理完成之后，直接在工作线程中向客户端发送数据。或者再创建一个写入队列，将返回数据和客户端信息（比如fd）放入该队列，然后由一组专门负责发送的线程来取元素和发送

半同步半反应堆IO：

Half-Sync/Half-Reactor（HSHR）：同步IO+队列+同步处理层



死循环之初，Polling API只监听服务端socket，当监测到服务端socket可读，进行accept，获得客户端fd，将其加入Polling API的监控集合中，继续进入循环。

循环体内Polling API返回，将fd取出。如果是服务socket再重复accept，如果是其他fd就丢进队列中。

和HSHA不同，HSHR的队列中存放的不是请求数据，而是fd。

多个工作线程竞争消费队列，从队列中取到的不是数据，而是客户端fd以及一些状态信息。工作线程的逻辑循环内从队列取到fd后，对fd进行read/recv获取请求数据，然后进行处理，最后直接write/send客户端fd，将数据返回给客户端。

**模板：**

模板类的模板构造函数：

template<class T>

class A{

public:

template<class T>

A(T) {cout << typeid(T).name() << endl;}

void show() {cout << typeid(T).name() << endl;}

};

int main(){

A<double> a(3); //int

a.show(); //double

}

There is no way to explicitly specify the template arguments when calling a constructor template, so they have to be deduced through argument deduction. This is because if you say:

Foo<int> f = Foo<int>();

The <int> is the template argument list for the type Foo, not for its constructor.

模板特化实现模板参数变换：

//默认特化不实现

template <typename T >

class Magic;

template<class Ret, class Arg>

struct Magic<Ret(Arg)>

{

void show(){ cout << typeid(Ret).name() << " " << typeid(Arg).name() << endl;}

};

int main(){

Magic<double(int)> m; //double int

m.show();

}

多态实现类型擦除：

用Function类包装所有返回值为void参数为int的可调用对象，如下：

class Function

{

private:

struct CallableBase{

virtual void operator()(int a0) = 0;

virtual struct CallableBase \*copy() const = 0;

virtual ~CallableBase() {};

};

template <typename T>

struct CallableDerived : public CallableBase{

T f;

CallableDerived(T functor) : f(functor) {}

void operator()(int arg) { return f(arg); }

CallableBase \*copy() const { return new CallableDerived<T>(f); }

};

private:

CallableBase \*m\_ptr;

public:

template <typename T>

Function(T functor) : m\_ptr(new CallableDerived<T>(functor)) {}

Function() : m\_ptr(nullptr) {}

void operator()(int arg) { return (\*m\_ptr)(arg); }

Function(const Function &f) { m\_ptr = f.m\_ptr->copy(); }

Function &operator=(const Function &f) {

if (this != &f){

delete m\_ptr;

m\_ptr = f.m\_ptr->copy();

}

return \*this;

}

Function(Function &&f) {

m\_ptr = f.m\_ptr;

f.m\_ptr = nullptr;

}

Function &operator=(Function &&f) {

if (this != &f){

delete m\_ptr;

m\_ptr = f.m\_ptr;

f.m\_ptr = nullptr;

}

return \*this;

}

~Function() { delete m\_ptr; }

};

int main(int argc, char \*\*argv)

{

Function showFunc = [](int a) {cout << "par=" << a << endl; };

showFunc(9);

}

**Lambda优于std::bind**

lambda更易内联

lambda延迟求值

using namespace std::chrono;

using Time = steady\_clock::time\_point;

enum class Sound { Beep, Siren, Whistle };

//在时间t，使用s声音响铃

void setAlarm(Time t, Sound s){

cout << "start time=" << duration\_cast<seconds>(t.time\_since\_epoch()).count() << " alram" << endl;

}

int main(int argc, char \*\*argv)

{

//设置一个小时后响铃的警报器， 但是，具体声音仍未确定

auto setSoundL =[](Sound s){setAlarm(steady\_clock::now() + hours(1), s);};

//警报器将被设置为在调用setSoundL后一小时发出声音

setSoundL(Sound::Beep); //setAlarm函数体在这可以很好地内联

//在std::bind调用中，将steady\_clock::now() + 1h作为实参传递给了std::bind，而不是setAlarm。这意味着将在调用std::bind时对表达式进行求值，并且该表达式产生的时间将存储在产生的bind对象中

//警报器将被设置为在调用std::bind后一小时发出声音，而不是在调用setAlarm一小时后发出

auto setSoundB = std::bind(&setAlarm, steady\_clock::now() + hours(1), std::placeholders::\_1);

setSoundB(Sound::Beep); //对std::bind的调用是将函数指针传递给setAlarm，对setAlarm的调用是通过一个函数指针，编译器不太可能通过函数指针内联函数

}

按值传递、按引用传递

struct Widget{

int size = 8;

Widget() = default;

Widget(const Widget& rhs) :size(rhs.size) { cout << "Widget Copy Constructor" << endl; }

};

struct CompLevel{

int lev = 0;

CompLevel() = default;

CompLevel(const CompLevel& rhs) :lev(rhs.lev) { cout << "CompLevel Copy Constructor" << endl; }

CompLevel(CompLevel&& rhs) :lev(rhs.lev) { cout << "CompLevel Move Constructor" << endl; }

};

void compress(const Widget& w, CompLevel lev) { //制作w的压缩副本

cout << "compressed , size=" << w.size << endl;

}

int main(int argc, char \*\*argv)

{

Widget w; //w.heavy=8

auto compressRateL = [&w](CompLevel lev) {return compress(w, lev); }; //显示指明按引用传递

auto compressRateB = std::bind(compress, w, std::placeholders::\_1); //默认按值传递，CompLevel拷贝构造

auto compressRateBRef = std::bind(compress, std::ref(w), std::placeholders::\_1); //引用传递

w.size = 9;

CompLevel lev;

compressRateL(lev); //实参lev按值传递，总共两次CompLevel的拷贝构造，size=9

compressRateB(lev); //实参lev默认按引用传递，一次CompLevel的拷贝构造，size=8

}

**emplace插入：**

拷贝初始化和直接初始化：

struct A {

explicit A(int x) { }

};

int main(){

A r1 = 10; //错误！不能编译

A r2(10); //正确

}

在标准术语中，用于初始化r1的语法（使用等号“=”）称为**拷贝初始化**；

用于初始化r2的语法（使用小括号或花括号）称为**直接初始化**

拷贝初始化不允许使用explicit构造函数

对r1，int类型10首先隐式转换为A，而根据构造函数定义中的explicit，这样的隐式转换不被允许，从而产生编译时期的报错。如果无explicit，则仅会调用构造函数，而非构造函数+拷贝\移动赋值函数

emplace插入和explicit：

struct A {

explicit A(int x) { }

};

int main(){

std::vector<A> vec;

vec.push\_back(10); //错误

vec.emplace\_back(11); //正确

}

对push\_back(10)，实际是拷贝初始化

emplace\_back实际是直接初始化，不受explicit影响，相当于A a(10)

emplace插入和存储资源管理类对象的容器：

对存储资源管理类对象的容器（比如std::list<std::shared\_ptr<Widget>>）调用插入函数时，函数的形参类型通常确保在资源的获取（比如使用new）和资源管理对象的创建之间没有其他操作。在emplace函数中，完美转发推迟了资源管理对象的创建，直到可以在容器的内存中构造它们为止，这给“异常导致资源泄漏”提供了可能。

class Widget {

};

int main(){

void killWidget(Widget\* pWidget);

std::list<std::shared\_ptr<Widget>> ptrs;

ptrs.push\_back(std::shared\_ptr<Widget>(new Widget, killWidget));

ptrs.emplace\_back(new Widget, killWidget);

}

对push\_back(std::shared\_ptr<Widget>(new Widget, killWidget))

1. 在上述的调用中，一个std::shared\_ptr<Widget>的临时对象被创建来持有“new Widget”返回的原始指针。称这个对象为temp。
2. push\_back通过引用接受temp。在存储temp的副本的*list*节点的内存分配过程中，内存溢出异常被抛出。
3. 随着异常从push\_back的传播，temp被销毁。作为唯一管理这个Widget的std::shared\_ptr，它自动销毁Widget，在这里就是调用killWidget。

对ptrs.emplace\_back(new Widget, killWidget)：

1. 通过new Widget创建的原始指针完美转发给emplace\_back中，*list*节点被分配的位置。如果分配失败，还是抛出内存溢出异常。
2. 当异常从emplace\_back传播，原始指针是仅有的访问堆上Widget的途径，但是因为异常而丢失了，那个Widget的资源（以及任何它所拥有的资源）发生了泄漏。

**=delete：**

C++中只声明一个函数，不定义，不调用，是可以编译链接并运行的

C++11前实现禁止拷贝语义，常

class basic\_ios {

private:

basic\_ios(const basic\_ios& ); // not defined

basic\_ios& operator=(const basic\_ios&); // not defined

};

将它们声明为private可以防止客户端调用这些函数。故意不定义它们意味着假如还是有代码用它们（比如成员函数或者类的友元friend），就会在链接时报错缺少函数定义（missing function definitions）。

C++11后，

class basic\_ios {

public:

basic\_ios(const basic\_ios& )=delete;

basic\_ios& operator=(const basic\_ios&)=delete;

};

deleted函数不能以任何方式被调用，否则会在编译期报错，而使用私有未定义函数的方式，可能在链接时才报错。

deleted函数一般被声明为public而不是private。当客户端代码试图调用成员函数，C++会在检查deleted状态前检查它的访问性。当客户端代码调用一个私有的deleted函数，一些编译器只会给出该函数是private的错误（而没有诸如该函数被deleted修饰的错误），即使函数的访问性不影响它是否能被使用。

**{}初始化：**

Widget w1(10); //使用实参10调用Widget的一个构造函数

但是如果你尝试使用相似的语法调用没有参数的Widget构造函数，它就会变成函数声明：

Widget w2(); //最令人头疼的解析！声明一个函数w2，返回Widget

由于函数声明中形参列表不能使用花括号，所以使用花括号初始化没有问题：

Widget w3{}; //调用无参构造函数构造对象

C++规定任何能被决议为一个声明的东西必须被决议为声明

当auto声明的变量使用花括号初始化，变量类型就会被推导为std::initializer\_list，尽管使用相同内容的其他初始化方式会产生正常的结果。如auto y = { 10 };//y被推导为std::initializer\_list<int>类型

**SIMD：**

SIMD的全称是Single Instruction Multiple Data，中文名"单指令多数据"。顾名思义，一条指令处理多个数据。

<https://www.intel.com/content/www/us/en/docs/intrinsics-guide/index.html>

SIMD简史：

1997年，Intel推出了第一个SIMD指令集——MultiMedia eXtensions（MMX）。MMX使用MM0~MM7共8个寄存器，每个寄存器为64位宽。

1999年，Intel推出SSE（Streaming SIMD Extensions，流式SIMD扩展）。SSE引入了16个新的寄存器XMM0~XMM15，每个寄存器为128位宽。

2000年，Intel发布SSE2。

2004年，Intel发布SSE3。

2006年，Intel发布SSE4指令集，并在2007年推出的cpu上实现。

2008年，Intel和AMD提出了AVX（Advanced Vector eXtentions），并于2011年分别在Sandy Bridge以及Bulldozer架构上实现。AVX对XMM寄存器做了扩展，从原来的128位扩展到了256位。

2013年，Intel发布AVX2，并于同年提出了AVX-512。AVX-512的主要改进是把SIMD寄存器扩展到了512位。

2016年，Intel发布第一款支持了AVX-512的CPU，即Xeon Phi x200(Knights Landing)。

查看SIMD指令文档：

新版本文档：https://learn.microsoft.com/en-us/cpp/intrinsics/x86-intrinsics-list?view=msvc-170

旧版本文档：https://learn.microsoft.com/zh-cn/previous-versions/visualstudio/visual-studio-2008/bb531427(v=vs.90)

新版本搜索：

进入microsoft learn搜索，https://learn.microsoft.com/en-us/cpp/intrinsics/?view=msvc-170

右上角搜索栏默认显示最新版vs，如vs2022

在vs2022下搜索\_mm\_shuffle\_epi8，在跳转页选中porduct->Visual Studio，继续搜索，进入x86 intrinsics list，查找到\_mm\_shuffle\_epi8指令，点击链接会跳转到intel官网下对该指令的说明

旧版本搜索：

进入microsoft learn搜索，https://learn.microsoft.com/en-us/visualstudio/ide/?view=vs-2022

右上角搜索栏默认显示最新版vs，切换到Previous versions，然后搜索\_mm\_shuffle\_epi8





SIMD指令头文件：

<immintrin.h>中包含了所有SIMD指令

但要使编译通过，在gcc and clang中还必须指定编译选项-march=native；

msvc默认允许使用SIMD指令，但在使用AVX指令时依旧需要显式enbale AVX

call gcc with march=native and ask it what flags it would use：

gcc -march=native -Q --help=target：查询（-Q --help=target）GCC编译器在本地计算机上（-march=native）支持的目标体系结构

对Intel(R) Xeon(R) Gold 6148 CPU @ 2.40GHz

gcc 4.8.5显示gcc -march=native -Q --help=target | grep march为core-avx2

gcc 8.3.1显示gcc -march=native -Q --help=target | grep march为skylake-avx512

cat /sys/devices/cpu/caps/pmu\_name显示skylake

google搜索”Intel(R) Xeon(R) Gold 6148 CPU @ 2.40GHz”，intel官网显示：Code Name：Products formerly Skylake

具体各指令集对应的头文件如下：

|  |  |  |
| --- | --- | --- |
| 头文件 | 指令集 | cpu |
| <mmintrin.h> | MMX | Pentium MMX! |
| <xmmintrin.h> | MMX + SSE | Pentium 3, Athlon XP |
| <emmintrin.h> | MMX + SSE+ SSE2 | Pentiuem 4, Ahtlon 64 |
| <pmmintrin.h> | MMX + SSE+ SSE2 + SSE3 | Pentium 4 Prescott, Ahtlon 64 San Diego |
| <tmmintrin.h> | MMX + SSE+ SSE2 + SSE3 + SSSE3 | Core 2, Bulldozer |
| <ammintrin.h> | MMX + SSE+ SSE2 + SSE3 + SSE4A | Phenom |
| <smmintrin.h> | MMX + SSE+ SSE2 + SSE3 + SSSE3+SSE4.1 | Core i7, Bulldozer |
| <nmmintrin.h> | MMX + SSE+ SSE2 + SSE3 + SSSE3+SSE4.1 +SSE4.2 | Core i7, Bulldozer |
| <wmmintrin.h> | AES | Core i7 Westmere, Bulldozer |
| <immintrin.h> | MMX + SSE+ SSE2 + SSE3 + SSSE3+SSE4.1 +SSE4.2, AVX, AVX2, FMA | Core i7 Sandy Bridge, Bulldozer |

gcc选项

查看gcc默认支持的SIMD指令集：gcc -dM -E - < /dev/null | grep -e SSE -e AVX

-E：表示只进行预处理

-dM：表示在预处理时，输出所有的宏定义，包括系统头文件中定义的宏

-：表示从标准输入读取源代码

< /dev/null：将标准输入重定向为 /dev/null，即不提供任何输入内容

[root@zhs home]$ gcc -dM -E - < /dev/null | grep -e SSE -e AVX | sort

#define \_\_SSE\_\_ 1

#define \_\_SSE2\_\_ 1

#define \_\_SSE2\_MATH\_\_ 1

#define \_\_SSE\_MATH\_\_ 1

[root@zhs home]$ gcc -msse2 -dM -E - < /dev/null | grep -e SSE -e AVX | sort

#define \_\_SSE\_\_ 1

#define \_\_SSE2\_\_ 1

#define \_\_SSE2\_MATH\_\_ 1

#define \_\_SSE\_MATH\_\_ 1

[root@zhs home]$ gcc -mno-sse2 -dM -E - < /dev/null | grep -e SSE -e AVX | sort

#define \_\_SSE\_\_ 1

#define \_\_SSE\_MATH\_\_ 1

[root@zhs home]$ gcc -msse3 -dM -E - < /dev/null | grep -e SSE -e AVX | sort

#define \_\_SSE\_\_ 1

#define \_\_SSE2\_\_ 1

#define \_\_SSE2\_MATH\_\_ 1

#define \_\_SSE3\_\_ 1

#define \_\_SSE\_MATH\_\_ 1

GCC默认即支持sse2指令，因此编译包含\_mm\_store\_si128、\_mm\_unpacklo\_epi8等sse2指令的代码时，可以不用显式指定-msse2。如果要禁用sse2指令，可使用-mno-sse2选项。

|  |  |
| --- | --- |
| gcc选项 | 引入的simd宏 |
| default | \_\_SSE\_\_、\_\_SSE\_MATH\_\_、\_\_SSE2\_\_、\_\_SSE2\_MATH\_\_ |
| gcc -msse2 | \_\_SSE\_\_、\_\_SSE\_MATH\_\_、\_\_SSE2\_\_、\_\_SSE2\_MATH\_\_ |
| gcc -msse3 | -msse2、\_\_SSE3\_\_ |
| gcc -msse4 | -msse2、\_\_SSE3\_\_、\_\_SSSE3\_\_、\_\_SSE4\_1\_\_、\_\_SSE4\_2\_\_ |
| gcc -mavx2 | -msse4、\_\_AVX\_\_、\_\_AVX2\_\_ |
| gcc -mavx512f | -mavx2、\_\_AVX512F\_\_ |
| gcc -mavx512vl | -mavx512f 、\_\_AVX512VL\_\_ |
| gcc -mavx512bw | -mavx512f 、\_\_AVX512BW\_\_ |
| gcc -mavx512vbmi | -mavx512f 、\_\_AVX512BW\_\_、\_\_AVX512VBMI\_\_ |
| gcc -mavx512vbmi2 | -mavx512f 、\_\_AVX512VBMI2\_\_ |

调用SIMD程序示例：

调用编译器内置函数(intrinsics)，是使用simd指令的方法之一

如，使用内置函数\_mm\_add\_ps，一次执行8个单精度浮点数的加法：

#include <cstdio>

#include <immintrin.h>

int main()

{

\_\_m128 v0 = \_mm\_set\_ps(1.0f, 2.0f, 3.0f, 4.0f); //将4个float按相反顺序赋值给\_\_m128中的4个float

\_\_m128 v1 = \_mm\_set\_ps(5.0f, 6.0f, 7.0f, 8.0f);

\_\_m128 result = \_mm\_add\_ps(v0, v1);

float fs[4];

\_mm\_storeu\_ps(fs, result);

printf("%f, %f, %f, %f\n", fs[0], fs[1], fs[2], fs[3]);//12.0 10.0 8.0 6.0

}

g++ - march = native - o main main.cpp - std = c++11

SSE/AVX支持的数据类型：



SSE引入了16个128位寄存器(XMM0至XMM15)，单个寄存器支持存储4个32位单精度浮点数

SSE2支持单个寄存器存储2个64位双精度浮点数，2个64位整数或4个32位整数或8个16位整数

AVX引入了16个256位寄存器(YMM0至YMM15)

AVX的256位寄存器和SSE的128位寄存器存在着相互重叠的关系(XMM寄存器为YMM寄存器的低位)，所以最好不要混用AVX与SSE指令集，否在会导致transition penalty(过渡处罚)

数据类型通常以\_mxxx(T)的方式进行命名，其中

1. \_m是固定前缀
2. xxx代表数据的位数，如\_\_m128为128位（SSE提供），\_\_m256为256位（AVX提供）；
3. T为类型，若为整形则为i，若为单精度浮点型则省略，若为双精度浮点型则为d，如\_\_m128i表示128位整数，\_\_m256d表示256位双精度浮点型。

以下为msvc中的数据类型定义：

typedef union \_\_declspec(intrin\_type) \_\_declspec(align(16)) \_\_m128i {

\_\_int8 m128i\_i8[16];

\_\_int16 m128i\_i16[8];

\_\_int32 m128i\_i32[4];

\_\_int64 m128i\_i64[2];

unsigned \_\_int8 m128i\_u8[16];

unsigned \_\_int16 m128i\_u16[8];

unsigned \_\_int32 m128i\_u32[4];

unsigned \_\_int64 m128i\_u64[2];

} \_\_m128i;

typedef union \_\_declspec(intrin\_type) \_\_declspec(align(16)) \_\_m128 {

float m128\_f32[4];

\_\_int8 m128\_i8[16];

\_\_int16 m128\_i16[8];

\_\_int32 m128\_i32[4];

\_\_int64 m128\_i64[2];

unsigned \_\_int8 m128\_u8[16];

unsigned \_\_int16 m128\_u16[8];

unsigned \_\_int32 m128\_u32[4];

unsigned \_\_int64 m128\_u64[2];

} \_\_m128;

typedef union \_\_declspec(intrin\_type) \_\_declspec(align(32)) \_\_m256i {

\_\_int8 m256i\_i8[32];

\_\_int16 m256i\_i16[16];

\_\_int32 m256i\_i32[8];

\_\_int64 m256i\_i64[4];

unsigned \_\_int8 m256i\_u8[32];

unsigned \_\_int16 m256i\_u16[16];

unsigned \_\_int32 m256i\_u32[8];

unsigned \_\_int64 m256i\_u64[4];

} \_\_m256i;

typedef union \_\_declspec(intrin\_type) \_\_declspec(align(32)) \_\_m256 {

float m256\_f32[8];

} \_\_m256;

typedef struct \_\_declspec(intrin\_type) \_\_declspec(align(32)) \_\_m256d {

double m256d\_f64[4];

} \_\_m256d;

SSE/AVX支持的函数的命名规则：

操作整形的内置函数通常以\_mm(xxx)\_name\_epUY的方式进行命名，其中

1. xxx为SIMD寄存器的位数，若省略则表示\_m128；
2. name为函数操作名字；
3. ep为固定字串
4. U，取值为u或者i，代表整数的类型，若为u则为无符号类型，若为i则为有符号类型，如\_mm\_adds\_epu16和\_mm\_adds\_epi16；
5. Y，代表操作的数据类型的位数

操作浮点数的内置函数通常以\_mm(xxx)\_name\_PT的方式进行命名，其中

1. xxx为SIMD寄存器的位数，若省略则表示\_m128，如\_mm\_addsub\_ps，若为256则表示\_256m，如\_mm256\_add\_ps；
2. name为函数操作名字，如加法运算\_mm\_add\_ps，减法运算\_mm\_sub\_ps；
3. P，取值为s或p，代表是对标量(scalar)还是矢量(packed data vector)进行操作，如\_mm\_add\_ss是只对最低位的32位浮点数执行加法，而\_mm\_add\_ps则是对4个32位浮点数都执行加法操作；
4. T，取值为s或d，代表浮点数的类型，若为s则为单精度浮点型，若为d则为双精度浮点，如\_mm\_add\_pd和\_mm\_add\_ps。

右上述规则可知，函数名第三个部分：

\_epixx表示操作向量中所有的xx位的有符号整型数据，向量寄存器长度为128位；

\_epuxx表示操作向量中所有的xx位的无符号整形数据，向量寄存器长度为128位；

\_ps表示操作向量中所有的单精度数据；

\_pd表示操作向量中所有的双精度数据；

\_ss表示只操作向量中第一个单精度数据；

此外：

\_pixx表示操作向量中所有的xx位的有符号整型数据，向量寄存器长度为64位；

\_si64表示操作向量寄存器中第一个64位的有符号整型数据；

\_si128表示操作向量寄存器中第一个128位的有符号整型数据。

SSE/AVS指令示例：

算术运算：

* \_\_m128 \_mm\_cmpeq\_ps(\_\_m128 \_A, \_\_m128 \_B);

功能：逐分量比较4个浮点数是否相等

The result in the output vector will be 0xffffffff if the input elements were equal, or 0 otherwise

\_\_m128 v0 = \_mm\_set\_ps(4.0f, 3.0f, 2.0f, 1.0f);

\_\_m128 v1 = \_mm\_set\_ps(8.0f, 7.0f, 2.0f, 5.0f);

\_\_m128 r3 = \_mm\_cmpeq\_ps(v0, v1);//逐分量比较是否相等，{0,-nan,0,0}，即00 00 00 00 ff ff ff ff 00 00 00 00 00 00 00 00

* \_\_m128 \_mm\_dp\_ps(\_\_m128, \_\_m128, const int /\* mask \*/);

功能：计算点积，并考虑mask

First performs a SIMD multiplication of the lower four packed single-precision floating-point elements (float32 elements) from the first source vector m1 with corresponding elements in the second source vector m2.

Each of the four resulting single-precision elements is conditionally summed depending on the high four bits in the mask parameter.

The resulting summed value is broadcast to each of the lower 4 positions in the destination vector, if the corresponding lower bit of the mask is "1". If the corresponding lower bit of the mask is zero, the corresponding lower element in the destination vector is set to zero.

The process is then replicated with the high elements of the source vectors.

\_\_m128 v0 = \_mm\_set\_ps(4.0f, 3.0f, 2.0f, 1.0f);

\_\_m128 v1 = \_mm\_set\_ps(8.0f, 7.0f, 6.0f, 5.0f);//将4个32位浮点数按相反顺序赋值给\_\_m128中的4个浮点数

\_\_m128 r7 = \_mm\_dp\_ps(v0, v1, 0xff); //点乘，{70,70,70,70}

\_\_m128 r8 = \_mm\_dp\_ps(v0, v1, 0xf0); //点乘，{0,0,0,0}

\_\_m128 r9 = \_mm\_dp\_ps(v0, v1, 0xf4); //点乘，{0,0,70,0}

\_\_m128 r10 = \_mm\_dp\_ps(v0, v1, 0x0f); //点乘，{0,0,0,0}

\_\_m128 r11 = \_mm\_dp\_ps(v0, v1, 0x1f); //点乘，{5,5,5,5}

\_\_m128 r12 = \_mm\_dp\_ps(v0, v1, 0x3f); //点乘，{17,17,17,17}

逻辑运算：

* \_mm\_and\_pd对两个数据逐分量and
* \_mm\_andnot\_ps先对第一个数进行not，然后再对两个数据进行逐分量and
* \_mm\_or\_pd对两个数据逐分量or
* \_mm\_xor\_ps对两个数据逐分量xor

{

\_declspec(align(16)) uint8\_t f0[16] = { 0x00, 0x01, 0x03, 0x11 };

\_declspec(align(16)) uint8\_t f1[16] = { 0x01, 0x01, 0x02, 0x44 };

\_\_m128 vf0 = \_mm\_load\_ps((float\*)f0);

\_\_m128 vf1 = \_mm\_load\_ps((float\*)f1);

\_\_m128 r1 = \_mm\_andnot\_ps(vf0, vf1); //01 00 00 44 00 00 00 00 00 00 00 00 00 00 00 00

\_\_m128 r2 = \_mm\_xor\_ps(vf0, vf1); // 01 00 01 55 00 00 00 00 00 00 00 00 00 00 00 00

\_declspec(align(16)) uint8\_t d0[16] = { 0x00, 0x01, 0x03, 0x11 };

\_declspec(align(16)) uint8\_t d1[16] = { 0x01, 0x01, 0x02, 0x44 };

\_\_m128d vd0 = \_mm\_load\_pd((double\*)d0);

\_\_m128d vd1 = \_mm\_load\_pd((double\*)d1);

\_\_m128d r3 = \_mm\_and\_pd(vd0, vd1); //00 01 02 00 00 00 00 00 00 00 00 00 00 00 00 00

\_\_m128d r4 = \_mm\_or\_pd(vd0, vd1);//01 01 03 55 00 00 00 00 00 00 00 00 00 00 00 00

printf("0x%x\n", \*(unsigned\*)&vd0);

}

shuffle运算：

* \_mm\_shuffle\_ps读取两个\_\_m128类型的数据a和b，并按照\_MM\_SHUFFLE提供的索引将返回的\_\_m128类型数据的低两位设置为a中按索引值取得到的对应值，将高两位设置为按索引值从b中取得到的对应值。索引值在0到3之间，分别以相反的顺序对应\_\_m128中的四个浮点数

\_declspec(align(16)) float p0[] = { 1, 2, 3, 4 };

\_declspec(align(16)) float p1[] = { 5, 6, 7, 8 };

\_\_m128 a = \_mm\_load\_ps(p0);

\_\_m128 b = \_mm\_load\_ps(p1);

\_\_m128 v0 = \_mm\_shuffle\_ps(a, b, \_MM\_SHUFFLE(1, 0, 3, 2)); // 3, 4, 5, 6

\_\_m128 v1 = \_mm\_shuffle\_ps(a, b, \_MM\_SHUFFLE(2, 2, 3, 3)); // 4, 4, 7, 7

\_\_m128 v2 = \_mm\_shuffle\_ps(a, b, \_MM\_SHUFFLE(1, 3, 2, 3)); //4, 3, 8, 6

* SSSE3 instruction \_mm\_shuffle\_epi8

\_\_m128i \_mm\_shuffle\_epi8(

\_\_m128i a,

\_\_m128i mask

);

参数：

[in] a：A 128-bit parameter that contains sixteen 8-bit integers.

[in] mask：A 128-bit byte mask.

返回值：

The return value can be expressed by the following equations:

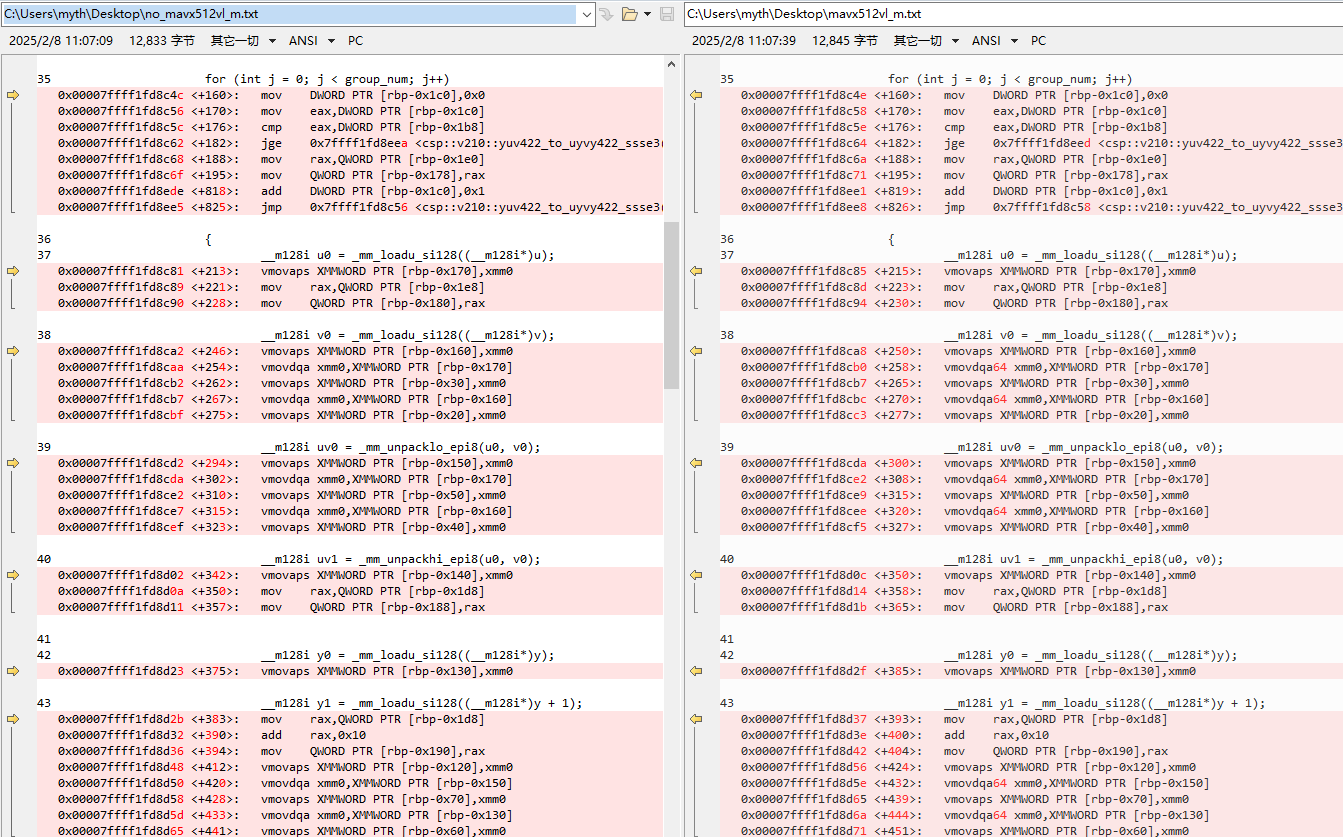
r0 = (mask0 & 0x80) ? 0 : SELECT(a, mask0 & 0x0f)

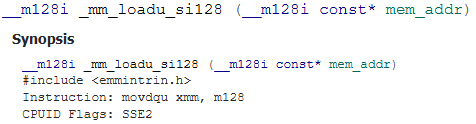
r1 = (mask1 & 0x80) ? 0 : SELECT(a, mask1 & 0x0f)

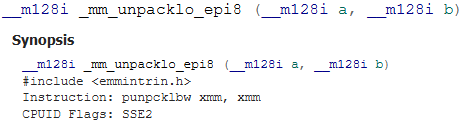
...

r15 = (mask15 & 0x80) ? 0 : SELECT(a, mask15 & 0x0f)

低等级的simd intrinsic函数的实现可能会用到高等级simd指令：

如上图，





左边程序的gcc编译选项为-msse2 -msse3 -msse4 -mavx2 -mavx512f

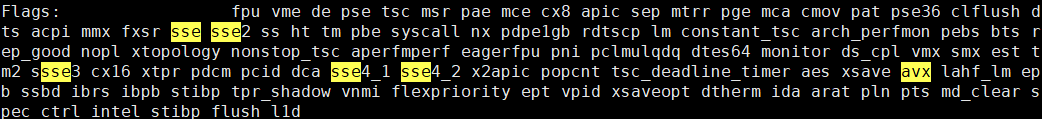
右边程序的gcc编译选项为-msse2 -msse3 -msse4 -mavx2 -mavx512f -mavx512vl

vmovdqa的CPUID Feature Flag为AVX

vmovdqa64的CPUID Feature Flag为AVX512F或AVX512VL

（https://www.felixcloutier.com/x86/movdqa:vmovdqa32:vmovdqa64）

右边编译出的二进制，无法在如下cpu上运行，gdb下报错“Program received signal SIGILL, Illegal instruction”



**内存虚拟化：**

进程虚拟内存，经cpu芯片上的MMU查页表(PT)或快表，被翻译为物理内存

如果进程是运行在虚拟机中的，那么需要对虚拟内存再进行虚拟化，即VMM(vitualizing virtualized memory)。

虚拟机中，虚拟内存转换到的物理地址是一个中间的物理地址（Intermediate Phyical Address, IPA），需要经过VMM/hypervisor的转换，才能得到最终的物理地址（Host Phyical Address, HPA）。

从VMM的角度，guest VM中的虚拟地址即GVA(Guest Virtual Address)，IPA即GPA(Guest Phyical Address)。



GVA->GPA->HPA转换的实现方案：

（1）软件实现—影子页表：

计算出GVA->HPA的映射关系，将其写入一个单独的影子页表（sPT, shadow Page Table）。

* GVA->GPA：guest VM中的每个进程都有一个由内核维护的页表（记为gPT），用于GVA->GPA的转换
* GPA->HPA：VMM层的软件会将gPT本身使用的物理页面设为write protected的，那么每当gPT变动（比如添加或删除了一个页表项），就会产生被VMM截获的page fault异常，之后VMM重新计算GVA->HPA的映射，更改sPT中对应的页表项。

影子页表方案存在两个缺点：

* 实现较复杂，需要为每个guest VM中的每个进程的gPT都维护一个对应的sPT，增加了内存的开销；
* VMM使用的截获方法增多了page fault和trap/vm-exit的数量，加重了CPU的负担。

（2）硬件辅助—EPT/NPT：

CPU厂商推出了硬件辅助的内存虚拟化技术，如Intel的EPT(Extended Page Table)和AMD的NPT(Nested Page Table），它们都能够从硬件上同时支持GVA->GPA和GPA->HPA的地址转换。

* GVA->GPA：依然通过查找gPT页表完成；
* GPA->HPA：通过查找nPT页表来完成。每个guest VM有一个由VMM维护的nPT。其实，EPT/NPT就是一种扩展的MMU，它可以交叉地查找gPT和nPT两个页表。





**IO虚拟化：**

在虚拟化系统中，I/O外设只有一套，需要被多个guest VMs共享。VMM/hypervisor提供了两种机制来实现对I/O设备的访问，一种是透传（passthrough），一种是模拟（emulation）

## device passthrough

passthrough是指guest VM可以透过VMM，直接访问I/O硬件，这样guest VM的I/O操作路径和非虚拟化环境下的I/O路径几乎相同，性能自然是非常高的

MMIO：内存映射I/O，IO设备内存和物理内存统一编址，映射到进程的虚拟内存空间

在虚拟化环境下，guest VM使用的物理地址是GPA，如果直接用guest OS中的驱动程序去操作I/O设备的话（这里的I/O限定为MMIO），那么设备使用的地址也是GPA。容易想到，使用CPU中的EPT/NPT MMU查询对应guest VM的nPT页表，进行一下GPA->HPA的转换就可以了。

但有一些I/O设备是具备DMA(Direct Memory Access)功能的。DMA直接在设备和物理内存之间传输数据，最终使用的必须是实际的物理地址（即HPA）。而DMA本身是为了减轻CPU的处理负担而存在的，其传输过程并不经过CPU，因此无法使用CPU中的EPT/NPT MMU。

那如何实现对DMA控制器发出的地址进行GPA->HPA的转换呢？再来一个类似于EPT/NPT的MMU？没错，这种专用于I/O设备转换地址的硬件单元在x86的阵营里即IOMMU。IOMMU为AMD的命名，Intel的命名为VT-d，即Virtualization Technology for Direct I/O。

IOMMU查找的页表通常是专门的I/O page tables。既然都是进行GPA->HPA的转换，为什么不和EPT/NPT MMU共享nPT页表呢？这是因为使用专门的I/O页表可以获得更快的查找速度。

为了加速查找过程，IOMMU中也有类似于EPT/NPT TLB的IOTLB硬件单元。

这里为了支持device passthrough模式下的DMA传输，IOMMU进行的是GPA->HPA的转换。既然EPT/NPT MMU都可以同时支持GVA->GPA和GPA->HPA的转换，那IOMMU是否也可以呢？答案是肯定的：

* 在虚拟化的环境中，借助IOMMU的GPA->HPA转换，DMA控制器可以直接以guest VM提供的GPA作为source或者destination，减少了VMM的参与；
* 在非虚拟化的环境中，借助IOMMU的HVA->HPA转换，DMA控制器也可以直接以user space buffer的HVA作为source/destination，而不需要OS kernel将user process的VA转换后的PA填入DMA控制器。

Intel的VT-d规定了一个domain对应一个IO页表。在具体的实现中，通常是一个guest VM作为一个domain，因此分配给同一个guest VM的设备将共享同一个IO页表。



device passthrough机制要求VMM为guestVM分配好设备，并提供隔离。假设系统中现在有三个guest VMs，编号分别是0,1,2，如果VM0分配到了网卡A，就要阻止VM1和VM2对网卡A的访问。

可以采用的方法是在拥有设备的guest VM加载驱动程序前，先给要分配出去的设备加载一个伪驱动作为占位符，由于没有真正的驱动程序，这个设备对于其他的guest VM来说就相当于是“隐藏”了。

这同时也暴露了使用device passthrough存在的一个问题，就是同一个I/O设备通常无法在不同的guest VM之间实现共享和动态迁移（比如PCI设备的热插拔）

**GPU底层架构：**

原文：Understanding the architecture of a GPU（<https://medium.com/codex/understanding-the-architecture-of-a-gpu-d5d2d2e8978b>）

CPU和GPU架构示意图如下：



图中，绿色代表computational units（可计算单元）或者称之为cores（核心），

橙色代表memories（内存），

黄色代表的是controlunits（控制单元）

## Computational units(cores)

CPU的Computational units是“大”而“少”的，然而GPU的Computational units是“小”而“多”的。

多少指数量，大小指算力。衡量计算能力的一个重要指标是GFLOP (Giga Floating-point Operations Per Second)，32 位浮点为Single Precision，64 位浮点为 Double Precision。

CPU的core比GPU的core更快，更聪明(smarter)，这也即所谓“大”的特点：

* 在过去的很长时间里，CPU core计算能力的增长得益于主频时钟的频率增长。相反，GPU不仅没有主频时钟的提升，而且还经历过主频下降的情况，因为GPU需要适应嵌入式应用环境，在这个环境下对功耗的要求比较高，不能容忍超高主频的存在。
* CPU比GPU聪明，很大一个原因就是CPU拥有"out-of-order exectutions"（乱序执行）功能，能进行指令重排序、分支预测。相比之下，GPU的core不能做任何类似out-of-order exectutions这样复杂的事情，而只能做一些简单的浮点运算，例如multiply-add(MAD)指令、fused multiply-add(FMA)。

floatn mad(floatn a,floatn b,floatn c)：返回a\*b+c，对中间的乘法结果进行了近似取舍，运行速度快

floatn fma(flaotn a,floatn b, floatn c)：返回a\*b+c，不对中间结果运算结果作近似取舍，精度更高

实际上，现代GPU core除了执行MAD、FMA这样简单的运算外，也支持更加复杂的操作，如张量 (tensor core，旨在服务深度学习场合)、光线追踪(ray tracing core，旨在服务hyper-realistic实时渲染的场合)。但总体来说GPU core的计算灵活性还是比不上CPU core。

tensor core的张量运算示意图如下（即矩阵乘法加及加法）：



## Memories

CPU的存储系统由寄存器、cache（SRAM）、主存（DRAM）层次结构组成。

从上文图可以看到，GPU中有一大片橙色的内存，为DRAM，这一块被称为全局内存（GMEM）。

左上角的小橙色块是GPU的cache

## SM结构：

如果是长度为8的数组两两并行求和计算，如下图计算方式，那么只需要三个时钟周期就可以计算出结果。图中绿色圆表示一个GPU core，数字为core编号。第一个时钟下，两两相加的结果通过0号core计算，放入了0号core可以访问到的显存中，以此类推……

如果GPU想要完成图示计算过程，显然，多个core之间要共享一段显存空间以此来完成数据之间的交互，需要多个core可以在共享的显存空间中完成读/写操作。

我们希望每个Cores都有交互数据的能力，但是不幸的是，一个GPU里面包含数以千计的core，使这些core都可以访问共享的显存段是非常困难和昂贵的。

出于成本的考虑，折中的解决方案是将各类GPU的core分类为多个组，形成多个流处理器(Streaming Multiprocessor，简称为SM)

引入SM后的GPU架构如下图：



上图的绿色部分为Core计算单元，绿色的块为Streaming Multiprocessor，理解为Core的集合。离SMs非常近的黄色部分名为RT COREs。单个SM的图灵架构如下图所示：

在SM的图灵结构中，绿色的部分CORE相关的，我们进一步区分了不同类型的CORE。主要分为INT32、FP32、TENSOR CORES等。

FP32 Cores：执行单精度浮点运算，在TU102卡中，每个SM有64个FP32核，TU120由72个SMs组成，因此，FP32 Core的数量是72 \* 64。

FP64 Cores：每个SM都包含2个64位浮点计算核心，用来计算双精度浮点运算，虽然图中没有画出，但实际是存在的。

Integer Cores：执行一些对整数的操作，例如地址计算，可以和浮点运算同时执行指令。在前几代GPU中，执行这些整型操作指令都会使得浮点运算的管道停止工作。在TU102卡中，每个SM有64个Integer核，总共有72\*64=4608个Integer Cores。

Tensor Cores：张量core是FP16单元的变种，认为是半精度单元，致力于张量计算加速常见的深度学习操作。张量Core还可以执行INT8和INT4精度的操作，用于可以接受量化而且不需要FP16精度的应用场景，在TU102中，每个SM有8个张量Core，一共有72\*8 个Tensor Cores。

各个核心之间如何完成彼此的协作？

在SM块的底部有一个96KB的L1 Cache，用蔚蓝色标注。这个cache段是允许各个Core都可以访问的段。每个SM都有一个专用的L1 Cache。芯片上的L1 cache大小是很有限的，但访问它非常快，比起GMEM快得多。

实际上L1 cache有两个功能，一个是用于SM上Core之间相互共享内存，另一个则是普通的cache功能。

当Core需要协同工作，并且彼此交换结果的时候，编译器编译后的指令会将部分结果储存在共享内存中，以便于不同的core获取到对应数据。

当用做普通cache功能的时候，当core需要访问GMEM数据的时候，首先会在L1中查找，如果没找到，则回去L2 cache中寻找，如果L2 cache也没有，则会从GMEM中获取数据，L1访问最快，L2 以及GMEM递减。缓存中的数据将会持续存在，除非出现新的数据做替换。从这个角度来看，如果Core需要从GMEM中多次访问数据，那么编程者应该将这块数据放入L1中，以加快读写速度。其实可以将共享内存理解为一段受控制的cache，事实上L1 cache和共享内存是同一块电路中实现的。编程者有权决定L1 的内存多少是用作cache多少是用作共享内存。



半导体IP指已验证的、可重复利用的、具有某个确定功能的集成电路模块，可以在做芯片设计的时候可以直接使用，类似软件编程时候的动态库静态库。

GPU内部往往有多个IP，比如用于图形编码的、解码的，有些IP上会运行firmware，为了让它们对物理内存的访问互不影响，也可能使用虚拟地址，经GPU内部的地址翻译单元（即GpuMmu），转换成物理地址（如果是访问system memory，理论上也可以使用CPU侧的IOMMU来转换）。

每个SM/CU有自己的TLB，但共享Page table worker。相比于CPU的TLB平均10%的miss rate，GPU的translation量更大，TLB miss rate可高达30%



**理解二级指针**

linus-torvalds：

At the opposite end of the spectrum, I actually wish more people understood the really core low-level kind of coding. Not big, complex stuff like the lockless name lookup, but simply good use of pointers-to-pointers etc. For example, I've seen too many people who delete a singly-linked list entry by keeping track of the "prev" entry, and then to delete the entry, doing something like

if (prev)

prev->next = entry->next;

else

list\_head = entry->next;

and whenever I see code like that, I just go "This person doesn't understand pointers". And it's sadly quite common.

People who understand pointers just use a "pointer to the entry pointer", and initialize that with the address of the list\_head. And then as they traverse the list, they can remove the entry without using any conditionals, by just doing a "\*pp = entry->next".

struct node

{

int data;

node\* next;

};

typedef bool(\*remove\_fn)(node const \* v);

// Remove all nodes from the supplied list for which the

// supplied remove function returns true.

// Returns the new head of the list.

node \* remove\_if(node \* head, remove\_fn rm){

/\*这里需要判断待删除的node是否头结点，导致了额外的指针定义（prev）和不必要的if判断（if (prev)\*/

for (node \* prev = NULL, \*curr = head; curr != NULL; ){

node \* const next = curr->next;

if (rm(curr)){

if (prev)

prev->next = next;

else

head = next;

free(curr);

}else

prev = curr;

curr = next;

}

return head;

}

void remove\_if(node \*\* head, remove\_fn rm){

/\*通过使用二级指针，可以一致的对待头结点和其它节点，并且函数不需要再返回值了\*/

for (node\*\* curr = head; \*curr; ){

node \* entry = \*curr;

if (rm(entry)){

\*curr = entry->next;

free(entry);

}

else{

curr = &entry->next;

}

}

}

**汇编：**

要让C++代码编译时生成汇编：配置属性->C/C++->命令行->添加”/FAs”

vs2017配置MASM汇编环境：

* 1. 建一个空项目
  2. 在项目里面新添加一个源文件test.asm（实际是添加一个cpp文件，但是重命名为test.asm）
  3. 在test.asm中输入一段64位汇编程序

;vs2017x64下，配置属性->链接器->高级->入口点为main

ExitProcess proto

.code

main proc

 mov eax,1234h  ;把十六进制1234放入eax寄存器

 mov ecx,0

 call ExitProcess

main endp

end

* 1. 把目标平台改为x64
  2. 点击项目生成，没有反应，因为默认是C++项目，不支持MASM汇编
  3. 为了编译此项目，需要让VC能编译汇编。右键点击test.asm，调出属性菜单，配置属性->常规->项类型，显示是“不参与生成”。显然，此asm文件没有参与项目的生成，那么在这里能不能把MASM加进去呢？不行，VC默认是没有MASM编译选项的，下拉框列表里没有。
  4. 右键点击项目->生成依赖项->生成自定义，然后在弹出的窗口中，选中masm选项，点击确定，此时VC项目就能够编译MASM汇编了
  5. 添加程序入口点。右键点击项目->属性->链接器->高级->入口点->设置为main
  6. 再次点击test.asm文件的属性，在项类型下拉框中选择：Microsoft Macro Assembler
  7. 重新生成项目，成功
  8. 配置代码颜色高亮及智能提示。安装扩展，工具->扩展和更新，搜索AsmDude，安装。

32位汇编示例：



vs中自动窗口和局部变量窗口中不显示变量，但添加到监视可以查看变量值及地址

反汇编后也会显示变量地址

;vs2017x86下，配置属性->链接器->高级->入口点为空

;该程序对数组{1,2,3,4,5}求和

.386              ;告诉汇编器生成的目标代码时使用Intel 80386指令集

.model flat,stdcall      ;指定内存模型为flat（扁平内存模型），使用stdcall调用约定

.stack 4096      ;运行时为栈保留4096字节内存空间

ExitProcess PROTO,dwExitCode:*DWORD*

.data

array *DWORD* 1h,2h,3h,4h,5h  ;声明数组array={1,2,3,4,5}，元素为DWORD类型（32bit）

theSum *DWORD* ?

.code

main PROC

    mov esi,OFFSET array          ;ESI中存放数组array的地址

    mov ecx,LENGTHOF array        ;ECX 中存放数组长度

    call ArraySum    ;跳转到ArraySum，ArraySum的值即函数中第一条指令push esi的地址，且栈增长4个字节

    mov theSum,eax                ;用EAX中的值保存到地址theSum

    INVOKE ExitProcess,0

main ENDP

;------------------------------------

;ArraySum

;计算int32数组元素之和

;接收：ESI = 数组偏移量

;       ECX = 数组元素的个数

;返回：EAX = 数组元素之和

;-------------------------------------

ArraySum PROC

    push esi                 ;保存ESI和ECX，即将其中的值压栈，后续将对这两个寄存器中的值进行修改

    push ecx

    mov eax,0                ;设置和数为0，记录在eax寄存器中

L1:   add eax,[esi]        ;给eax寄存器中的值加上，esi寄存器中的内存地址处的值

    add esi,TYPE *DWORD*      ;给esi寄存器中的值加上4，指向数组中下一个元素

    loop L1                 ;ECX寄存器中的值减1，然后跳转到L1处重复执行，如果发现减一后为0，则不重复，执行下一句

    pop ecx                 ;弹出栈顶值到EXC，即恢复ECX

    pop esi               ;弹出栈顶值到ESI，即恢复ESI

    ret                     ;继续将栈顶4个字节弹栈

ArraySum ENDP

END  main

call时栈增长4字节，ret时栈收缩4字节，是指令指针寄存器的压栈、出栈

整数常量语法：

26d：十进制

11010011b ：二进制

42o ：八进制

0A3h ：十六进制

C语言调用MASM汇编函数：

创建main.cpp，内容如下：

#include <stdio.h>

extern "C" int ArraySum(int a[], int len);

int main()

{

    int a[] = { 1,3,5,7};

    int sum = ArraySum(a, sizeof(a) / sizeof(int));

    printf("sum=%d\n", sum);

}

创建ArraySum.asm，内容如下：

.386

.model flat,c

.code

ArraySum PROC

    push ecx

    push ebx

    mov ecx,[esp+16]    ;数组长度保存到ecx

    mov ebx,0                ;设置和数为0，记录在ebx寄存器中

L1:   add ebx,[eax]        ;给ebx寄存器中的值加上，eax寄存器中的内存地址处的值

    add eax,TYPE *DWORD*      ;给eax寄存器中的值加上4，指向数组中下一个元素

    loop L1                 ;ECX寄存器中的值减1，不为0则跳转到L1处循环

    mov eax,ebx  ;将ebx中的值赋给eax

    pop ebx

    pop ecx

    ret

ArraySum ENDP

end

编译运行即可

注意，上述.asm文件中“.model flat,c”行设置调用约定为c，而非标准调用约定stdcall

调用约定描述了函数参数的传递规则，包括参数压栈顺序、使用哪些寄存器等

反汇编main.cpp文件：



如上，可以发现，在.cpp中调用汇编函数ArraySum时：

* + - 先依次将数组长度（编译期值）、数组地址压栈，然后放置数组地址到eax寄存器中，最后跳转到指令(\_ArraySum)处执行，\_ArraySum即汇编函数中的第一条指令的地址
    - 在call指令之后有一条mov dword ptr [sum],eax指令，所以函数返回值在eax中

所以在.asm中实现汇编函数时：

* + - 对于函数参数，数组长度必须从栈里获取，数组地址既可以从eax寄存器获取也可以从栈里获取
    - 函数实现求和功能时会用到一些寄存器（ecx、ebx），所以在函数开头需要对这些寄存器的值压栈保存，函数跳出时弹栈恢复，从而对调用者来说，调用函数后，不会出现某些寄存器的值被意外修改。

NASM汇编：

ubuntu+gcc+nasm汇编环境：

install nasm

install gcc-multilib

C++程序first.cpp：

int main() {

return 0;

}

对应的汇编程序first.asm：

global main

main:

mov eax, 0

ret

编译运行：

$ nasm -f elf first.asm -o first.o

$ gcc -m32 first.o -o first

$ ./first ; echo $? #打印出0

按照约定，eax中的值，就是函数的返回值

$?表示的是上一个命令、函数、或脚本的退出状态码值

因此，如果是mov eax 30，则echo打印出30

mov、add\sub、寄存器、内存、.data数据段示例：

global main

main:

    mov ebx, 1

    mov ecx, 2

    add ebx, ecx

    mov eax, [ga]

    add eax, [gb]

    sub eax, ebx

    mov [gr], eax

    ret

section .data

ga dw 7

gb dw 10

gr dw 0

./main ; echo $?输出14

mov指令也可以直接将常数mov到内存中：

mov eax, 100

mov [gr], eax

如果直接被精简为语句：

mov [gr], 100

则nasm汇编时会报错：error: operation size not specified

如下修改则正常：

mov dword [gr], 100

gdb查看某个寄存器的值：info register eax或p $eax

gdb查看所有寄存器的值：info registers

gdb反汇编：disas <func name>

gdb disas可以看到指令的地址，但此时直接b \*<addr>会提示不能打上断点，只有程序run起来后，才能设置上断点

gdb ./main

gdb-peda$ disas main

Dump of assembler code for function main:

   0x565561a0 <+0>:     mov    ebx,0x1

   0x565561a5 <+5>:     mov    ecx,0x2

   0x565561aa <+10>:    add    ebx,ecx

   0x565561ac <+12>:    mov    eax,ds:0x4008

   0x565561b1 <+17>:    add    eax,DWORD PTR ds:0x400a

   0x565561b7 <+23>:    sub    eax,ebx

   0x565561b9 <+25>:    mov    ds:0x400c,eax

   0x565561be <+30>:    ret

   0x565561bf <+31>:    nop

End of assembler dump.

gdb-peda$ b main

Breakpoint 3 at 0x565561a0

gdb-peda$ r

[-------------------------------------code-------------------------------------]

   0x5655619d <\_\_x86.get\_pc\_thunk.dx+4>:        xchg   ax,ax

   0x5655619f <\_\_x86.get\_pc\_thunk.dx+6>:        nop

=> 0x565561a0 <main>:   mov    ebx,0x1

   0x565561a5 <main+5>: mov    ecx,0x2

gdb-peda$ b \*0x565561b7

gdb-peda$ c

[-------------------------------------code-------------------------------------]

   0x565561ac <main+12>:        mov    eax,ds:0x56559008

   0x565561b1 <main+17>:        add    eax,DWORD PTR ds:0x5655900a

=> 0x565561b7 <main+23>:        sub    eax,ebx

   0x565561b9 <main+25>:        mov    ds:0x5655900c,eax

gdb-peda$ info register eax

eax            0xa0011             0xa0011

gdb-peda$ info register ebx

ebx            0x3                 0x3

gdb-peda$ n

[-------------------------------------code-------------------------------------]

   0x565561b1 <main+17>:        add    eax,DWORD PTR ds:0x5655900a

   0x565561b7 <main+23>:        sub    eax,ebx

=> 0x565561b9 <main+25>:        mov    ds:0x5655900c,eax

   0x565561be <main+30>:        ret

gdb-peda$ info register eax

eax            0xa000e             0xa000e

流程控制，比较、跳转指令：

C++程序：

int main() {

    int a = 50;

    if( a > 10 ) {

        a = a - 10;

    }

    return a;

}

等价于汇编程序：

global main

main:

    mov eax, 50

    cmp eax, 10    ; 对eax和10进行比较

    jle END            ; 小于或等于的时候跳转

    sub eax, 10

END:

    ret

CPU里面有一个专用的寄存器eflags，用来专门“记住”cmp指令的比较结果

后续的跳转指令，根据eflags寄存器中的状态，来决定是否要进行跳转的

跳转指令首字母都是j，后边的字母：

a: above

e: equal

b: below

n: not

g: greater

l: lower

s: signed

z: zero

因此有：

ja 大于时跳转

jae 大于等于

jb 小于

jbe 小于等于

je 相等

jna 不大于

jnae 不大于或者等于

jnb 不小于

jnbe 不小于或等于

jne 不等于

jg 大于(有符号)

jge 大于等于(有符号)

jl 小于(有符号)

jle 小于等于(有符号)

jng 不大于(有符号)

jnge 不大于等于(有符号)

jnl 不小于

jnle 不小于等于

jns 无符号

jnz 非零

js 如果带符号

**函数调用传值、传引用的汇编分析：**

x64平台上主要有两套常用的 ABI：

* 用于64位Windows操作系统上的Windows x64 ABI（https://learn.microsoft.com/en-us/cpp/build/x64-software-conventions?view=msvc-170）
* 用于64位Linux以及一众UNIX-like的操作系统上的x86-64 System V ABI（https://gitlab.com/x86-psABIs/x86-64-ABI）

System V ABI规定了：

* 前六个整形参数，依次可以使用rdi、rsi、rdx、rcx、r8、r9寄存器传递；
* 如果寄存器用完了，就通过栈传递；
* 整形参数即char、short、int、long、long long等基础整数类型外加指针类型；
* 浮点参数和SIMD类型的参数则有专门的寄存器负责

一种常见说法：Linux环境上小于等于16字节大小的结构体值传递效率比引用效率更高

原因分析：如果结构体大小小于等于8字节，可以直接塞进一个64位寄存器里面传参，通过寄存器传参比通过引用传参要少几次访存，效率要高一些。那对于16字节呢？System V ABI 允许将一个16字节大小的结构体拆成两个8个字节的部分，然后分别使用寄存器传递。那么在这种情况下传值也确实比传引用要高一些。观察如下示例代码：

#include <cstdio>

struct X {

    size\_t x;

    size\_t y;

};

extern void f(X);

extern void g(const X&);

int main() {

    f({1, 2}); *// pass by value*

    g({1, 2}); *// pass by reference*

}

*//g++ -O2 -c main.cpp -std=c++11*

*//objdump -d main.o -M intel*

在centos7.7，gcc4.8.5环境编译，机器码为：

[root@zhs study]$ g++ -O2 -c main.cpp -std=c++11

[root@zhs study]$ objdump -d main.o -M intel

main.o:     file format elf64-x86-64

Disassembly of section .text.startup:

0000000000000000 <main>:

   0:   48 83 ec 18             sub    rsp,0x18

   4:   bf 01 00 00 00          mov    edi,0x1

   9:   be 02 00 00 00          mov    esi,0x2

   e:   e8 00 00 00 00          call   13 <main+0x13>

  13:   48 89 e7                mov    rdi,rsp

  16:   48 c7 04 24 01 00 00    mov    QWORD PTR [rsp],0x1

  1d:   00

  1e:   48 c7 44 24 08 02 00    mov    QWORD PTR [rsp+0x8],0x2

  25:   00 00

  27:   e8 00 00 00 00          call   2c <main+0x2c>

  2c:   31 c0                   xor    eax,eax

  2e:   48 83 c4 18             add    rsp,0x18

  32:   c3                      ret

从反汇编结果可以发现，1、2分别通过寄存器edi和esi传递给了f函数，而g则是把临时变量的地址传递给了g函数。

mov rdi,rsp ;rdi即第一个入参，即临时变量的地址为rsp的值

edi和rdi寄存器：rdi寄存器是64位的，edi寄存器的范围为rdi寄存器的低32位

**构造函数和拷贝构造函数的汇编分析：**

C+代码：

#include <cstdio>

struct X {

    size\_t x;

    size\_t y;

    X(int a,int b){x=a;y=b;}

    X(const X& rhs){x=rhs.x;y=rhs.y;printf("copy cons\n");}

};

void f(X a)

{

    printf("f x=%d,y=%d\n",a.x,a.y);

}

int main()

{

    X a{1,2};

    f(a); *// pass by value*

}

生成的汇编：

[root@zhs study]$ g++ -O2 -o main  main.cpp -std=c++11 -fno-inline

[root@zhs study]$ objdump -d main -M intel  | c++filt

00000000004005f0 <X::X(int, int)>:

  4005f0:       48 63 f6                movsxd rsi,esi

  4005f3:       48 63 d2                movsxd rdx,edx

  4005f6:       48 89 37                mov    QWORD PTR [rdi],rsi

  4005f9:       48 89 57 08             mov    QWORD PTR [rdi+0x8],rdx

  4005fd:       c3                      ret

  4005fe:       66 90                   xchg   ax,ax

0000000000400600 <X::X(X const&)>:

  400600:       48 8b 06                mov    rax,QWORD PTR [rsi]

  400603:       48 89 07                mov    QWORD PTR [rdi],rax

  400606:       48 8b 46 08             mov    rax,QWORD PTR [rsi+0x8]

  40060a:       48 89 47 08             mov    QWORD PTR [rdi+0x8],rax

  40060e:       bf b0 06 40 00          mov    edi,0x4006b0

  400613:       e9 78 fe ff ff          jmp    400490 <puts@plt>

  400618:       0f 1f 84 00 00 00 00    nop    DWORD PTR [rax+rax\*1+0x0]

  40061f:       00

00000000004004b0 <main>:

  4004b0:       48 83 ec 28             sub    rsp,0x28

  4004b4:       ba 02 00 00 00          mov    edx,0x2

  4004b9:       be 01 00 00 00          mov    esi,0x1

  4004be:       48 89 e7                mov    rdi,rsp

  4004c1:       e8 2a 01 00 00          call   4005f0 <X::X(int, int)>

  4004c6:       48 8d 7c 24 10          lea    rdi,[rsp+0x10]

  4004cb:       48 89 e6                mov    rsi,rsp

  4004ce:       e8 2d 01 00 00          call   400600 <X::X(X const&)>

  4004d3:       48 8d 7c 24 10          lea    rdi,[rsp+0x10]

  4004d8:       e8 f3 00 00 00          call   4005d0 <f(X)>

  4004dd:       31 c0                   xor    eax,eax

  4004df:       48 83 c4 28             add    rsp,0x28

  4004e3:       c3                      ret

00000000004005d0 <f(X)>:

  4005d0:       48 8b 57 08             mov    rdx,QWORD PTR [rdi+0x8]

  4005d4:       48 8b 37                mov    rsi,QWORD PTR [rdi]

  4005d7:       31 c0                   xor    eax,eax

  4005d9:       bf ba 06 40 00          mov    edi,0x4006ba

  4005de:       e9 8d fe ff ff          jmp    400470 <printf@plt>

  4005e3:       66 2e 0f 1f 84 00 00    nop    WORD PTR cs:[rax+rax\*1+0x0]

  4005ea:       00 00 00

  4005ed:       0f 1f 00                nop    DWORD PTR [rax]

汇编分析：

X a{1,2};对应汇编中main函数第2、3、4、5行，调用构造函数时使用rdi、esi、edx三个寄存器传参，参数分别为分配给待构造对象的内存地址、1、2。

f(a);先在main函数中调用拷贝构造函数生成对象a的临时副本，再调用f()函数。

调用拷贝构造函数，对应汇编中main函数的第6、7、8行，使用rdi、rsi二个寄存器传参，参数分别为分配给待构造对象的内存地址、a的内存地址。

调用f()函数，对应汇编中main函数的第9、10行，使用rdi寄存器传参，参数为上文拷贝构造生成的临时副本的内存地址。

**unique\_ptr和raw\_ptr的效率不一致:**

这里不谈异常导致的额外开销（析构函数导致编译器必须额外生成清理栈帧的代码），仅仅讨论一个C++对象（小于8字节）能使用寄存器传参吗？对于一个完全trivial的类型来说，是没问题的，它表现得和一个C语言的结构体几乎完全一样。不过不满足呢？

比如自定义了拷贝构造函数，还能放寄存器里面吗？其实从逻辑上就不能，为什么呢？我们知道，C++是允许我们对函数参数取地址的，那么如果参数是整形，通过寄存器传参，那取地址的结果哪里来的呢？实验一下：

#include <cstdio>

extern void f(int&);

int g(int x) {

    f(x);

    return x;

}

生成的对应汇编如下：

[root@zhs study]$ g++ -O2 -c main.cpp -std=c++11

[root@zhs study]$ objdump -d main.o -M intel

0000000000000000 <\_Z1gi>:

   0:   48 83 ec 18             sub    rsp,0x18

   4:   89 7c 24 0c             mov    DWORD PTR [rsp+0xc],edi

   8:   48 8d 7c 24 0c          lea    rdi,[rsp+0xc]

   d:   e8 00 00 00 00          call   12 <\_Z1gi+0x12>

  12:   8b 44 24 0c             mov    eax,DWORD PTR [rsp+0xc]

  16:   48 83 c4 18             add    rsp,0x18

  1a:   c3                      ret

可以发现，这里把edi（用于传递第一个整形参数）里面的值拷贝到了rsp+12这个地址，也就是栈上，之后把这个地址传递给了f。也就是说，如果一个函数参数通过寄存器传递，如果在某些情况下需要它的地址，编译器会把这个参数拷贝到栈上。但是无论如何，用户是观察不到这些拷贝过程的，因为它们的拷贝构造函数是trivial的。不影响最终代码执行结果的任何优化都是符合as if原则的。

那么如果这个对象有用户定义的拷贝构造函数，假设参数通过寄存器传递，就可能会导致额外的拷贝构造函数调用，并且用户可以观察到这个副作用。显然这是不合理的，所以不允许拥有自定义拷贝构造函数的对象通过寄存器传参注意。注意**，**给拷贝构造显式标记为delete也算是自定义拷贝构造函数**。**

所以对于unique\_ptr来说，不能通过寄存器传递。但是裸指针却可以通过寄存器安全的传递。综上所述，unique\_ptr的效率逊色于裸指针。

C++代码：

#include <cstdio>

struct X {

    size\_t x;

    size\_t y;

    X(int a,int b){x=a;y=b;}

*//X(const X& rhs){x=rhs.x;y=rhs.y;}*

};

extern void f(X);

extern void g(const X&);

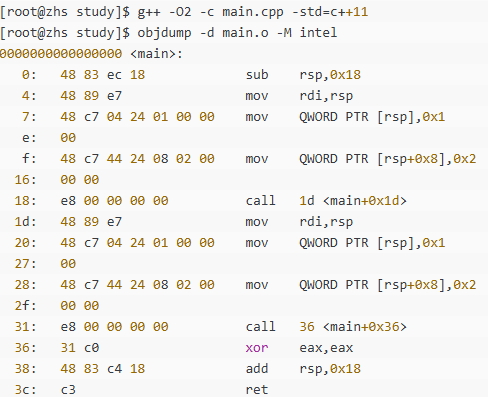
int main() {

    f({1, 2}); *// pass by value*

    g({1, 2}); *// pass by reference*

}

生成的汇编（下左图为原始代码的汇编，下右图为去掉对拷贝构造函数的注释后的汇编）：

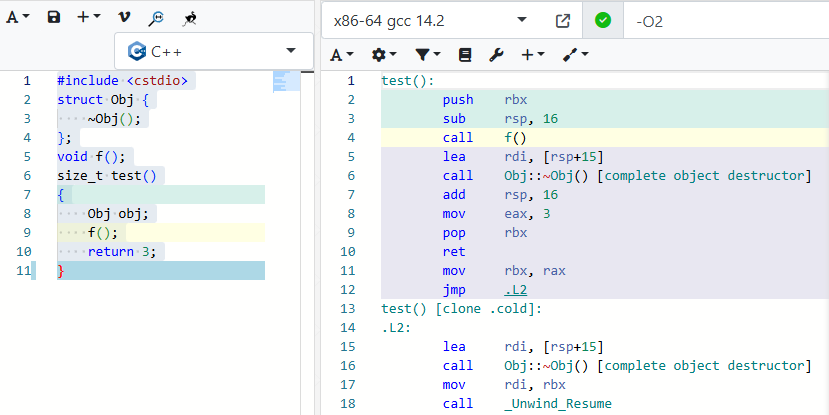
 

可以看到，有用户自定义拷贝构造函数时，未通过寄存器传参，而是传地址。

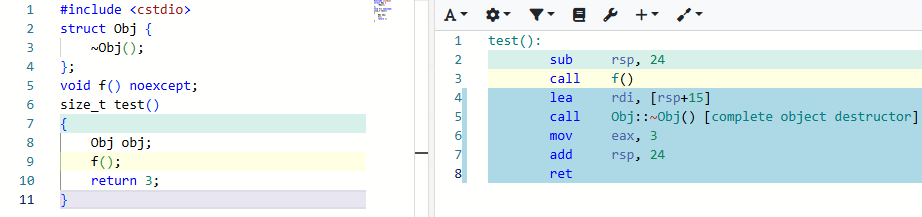
**noexcept优化：**

对于函数声明void f();

编译器无从知道这个函数实际会不会抛异常。所以在为test生成汇编时就要假定会抛异常。万一它抛了异常，test的流程就不再继续往下走，立即做栈回溯，析构掉obj。



将函数声明标为noexcept只是个优化手段，让编译器知道foo里绝不会抛异常，不必额外生成代码来保护栈上对象安全析构。示例如下：



**GDB汇编调试：**

objdump -d <file>：显示executable section的反汇编代码，包括.text段、.plt段等

objdump -d <file> -M intel：以intel风格显示汇编

objdump -d -j .plt <file>：显示.plt段的反汇编代码，即plt表的内容

disas main

disas Rect::Rect

disas 0x00007ffff7df1a30：显示内存地址0x00007ffff7df1a30处的汇编代码

后文以如下源码为例进行说明：

#include <cstdio>

struct X {

    size\_t x;

    size\_t y;

    X(int a,int b){x=a;y=b;}

    X(const X& rhs){x=rhs.x;y=rhs.y;printf("copy cons\n");}

};

void f(X a)

{

    printf("f x=%d,y=%d\n",a.x,a.y);

}

int main()

{

    X a{1,2};

    printf("finish create\n");

    f(a); *// pass by value*

    printf("finish call f\n");

}

设置汇编指令格式：

Intel x86处理器，gdb默认的汇编格式是att

show disassembly-flavor

set disassembly-flavor intel

set disassembly-flavor att

查看源代码的汇编指令：

查看函数的汇编指令：

disas <函数名>，

或者disas /m <函数名>，加上/m参数后，能和源码逐行对应显示

或者disas /r、disas /rm，加上/r参数后，能显示原始机器码

查看某一行源码对应的汇编指令：

info line <行号>

disas <起始内存地址, 截止内存地址>

查看汇编对应的源码行号：info line 0x1151

示例，查看main函数对应的汇编：

(gdb) disas /m main

Dump of assembler code for function main():

13      {

   0x00000000004005cc <+0>:     push   %rbp

   0x00000000004005cd <+1>:     mov    %rsp,%rbp

   0x00000000004005d0 <+4>:     sub    $0x20,%rsp

14          X a{1,2};

   0x00000000004005d4 <+8>:     lea    -0x20(%rbp),%rax

   0x00000000004005d8 <+12>:    mov    $0x2,%edx

   0x00000000004005dd <+17>:    mov    $0x1,%esi

   0x00000000004005e2 <+22>:    mov    %rax,%rdi

   0x00000000004005e5 <+25>:    callq  0x400624 <X::X(int, int)>

15          printf("finish create\n");

   0x00000000004005ea <+30>:    mov    $0x400737,%edi

   0x00000000004005ef <+35>:    callq  0x400490 <puts@plt>

16          f(a); *// pass by value*

   0x00000000004005f4 <+40>:    lea    -0x20(%rbp),%rdx

   0x00000000004005f8 <+44>:    lea    -0x10(%rbp),%rax

   0x00000000004005fc <+48>:    mov    %rdx,%rsi

   0x00000000004005ff <+51>:    mov    %rax,%rdi

   0x0000000000400602 <+54>:    callq  0x400650 <X::X(X const&)>

   0x0000000000400607 <+59>:    lea    -0x10(%rbp),%rax

   0x000000000040060b <+63>:    mov    %rax,%rdi

   0x000000000040060e <+66>:    callq  0x40059d <f(X)>

17          printf("finish call f\n");

   0x0000000000400613 <+71>:    mov    $0x400745,%edi

   0x0000000000400618 <+76>:    callq  0x400490 <puts@plt>

18      }

   0x000000000040061d <+81>:    mov    $0x0,%eax

   0x0000000000400622 <+86>:    leaveq

   0x0000000000400623 <+87>:    retq

End of assembler dump.

(gdb)

示例：查看第16行源码对应的汇编：

(gdb) list 16

11      }

12      int main()

13      {

14          X a{1,2};

15          printf("finish create\n");

16          f(a); *// pass by value*

17          printf("finish call f\n");

18      }

19

(gdb) info line 16

Line 16 of "main.cpp" starts at address 0x4005f4 <main()+40> and ends at 0x400613 <main()+71>.

(gdb) disas 0x4005f4,0x400613

Dump of assembler code from 0x4005f4 to 0x400613:

   0x00000000004005f4 <main()+40>:      lea    -0x20(%rbp),%rdx

   0x00000000004005f8 <main()+44>:      lea    -0x10(%rbp),%rax

   0x00000000004005fc <main()+48>:      mov    %rdx,%rsi

   0x00000000004005ff <main()+51>:      mov    %rax,%rdi

   0x0000000000400602 <main()+54>:      callq  0x400650 <X::X(X const&)>

   0x0000000000400607 <main()+59>:      lea    -0x10(%rbp),%rax

   0x000000000040060b <main()+63>:      mov    %rax,%rdi

   0x000000000040060e <main()+66>:      callq  0x40059d <f(X)>

End of assembler dump.

自动反汇编要执行的代码：

display /i $pc：自动显示将要执行的下一条汇编指令

undisplay：关闭自动显示功能

手动查看将要执行的汇编：

x/i $pc：查看1行汇编

x/4i $pc：查看4行汇编

set disassemble-next-line on：总是显示将要执行的下一行源码的反汇编，如后续执行n、step、si等命令后，自动跟随展示汇编

set disassemble-next-line off：关闭自动反汇编

如下所示，借助set disassemble-next-line on和单步调试n指令，可以方便地看出，每行C++代码，对应哪些汇编指令

(gdb) set disassemble-next-line on

(gdb) b main

Breakpoint 1 at 0x4005d4: file main.cpp, line 14.

(gdb) r

Breakpoint 1, main () at main.cpp:14

14          X a{1,2};

=> 0x00000000004005d4 <main()+8>:       48 8d 45 e0     lea    -0x20(%rbp),%rax

   0x00000000004005d8 <main()+12>:      ba 02 00 00 00  mov    $0x2,%edx

   0x00000000004005dd <main()+17>:      be 01 00 00 00  mov    $0x1,%esi

   0x00000000004005e2 <main()+22>:      48 89 c7        mov    %rax,%rdi

   0x00000000004005e5 <main()+25>:      e8 3a 00 00 00  callq  0x400624 <X::X(int, int)>

(gdb) n

15          printf("finish create\n");

=> 0x00000000004005ea <main()+30>:      bf 37 07 40 00  mov    $0x400737,%edi

   0x00000000004005ef <main()+35>:      e8 9c fe ff ff  callq  0x400490 <puts@plt>

(gdb) n

finish create

16          f(a); *// pass by value*

=> 0x00000000004005f4 <main()+40>:      48 8d 55 e0     lea    -0x20(%rbp),%rdx

   0x00000000004005f8 <main()+44>:      48 8d 45 f0     lea    -0x10(%rbp),%rax

   0x00000000004005fc <main()+48>:      48 89 d6        mov    %rdx,%rsi

   0x00000000004005ff <main()+51>:      48 89 c7        mov    %rax,%rdi

   0x0000000000400602 <main()+54>:      e8 49 00 00 00  callq  0x400650 <X::X(X const&)>

   0x0000000000400607 <main()+59>:      48 8d 45 f0     lea    -0x10(%rbp),%rax

   0x000000000040060b <main()+63>:      48 89 c7        mov    %rax,%rdi

   0x000000000040060e <main()+66>:      e8 8a ff ff ff  callq  0x40059d <f(X)>

(gdb) n

copy cons

f x=1,y=2

17          printf("finish call f\n");

=> 0x0000000000400613 <main()+71>:      bf 45 07 40 00  mov    $0x400745,%edi

   0x0000000000400618 <main()+76>:      e8 73 fe ff ff  callq  0x400490 <puts@plt>

在汇编上打断点：

b main：在main函数人口打断点

b \*main：在main函数的第一条汇编上打断点

其中，\*表示要在一个具体的地址上设置断点，而不是仅仅在函数名上，

如b \*0x4005e2表示在地址0x4005e2处打断点

(gdb) b \*main

Breakpoint 1 at 0x4005cc: file main.cpp, line 13.

(gdb) r

(gdb) disas

Dump of assembler code for function main():

=> 0x00000000004005cc <+0>:     push   %rbp

   0x00000000004005cd <+1>:     mov    %rsp,%rbp

   0x00000000004005d0 <+4>:     sub    $0x20,%rsp

   0x00000000004005d4 <+8>:     lea    -0x20(%rbp),%rax

   0x00000000004005d8 <+12>:    mov    $0x2,%edx

   0x00000000004005dd <+17>:    mov    $0x1,%esi

   0x00000000004005e2 <+22>:    mov    %rax,%rdi

   0x00000000004005e5 <+25>:    callq  0x400624 <X::X(int, int)>

b \*<内存地址>：

(gdb) b \*0x00000000004005dd

Breakpoint 1 at 0x4005dd: file main.cpp, line 14.

(gdb) r

(gdb) disas

Dump of assembler code for function main():

   0x00000000004005cc <+0>:     push   %rbp

   0x00000000004005cd <+1>:     mov    %rsp,%rbp

   0x00000000004005d0 <+4>:     sub    $0x20,%rsp

   0x00000000004005d4 <+8>:     lea    -0x20(%rbp),%rax

   0x00000000004005d8 <+12>:    mov    $0x2,%edx

=> 0x00000000004005dd <+17>:    mov    $0x1,%esi

   0x00000000004005e2 <+22>:    mov    %rax,%rdi

   0x00000000004005e5 <+25>:    callq  0x400624 <X::X(int, int)>

b <函数名>：

(gdb) b main

Breakpoint 1 at 0x4005d4: file main.cpp, line 14.

(gdb) r

(gdb) disas

Dump of assembler code for function main():

   0x00000000004005cc <+0>:     push   %rbp

   0x00000000004005cd <+1>:     mov    %rsp,%rbp

   0x00000000004005d0 <+4>:     sub    $0x20,%rsp

=> 0x00000000004005d4 <+8>:     lea    -0x20(%rbp),%rax

   0x00000000004005d8 <+12>:    mov    $0x2,%edx

   0x00000000004005dd <+17>:    mov    $0x1,%esi

   0x00000000004005e2 <+22>:    mov    %rax,%rdi

   0x00000000004005e5 <+25>:    callq  0x400624 <X::X(int, int)>

**PCIe：**

Intel在1992年提出PCI（Peripheral Component Interconnect）总线协议，并召集组成了PCI-SIG (PCI Special Interest Group，PCI 特殊兴趣组）企业联盟。从此PCI-SIG就负责PCI和其继承者们（PCI-X和PCIe）的标准制定与推广。

2004年，Intel推出PCI express（PCIe）。

**PCI架构：**

PCI是总线结构，一个典型的桌面系统的PCI架构如下图：



桌面系统一般只有一个Host Bridge用于隔离存储器域与PCI总线域，并完成处理器与PCI设备间的数据交换。

桌面系统一般也只有一个Root Bridge，每个Root Bridge管理一个Local Bus空间，它下面挂载了一颗PCI总线树，同一颗PCI总线树上的所有PCI设备属于同一个PCI总线域。一颗典型的PCI总线树如下图：



从图中可以看出PCI总线树主要包括三部分：

1. PCI设备。符合PCI总线标准的设备就被称为PCI设备。上图中的Audio、LAN都是一个PCI设备。
2. PCI总线。PCI总线在系统中可以有多条，类似于树状结构进行扩展，每条PCI总线都可以连接多个PCI设备/桥。图中有两条PCI总线。
3. PCI桥。当一条PCI总线的承载量不够时，可以用新的PCI总线进行扩展，而PCI桥则是连接PCI总线之间的纽带。

PCI标准的特点：

1. PCI是并行总线，在一个时钟周期内32个bit（后扩展到64）同时被传输。
2. PCI空间与处理器空间隔离。PCI设备具有独立的地址空间，即PCI总线地址空间，该空间与存储器地址空间通过Host Bridge隔离。在Host Bridge中有许多缓冲，这些缓冲使得处理器总线与PCI总线工作在各自的时钟频率上，彼此互不干扰。处理器访问PCI设备时、PCI设备访问主存储器时，都必须通过Host Bridge进行地址转换。
3. PCI总线扩展性强。在PCI总线中，Root Bridge可以直接连出一条PCI总线，这条总线是该Root Bridge所管理的第一条PCI总线，该总线还可以通过PCI桥扩展出一系列PCI总线，并以Root Bridge为根节点，形成一颗PCI总线树。在同一条PCI总线上的设备可以直接通信，并不会影响其他PCI总线上设备间的数据通信。同一颗PCI总线树上的PCI设备，也可以直接通信，但是需要通过PCI桥进行数据转发。

**PCIe架构：**

PCIe是点对点结构，一个典型的PCIe系统架构如下图：



典型的结构是一个root port和一个endpoint直接组成一个点对点连接对。而Switch可以同时连接几个endpoint。一个root port和endpoint对就需要一个单独的PCI bus。

而PCI是在同一个总线上的设备共享同一个bus number。过去主板上的PCI插槽都公用一个PCI bus，而现在的PCIe插槽却连在芯片组不同的root port上。

**PCIe插槽：**

PCI插槽都是等长的，防呆口位置靠上，大多为乳白色。

PCIe插槽有大有小，最小的x1，最大的x16，防呆口位置靠下。PCIe插槽的物理配置x后面的数字表示有多少个通道。PCIe x1插槽有一个通道，每个周期可以移动一位数据。PCIe x2插槽有两个通道，每个周期可以移动两位数据，依此类推。



Q：x1的串口卡能不能插在x4的插槽里？

A：完全没有问题，串口卡也将以x1的方式工作。

Q：x16的RAID卡能不能插在x8的插槽内？

A：能，RAID卡将以x8的方式工作。实际上，可以将任何PCIe卡插入任何PCIe插槽中! PCIe在链接training的时候会动态调整出双方都可以接受的宽度。但有些主板厂商会把PCIe插槽尾部开口，方便这种行为。

Q：显卡是PCIe 3.0的，主板是PCIe2.0的，能工作吗？

A：能，会以2.0工作。反之，亦然。

Q：x16的显卡插在主板上最长的x16插槽中，但是benchmark却显示跑在x8下，怎么回事?

A：主板插槽x16不见得就连在支持x16的root port上，有些主板实际上是x8，详细见主板原理图。

**PCIE设备的BDF：**

一个PCIe设备可以只有一个功能，称之为Single Function Device；也可以拥有最多8个功能，称之为Multi Function Device。

每一个PCIe功能都有一个唯一的标识符BDF（Bus，Device，Function）。BDF由总线号、设备号和功能号共同组成，格式为Bus:Device.Function。lspci命令可以查看BDF，如下：

[root@localhost home]$ lspci | grep Blackmagic

37:00.0 Multimedia video controller: Blackmagic Design DeckLink 4K Extreme 12G

上述打印显示BDF为37:00.0，即标识的是55号总线上的0号设备的功能0。

总线号类似于街道的名字，设备号类似于这条街道上的某栋楼，而功能号类似于这栋楼里的某个房间。

总线号由8bit表示，因此，最大总线号为255；

设备号由5bit表示，因此，最大设备号为31；

功能号由3bit表示，因此，最大功能号为7。

由于总线号占8bit，故单个系统最多拥有256个总线，这对于大型系统而言是不够的，所以引入了域的概念，这样，每个PCI域就可以拥有最多256个总线

lspci的-D参数可以查看PCI域：

[root@localhost home] lspci -D | grep net

0000:88:00.0 Ethernet controller: Intel Corporation Ethernet Connection X722 for 10GbE SFP+ (rev 04)

0000:88:00.1 Ethernet controller: Intel Corporation Ethernet Connection X722 for 10GbE SFP+ (rev 04)

第一个冒号前的16bit即为域编号。

上述第一行描述的是x722 10G网卡的第一个网口，即0号域136号总线上的0设备上的0号功能

x722为双网口网卡（即Multi Function Device），物理上只占用一个PCIE插槽，但有两个功能，

[root@localhost home] lspci -D | grep net

0000:88:00.0 Ethernet controller: Intel Corporation Ethernet Connection X722 for 10GbE SFP+ (rev 04)

0000:88:00.1 Ethernet controller: Intel Corporation Ethernet Connection X722 for 10GbE SFP+ (rev 04)

Blackmagic Design DeckLink Quad 2采集卡，物理上只占用一个PCIE插槽，但实际占用2个总线号：

[root]$ lspci | grep Blackmagic

3e:00.0 Multimedia video controller: Blackmagic Design DeckLink Quad 2

3f:00.0 Multimedia video controller: Blackmagic Design DeckLink Quad 2

Quad2卡总共有8个子设备，总线号依次为3e、3e、3f、3f、3e、3e、3f、3f，即靠近ref口的四个物理端口占一个pcie总线号，另外四个物理端口占用一个pcie总线号

总线号由卡槽位置、卡槽上插的卡的类型等因素共同决定。

如下示例，服务器上插了三张Blackmagic卡，依次为8K pro、Intensity 4K、Quad2，调换卡槽位置前后，占用的总线号情况依次如下图：





**PCIE带宽：**

不同PCIe版本的传输速率如下：

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| PCIE版本 | 编码方案 | 传输速率 | 吞吐量 | | | |
| ×1 | ×4 | ×8 | ×16 |
| 1.0 | 8b/10b | 2.5[GT](https://baike.baidu.com/item/GT)/s | 250[MB](https://baike.baidu.com/item/MB)/s | 1[GB](https://baike.baidu.com/item/GB)/s | 2GB/s | 4GB/s |
| 2.0 | 8b/10b | 5GT/s | 500MB/s | 2GB/s | 4GB/s | 8GB/s |
| 3.0 | 128b/130b | 8GT/s | 984.6MB/s | 3.938GB/s | 7.877GB/s | 15.754GB/s |
| 4.0 | 128b/130b | 16GT/s | 1.969GB/s | 7.877GB/s | 15.754GB/s | 31.508GB/s |
| 5.0 | 128b/130b | 32 or 25GT/s | 3.9 or 3.08GB/s | 15.8 or 12.3GB/s | 31.5 or 24.6GB/s | 63.0 or 49.2GB/s |

传输速率为每秒传输量GT/s，传输量包括开销位，比如PCIe1.x和PCIe2.x使用8b/10b编码方案，导致占用了20%（=2/10）的原始信道带宽。

GT/s（Giga transation per second），千兆传输/秒，重点在于描述物理层通信协议的速率属性

PCIe吞吐量（可用带宽）计算方法：

吞吐量=传输速率\*编码方案

例如：PCI-e2.0协议支持5.0GT/s，即每一条Lane上支持每秒钟内传输5G个Bit；但这并不意味着PCIe2.0协议的每一条Lane支持5Gbps的速率。因为PCIe2.0的物理层协议中使用的是8b/10b的编码方案。即每传输8个Bit，需要发送10个Bit；这多出的2个Bit并不是对上层有意义的信息。

那么，PCIe2.0协议的每一条Lane支持5\*8/10=4Gbps=500MB/s的速率，x8的PCIe 2.0的可用带宽则为32Gbps=4GB/s

PCIE带宽查询：

1. lspci | grep NVIDIA查看BDF，格式为bus:device.function
2. lspci –nn | grep <BDF>查看class id、vendor id和device id：

[root]$ lspci -nn | grep Blackmagic

3e:00.0 Multimedia video controller [0400]: Blackmagic Design DeckLink Quad 2 [bdbd:a13f]

3f:00.0 Multimedia video controller [0400]: Blackmagic Design DeckLink Quad 2 [bdbd:a13f]

86:00.0 Multimedia video controller [0400]: Blackmagic Design DeckLink 8K Pro [bdbd:a14b]

[root@]$ lspci -n | grep 3e:00.0

3e:00.0 0400: bdbd:a13f

Linux 使用Class ID + Vendor ID + Device ID来代表设备。上图里的0400: bdbd:a13f所代表的设备名称为(Class ID = 0400 , Vendor ID = bdbd, Device ID = a13f)。所以3e:00.0、3f:00.0总线地址实际属于同一张卡，即Quad2卡。

-n：显示设备的数字标识符（Class ID、Vendor ID和Device ID）

-nn：显示设备的数字标识符（Class ID、Vendor ID和Device ID）以及设备的名称

1. lspci -n -d <vendor\_id:device\_id> -vvv | grep -i width，查看-d指定的设备的详细信息





LnkSta和LnkCap这两个速度有可能不一样，典型情况如：系统提供的是PCIe是3.0但设备使用的还是2.0的

LnkCap：表示链路能力（Link Capability），即PCI设备链路的能力。如支持的速度、宽度、协商能力等。LnkStat：表示链路状态（Link Status），即PCI设备链路的当前状态。如当前的速度、宽度、协商状态等。

通过比较LnkCap和LnkStat字段，可以了解PCI设备链路的最大能力和当前状态之间的差异。这有助于评估链路的实际性能、是否达到了预期的带宽要求。

**PCIE相关命令：**

查看pcie所在的cpu：

cd /sys/bus/pci/devices/0000:d8:00.0

cat local\_cpulist

查看pcie树形结构：

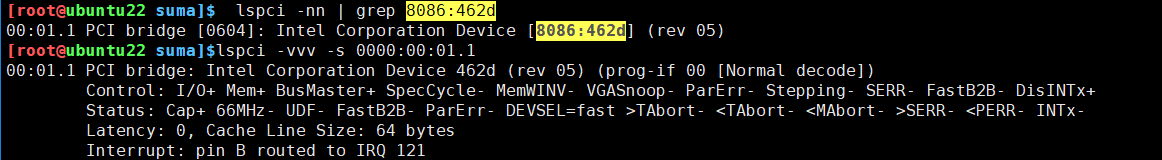
lspci -vt

[ ]里边是bus号；

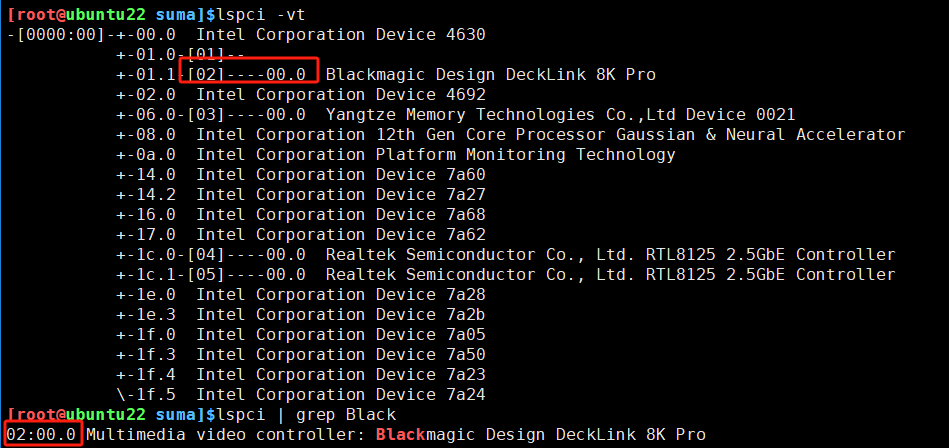
-表示link，即-两端是两个PCIe设备，这两个设备组成一条PCIe Link。这条Link两端的设备，更靠近根端口的，也就是最左边起点的设备称为下行设备，而远离根端口的设备称为上行设备。

**dmesg的pcie报错：**

****

****

如上图，是一个pci bridge，设备标识为8086:462d

****

如上图，pci桥00:01.1连接的BDF为02:00.0，是个Blackmagic卡

**PCIE配置空间：**

PCIe设备的每一个功能（Function）都有一个唯一独立的配置空间（Configuration Space）与之对应。

PCIE为了兼容PCI，几乎完整地保留了PCI总线的配置空间，将配置空间从256B扩展到4KB

从PCI总线中继承过来的配置空间的示意图：



如上图，有两种类型的配置空间，Type0和Type1，Type0对应非桥设备（Endpoint），Type1对应桥设备（Root Complex和Switch端口中的P2P桥）。

配置空间中的Device ID和Vendor ID是区分不同设备的关键，OS和UEFI在很多时候就是通过匹配它们来找到不同的设备驱动（Class Code有时也起一定作用）。这些都是是出厂时就固定在PCI硬件设备中的数据。为了保证唯一性，Vendor ID应当向PCI特别兴趣小组(PCI SIG)申请而得到。

每个Type0型的Header中有6个Base Address Registers（BAR）寄存器，用来记录PCI需要占用的地址空间大小、配置PCI设备占用的地址空间：每个PCI设备在BAR中描述自己需要占用多少地址空间，UEFI通过汇总所有设备的这些信息构建一张完整的关系图，描述系统中资源的分配情况，然后再合理地将地址空间配置给每个PCI设备。

每个Type1型的Header中都包含最后一级总线号（Subordinate Bus Number）、下一级总线号（Secondary Bus Number）和上一级总线号（Primary Bus Number），用于BDF路由。

PCIe新增的配置空间的示意图：



处理器一般不能够直接发起配置读写请求，因为其只能产生Memory Request和IO Request。这就意味着Root Complex必须要将处理器的相关请求转换为配置读写请求。

针对传统的PCI设备（Legacy PCI），采用的是IO间接寻址访问（IO-indirect Accesses）；针对PCIe设备，采用的是Memory-Mapped Accesses。

Root Complex和Switch的每一个端口中都包含一个P2P桥，桥的配置空间头（Configuration Space Header）是Type1型的。每个Type1型的Header中都包含最后一级总线号（Subordinate Bus Number）、下一级总线号（Secondary Bus Number）和上一级总线号（Primary Bus Number）等信息。当配置请求进行BDF路由的时候，正是依靠这些信息来确定要找的设备的。一个简单地例子如下图所示：



PCIe中只有Root Complex才可以发起配置空间读写请求，Root的每个端口中都包含有一个P2P桥。

当Root Complex发起配置空间读写请求时，相应的桥首先检查请求的BDF中的Bus号是否与自己的下一级总线号（Secondary Bus Number）相等，如果相等，则先将Type1转换为Type0，然后发给下一级（即Endpoint）。

如果不相等，但是在自己的下一级总线号（Secondary Bus Number）和最后一级总线号（Subordinate Bus Number）之间，则直接将Type1型请求发送给下一级。如果还是不相等，则该桥认为这一请求和自己没什么关系，则忽略该请求。

注：Root Complex最先发送的配置请求一定是Type1型的。非桥设备（endpoint）会直接忽略Type1型的配置请求。

**PCIE枚举扫描拓扑结构：**

深度优先遍历

PCIE的枚举过程只分配bus号，并不分配device号。因为对于switch而言，brige上的device号需要硬件确定，无需进行分配。switch在物理上呈现出来是芯片，在设计的时候就要考虑device号，否则无法正常工作。对于endpoint而言，device只能是0，因为endpoit会单独占一个bus，这个bus下不会有两个endpoint。



**PCIe时钟：**

任何电路都需要时钟进行驱动，尤其是总线信号，发送端需要时钟驱动打出信号，接收端则需要时钟采样信号，从而识别信息。

PCIe总线物理链路间的数据传送使用基于时钟的同步传送机制，但是在物理链路上并没有时钟线，PCIe总线的接收端含有时钟恢复模块CDR(Clock Data Recovery)。CDR从接收报文中提取接收时钟，从而进行同步数据传递。

PCIe的时钟是嵌入到差分信号中的，PCIe总线在训练之初会完成一个bit lock，在这个过程中链路上会传输一段0和1间隔序列，即连续的高低电平，用来给Receiver提取时钟。Receiver提取到时钟后，才能够继续捕获一位一位的数据，也就是做bit的识别，再往后则是做多个bit的识别，即序列识别。在PCIe Transmitter里会进行编码操作，让0和1的数量更加均衡，以防止总线时钟丢失。此外，总线处于空闲状态时，也会持续传输时钟对齐码流，以防止丢了时钟，突然有数据过来的时候无法正确传输。

PCIe协议规定参考时钟为100MHz时钟，Gen1~Gen4要求参考时钟精度在±300ppm以内，Gen5要求±100ppm以内。PCIe controller内部电路运行是需要时钟的，100MHz参考时钟即是提供给controller使用的时钟，和任何一个电路系统需要时钟是相同的概念。100MHz时钟输入controller后，经内部PLL进行倍频再分给各部分电路使用。

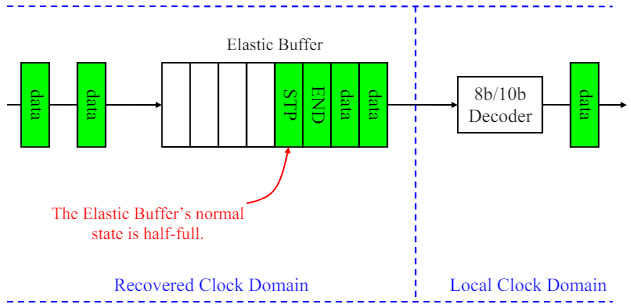
独立时钟架构（Separate Clock Architecture）：发送端和接收端使用独立的100MHz参考时钟源。

数据时钟架构（Data Clock Architecture）：仅发送端需要参考时钟，接收端无需外部参考时钟，接收端物理层从数据流中恢复出时钟作为参考时钟。

同源时钟架构（Common Refclk Architecture），即root和endpoint使用的是同一个100MHz参考时钟源，目前应用最广泛的架构。在这种架构中，会对时钟走线长度、接收端数据和时钟之间的传输延迟增量（时钟偏斜，Clock Skew）进行限制。为什么会有这样的限制？

既然PCIe Receiver接收数据是基于总线上使用CDR恢复的时钟，那么和100MHz参考时钟有什么关系，为什么要对100MHz时钟进行限制？这两个时钟有什么关系？这里涉及到Receiver内部的工作机制，其中有一个部件叫做Elastic Buffer——弹性缓冲，这个部件的作用是，将按照CDR恢复的时钟采样得到的数据继续向后面电路传输时，进行一个过渡。为什么需要过渡？因为后面电路所采用的时钟不是CDR恢复的时钟了，而是Receiver使用100MHz参考时钟进行倍频而得到的时钟。这里存在2个时钟，数据需要从一个时钟域传输到另一个时钟域，虽然前后的时钟被处理到了同一个频率，但无法保证频率、相位没有任何一丝差异。

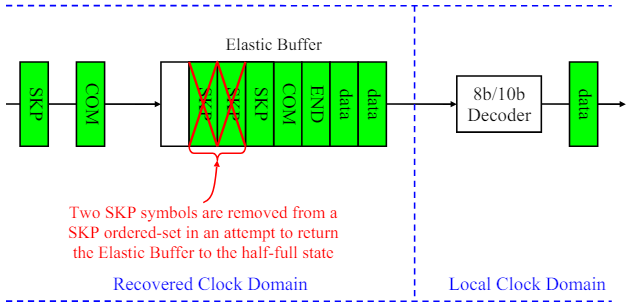
Elastic Buffer原理如下：



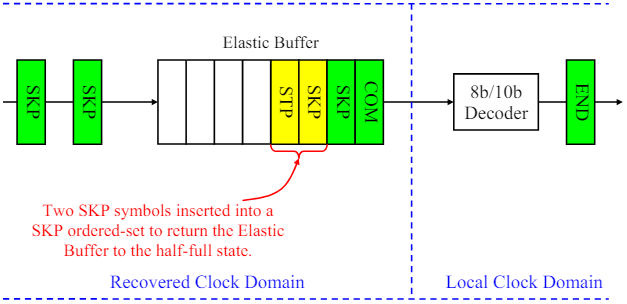
从buffer左侧进入的是使用CDR恢复的时钟采样的数据，属于恢复时钟域；buffer右侧取出则是controller使用100MHz参考时钟进行倍频后的本地时钟，属于本地时钟域。两个时钟由于各种不可控因素会存在很小的频率差异和相位差异。Elastic Buffer是一个先进先出的缓冲空间，数据从左边进，然后按照进来的顺序再从右边出。入口和出口的差异使用SKP序列进行处理。

SKP序列是PCIe总线专门用于补偿时钟差异的特殊序列。SKP序列会在总线上被定期插入。当SKP序列和有效数据序列一起被receiver采样进入到Elastic Buffer后，也占用了Elastic Buffer的若干位置。

当数据进入的时钟频率（CDR恢复的时钟频率）大于数据出的时钟频率（本地时钟频率）时，数据进的比出的快，会导致buffer中的数据累积越来越多，甚至可能溢出丢失，在这种情况下，则从buffer中移除几个SKP。



当数据进入的时钟频率（CDR恢复的时钟频率）小于数据出的时钟频率（本地时钟频率）时，数据出的比进的快，会导致buffer的数据越来越少，甚至下溢，在这种情况下，则向buffer中插入几个SKP，填充buffer



通过抽走和插入SKP序列的方式，可以解决两边时钟频率差异的问题。Elastic Buffer自身有状态检测功能，能够不断监测buffer内的数据量，从而做出响应。

从Elastic Buffer的工作原理可以看到，其对时钟差异的补偿量级，可能也只是可以操作的SKP数量所对应的时间，而一个SKP序列，在PCIE 5.0只有16bit数据，所对应的时间极短，因此弹性缓冲能够补偿的时间也是极短的。如果buffer两边的时钟相差很大，那么弹性缓冲也无能为力，故同源时钟架构中，需要对时钟走线长度以及收发链路的时钟延时进行一个限制。

**float浮点数精度分析：**



在十进制中，一个数字用科学计数法表示为的形式，a是0-9中的一个整数。

同理，在二进制中，一个数字用科学计数法表示为 的形式，a为0或1

十进制小数转二进制的规则为“乘2取整，顺序排列”，原理是通过不断的左移反推每一位小数。

以0.7为例，结果为0.10接上1100的无限循环；

以0.15625为例，结果为0.00101。

由此发现，0.7的有效小数位更少，但反而不能被精确表示。

计算过程见下表：

IEEE754规定：

1. 在二进制数中，通过移位将小数点前面的值固定为1，即让尾数的第一位总是1，从而一个浮点数的尾数、指数部分就能唯一确认了
2. 尾数第一位的1省略不写，从而float的23位尾数可以表示24位有效数字
3. float的指数是个无符号整数，取值范围为0~255，对数字做了偏移127处理，从而对应的指数值为-127~128

示例，十进制数25.125的float存储结构：

25.125转为二进制是11001.001。为了让第1位为1，执行逻辑左移4位，即表示为。

指数部分是4，4+127=131，所以指数部分存储的是二进制的10000011。

尾数部分是1.1001001，第1位的1不存储，所以只需存1001001

所以整个float的32bit为：

0 10000011 10010010 00000000 0000000

如果是小端模式，则在内存中即为：

0x00 00 C9 41

float表示整数的精度：

float尾数的23位加上显式为1的整数位共24位，而对于一个24位的二进制数，能够表示的最大数字为  ，转换为十进制即16777215，使用二进制科学计数法表示为   。

从这里可以看到，小于等于16777215的整数，都能够用尾数确定数字构成、指数控制小数点移动这种方式精确表示出来。

那大于16777215的整数呢？

* 16777216比较特殊，是，因此可以表示为 ，只靠控制指数位就表示了出来。
* 接下来看16777217，即 ，按照二进制规则可以写成1(23个0)1，共25位，转换为float存储时可以看到，最后的一个1在第25位，会被尾数位丢失掉，导致记录的数字还是16777216，造成了精度的丢失。
* 接下来看16777218，即  ，按照二进制规则可以写成1(22个0)10，共25位，转换为科学计数法时可以看到，因为1在第24位，因此没有被丢失造成精度丢失。

由此可知：

1. 小于等于16777216的所有整数可以被精确表示，这就是float保证7位有效数字的原因
2. 并非所有超过16777216的整数都会遇到精度丢失的问题
3. 实际上，所有大于16777216的整数，只要能够表示为0~16777215之间一个数字进行移位（  ）操作，就能够保证不丢失精度

float表示小数的精度：

如下图，为IEEE754给出的float和double精度范围：



**GCC 的-fno-omit-frame-pointer：**

#include <stdio.h>

void test() {

    int a=3;

    int b=5;

    a+=b;

}

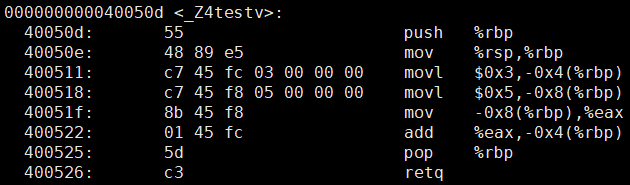
int main() {

    test();

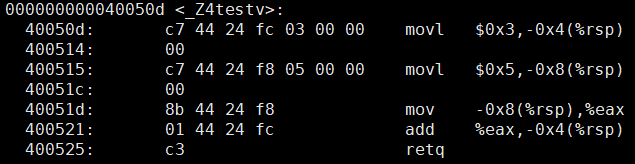
}

g++ -o main main.cpp

objdump -d main可以看到包含frame pointer：



如果g++ -o main main.cpp -fomit-frame-pointer，则



GCC的-O (-O1) 会默认打开-fomit-frame-pointer，此时可以-fno-omit-frame-pointer关闭之

**std::any**

#include <iostream>

#include <cassert>

#include <string>

#include <any>

int main()

{

    std::any a0;

    std::any a1 = 42;

    std::any a2 = std::string("sky fly");

    std::any a3 = a0; *// Copies the empty any*

    std::any a4 = a1; *// Copies the "std::string"-containing any*

    a4 = a0;          *// copy assignment works, and properly destroys the old value*

    assert(!a0.has\_value());            *// a0 is still empty*

    assert(a1.type() == typeid(int));

    assert(a2.type() == typeid(std::string));

    assert(!a4.has\_value());

    assert(a4.type() == typeid(void));  *// type() returns typeid(void) when empty*

*//访问any内部数据*

    std::string& a2ref = std::any\_cast<std::string&>(a2); *//引用*

    a2ref += " land";

    std::cout << std::any\_cast<std::string&>(a2) << std::endl;  *//"sky fly land"*

    std::string a2cp = std::any\_cast<std::string>(a2); *//副本*

    a2cp += " sea";

    std::cout << std::any\_cast<std::string&>(a2) << std::endl; *//"sky fly land"*

    std::any a5 = 0.3;

    if (auto ptr = std::any\_cast<double>(&a5)) *//指针*

    {

        printf("a5=%f\n", \*ptr); *//"a5=0.300000"*

    }

    else if (auto ptr = std::any\_cast<std::string>(&a5))

    {

        printf("a5 not contain std::string\n");

    }

}

**std::variant**

#include <string>

#include <variant>

#include <iostream>

int main()

{

    using JsonVal = std::variant<int, bool, std::string>;

    JsonVal jval;

    jval = std::string("hello");

    std::cout << jval.index() << std::endl; *// 2*

    std::cout << std::get<std::string>(jval) << std::endl; *//或std::get<2>(jval)*

    jval = 1; *//jval.index()为0*

    jval = true; *//jval.index()为1*

}

**模板SFINAE：**

SFINAE是C++模板进阶的门槛之一

它来自于substitution failure is not an error的首字母缩写

error就是一般意义上的编译错误。一旦出现编译错误，大家都知道，编译器就会中止编译，并且停止接下来的代码生成和链接等后续活动

substitution，模板实例化的过程中，模板参数被替换为实际的参数类型或值的过程。如gcc编译器报错常出现“template argument deduction/substitution”字样。

假设有如下函数签名：

template <

  typename T0,

*// 一大堆其他模板参数*

  typename U = */\* 和前面T有关的一大堆 \*/*

>

RType */\* 和模板参数有关的一大堆 \*/*

functionName (

   PType0 */\* PType0 是和模板参数有关的一大堆 \*/*,

   PType1 */\* PType1 是和模板参数有关的一大堆 \*/*,

*// ... 其他参数*

) {

*// 实现，和模板参数有关的一大堆*

}

那么，所有函数签名上的“和模板参数有关的一大堆”，都是substitution时要处理的东西（当然也有一些例外）。而函数体，即整个实现部分，都不属于substitution。

一个具体的例子如下：

template <

  typename T,

  typenname U = typename vector<T>::iterator *// 1*

>

typename vector<T>::value\_type  *// 1*

  foo(

      T\*, *// 1*

      T&, *// 1*

      typename T::internal\_type, *// 1*

      typename add\_reference<T>::type, *// 1*

      int

  )

{

*// 根据定义，substitution只发生在函数签名上。*

*// 故而整个实现部分，都没有 substitution。这是一个重点需要记住。*

}

所有标记为1的部分都是需要替换的部分，而它们在替换过程中的失败，就称之为替换失败（substitution failure）。

函数重载决议之后（当然也是模板参数推导替换之后），才会进行函数体内编译检查，如下示例：

#include <iostream>

#include <vector>

#include <string>

template<typename T>

struct A

{

    T m;

    A(const T x) : m(x) {}

    A operator+(const A rhs)

    {

        return A(rhs.m + m);

    }

};

template<typename T>

auto add(T a, T b) *// #接口实现1*

{

    return a + b;

}

template<template<typename >class Container, typename T>

T add(Container<T> a, Container<T> b)*// #接口实现2，是实现1的重载*

{

    T sum = 0;

    for (auto i : a) sum += i;

    for (auto i : b) sum += i;

    return sum;

}

int main()

{

    int x = 1, y = 2;

    auto z = add(1, 2); *// ok，匹配到接口实现1上*

    printf("z=%d\n",z);

    std::vector<int> i = { 1,2,3 }, j = { 4,5,6 };

    auto k = add(i, j); *// ok，匹配到接口实现2上*

    printf("k=%d\n", k);

    A<float> a1(3);

    A<float> a2(4);

*//auto a3 = add(a1, a2); //error，编译器会匹配到接口实现2，但是A类型中没有begin，end等函数，故而实例化出错；如果没有实现2，则编译器会匹配到实现1，编译通过*

}

如上示例代码，我们本来想让接口实现1服务基础类型、支持加法的自定义类型，让接口实现2服务容器类型。但在为自定义A<float>类型的变量调用接口的时候，实际匹配到了接口实现2上去，与预期不同。

编译器通过命名查找机制找到重载1和重载2两种函数模板，对于重载1，模板参数T被推导为A<float>类型，对于重载2，模板参数Container、T被推导为A、float，假如还有add的第三个重载template <typename T, typename T\*> auto add(T a, T\* b) { }，此时T无法被推导，但编译器并不会立即报错，只是将其从重载集中移除，后面的重载决议也就不会再选择它，这个机制就是SFINAE。

这个问题的根本原因在于根据c++重载决议机制，模板函数T add(Container<T> a, Container<T> b)与自定义类型A<float>匹配的更好。

然后重载决议之后，才进行第二阶段的编译检查，这时候的检查更加严格，会检查函数内部的表达式是否能够成立，即发现for (auto i : a)对类型A<float>不合法，编译出错。

SFINE使用示例：

#include <iostream>

#include <string>

struct ICounter {

    virtual void increase() = 0;

    virtual ~ICounter() {}

};

struct Counter : public ICounter {

    void increase() override { printf("ICounter++\n"); }

};

template <typename T>

void inc\_counter(T& counterInt, typename std::enable\_if<std::is\_integral<T>::value>::type\* = nullptr){

    ++counterInt;

}

template <typename T> void inc\_counter(T& counterObj, typename std::enable\_if<std::is\_base\_of<ICounter, T>::value>::type\* = nullptr){

    counterObj.increase();

}

int main()

{

    int a = 10;

    Counter b;

    inc\_counter(a);

    printf("a=%d\n", a); *//"a=11"*

    inc\_counter(b); *//"ICounter++"*

}

上述程序中，typename std::enable\_if<std::is\_integral<T>::value>::type是为了让enable\_if参与到函数签名中，从而参与到substitution过程中。

is\_integral<T>::value返回一个bool类型的编译期常数，描述T是不是一个integral。enable\_if<C>的作用是，如果C值为true，那么type就会被推断成一个void类型，让整个函数匹配后的类型变成void inc\_counter<int>(int& counterInt, void\* dummy=nullptr)；如果C值为false，那么enable\_if<false>这个特化形式中，不存在::type，于是substitution就失败了——所以这个函数原型根本就不会被产生出来，也就不会成为重载候选。

也可以将enable\_if放到模板参数中，即

template <typename T, typename std::enable\_if<std::is\_integral<T>::value>::type\* = nullptr>

void inc\_counter(T& counterInt)

template <typename T, typename std::enable\_if<std::is\_base\_of<ICounter, T>::value>::type\* = nullptr>

void inc\_counter(T& counterObj)

上述示例中，如果Counter不继承于ICounter，则可以使用C++11提供的Expression SFINAE实现相同功能：

#include <iostream>

#include <string>

struct Counter {

    void increase() { printf("ICounter++\n"); }

};

template <typename T>

void inc\_counter(T& intTypeCounter, decltype(++intTypeCounter)\* = nullptr) {

    ++intTypeCounter;

}

template <typename T>

void inc\_counter(T& counterObj, decltype(counterObj.increase())\* = nullptr) {

    counterObj.increase();

}

更一般化地，借助Expression SFINAE，我们可以写出以下程序：

#include <iostream>

#include <string>

struct A { void work() { printf("A::work\n"); } };

struct B {    void rest() { printf("B::rest\n"); } };

template <typename T>

void done(T& obj, decltype(obj.work())\* = nullptr) { obj.work();}

template <typename T>

void done(T& obj, decltype(obj.rest())\* = nullptr) { obj.rest();}

int main(){

    A a;

    B b;

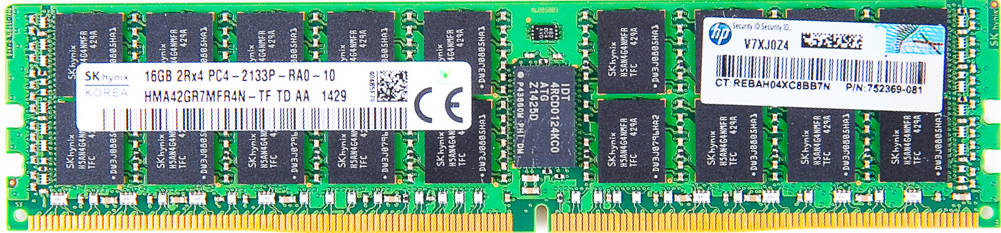
    done(a);

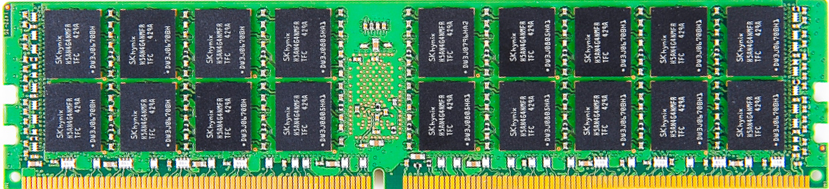
    done(b);

}

**内存的Rank及位宽：**

下图是一个16 GB服务器内存条（海力士sk hynix 16gb 2rx4 pc4 2133P RA0 10）的正反面图：





每个黑色的内存颗粒即一个chip，图中总共有37个内存颗粒，正面19个，背面18个。

内存条正面标签上有一串字符串标识16 GB 2R\*4 PC4-2133P-RA0-10，其中含义为

1. 16GB：内存总容量
2. 2R：该内存条有2个Rank
3. \*4：每个内存颗粒的位宽是4 bit

Rank指的是属于同一个组的chip的总和。这些chip并行工作，组成一个64bit的数据，供CPU来同时读取。

内存颗粒的位宽为4 bit时，因为CPU要同时读写64bit的数据，所以16个chip组成一个rank。

根据Rank和位宽可以计算出，这跟内存条总共需要2\*(64/4)=32个内存颗粒。

但实际上图中显示有37个内存颗粒。

支持ECC（“Error Checking and Correcting”，错误检查和纠正）功能的内存，需要为CPU同时提供72位的读写，其中64位是数据，8位用于ECC校验。在实现上，会在板上额外添加内存颗粒来专门负责ECC校验。

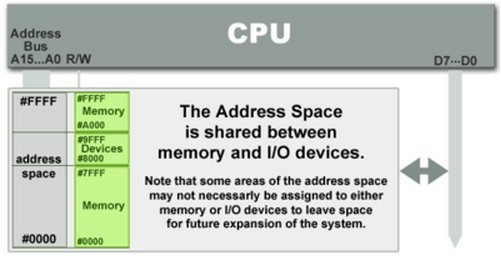
1. 位宽是4，为了提供8位的ECC校验数据，每个Rank需要额外2个内存颗粒来存储ECC数据。所以每个Rank总共需要16+2=18个内存颗粒。
2. Rank数量是2，所以总共需要18\*2=36个内存颗粒来存储用户数据以及校验位

此外，多出来的一个颗粒（正面示意图中间位置较大的那个内存颗粒）是用于缓存地址等控制信号的。服务器一般采用的是RDIMM（带寄存器双列直插模块），比普通的内存多一个带寄存器的时钟驱动器（RCD）模块。RCD从主机存储器控制器获取命令地址总线、控制信号和时钟信号，所有的信号通过RCD进入RDIMM后都会重新计时和清理。有了这个RCD模块的支持，能显著降低信号干扰，进而大幅提升稳定性。这样单条内存的容量可以做到更大一些。

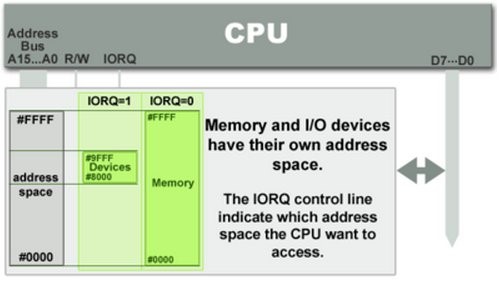
此外，当RDIMM提供的单条内存容量还不能满足服务器需求时，引入了LRDIMM（低负载双列直插内存模块）。LRDIMM 在 RDIMM 引入RCD 的基础上，又进一步引入了数据缓冲器 DB（Data Buffer），来缓冲来自内存控制器或内存颗粒的数据信号。这样实现了对地址、控制信号、数据的全缓冲，使得可以支持更大容量。

**MMIO\PMIO和内存类型：**

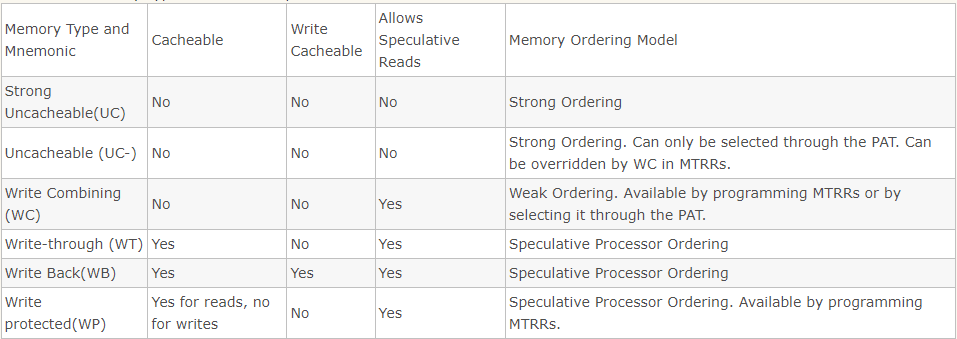
Memory-mapped I/O(MMIO)，内存映射IO。内存和I/O设备共享同一个地址空间。用于访问内存的CPU指令也用来访问I/O设备。每个I/O设备监视CPU的地址总线，一旦发现CPU访问的是分配给自身的地址，就做出响应，将数据总线连接到需要访问的设备硬件寄存器。



Port-mapped I/O(PMIO)，端口映射IO，又叫做被隔离的I/O（isolated I/O）。内存和I/O设备有各自独立的地址空间。端口映射I/O通常使用一种特殊的CPU指令，专门执行I/O操作。在Intel的微处理器中，使用的指令是IN和OUT。为了实现地址空间的隔离，要么在CPU物理接口上增加一个I/O引脚，要么增加一条专用的I/O总线。



IA32现在一共有5种memory type



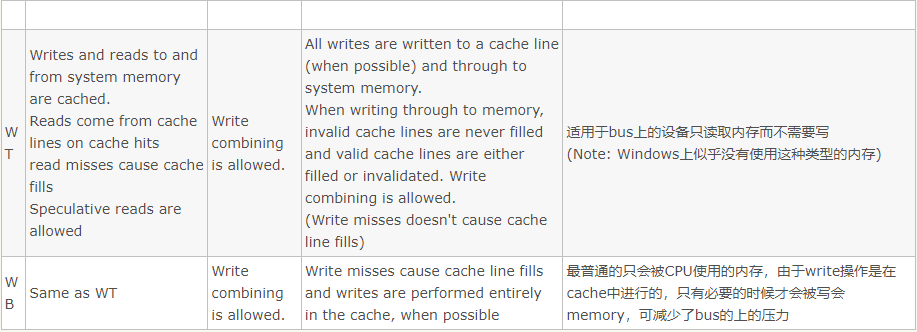
(Table 10-2. Memory Types and Their Properties)

1. Strong Uncacheable(UC) ：对于UC的内存读写操作都不会写到cache里，不会被reordering。

这种类型的内存适用于memory-mapped I/O device，比如说集成显卡。对于被memory-mapped I/O device使用的内存，由于会被CPU和I/O device同时访问，那么就会导致CPU的cache一致性问题。reordering也会导致I/O device读到dirty data，比如说I/O device把这些内存作为一些控制用的寄存器使用。  
对于普通用途的内存，UC会导致性能的急剧下降。

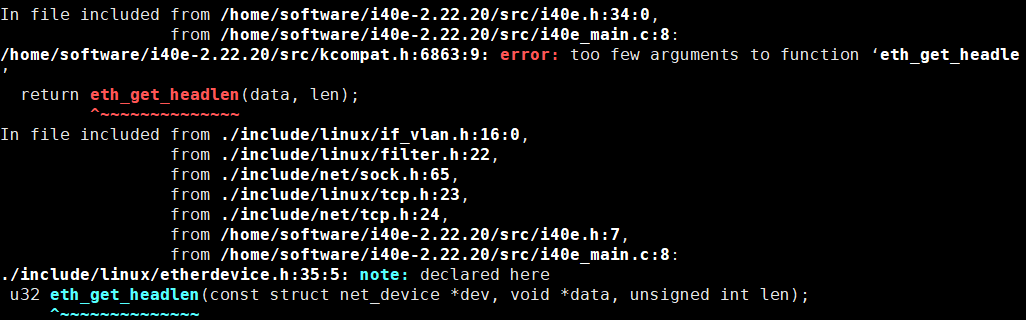
注：有些I/O device支持bus coherency protocol，可以和CPU保持cache一致性，从而可以使用cacheable内存，但这种总线协议也是有代价的。

1. Uncacheable (UC-)：和UC类型一样，除了UC- memory type可以通过设置MTRRs被改写为WC memory type.
2. Write Combining (WC)：WC内存不会被cache, bus coherency protcoal不会保证WC内存的读写。对于WC类型的写操作，可能会被延迟，数据被combined in write combining buffer, 这样可以减少总线上的访存操作。Speculative reads are allowed。  
   对于video frame buffer, 适合使用WC类型的内存。因为CPU对于frame buffer一般只有写操作，没有读，并不需要cache。对frame buffer而言，的写操作是否按顺序没有关系。  
   (注:  Speculative read是指读之前并不验证内存的有效性，先冒险的读进来，如果发现不是有效数据再取消读取操作，并更新内存后再读取. 比如说数据还是被buffer在WC buffer中)
3. Write-through (WT) and Write-back(WB)



**gcc函数传参报错示例：**

在Kylin Linux Advanced Server V10 (Halberd)系统上编译i40e-2.22.20网卡驱动时，gcc报错如下图：



/usr/src/kernels/目录下有linux内核源码

i40e驱动源码里kcompat.h文件的第6863行进行了函数调用eth\_get\_headlen(data, len)，此时gcc报错传递的实参个数太少

i40e\_main.c第8行包含了i40e.h，i40e.h的第7行包含了<net/tcp.h>，

tcp.h是linux内核源文件，实际路径为/usr/src/kernels/4.19.90-89.17.v2401.ky10.x86\_64/include/net/tcp.h

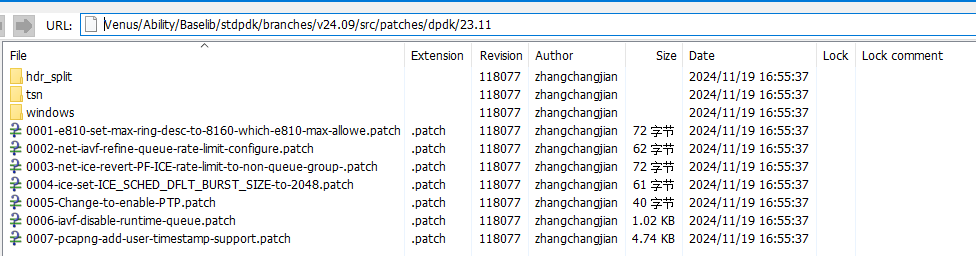
……

if\_vlan.h的第16行包含了<linux/etherdevice.h>，

etherdevice.h的第25行声明了函数eth\_get\_headlen，函数签名为u32 eth\_get\_headlen(const struct net\_device \*dev, void \*data, unsigned int len)，共三个形参

**Git应用patch示例：**

Media-Transport-Library（MTL）开源库依赖dpdk23.11，但项目在其子目录dpdk/23.11下并未提供dpdk的源码，而是包含了数个.patch文件，如下图：



项目的构建说明build.md中提示：



大意是用户需要自己clone dpdk23.11源码，然后用git am命令应用MTL提供patch，从而对官方的dpdk源码进行定制化改动

由于git clone dpdk太慢，因此选择直接在github上下载了dpdk23.11分支的.zip源码包

但解压后，进入dpdk源码目录，发现其中不包含.git目录，也即表示当前目录不是个git仓库，故执行git am时会报错：

[root@localhost 23.11]$ git am 0006-iavf-disable-runtime-queue.patch

fatal: not a git repository (or any of the parent directories): .git

为了成功应用patch，先将当前目录初始化为一个git仓库：

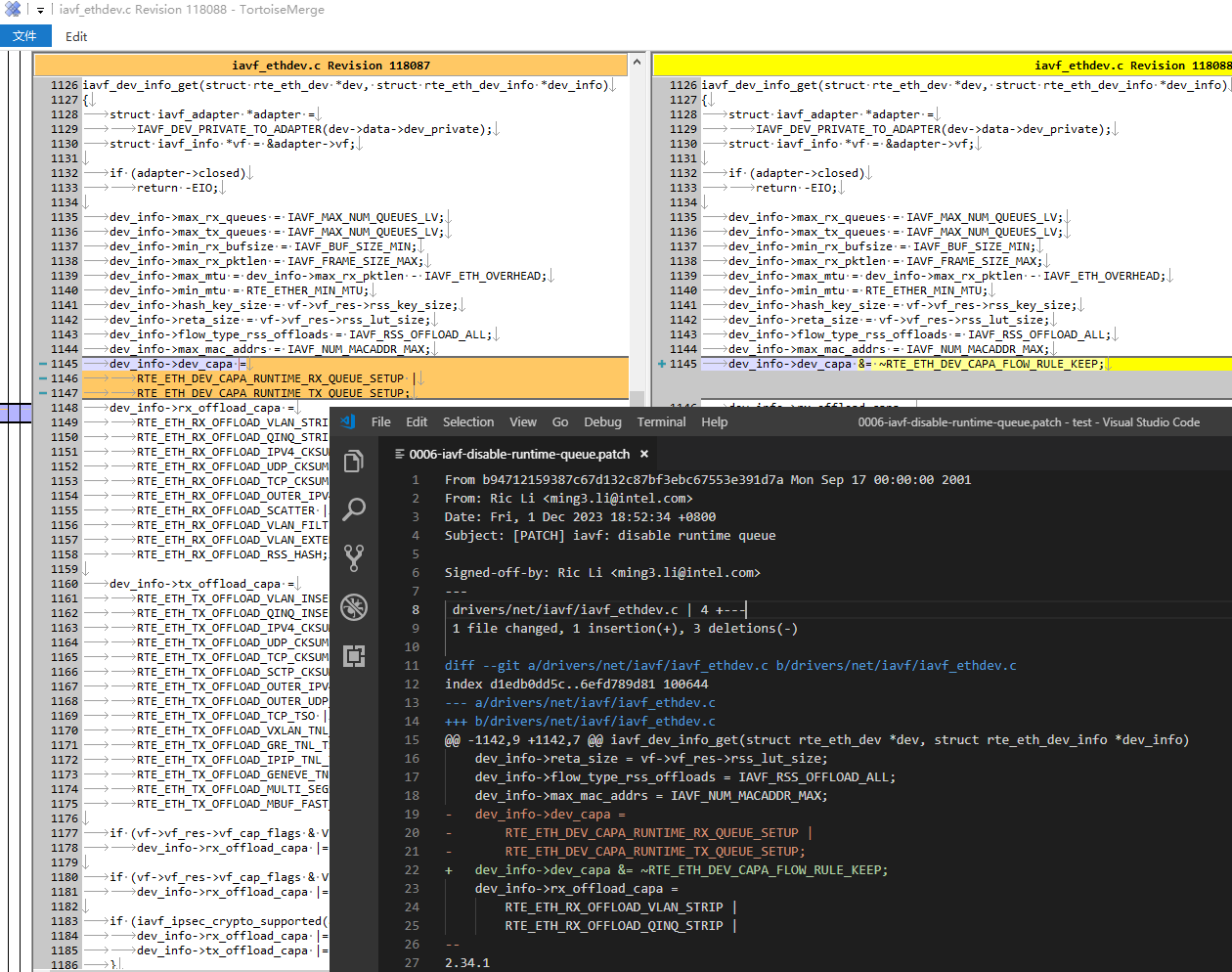
[root@localhost 23.11]$ git init

[root@localhost 23.11]$ git add .

[root@localhost 23.11]$ git commit -m "first commit"

然后执行git am 0006-iavf-disable-runtime-queue.patch，则成功应用这个patch

比较patch文件内容及应用patch前后dpdk源码的变化，发现是对应的，如下图



对于部分patch，git am时报错“Patch format detection failed”，查看patch内容发现里边似乎是指定的另一个patch的路径，因此实际git am时应该应用的是最终被指向的patch：

[root@localhost 23.11]$ git am 0005-Change-to-enable-PTP.patch

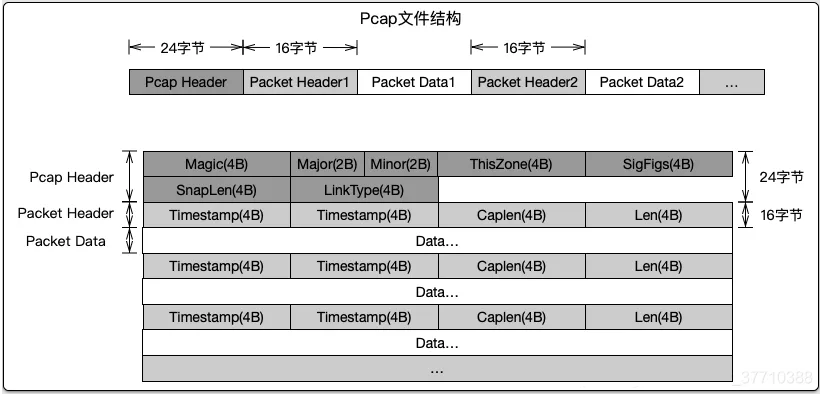
Patch format detection failed.

[root@localhost 23.11]$ cat 0005-Change-to-enable-PTP.patch

../22.07/0008-Change-to-enable-PTP.patch

[root@localhost 23.11]$ git am ../22.07/0008-Change-to-enable-PTP.patch

**pcap的snaplen：**



Pcap Header：文件头，每一个 pcap 文件只有一个文件头，共占 24 字节，其中一个字段为LinkType（链路类型），即链路类型。解析数据包首先要判断LinkType，一般值为1，即以太网常用的LinkType

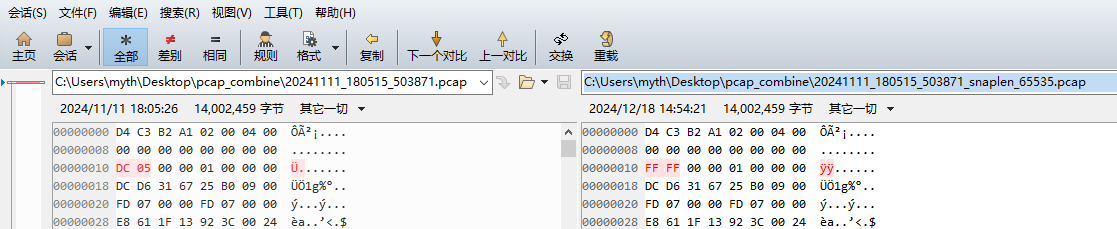
pcap中有多个数据包，每个数据包包括Packet Header和Packet Data。

Packet Header中记录着时戳和Packet Data的长度

Packet Data是一个数据链路层帧

现象：使用libpcap库的pcap\_open\_offline函数离线读取一个pcap文件时，出现pcap包头（struct pcap\_pkthdr）中caplen小于len的情况，最大的caplen为1550字节。但是，使用Wireshark打开该pcap文件是正常的。

分析：使用pcap\_open\_offline函数打开pcap文件时，在读取报文时会根据pcap文件头部的SnapLen对报文进行截断。上述pcap文件的第16到20字节的内容为0xDC 0x05 0x00 0x00，即SnapLen字段的值为0x05DC，即1550。使用UltraEdit修改四字节为0xFF 0xFF 0x00 0x00，即65535，再用pcap\_open\_offline读取则正常。



*// 读取输入文件每个包并写入输出文件*

struct pcap\_pkthdr\* header;

const u\_char\* packet;

while(pcap\_next\_ex(pcap\_in, &header, &packet)>=0)

{

    if(header->caplen < header->len)

    {

        printf("warn: caplen=%u, len=%u\n",header->caplen, header->len);

    }

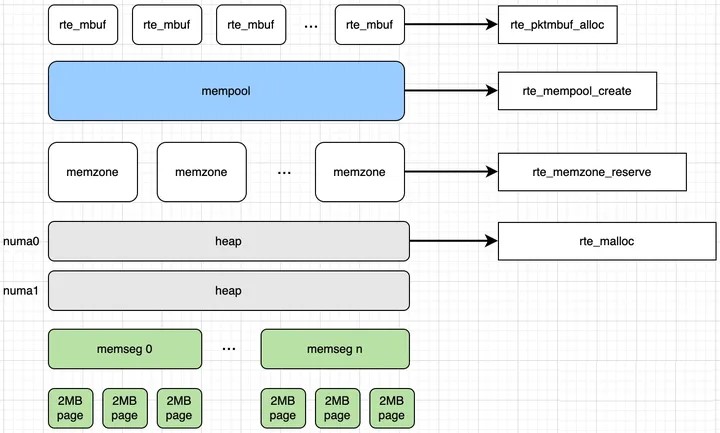
    pcap\_dump((u\_char\*)dumper, header, packet);

}

如上代码片段，如果进入if(header->caplen < header->len)分支，则最终输出的pcap文件比原始文件的size小。

**dpdk内存管理：**

dpdk内存管理总体架构图如下：



rte\_mbuf用于存储网络数据包。rte\_mbuf是一种固定大小的对象，一个rte\_mbuf占用的物理内存是连续的。一个网络数据包可以跨越多个rte\_mbuf，以链表的形式串在一起。

mempool是用于存储固定大小对象的内存池。如创建存储rte\_mbuf的mempool的接口为：

struct rte\_mempool\*

rte\_pktmbuf\_pool\_create(const char \*name, *//mempool的名字，在dpdk中，mempool由名称唯一标识*

                        unsigned n,                *//要创建对象（rte\_mbuf）的个数*

                        unsigned cache\_size,

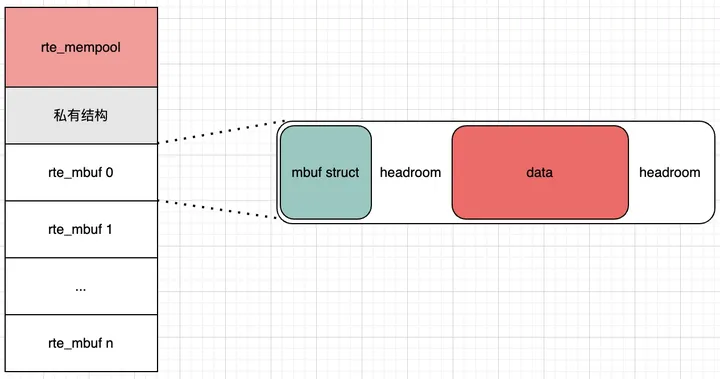
                        uint16\_t priv\_size,

                        uint16\_t data\_room\_size,   *//head\_room + data\_buff + tailroom的大小*

                        int socket\_id              *//在指定的numa内存上创建*

)

mempool内存池结构图：



一个mempool占用的物理内存不一定连续，且可能会跨多个大页。

rte\_mbuf的结构分为mbuf struct、headroom、data 和tailroom四部分：

1. mbuf struct存放一些描述性信息，比如包大小，协议类型，以及网络数据包跨越几个rte\_mbuf等。
2. headroom存放用户自己的metadata信息，这些信息不会在网络中传输，例如virtio PMD在其中存储结构体virtio\_net\_hdr。
3. data存放真正的网络数据包，包括ether/ip/tcp首部，以及payload。

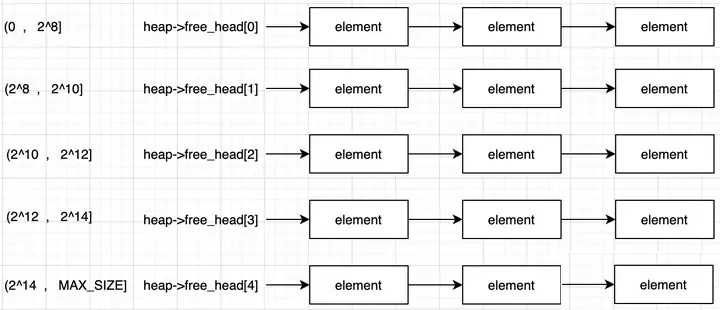
创建mempool时是从memzone分配内存

每个memzone描述的都是一块连续的物理内存，memzone结构本身并不在大页上，但是它描述的连续物理内存在大页上。

当一个memzone描述的连续物理内存大小不能满足创建mempool的需求时，一个mempool就需要消耗多个memzone。

memzone描述的内存块是从heap上申请的。申请时会根据申请的目标大小找到对应的堆链表里面能满足需求的最小element，每个element都是一块连续物理内存。一旦申请后，这块被申请的内存将从heap的链表里面的element切割出去，交由memzone管理维护。

dpdk设计了5种不同大小的堆链表，如下图：



memseg是dpdk比较底层的数据结构，对用户不可见。一个segment表示一块连续的物理内存，也即是heap链表里面的element。

一个memseg由物理内存连续的多个大页组成

一个memseg结构维护的大页内存必然是同一个物理cpu上的