

目录

2021/3/3.....	2
串行 Y86.....	2
2021/3/10.....	4
流水线技术.....	4
流水线中的依赖问题.....	5
流水线的实现.....	5
电路的反馈部分.....	7
True Dependencies (Data Hazard)	8
2021/3/17.....	8
Data hazard.....	10
2021/3/22.....	14
Y86 中的 Exception.....	14
性能计算方法.....	19
2021/3/24.....	19
现代处理器的乱序执行.....	20
2021/3/31.....	20
超标量处理器.....	20
数据流图和关键路径.....	21
整数乘法的展开优化.....	23
分支预测.....	24
2021/4/7.....	25
插桩分析 (profiling)	27
RAM (random access memory)	28
DRAM.....	28
2021/4/14.....	30
硬盘.....	30
CPU 如何控制和访问外部设备.....	32
2021/4/19.....	33
局部性.....	33
使用硬件利用局部性原理.....	34
Cache Miss.....	35
Cache Memory.....	35
2021/4/21.....	37
Set-associated cache.....	38
Fully-associated cache.....	39
2021/4/28.....	40
第七章：链接.....	42
编译的过程.....	43
链接器的作用.....	44
ELF 表.....	45
2021/5/8.....	46
ELF 字符串表.....	47
ELF 中的 Symbol Table.....	48

强名和弱名.....	52
2021/5/12.....	54
静态链接.....	54
Executable Object File.....	59
动态链接.....	60
2021/5/17.....	61
位置无关代码.....	62
GOT 表.....	62
动态内存分配.....	64
动态分区分配算法.....	65
2021/5/19.....	66
动态内存分配的实现.....	66
Segregated Storage.....	70
Segregated Fits.....	70
Buddy System.....	71
Virtual Memory.....	71
Page Table.....	73
2021/5/26.....	74
虚地址到实地址的映射.....	74
为什么要使用虚拟内存（VM）.....	76
Page Fault.....	80
硬件 TLB.....	81
2021/5/31.....	82
多级页表.....	82
Intel Core i7.....	84
Linux 里对虚拟地址的支持.....	87
2021/6/2.....	92
Copy-On-Write（写时复制）.....	93
最优替换策略.....	95
1. 先进先出策略.....	96
2. 随机选择策略.....	96
3. LRU 替换策略.....	97
2021/6/9.....	98
时钟算法（LRU 的近似实现）.....	99
垃圾回收.....	101

2021/3/3

串行 Y86

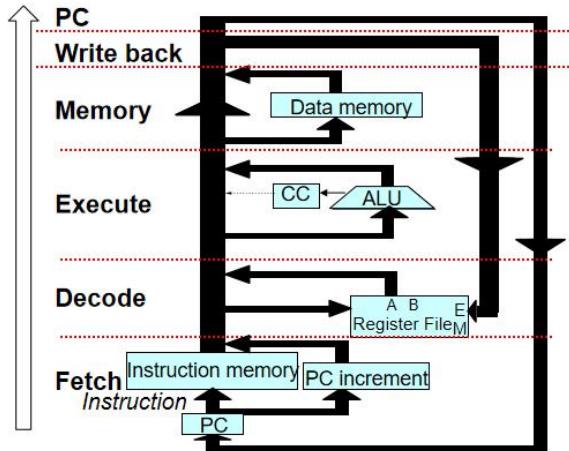
我们目前设计出来的 cpu 可以执行 y86，但是效率上并不快。先复习一下 12 类指令，看到第一个 byte 长度就确定了，第一个 byte 可以分为 icode，第二部分称为 func（cmov，

opq, jxx)。Instruction byte 必须要，后面两个可选。

Y86 有程序员可以看到的状态: PC, 15 个寄存器(F 作为无寄存器处理), 三个 CC(OF,ZF,SF), 还有状态码(X86 其实也有，但是没有介绍), 内存。

Y86 的状态码(status code), 分成了四个: HLT(停机的缩写), AOK(指令正常运行), ADR(地址的缩写, 地址是错的就置这个状态码), INS(指令的缩写, 指令编码是错的时候就置这个状态码)

有了这些以后，就可以为 Y86 设计 CPU，其设计原则是把一条指令切分成一些顺序执行的小的 stage。好处就是硬件的利用率高，challenge 就是要用一套相同的普适的 stage 来描述这 12 个指令。



这些阶段可以分为寻址、解码、执行、内存、写回、PC 更新。

程序一开始一定是从 PC 开始，知道从内存的哪里去拿下一条指令。取出来以后，计算一下当前指令的下一条指令在哪里，如果当前指令是 pop，那么下一条指令是 PC+2。然后我们进入 decode 阶段，读了对应的寄存器。然后到执行阶段，用 ALU 执行算数逻辑运算，同时有 CC 出来。第四步是读写内存。第五步是，写回寄存器。第六步是确定下一步的 PC，对于 jxx 要判断是否跳转，ret 是要从内存中得到下一步 PC 的地址。

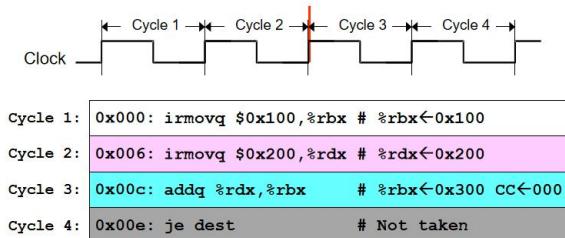
注意在此 CPU 中，寄存器文件(register file)存在两个写口(dstE 和 dstM, E 是 execute 出来的值, M 是 memory 出来的值)，因为有一个指令 pop 要同时写两个寄存器。实际上一般不用两个写口，我们假设时钟上升沿是 dstM 先写，dstE 再写。

机器只要开着，就会按照以上六步无限循环，直到停机或者碰到异常。

icode 和 ifun 分成了两簇四根线作为信号。RA 和 RB 同理。内存分为两块，指令内存和数据内存。实际上第一步 fetch 中，icode:ifun = M1[PC] 实际上就是 decode，才能知道后面有几个字节。

一个时钟周期执行一条指令，所以只有一次时钟上升的机会去更新所有待修改的值。

所以 CC、valE 什么时候写进去不影响指令的执行，可以等到时钟上升沿时加上新的 PC，三个东西同时修改。



现在关心第二条指令执行结束，第三条指令开始阶段。

每个时钟周期的最后，有三个状态即将被修改，故有 CC 新 PC memory 待修改

只有算数逻辑运算的四条指令才会修改 CC

Return 是从栈顶获得返回地址，退栈，然后跳到返回地址继续执行

Cmove 其他地方都和 move 差不多，但是在 execute 中，要根据条件码是否把原始值移动到目的中去。所以要根据 CC 和 ifun 来判断条件是否成立。

CPU 指令的执行是一个时钟周期上升到下一个时钟周期上升之前的阶段，这个时长不能太短，因为信号的传播和组合电路的计算都需要时间。只有当这些红色的信号都产生好了稳定了。我们才能触发下一条指令的执行。

本来要写的东西可以分几个阶段分别写进去，但是实现的时候我们只能等下一个时钟周期的上升沿同时把所有东西写进去。Pop rsp 定义从内存中的写指令优先写。

接下来我们要来设计 CPU

+8 -8 valC 都是从 fetch 得到的值，而 valA 是从 decode 阶段得到的值

在执行阶段中，valC 和 0

指令流和数据流是分开的，这在第六章是有意义的。指令是只读的，数据内存是可读可写的。这种分开的结构叫做 https://en.wikipedia.org/wiki/Harvard_architecture

2021/3/10

串行 CPU 的缺点：效率不高，太慢了。因为在一个 cycle 要把六步都执行完，每一步会产生新的信号，下一步有新产生的信号在往前走，要等六个阶段信号都稳定了，才能开始下一个 cycle。fetch 产生的信号到 execute 电路后，fetch 就空闲在那里，在 cycle 的其他时候，就没有起作用了。时间和硬件的利用率都很低，因此我们需要更好的实现。

流水线技术

我们先要讲流水线技术，先介绍其一般的思想。把一条指令分成独立的几步，在一个给定的时间，系统在服务很多物体。要改成流水线，把系统分成独立的 stage（三个 100ps, 3.12GOPS），每个 stage 中加 register。从 320ps 到 360ps 变慢了，但是在理想的情况下，每 120ps 就能完成一次操作，吞吐量 $1/120 = 8.33*10^9\text{ GOPS}$ 。理论上提高三倍，但是我们加了一点 register，所以没有到三倍。单条指令看变慢了，但是整体系统吞吐量变高了。为什么中间要加两个 register，中间任何一个时刻，系统会处理好几个操作。第二个 cycle 完成之前，系统在组合逻辑 A 和 B 中传播，加了这两个寄存器阻断了 A 和 B 之间，使得 A 的结果不会影响 B 寄存器中逻辑的传播。寄存器起到了隔离两段电路的作用。时钟上升后，A 和 B 的计算结果存储到寄存器中，A 准备读入新的指令。刚才流水线是平均分成三份，假设其分的并不均匀，比如 50ps, 150ps, 100ps。此时时钟会按照 150+20ps（50 和 100 对应的组合电路传播完成后等待）来跑，所以执行一条指令会使用 $3*(150+20)=510\text{ ps}$ 。这样吞吐量是 $1/510$ ，

相对于平均分较慢，吞吐率也会下降。流水线每分割一次，都要在其中加一次寄存器，流水线可以分成六部分，原则上会比三部分快，但是我们加了额外的 **overhead** 也就是寄存器的 20ps，总体上从 360ps 到 420ps，从吞吐量 1/120 提高到了 1/70。随着流水线越来越深，级数越来越多，额外开销就会越来越多。6 级流水线中，28.57%都是额外寄存器的开销。

流水线中的依赖问题

真正导致流水线有问题的原因是在一个流水线中，同时处理几条指令。但是如果指令之间有依赖关系，比如 2、3 指令需要 1 指令的结果，那必须等第一条指令做完。如果强行在最后连一条结果到输入，那么会错误的给错指令。

如何解决数据依赖问题？我们叫做 **data hazards**，另一个问题就是 **control dependence**，比如有一个条件跳转指令。要等这条指令结束才能知道下一条指令是谁，这会导致流水线甚至不知道下一条指令放什么。

流水线的实现

下面来讲流水线的实现。前面串行的实现叫做 **seq**，流水线叫做 **pipe**。**seq** 已经把一条指令分成了若干个步。改造成 **pipe** 前，要先改成 **seq+**，然后再改造成 **pipe**。如何把 **seq** 改成 **seq+**，我们把最后一步 PC (**seq** 中在最后一个 stage)。我们需要把 PC 改到第一步，因为这样第一个阶段完成后，能立马知道下一条指令是谁。这样流水线第二个周期中能继续读入指令。确定 PC 需要知道 VALP,VALC,RETURN ADDRESS，三个东西选哪个和 icode 有关系 (return 需要 pValM) , 和条件有关 (jump 和 call 和 condition jump)

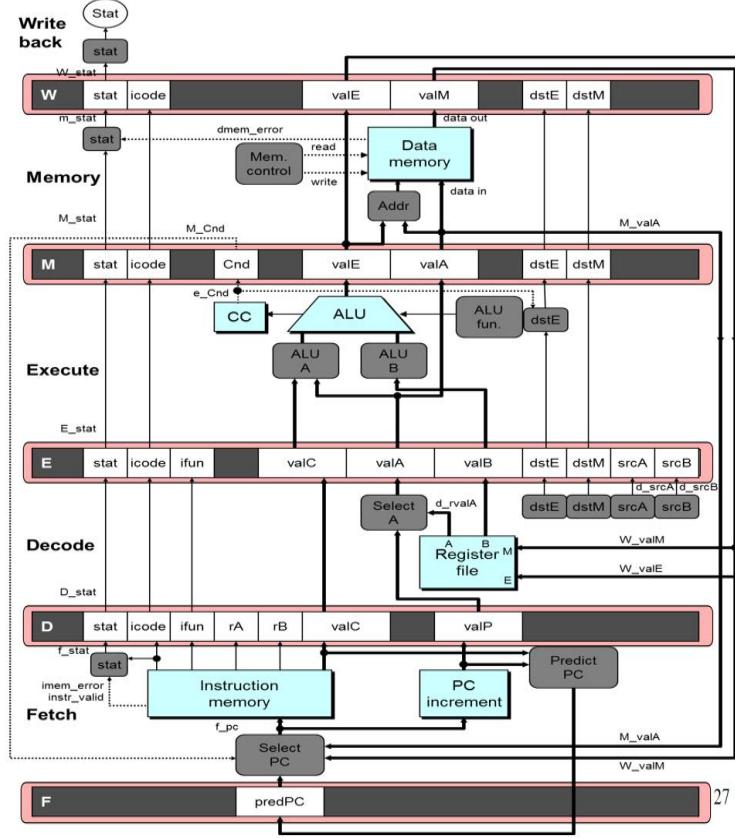
为了实现这个，把上一条指令的五个信号存进这五个寄存器。直接根据这五个信号，来确定 PC 是谁。

PC Computation

```
int pc= [
    picode == ICALL : pValC;
    picode == IJXX && pCnd : pValC;
    picode == IRET : pValM;
    1 : pValP;
];
```

小写的 **p** 代表是 **previous** 的信号，是时钟上升时，上一个阶段写入寄存器的。在 **seq** 有一个寄存器叫 PC，但在 **seq+** 中没有一个叫做 PC 的寄存器，只有 PC 的信号线。但是程序员想要的时候，还是能拿到 PC 这个值的。我们把 PC 和 Fetch 阶段合并新的 Fetch，这样就把 **seq** 的六个阶段合并成了 **seq+** 的五个阶段。我们的实现中有一个不太好的地方，因为 fetch 这个 stage 做的事情特别多。要对五个寄存器做了计算才能去取指令 (解码)，然后再把指令中的东西拿出来 (访问内存)，还要对 PC increment 做加法 (计算)。这导致了五个阶段之间，任务量不平均。这是因为从一开始，我们的分解就不是特别平衡。我们的 y86 是一个 cisc 的结构，指令是变长的。指令解码阶段本来应该放在第二步做，但是为了计算出下一个周期第一阶段的指令地址，我们需要在第一阶段中对指令解码得到 icode 传到下一步中。

Seq+分是分成了五步，但是还不是流水线化的。我们已经切成五块了。在每一块之间，我们要插入一个寄存器，这样才能把两个 stage 隔开，让相邻的 stage 之间互相不被干扰。



如上图所示，为了流水线化，我们要插入五个寄存器，初步地把原来的电路改造为了流水线。Clock register 会记录一些信号：状态信号，icode 和 ifun（这些长期需要的数据，需要持续传播）。但 ra、rb 在读写寄存器阶段是需要的，算出来以后 ra 和 rb 就没有用了，不会往前传播。而计算出的 write back 需要写的寄存器，我们需要一步一步地传上去。数据不能跨阶段传输，因为当前这条指令会在流水线中一步一步地往前走，如果直接拉到上面去，就变成了其他指令的信号了。比如说 srcA

```
#code from SEQ
int srcA = [
    icode in {IRRMOVL, IRMMOVL,
               IOPL, IPUSHL} : rA;
    icode in {IPOPL, IRET} : RRSP;
    1: RNONE; #Don't need register
];
#code from PIPE
int d_srcA = [
    D_icode in {IRRMOVL, IRMMOVL,
                IOPL, IPUSHL} : D_rA;
    D_icode in {IPOPL, IRET} : RRSP;
    1: RNONE; #Don't need register
];
```

输入输出发生了一些微小的变化，逻辑上是没有变动的。原来的地方只有一个 icode，fetch 出来的信号分裂即可。但是目前电路里有很多 icode，所以用了哪个是要区分开的。那么如何区分开呢？我们使用寄存器的第一个名字 (**D_icode**)，同样的 rA 也是使用 decode 阶

段的 D_rA

在 seq 中, call 要写内存, 要把 valP 传很远作为返回地址写到 memory 中。而在这里中, 我们没有这个电路。我们加了一个 Select A, 因为我们发现 valA 和 valP 二选一会向上传播, 所以我们把其合并为一个从而减少信号的传递。

电路的反馈部分

接下来我们关注电路的反馈部分。

1. 当前指令 fetch 后, 我们取到了 valP 或者 valC, 如果 call 和 jump 就是 valC, 如果是正常指令, 就是 valP。第一条指令刚刚结束, 反馈就来了非常及时, 这就是我们要把 seq 改成 seq+ 的理由。因为反馈很晚回来的话, 电路就没法做了。但是大部分但不完全是对的, 因为 return 和有条件跳转的情况下, 会不对。有影响的是有条件跳转, 我们要等到 execute 结束的时候, 我们才能算出 cmd, 要等到 execute 的下一个 cycle, 才能知道条件不成立 (valP) 或者条件成立 (valC)。发现错了, 我们要改回来, 已经错了怎么办? 以后再说。Return 要等到 memory 这个 stage 拿到了返回地址, 才能反馈到之前得到正确的地方。这两个我们叫做 control hazard。

- Instructions that Don't Transfer Control

- Predict next PC to be valP
- Always reliable

- Call and Unconditional Jumps

- Predict next PC to be valC (destination)
- Always reliable

这是可靠的。

- Conditional Jumps

- Predict next PC to be valC (destination)
- Only correct if branch is taken
 - Typically right 60% of time
 - Recovery: M_Cmd and M_valA (valP: next PC)

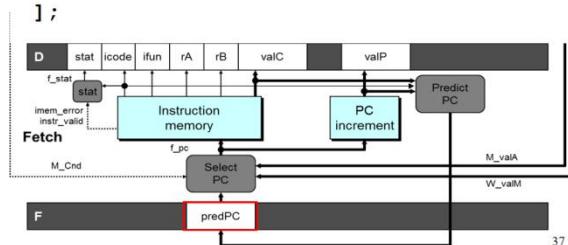
- Return Instruction

- Don't try to predict

这是不可靠的。ValC 和 valP 都是有的, 但

是我们不知道该选谁。反正是两个里面选一个, 我们就随便猜一个。我们猜 valC 总是成立, 我们发现 valC 的概率为 60%, 因为高级语言通常是循环, 跳转的次数比较多一点。我们猜错了怎么办, 一些错误的指令就进去了。Return 就不猜了, 就让程序错下去了。我们插入三条 nop 的指令, 就会不错了。我们先要做出猜测的电路。

```
Int F_predPC = [
  f_icode in {IJXX, ICALL} : f_valC;
  1: f_valP;
];
```



37

注意小写下划线是这个 stage 产生的 icode 和 valC 和 valP, 要写进状态的东西是大写的。因为我们有猜错的情况, 我们要及时的恢复。我们得到跳转条件不成立, 我们不要在从

下一条指令取了，我们拿 `M_valA` 及时修改。Return 没猜的情况，我们拿 `w_icode` 赶快把返回地址的那一条返回地址出来执行。

```

int f_PC = [
    #completion of RET instruction
    W_icode == IRET : W_valM;
    #mispredicted branch. Fetch at incremented PC
    M_icode == IJXX && !M_Cnd : M_valA;
    #default: Use predicted value of PC
    1: F_predPC
];

```

如果是条件不成立和 return 的情况，我们要先判定，并且第一条和第二条不能颠倒地写。Return 和 jxx 然后接在一起，应该 return 优先。

另一个是 data hazard，我们要往回写这寄存器，但是寄存器的计算要很晚很晚。

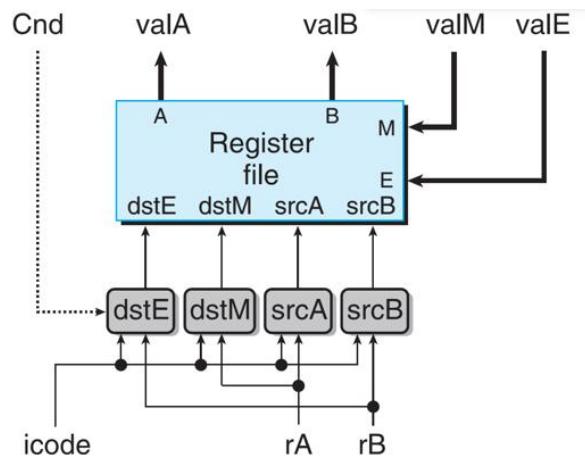
True Dependencies (Data Hazard)

Data hazard: 指令之间有 dependence, read (第一条指令的结果) after write (第一条指令计算完毕)

这叫做 true dependencies，我们要解决这个问题，让流水线正确工作。不能退回到原来的串行，因为我们还想要性能。怎么让我们程序正确执行，我们可以插入三个 nop。等待前面依赖的指令 write back 指令都结束。在执行我们当前的指令。

硬件能不能帮我插 nop？当我们发现 read after write 发生了，硬件判断这件事情发生了。让当前读的指令停在这里。把 nop 从半当中写进去。

2021/3/17



每个寄存器中有 64 位。可以用 4 位来表示 15 个寄存器。

写端口，需要两个值。一个叫做 `val_E` 和 `val_M`

写操作需要两个值，`name` (指定存储单元) 和 `value`。`M` 口读到的 `val_M` 是确定的，一定是从 `data memory` 里读出来的值往这里写。`E` 口的 `valE` 是从 `ALU` 这个地方来的(`execute` 的输出)，

可能往 rB, rA, rsp 中去写。往哪里去写是要在 decode 中计算出来的。因为现在是流水线化的，一个流水线中有五条指令，所以 decode 阶段得到的 desE 和 dstM（两个寄存器）都需要跟着流水线往前走，不能跳跃着往前走，这就会到别的指令那里去。因为在 write back 阶段才会需要到本指令的 desE 和 dstM，保证需要用的时候可以用得到。

srcA 和 srcB 是读取 valA 和 valB 的，是两个寄存器名字。这两个都是四位的寄存器。读操作是组合电路的行为模式，给定输入后在一个微小的 delay 后，输出就确定了。而写操作必须要在时钟上升的时候才能写进去。

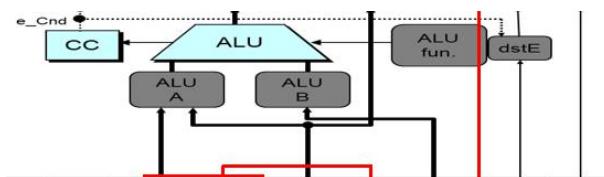
反馈问题，会导致 **data hazard** 的情况。为什么要把 PC 从第六步改到第一步？因为我们可以算出大量的 PC。程序在执行的过程中，最多的是串行指令 move、pop 等。能马上知道下一个 cycle 要取出的指令。对于 call 和无条件 jump 都是对的。

Opq 指令下一条指令的地址是由 valP（增加了的 PC）

我们还是不能确定 return 和有条件跳转 jxx。当我发现 return 的时候，我是不知道要干什么的，我们只知道 memory 之后才知道下一条指令在哪里。比较麻烦，但我们讲过一个 RISC 是比较早得到返回地址的。Return 的解决方案就是放弃。

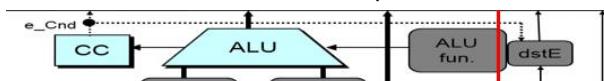
有条件跳转是二选一，条件成立的时候就跳，否则就不跳。不知道条件是否成立，条件跳转会最后由 CC 来定。二选一的比较后是 valC 胜出，循环里会大量使用跳转，所以我们去猜是 valC（选择跳的地址）。

发现错了以后，选错的两条指令在里头该怎么办？



Return 的时候，第一条错误指令已经修改了 CC。原则：绝对不能让错误的指令把 CC 写进去。

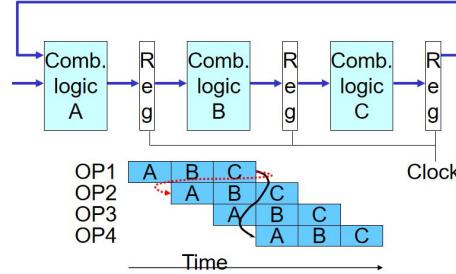
Return：反正也猜不对，就在 return 后面加三个 nop



如果成立就不改 dstE，不成立就把 dstE 改成 F

Data hazard

Data Hazards



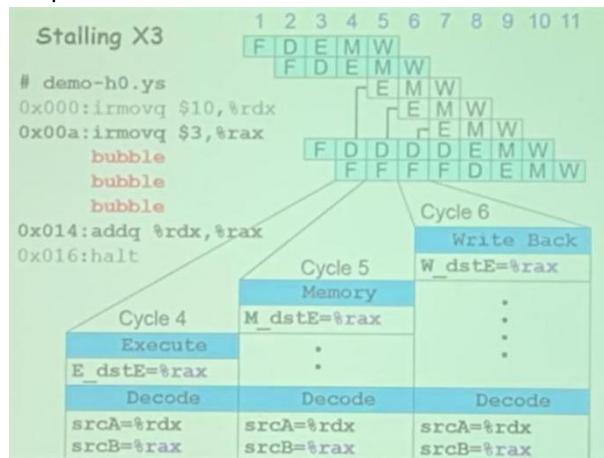
In attempting to speed up the system via pipelining,
we have changed the system behavior

16

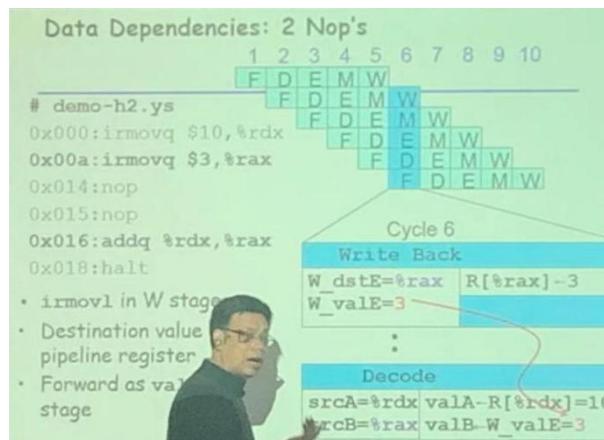
处理这个 hazards 只有加三个 nop

如果用软件加 nop, 比较土!

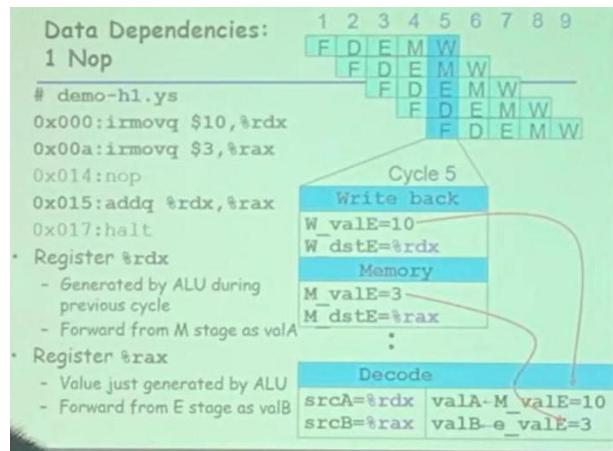
要用 rax, rax 但还没有写进去。但一个还没来得及写的 rax 在 pipe 里, 一个是 destE, 一个是 srcA。我们知道一定是错的, 此时我们可以让硬件来插入一个 nop, 即让代码待在 decode 阶段不往前走。这会导致之后的 execute 没有东西传入, 我们此时可以通过硬件在半当中 execute 插入一个 nop, 并且还要让之前在 decode 和 fetch 阶段的指令不动、原地待命。



Bubble 就是硬件 nop。插 bubble 这个操作好不好呢? 要读 rax 的时候发现还没写进去, 但是要写进去从值我们已经知道了。我们能不能直接把值拿过来用呢?

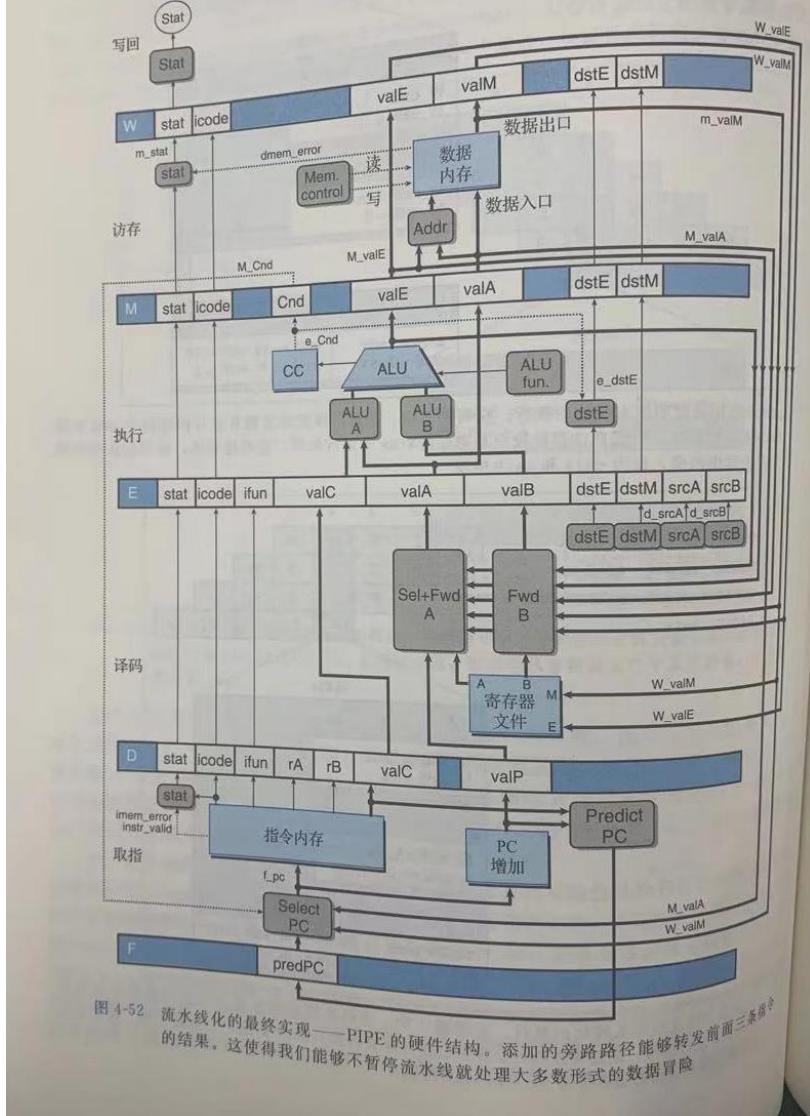


不需要等 writeback 结束了再拿来读，可以直接拿来用就可以少产生一些 bubble。



出来的时候 valP 和 valA 二选一

程序 prog2~prog4 中描述的转发技术，其实也可以转发从内存中读出的以及共用端口 E 的值。对端口 E 的值进行转发，我们可以从数据内存中读出的值（信号 m_valM）。从写回阶段，我们可以在写回阶段，我们可以转发刚刚从数据内存中读出的值（信号 m_valM）。这样一共就有五个不同的转发源（e_valE, m_valM, W_valE, W_valM 和 W_valA），以及两个不同的转发目的（valA 和 valB）。

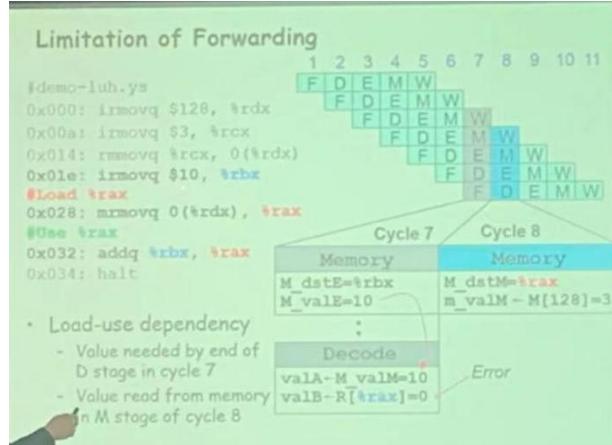


书上 P302

Practice Problem

- There is an instruction
addq %rax, %rbx
- What happens if the value needed by %rbx exists at several previous stages?
irmovq \$1, %rbx
irmovq \$2, %rbx
addq %rax, %rbx

所以使用转发的时候需要优先级，即我们要选择最近的转发值。通过 forwarding 好像不需要加 bubble 了，但是有一个例外。



Rax 只是刚刚算出了地址，还没有从地址中取到值放在 rax 里。这种情况叫做 load。这个时候就必须要插入一个 bubble。

这叫做 Load/Use Hazard

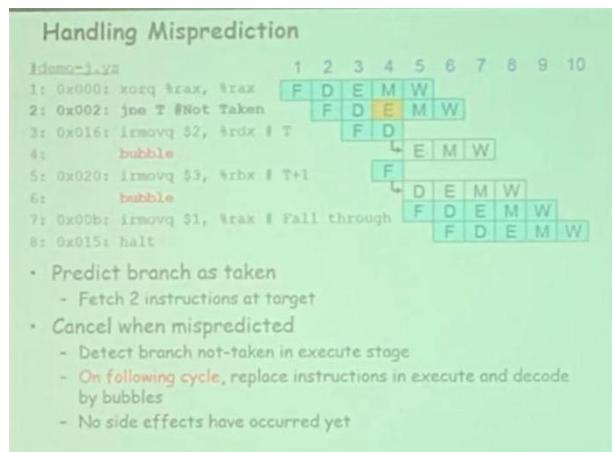
Select PC

```
int f__PC = [
    #completion of RET instruciton
    W_icode == IRET : W_valM;
    #mispredicted branch. Fetch at incremented PC
    M_icode == IJXX && !M_Cnd : M_valA;
    #default: Use predicted value of PC
    1: F_predPC
];
```

当场修正当前 cycle。

如果 return 后跟了一个 jxx，同时都成立，事实上我们应该选择 return，return 执行后 jump 就不应该执行。如果碰到 return 就插入三个 nop，这种情况就不会出现了。

如何处理错误执行的两条指令？再往上走就变成 Nop



2021/3/22

流水线中的反馈

一个是预测，一个是 valC 跳转的目的地， valP 当前指令的下一条指令

猜错的话是两个情况。

第一种情况是跳转条件不成立的情况。

第二种情况的 return， valC 和 valP 都不对。

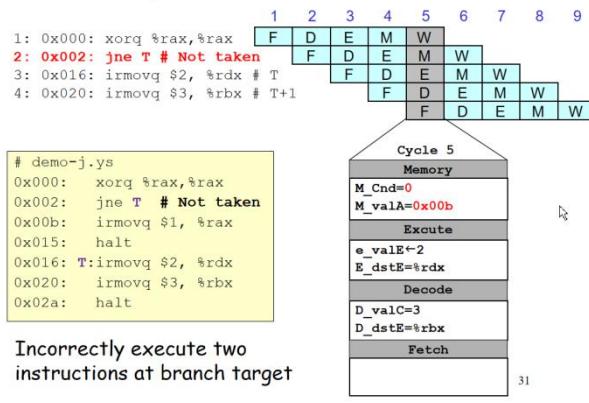
从 memory 阶段得到地址，到 write back 这个阶段，我们才能把 return 的地址写到 PC 里去。

我们要修正的时候，已经有三条指令读进去了。

碰到 return 的时候，当我找到正确的指令的时候 write back 把正确的指令反馈到 fetch。

所以在 return 的时候，我们要插入三个 nop。

Branch Misprediction Trace



在 memory 得到正确地址中，程序的状态（CC、PC、memory、register）还没被修改。CC 要时钟上升了才写进去，现在还没上升。

在 execute 阶段发现 condition 不成立，这个时候不去写这个指令。发现错误的时候，让两条错误的指令不往前走了。把 icode 和 ifun 替换为 nop。

为什么知道地址错了以后不直接把正确地址拉过来？当我算出 condition 的时候，这个 cycle 快结束的时候要让 fetch 重新算一遍。这会导致 fetch 这个阶段时间非常长，但是我们要避免一个 stage 时间特别长。

Y86 中的 Exception

Y86 中的 exception 是处理器的状态发生的变化以后，CPU 要对这样的状态变化（event）做出一个反应。

Exceptions

- Conditions under which processor cannot continue normal operation
- Causes
 - Halt instruction
 - Bad address for instruction or data
 - Invalid instruction
- Typical Desired Action
 - Complete some instructions
 - Either current or previous (depends on exception type)
 - Discard others
 - Call exception handler
 - Like an unexpected procedure call

一个是停机状态，一个是读到了非法的地址，一个是指令本身是错的。如下例所示：

```
Inst1  
Inst2  
Inst3
```

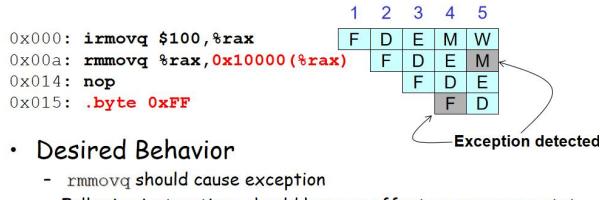
如果 `inst2` 出现了问题，按照第八章的 `exception`，当前这条指令之前的指令都应该完成。这条指令本身 (`inst2`) 该怎么办？有些 `exception` 是要执行的，而有些是不执行的。比如 `syscall` 本身是要执行的，而在 `y86` 中 `halt` 是要执行的，而非法地址、非法指令、`page fault` 是不能做的。

```
# demo-exc1.ys      I  
irmovq $100,%rax  
rmmovq %rax,0x10000(%rax) # invalid address  
nop  
nop  
halt             # Halt instruction
```

两条同时都会抛 `exception`，肯定是之前的先触发。

Exceptions in Pipeline Processor #2

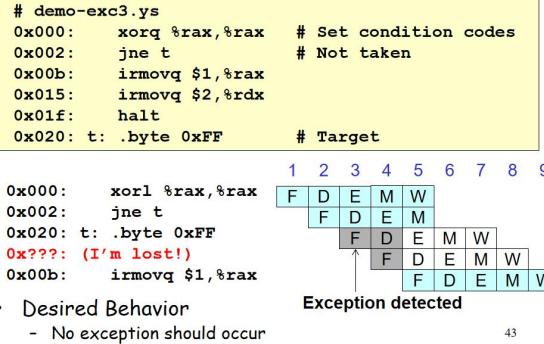
```
# demo-exc2.ys  
irmovq $100,%rax  
rmmovq %eax,0x10000(%rax) # Invalid address  
nop  
.byte 0xFF          # Invalid instruction code
```



- Desired Behavior
 - `rmmovq` should cause exception
 - Following instructions should have no effect on processor state

但在流水线中，下一条指令的非法指令比非法地址抛出的要早，怎么办呢？从道理上应该先抛上一条指令的 `exception`。所以我们不是发现 `exception` 就要立马抛出。

Exceptions in Pipeline Processor #3



43

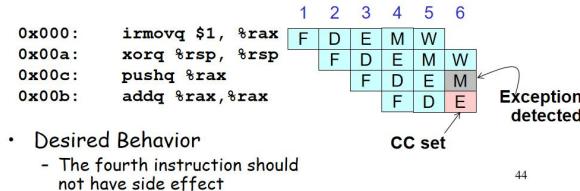
开始是非法的，后来发现是不需要的。所以我们不能过早地抛出异常。

Exceptions in Pipeline Processor #4

```

# demo-exc4.ys
0x000: irmovq $1,%rax
0x00a: xorq %rsp,%rsp # Set %rsp to 0 & Set CC to 100
0x00c: pushq %rax # attempt to write 0xfffffffffffffff8
0x00e: addq %rax,%rax # (Should not be executed)
                      # Would set CC to 000

```



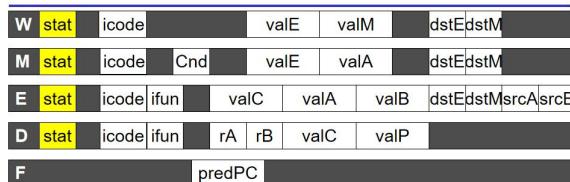
44

一定要在时钟上升之前阻止 CC 写进去。

Fetch 阶段发现 halt，指令不对，指令地址不对。

Memory 阶段要读写数据，发现异常。

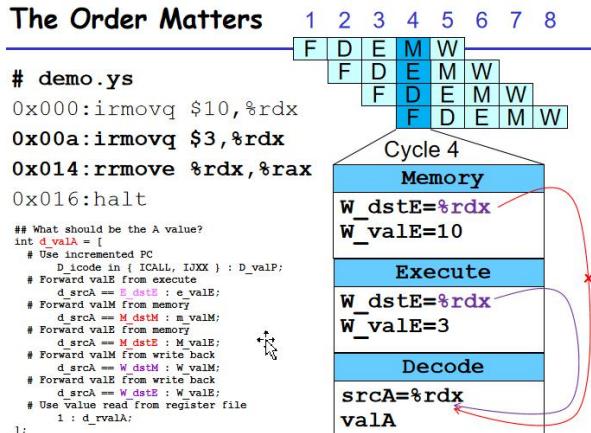
Maintaining Exception Ordering



- Add exception status field to pipeline registers
- Fetch stage sets to either "AOK", "ADR" (when bad fetch address), "HTL" (halt instruction) or "INS" (illegal instruction)
- Decode & Execute pass values through
- Memory either passes through or sets to "ADR"
- Exception triggered only when instruction hits Write-back

fetch 阶段，发现异常不能马上抛出来，要记录下来，直到 write back 的时候，确定要不要做完。所以我们要增加一个 stat 往上传递。后面的指令跟着到写 memory 的阶段时，不能置 CC，也不能写 memory。

因为有 forward 拿当前 sourceA 和 sourceB 拿 dstE 和 dstM 比较



Forwarding 的时候，选择顺序有关系。所以 EMW 要按顺序，dstM 和 dstE 不能写颠倒。

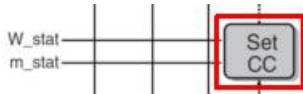
Write Back

```

int Stat = [
    # SBUB in earlier stages indicates bubble
    W_stat == SBUB : SAOK;
    1 : W_stat;
];

```

判断如果是 bubble 了，就不发生错误



写 CC 有两个条件

```

# Should the condition codes be updated?
bool set_cc = (E_icode == IOPL)
    # State changes only during normal operation
    && !m_stat in { SADR, SINS, SHLT }
    && !W_stat in { SADR, SINS, SHLT };

```

之前的流水线中发生错误的时候，就不写 CC

- Detection

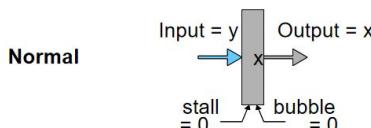
Condition	Trigger
Processing ret	IRET in { D_icode, E_icode, M_icode }
Load/Use Hazard	E_icode in { IMRMOVL, IPOPL } && E_dstM in { d_srcA, d_srcB }
Mispredicted Branch	E_icode = IJXX & le_Cnd

- Action

Condition	F	D	E	M	W
Processing ret	stall	bubble	normal	normal	normal
Load/Use Hazard	stall	stall	bubble	normal	normal
Mispredicted Branch	normal	bubble	bubble	normal	normal

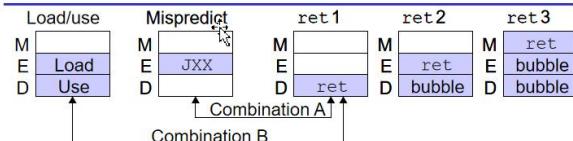
Bug:

Pipeline Register Modes



Stall 和 bubble 都是 1 的时候，行为未定义

Control Combinations



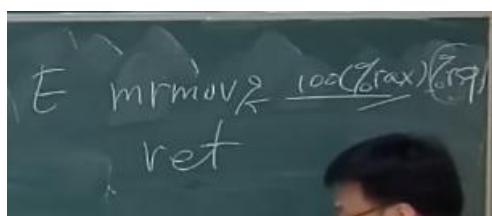
- Special cases that can arise on same clock cycle

- Combination A
 - Not-taken branch
 - ret instruction at branch target
- Combination B
 - Instruction that reads from memory to %rsp
 - Followed by ret instruction

31

比如

Mrmovq 跟着 ret



ret 要读 rsp, 读不到怎么办, return 必须停在这个地方。等待上一条指令往前走, 所以这种情况 bubble 和 stall 都要为 1 的情况下, 设置 stall 为 1,

Corrected Pipeline Control Logic

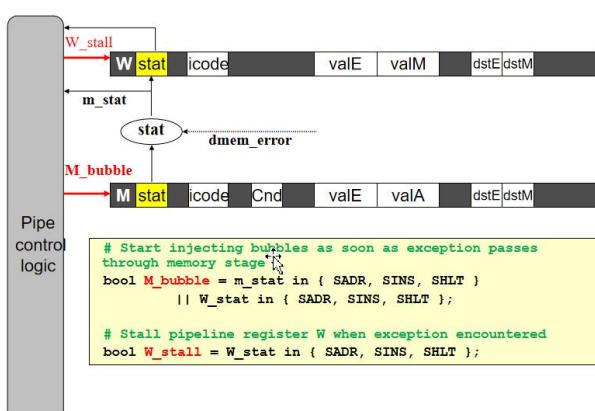
```
bool D_bubble = 
    # Mispredicted branch
    (E_icode == IJXX && !e_Cnd) ||
    # Stalling at fetch
    while ret passes through pipeline
    IRET in { D_icode, E_icode, M_icode }
    # but not condition for a load/use hazard
    && !(E_icode in { IMRMOVL, IPOPL } 
    && E_dstM in { d_srcA, d_srcB });


```

Condition	F	D	E	M	W
Processing ret	stall	bubble	normal	normal	normal
Load/Use Hazard	stall	stall	bubble	normal	normal
Combination	stall	stall	bubble	normal	normal

36

Ret 发生时, 排除掉同时发生 load/use hazard 的情况



38

发生 exception 的情况, M 的指令就停在这个地方。

性能计算方法

最后我们考虑一下 CPU 设计完的以后，我们考虑一下性能。有两种计算方法。

第一种是绝对的 1G 还是 2G 还是 500M，按照主频（每秒钟发出多少个 clock cycle）

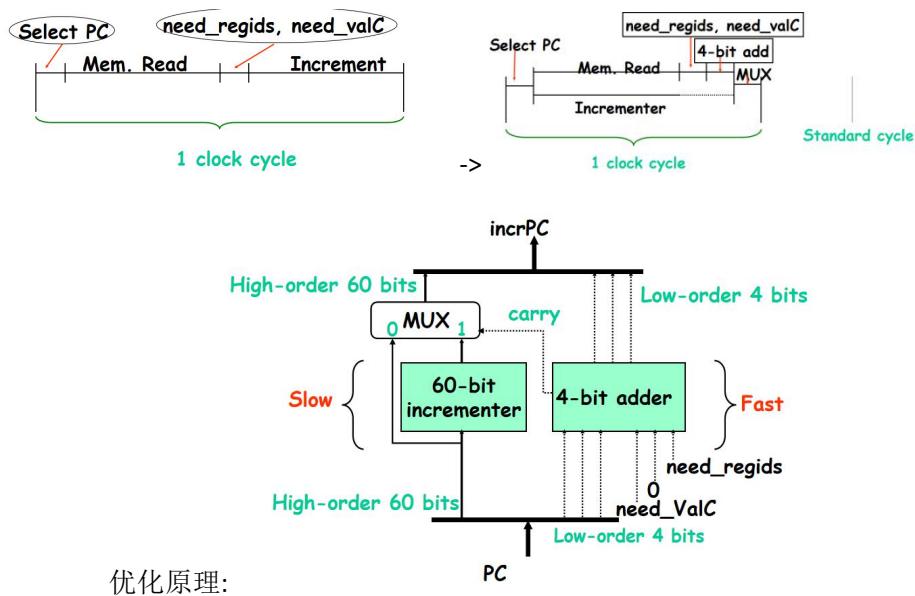
第二种是相对的，执行一条指令需要几个 cycle。一般来说，碰到 return 增加 3 个，

CPI for PIPE

- $CPI \approx 1.0$
 - Fetch instruction each clock cycle
 - Effectively process new instruction almost every cycle (throughput)
 - Although each individual instruction has latency of 5 cycles (latency)
- $CPI > 1.0$
 - Sometimes must stall or cancel branches

2021/3/24

在实际设计的过程中，尤其是 Y86，我们希望是每一个 stage 工作量差不多。现在我们的 fetch 是读寄存器，forwarding 是五选一，三个 valA 的选项。Execute 就是做一个 ALU。Memory 是读写 Memory。Write Back 是写寄存器。Fetch 是读 instruction memory，作判定，做 PC 加法，操作比较多，所以这个阶段特别长。有没有办法变短一点？目前 Select PC 和 PC increment 是一个串行任务，PC increment 只需要加 1、2、9、10. 64 位加法中，前六十位可以不需要 need_regids 和 need_ValC 就可以直接计算。后四位需要之后再计算。



最后用 HLC 写一个 PIPE，就要去思考线路是这么走的。

第五章其实还在讲处理器的结果，第四章叫做 pipeline 是现代处理器的基础。

Super scalar 超标量结构，可以同时执行几条指令 $CPI < 1$

我们要设置一个指标，主频快 CPI 也要小才有意义。

相对指标：Cycles Per Element 处理一个元素所要花的时间。

浮点寄存器%xmm (128位) 或%ymm (256位)

Ymm256位可以分成4个64位，其同时做运算，互相不受干扰。

优化主要就是减少函数调用次数和对内存的访问次数，为什么这两件事情不能让编译器来做。

因为我们有一些阻碍优化的程序因素（optimization blockers）。

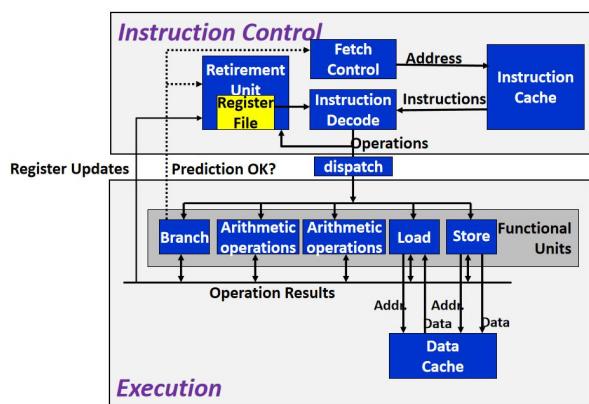
编译器优化的时候非常保守，不能做可能违背程序行为的优化。Alias 和 side effect。

现代处理器的乱序执行

接下来去做机器有关优化，所以要知道现代处理器的工作原理。

OutofOrder 乱序执行，分成两块：一块是执行，一块是指令控制。

Y86是标量处理器，一条指令处理一个标量。



指令存储和数据存储是分开的，而且比起存在内存里，存在了 cache 里。目标：同时执行好多条指令。

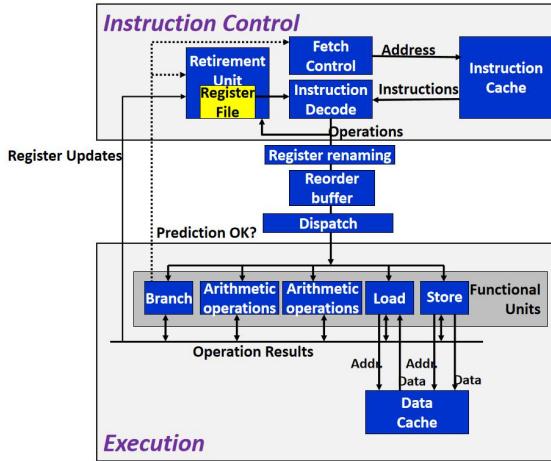
FetchControl 给一个地址拿一条指令，接下来就想办法拿下一条指令。第二部分就是 instruction decoder，它把一系列汇编指令分解为一些微指令。在这过程中会使用一些只有硬件可以使用的中间寄存器。有一个 renaming table

2021/3/31

超标量处理器

上节课讲了机器无关优化，进一步做优化可能就要根据处理器的特性。上节课就提到了超标量处理器（一次可以执行多条指令）。

Intel 为了做流水线，把一条指令翻译成几条。比如从内存中取值、计算乘法。里面寄存器要重新命名，比如 xmm0 又读又写，把读变成 xmm0.0 把写变成 xmm0.1. 比较指令会放到一个 bit 的寄存器中。这些中间寄存器都是系统内部的，汇编程序员是不能用的。产生比很多的 miu-code 放到 reorder buffer 中。



我们尽量的让指令同时执行。超标量提供了多个执行部件（多功能部件），每个部件可以在每个时刻执行一个 miu-code。比如 Core i7，这样的核里头，有 8 个多功能部件。处理整数加法有四个部件，浮点乘法有两个，load 有两个，store 只有一个，部件 6 还可以计算 cmp 操作。

- Haswell CPU (Core i7)

0. Integer arithmetic, FP multiplication, integer and FP division, branches
1. Integer arithmetic, FP addition, integer multiplication, FP multiplication
2. Load, address computation
3. Load, address computation
4. Store
5. Integer arithmetic(basic operations)
6. Integer arithmetic, branches
7. Store address computation

一条指令进去到出来有经过三个 cycle。除法没有流水线。

Miu-code 是存在 reorder buffer 里头。哪些 miu-code 可以分配到下面多功能部件去执行呢？两个原则：1.多功能部件空闲 2.指令本身是能够执行的

比如 mulq t.1, %xmm0.0 要等内存中的数据取出来存到 t.1 后才能执行，在没有取出 t.1 时，指令是不能执行的。

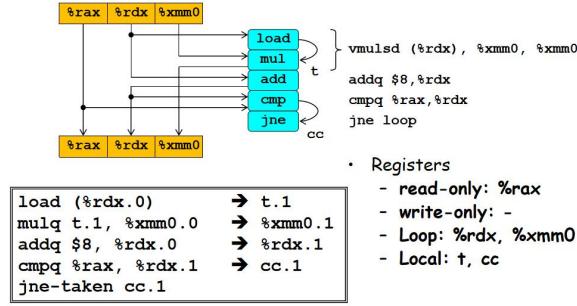
这件事情就是要靠 dispatch 来判断，如果满足了这两个条件，就送到相应的多功能部件中去执行。

上节课说的一个优化程序：

1. 把 len 提出循环
2. 把 data 头指针提出循环
3. 把循环中写内存改为写寄存器

数据流图和关键路径

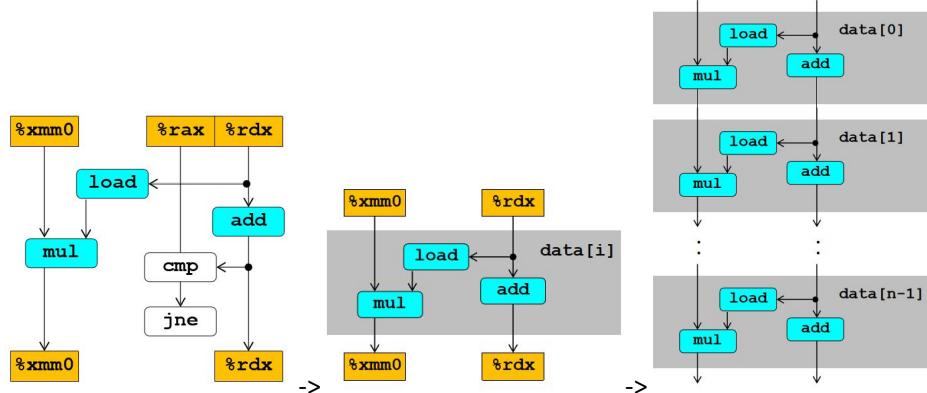
为了判定这样子优化是不是最优的，书上提供了数据流图。看关键路径到底有多长，可以得到程序最快长什么样子。



mul 接受 t 和 xmm0，存到 xmm0 里去。

由于 rax 直接从上面的 rax 拉下来，所以 rax 没有被写，是只读的。

t 和 cc 是局部寄存器，不需要往下传。



Cmp 和 jne 只影响循环内部。而蓝色的三个东西影响的是循环的这次迭代的输出。因为这次输出是下一次循环迭代的输入。蓝色连着的线表示了这次循环的数据流，我们发现 add 和 load 可以同时进行，因为它们的指令不影响，多功能部件也不冲突。

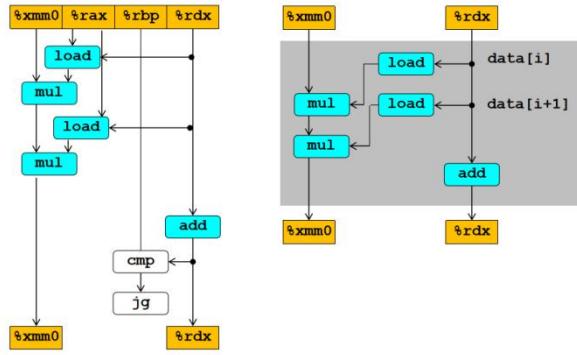
第二个问题是，t 和 cc 这样中间值，该如何传？我们有一个类似于总线一样的东西，可以把操作值输出到总线上，送到对应的寄存器上，这是一种高级的 **forwarding** 技术。

我们可以考察关键路径上的延迟。左边路径上关键的是 mul 浮点操作，其需要 5 个 cycle。

因为上一个循环的 mul 还没结束，这个循环的 load (4 个 cycle) 就开始做了。所以上一层的 mul 和这一层的 load 是同时结束的。Load 因为可以是流水线，大家可以一起开始做。但是乘法没有办法，有数据依赖，必须一个个做。所以乘法是关键路径。这个循环的消耗 cycle 数就是 5。

为什么整数加法是 1.27 而不是 1 呢，因为 cmp 和 jne 也要使用整数加法的多功能部件，一个循环中有四个多功能部件，这会导致有多余的 overhead，包括调度的时间。

现在我们的加法是 1.2，没到 1.下一步，我们的目标是把加法搞到 1.我们要做一下针对这个结构，去做一些有关的优化。比如 4 个加法部件不太够，我们该怎么样减少一些加法运算。本来每个循环中做一个加法，我们可以每次做两个加法。展开循环到每次 i+2，(最后奇偶的情况特判就行)，这叫做 **loop unrolling**。现在就是每两次 load 和 mul，比较一次加法。



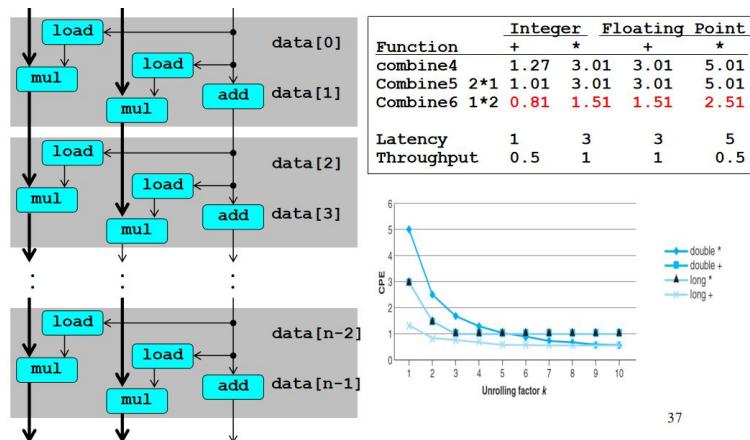
每处理两个元素，才有一次加法运算。加法的平均次数减少了。关键路径没变短，只是非关键路径上的计算变少了。展开 2 次到 3 次，都会从 1.27 下降到接近 1.

Function	Integer		Floating Point	
	+	*	+	*
combine4	1.27	3.01	3.01	5.01
Combine5*2	1.01	3.01	3.01	5.01
combine5*3	1.01	3.01	3.01	5.01
Latency	1	3	3	5
Throughput	0.5	1	1	0.5

基本上我们就达到了给定体系下的解。为什么浮点乘法的吞吐量是 0.5？因为有两个乘法计算部件。为什么有 4 个加法单元，但是吞吐量不是 0.25 而是 0.5？因为加法的速度被 `load` 限制了，`load` 只有两个。我们怎么让我们的程序到达系统的最优解？目前来说，乘法是有数据依赖的，限制了速度。

整数乘法的展开优化

怎样增加不相关的乘法，使其能够流水线计算呢？我们可以分布地来计算，换句话说，奇数归奇数，偶数归偶数进行计算。这样奇偶的乘法就无关了。最后再把最终结果合在一起即可。这样增加了并行性。但是这样运算在浮点运算是有问题的，因为浮点运算是不能做交换律和结合律的。在整数乘法的时候，编译器会帮我们做这个优化，但是浮点数乘法的时候，编译器是不敢这样做的。所以程序员主动去这样优化的时候，要保证优化结果造成的误差是在接受范围内的。



多展开一点，比如展开到 10 左右，就是 0.5 了。

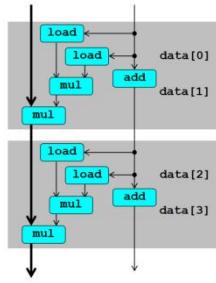
```

void combine7(vec_ptr v, int *dest)
{
    int i;
    int length = vec_length(v), limit = length-1;
    data_t *data = get_vec_start(v);
    data_t acc = IDENT;

    /* combine 2 elements at a time */
    for (i = 0; i < limit; i+=2) {
        acc = acc OPER (data[i] OPER data[i+1]);
    }

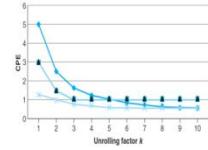
    /* finish any remaining elements */
    for (; i < length; i++)
        acc = acc OPER data[i];
    *dest = acc ;
}

```



Function	Integer		Floating Point	
	*	+	*	+
combine4	1.27	3.01	3.01	5.01
Combine5	2*1	1.01	3.01	5.01
Combine6	1*2	0.81	1.51	2.51
Combine7	2*1	1.01	1.51	2.51

Latency 1.00 3.00 3.00 5.00
Throughput 0.50 1.00 1.00 0.50



44

加法一做完，下面的两个 load 就可以开始做了。就等于四个 load 在流水线里头，load 结果出来以后，相邻循环的上 mul 就相邻开始做了。等上循环的下 mul 开始的时候，上循环的上 mul 已经结束了。所以相邻循环的下 mul 也是相邻开始做的（流水线做）。所以也能有提升。

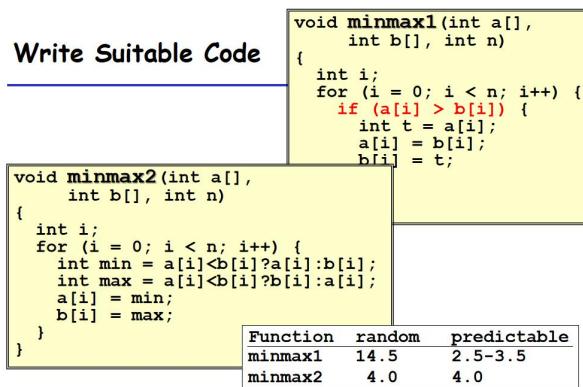
在这种情况下，展开十次以后，也能使得最终结果到达超标量这个结构的最好的结果。针对体系结构的优化，使得浮点数乘法从 5 变为了 0.5，提升了十倍性能。但是循环展开不是无限可以做的，比如我们不能展开二十次，因为浮点寄存器只有十六个，不能超过寄存器个数展开循环，否则寄存器溢出（不够）以后就要去读写内存，导致性能降低。

分支预测

最后，我们来讲一下分支预测（branch prediction）。我们一次要取出很多指令，解码后扔给很多的部件去执行。碰到跳转指令以后，我们就不知道要取哪个指令，我们必须去猜。这时候我们就叫做 speculative execution（投机执行）。有可能猜错，所以在这种指令执行的时候，即不能修改寄存器也不能修改内存，即用户可见的状态是不能被修改的。发现跳转不对的时候，这些 invalid 的指令都要清除掉。再去做正确的指令。因为会有错误的指令进来，所以不能轻易地修改内存和寄存器。什么时候可以修改呢？我们需要一个部件叫做 retirement unit。我们需要把指令按照原来的顺序留着，要遵循 sequential semantics。必须等到我们确定了错误指令消除了以后，执行了正确的指令，之前的指令才能真正把寄存器修改掉。

Retirement unit 需要按照先进先出来保留指令。如果指令正确地执行结束了，就可以修改寄存器和内存。所以 retirement unit 管理 register file。

当我们猜错的时候，就会有很多错误的指令进入到部件中。在 y86 里，猜错是浪费两个 cycle。多功能部件中，猜错的代价约为 19 个 cycle，这对于一般的循环，影响不是很大。但是对于很短的 if 来说，比较麻烦。



要是猜错了就错的很多。如果条件对错是随机的话，完成一个元素就是 14.5 个 cycle。

如果每次能猜对，那么就是 2.5~3.5 个 cycle。在这种情况下，我们将比较运算用 cmove 来实现。这样使用的 cycle 数量都是 4 了。

2021/4/7

上节课我们充分发挥了指令集的并行性（乱序执行），最后基本上都达到了整个结构提供的最高吞吐。超标量结构需要不断地提供指令，当一条指令取出来以后，不能等它执行结束，需要通过预测来取新的指令。指令执行过程中要修改寄存器和内存这些程序员可见的状态，但是如果预测错的指令修改了就很难修复。所以我们通过 register renaming，暂时写到临时的寄存器里去。等到指令 retire 的时候，也就是顺序结束的时候，确定没有预测错误了，才能把寄存器的修改真正写入到对应的寄存器里去。我一条指令，store 向 memory 中存数据，如果是猜测并且最后发现猜测错误，这条指令在执行的过程中，也不能真写入 memory。那 load 指令从哪里读呢？因为乱序执行的特性，在 store 接着 memory 后的情况下，可能还需要特殊处理一下。

这边我们有一个数据结构 list_ele 和 *list_ptr，实际上就是一个链表。有一个函数计算链表的长度。Load 是一个流水线，每个 cycle 可以扔进去一个，等 4 个 cycle 才能结束。因为第二个要等第一个的结果，要等 load 完了才能继续，所以这个程序的 cpe 是 4.

void write_read(long *src, long *dest, long n)		Example A: write_read(&a[0], &a[1], 3)			
		initial	Iter.1	Iter.2	Iter.3
cnt		3	2	1	0
a	-10 17	-10 0	-10 -9	-10 -9	-10 -9
val	0	-9	-9	-9	-9

Example B: write_read(&a[0], &a[0], 3)					
		initial	Iter.1	Iter.2	Iter.3
cnt		3	2	1	0
a	-10 17	0 17	1 17	2 17	2 17
val	0	1	2	3	3

Function	CPE
Example A	1.3
Example B	7.3

6

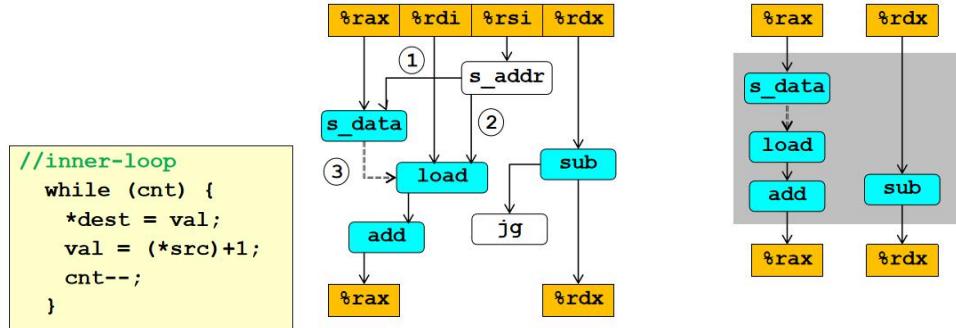
Dest 是往里面写，src 是从里面读出数据。写读会重复 n 次。

Example A 是两个不相同的地址，循环三次。Load 归 load，store 归 store，没有 dependency。

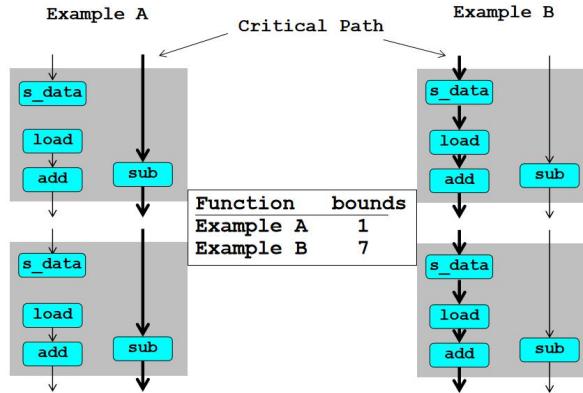
Example B 是两个相同的地址，读写在同一个地方。叫做 alias。

我们要引入一个 store buffer，我们不能直接去写 cache，因为这条指令是否要执行我们不知道。Store unit 是一个地址数据对。一个进程再执行时有用户态和内核态。前两年有一个攻击，程序在内核态执行的时候也是乱序的，intel 处理器为了性能，从 kernel 态到用户态的时候，store buffer 的东西还保留着。用户态可以看到内核态的一些信息。

我们回到 example B，load 的数据一定是刚 store 进去的，数据一定还存在于 store unit (store buffer) 中，所以 load 可以先通过 load unit 查询 store buffer，如果没有再从 data cache 中去拿。



如果 dest 和 src 地址相同，那么③这根线就存在，可以简化为最右这张图。



根据这根虚线是否为实线，不同例子下关键路径会有所不同。主要是要记住，我们引入了 store buffer，使得 load 比较复杂，要和最近的 store buffer 中的地址比较一下。

给你一个文件，统计里面的单词词频，最后再降序排列。现在是，我们要测试一个 n-gram $n=2$ ，要测试连续的 n 个单词组，出现的次数。（两万多个单词，有 36 万个 bi-gram）

基本做法：所有字符变为小写，用一个哈希，如果哈希冲突了就链接在哈希值结点链表最后。全部做完以后来一个排序。保研上机考三小时就做一道题。

第一章里介绍了 Amdahl's law 讲了程序优化效果会怎么样。一个程序跑 T_{old} 时间，优化以后跑 T_{new} 时间，一个程序可以优化的长度是 α ，加速了 k 倍。剩下一部分是不能优化的，叫做 1. 极限程度下， k 可以变成无穷大，极限加速比是 $1/(1-\alpha)$ ：

$$S = T_{old} / T_{new} = 1 / [(1 - \alpha) + \alpha / k]$$

$$S_\infty = 1 / (1 - \alpha)$$

$$T_{old} = 209.0$$

$$\alpha = T_{part} / T_{old} = 203.7 / 209.0 = 0.974$$

$$S_\infty = 1 / (1 - \alpha) = 39.0$$

$$T_{old} / T_{new} = 209.0 / 5.4 = 38.5$$

我们最终实验发现，加速比可以到达 38.5，理想加速比是 39. 接下来的问题是我们如何找到这部分，程序中花的时间最多的部分就是 hottest part。

插桩分析 (profiling)

我们使用 profiling 的，在编译的时候插桩一些分析函数进去。最终我们可以统计到每个函数执行了多少时间。https://blog.csdn.net/qq_20798591/article/details/105241074

Gcc 会额外产生一个 gmon.out 的统计文件。

% time	cumulative seconds	self seconds	calls	self s/call	total s/call	name
97.58	203.66	203.66	1	203.66	203.66	sort_words
2.32	208.50	4.85	965027	0.00	0.00	find_ele_rec
0.14	208.81	0.30	12511031	0.00	0.00	Strlen

Each line describes one function

name: name of the function

%time: percentage of time spent executing this function

cumulative seconds: [skipping, as this isn't all that useful]

self seconds: time spent executing this function

calls: number of times function was called (excluding recursive)

self s/call: average time per execution (excluding descendants)

total s/call: average time per execution (including descendants) ²³

By default, the timings for library functions are not shown

Sort 是排序, find_ele_rec 是在 hash 链上找元素。

时间是如何统计出来的呢？插桩编译运行以后，定时地在一个时间段里产生一个中断，查看停在那一条指令上，这条指令对应的是哪个函数。这个函数就记一个数。

calling history in the report

index	%	time	self	called	name
				158655725	find_ele_rec [5]
		4.85	0.10	965027/965027	insert_string [4]
[5]	2.4	4.85	0.10	965027+158655725	find_ele_rec [5]
		0.08	0.01	363039/363039	save_string [8]
		0.00	0.01	363039/363039	new_ele [12]
				158655725	find_ele_rec [5]

· Ratio : 158655725/965027 = 164.4

· The average length of a list in one hash bucket is 164

1.5 亿是递归调用的次数, 96 万是正常调用的次数。所以 1.5 亿/96 万=164 就是每次正常调用里递归的次数，那就是平均插入一个元素，在哈希值结点的链表中要往下找多少次。

把自己的排序变为标准库中的快排	209->5.
把插入元素进链表的最前头	5.->6.
把插入元素进链表的最后头（词频出现比较高的单词，出现的比较早）	6.->5..
原来哈希只有 1021 个 bucket，而 bi-gram 有 36 万个，平均长度是 355.6 个 bi-gram。之前	5..->5...

是平均每次调用递归 164 次，也就是平均查一半的长度可以查到。那么我们能不能把长度缩短一点呢？我们把 bucket 变大，变成 20 万个。我们发现变好了，但是没好太多。因为哈希简单地使用了每个字符的 asc2 码加起来，最大的就是 3371.用不到这么多 bucket。	
我们选用更好的哈希函数，使用移位异或作为新的哈希函数。此时我们发现 strlen 这个 0.3 在 0.4 中有很大比例。Strlen 我们是在把字母变小写的时候，使用这样的一个算法。	5...->0.4
我们先把循环中的 strlen 提出到循环外。	0.4->0.2

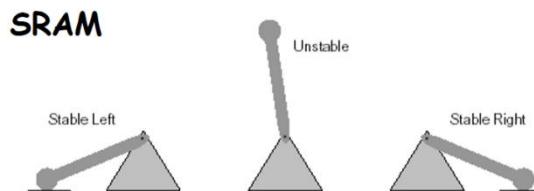
我们使用了库函数把时间降下来了，最后把 hash 和链表这部分变快了。

我们发现 gprofile 是和数据有关的。Gprofile 是一段时间停一下，对于程序执行时间太短的程序没什么用。

RAM (random access memory)

第六章和组成是比较密切的。我们首先会介绍很多存储。计算机中存信息有很多介质。这些介质我们会一个个来介绍。首先是内存条，叫做 RAM (random access memory)。内存条就是一个芯片，里面最基本的存储单元叫做一个 cell。一个 cell 会存一个 bit。内存条是 dynamic RAM (dRAM)，还有一类是 cache，是 static RAM (sRAM)。区别在于，sRAM 的一个 cell 是由 6 个晶体管构成。而 dRAM 的一个 cell 只有一个晶体管和一个电容。

sRAM 通电的情况下，可以长期存储这个信息，抗干扰性很强。

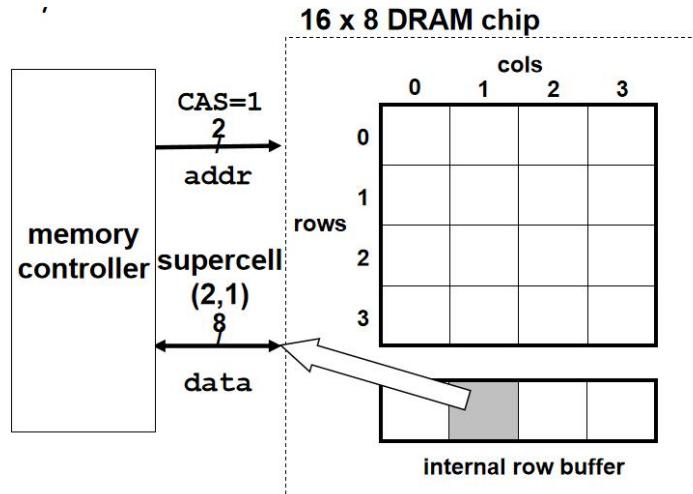


而 dRAM 不太一样，这个值存在那里，一开始是 1，过一会儿会自动漏电，值会掉。所以每过一段时间要去刷新一下这个值，否则这个值就没了，电路比较简单，容易被干扰。所以 dRAM 比较便宜，sRAM 比较贵。但是 sRAM 速度比较快并且抗干扰。

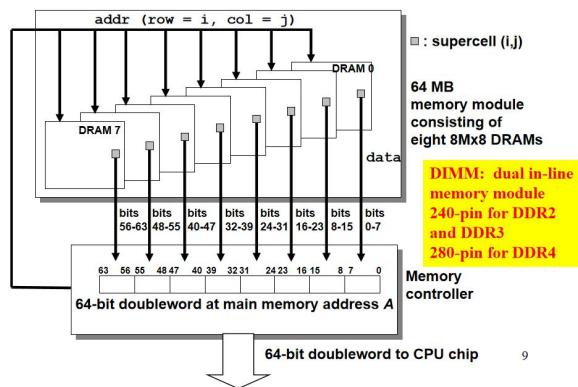
dRAM	sRAM
内存使用，集成度高	容量大
便宜	贵
不抗干扰	抗干扰
电路简单	电路复杂
定期刷新	只要有电，数据会一直存在，不需要定期刷新

DRAM

D*w dram 每一个方块叫做一个 supercell



十六个变成 4×4 的东西，有一个内部的行缓存。这个虚线就构成了内存，内存要和 CPU 打交道一定要通过一个 memory controller（负责内存和 CPU 之间的通信）。CPU 要访问内存就是把地址给 memory controller（16 个地址， 2^4 ），然后 memory controller 切成一半（ 2^2 ），前一半叫做行地址，后一半叫做列地址。然后我们可以找到这个数据，返回给 CPU。高的两位叫就行地址拿过来，我们把一行都选出来，copy 到内部行 buffer 里去。再送一个列地址，我们再从 buffer 中找到列地址中对应的 supercell，把八位数据返回给 CPU。这样只能访问一个 byte。



如果我们要同时访问 8 个 byte，我们就使用八片这样的东西，先取行再取列。最后拼起来，我们就得到了这个六十四位的数据。内存条上有很多引脚，有几百个，包括地址线、数据线、控制线等。这是最基本的 dRAM。

- All enhanced DRAMs are built around the conventional DRAM core
- Fast page mode DRAM (FPM DRAM)
 - Access contents of row with [RAS, CAS, CAS, CAS, CAS] instead of [(RAS,CAS), (RAS,CAS), (RAS,CAS), (RAS,CAS)].

在此之上有很多新的实现。比如为了访问数组，数组里的数据通常是存在同一行中的，可以发一个行，若干个列。

Extended data out DRAM (EDO DRAM)，这个模型下，CAS(列地址的信号)挨得比较近。

以上这些都是异步的，memory controller 把数据送出去以后到获取相应的数据后，不知道要等多少时间。还有时钟控制的同步的 dRAM，等待固定 cycle 后肯定就有数据了。

2021/4/14

最基本的 DRAM，访问的时候是 memory controller 把地址分成行和列。先把一整行放到 buffer，再访问列。

再发展就到现在主流的 DDR，上升沿和下降沿都可以用来控制，只需要等半个 cycle。DDR4 (16bits) 一次能够取出多少个列。

另外一个，在显卡里用的比较多的就是 video RAM，支持同时读写。

DRAM 和 SRAM (static ram) 都需要有电才能维持信息，这叫做易变的内存。停电以后保留信息的叫做非易失内存，最早的叫做 ROM (read-only memory)。其实在出厂的时候是可以写一次的。最初开机的时候，最初的程序的信息开始运行，放在 ROM 中。、

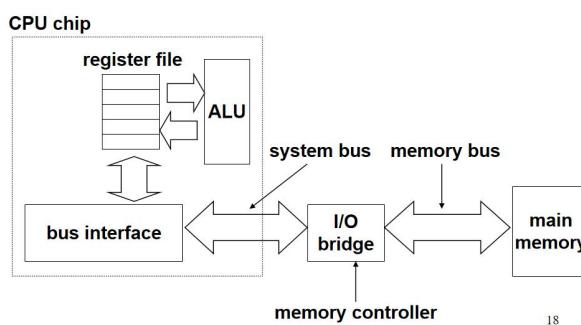
EPROM (可擦可编程 ROM)，使用紫外线照射来清除，使用特殊设备写，可以写一千次。

EEPROM (电子可擦除的 PROM)，可以用电子设备擦除和写，可以使用十万次。

Flash Memory, eg: U 盘、硬盘。

这些东西被叫做固件 (firmware)。启动的代码叫做 BIOS (basic IO system)，都要放在固件上，引导程序启动。

总线 (bus) 是一些并行的线，可以有数据、地址、读写控制信号，总线上面可以接很多设备。

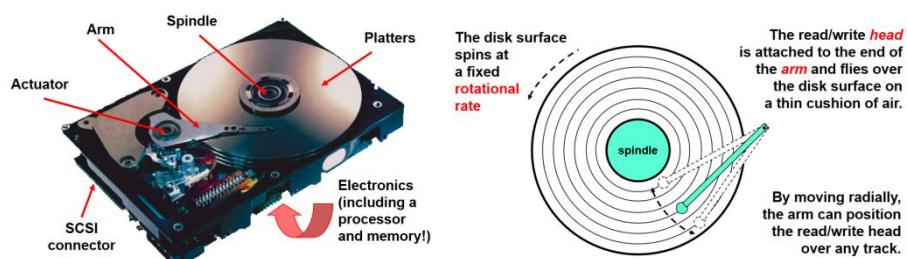


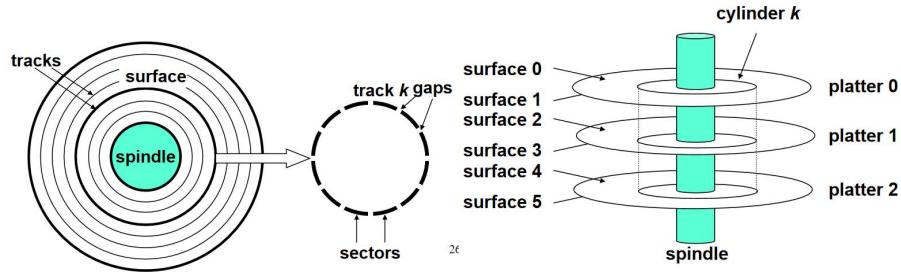
18

Memory 和 CPU 通过 bus 连在一起。如果要取值，经过一个虚地址翻译到实地址的过程后，通过 system bus 和 memory bus，从主存中获取数值。

硬盘

接下来，我们来介绍硬盘，基本上快过时了，只有在服务器端可能还在使用。





Platter 是一个盘，有正反两面。每个 surface 有一系列的同心圆，叫做 track。每个同心圆上有一系列不连续的 sector(扇区)。在每个面上，同直径的 track，构成了 cylinder。以下我们来计算一个硬盘的容量：

- Capacity

- maximum number of bits that can be stored
- Vendors express capacity in units of terabytes (TB), where $1 \text{ TB} = 10^{12}$.

- Capacity is determined by these technology factors:

- **Recording density** (bits/in): number of bits that can be squeezed into a 1 inch segment of a track.
- **Track density** (tracks/in): number of tracks that can be squeezed into a 1 inch radial segment.
- **Areal density** (bits/in²): product of recording and track density.

Example:

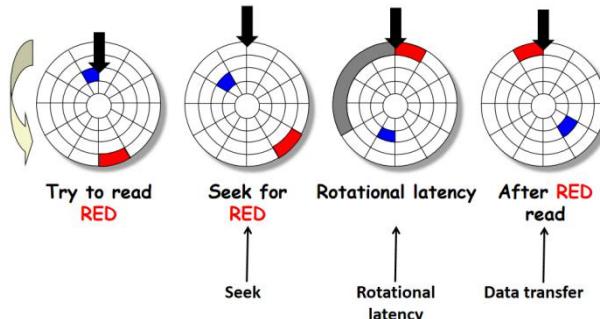
- 512 bytes/sector
- 300 sectors/track (on average)
- 20,000 tracks/surface
- 2 surfaces/platter
- 5 platters/disk

$$\begin{aligned}\text{Capacity} &= (\# \text{ bytes/sector}) \\ &\times (\text{avg. } \# \text{ sectors/track}) \\ &\times (\# \text{ tracks/surface}) \\ &\times (\# \text{ surfaces/platter}) \\ &\times (\# \text{ platters/disk})\end{aligned}$$

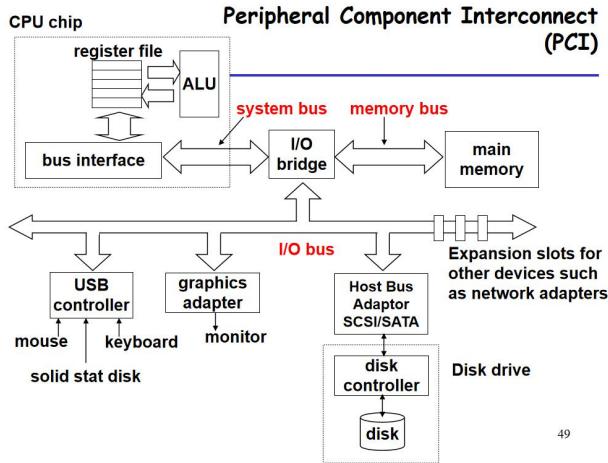
30

在早期的时候，不同直径的 track 中 sector 数量和每个 sector 的数据是一样的，只是 gap 不同。后来就变成一个 zone (几个 track 组成一个 zone) 里，sector 数量是一样的。Arm 在整个 cylinder 上是统一地在动。

硬盘读的时间由 seek、rotation 和数据传输时间组成。

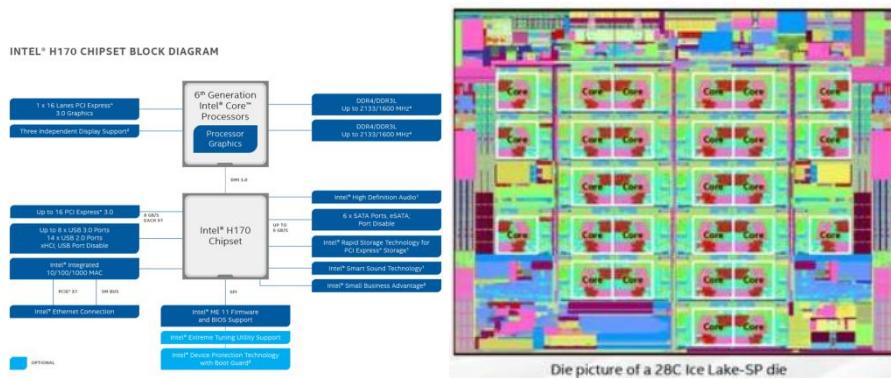


硬盘寻址是很麻烦的事情，要知道 surface、sector 和 track，才能找到对应的数据在哪里。实际在做的时候，把 sector 划为逻辑块，按照 0,1,2,3,…… 进行命名。把序号变为三元组，是硬盘自带的 disk controller 来做这件事。硬盘上一些空闲的空间用来存放翻译所需要的数据，这些空间用户就不能用了。



49

北桥上连 memory。南桥上连外部设备，可以插 USB、显卡、硬盘，还有一些扩展槽。



现代硬件架构已经没有北桥的概念了。Memory controller 实际在芯片里。

CPU 如何控制和访问外部设备

要访问硬盘，首先需要有一个概念 **IO port**。把 IO 外部设备里的寄存器变成了内存地址，这个操作就叫做 **IO port**。（在地址空间中保存了一些地址，把 **controller** 中的寄存器映射到这些地址上去（内存映射 IO），要访问这些寄存器就通过这些内存地址来访问。）

比如 CPU 要读磁盘上的 512 个 byte。需要给磁盘一个读命令（把逻辑块上的数据放到 memory 里去），以及磁盘上的哪一个逻辑块，以及读到内存中的哪一个地址。把这三个数据传到磁盘的 **IO port** 里。然后，**controller** 获取了命令需要的所有参数，CPU 就已经去干别的事情了。**controller** 负责把 **disk** 里一个 **sector** 中的数据复制到内存中。这个过程叫做 **DMA** (**direct memory access**)，这也是一个比较重要的概念。

复制完成后，**disk controller** 要通知 CPU 任务已经完成。把 **interrupt** 引脚置成 1，CPU 通过 **exception handler** 处理中断事件，需要找到发出者，再进行相应的处理。

现在硬盘不太用了，用得比较多的是固态硬盘 (SSD, solid state disk)，其使用了 **flash memory**。固态硬盘其实是可以一个 byte 一个 byte 访问的，但是我们还是当做硬盘设备来用。有的 SSD 可以插入 USB 口或者 SATA 口，或者 PCIE 口（更快）。因为还是当做一块设备，CPU 还是以逻辑块的形式进行访问。一个闪存中的块对应的是其中多个 **page**。要把 CPU 的逻辑块翻译成 **page**，需要 SSD 上一个类似于 **controller** 的东西 (**flash translation layer**)。要写 **page** 前，要先擦除 **block**。随机读写比起顺序读写要慢很多，因为涉及到块的擦除和数据的迁移。

SSD 优点：

1. 没有机械臂，没有移动的零件。省电、更加强壮（避免了硬盘高速运转时的空气动力学对读写头导致的影响）。

SSD 缺点：

1. SSD 的擦的次数是有限的
2. 是硬盘价格的四倍

2021/4/19

局部性

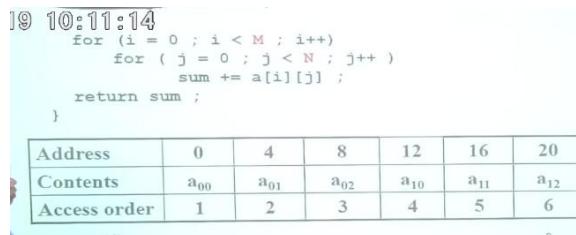
CPU 的计算速度越来越快，memory 的访问速度虽然也在变快，但是没有赶上 CPU 的增加速度。经常要从 memory 中取出数据，如果取的速度太慢的话，那么 CPU 就等于没有数据，相当于空转。指令也是一个问题，指令也是存在内存当中的。要弥补这个 CPU 和 memory 的差距呢，我们用到了局部性的原理。

这是一个典型的例子，向量加法。我们写一个循环，取一个元素，加到 sum 上，n 次就加完了。v 是一个长度为 4 的整数数组。数组在内存中 v0 和 v1 相邻，v1 和 v2 也是相邻的。在下一次循环的时候，总是要访问相邻的这个元素。Sum 是被重复地访问。这种情况在程序当中会经常碰到的。我们叫做 principle of locality，程序倾向访问的数据和指令，地址要么是相近的，要么是相同的。同一个地址的反复访问叫做时间的局部性。另外一个相邻地址的访问，叫做空间局部性。这个现象在程序当中比较普遍。

像刚才这个例子，数组相邻元素被访问就是空间局部性，求和的 sum 被反复访问就是时间局部性。如果我们把这个 C 语言翻译成汇编，我们可以看到循环中有若干条指令，是在内存中连续存放的。（空间局部性）因为是循环，这些指令会被反复地访问。（时间局部性）

在我们这个例子中，locality 是比较好的。访问数组的步长是 1，是最好的情况。如果这个程序只是加偶数位的，那么步长就是 2 了。我们为了达到同一个目的，对程序有不同的写法，功能都是一样的，但是最后的局部性可能差异比较大。

这是一个 2×3 的一个数组。



The screenshot shows a debugger interface with assembly code and memory dump. The assembly code is:

```
19 10:11:14
    for (i = 0 ; i < M ; i++)
        for ( j = 0 ; j < N ; j++ )
            sum += a[i][j] ;
    return sum ;
}
```

The memory dump table is:

Address	0	4	8	12	16	20
Contents	a ₀₀	a ₀₁	a ₀₂	a ₁₀	a ₁₁	a ₁₂
Access order	1	2	3	4	5	6

把两行的元素，先按照第 0 行求和，再按照第 1 行求和。因为数组是按照行来存放数据的，此时 步长为 1.

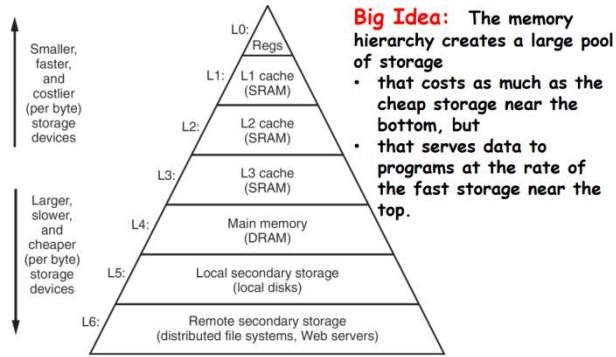
如果我们先固定一列，按照逐列求和来进行访问，也就是 a₀₀->a₁₀->a₀₁->a₁₁。这个局部性不太好，要隔三个。

有了这个局部性以后，设计计算机系统的时候，就要充分利用这个局部性原理来提高程序运行的速度。可以从各个层面去做，今天我们要讲的是纯靠硬件去做（用户无法干预这件事情），也就是 cache memory。第九章，虚拟存储就是用硬件和软件两个东西来应用局部性原理。现代，有很多应用程序也会做这件事情，比如访问网页，一开始从很远的地方去拿过

来，再去访问的可能就放到了交大服务器中心。有时候远端更新了，近端却没有及时更新，但是访问速度就快了，这是用软件来利用局部性原理。

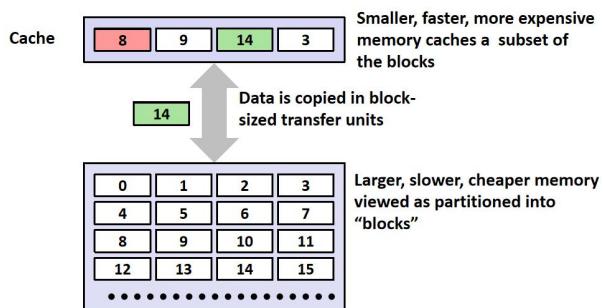
使用硬件利用局部性原理

这样实现的基础是有不同速度的存储介质，比如 sram（快但贵，做大以后就不快了），dram，ssd（可以做的比较大，但是比较慢）。因为有不同速度和容量的存储介质，最终我们想构建一个存储系统，里面的介质快慢都有。我们希望访问速度平均来讲接近最快的，存储的容量接近最大的，价格还不能太高。这就要求我们充分利用局部性原理。



我们各种存储技术和介质都放在一起以后，放在一起就构成了 **memory hierarchy**。寄存器是整个计算机中，访问速度最快的存储单元，访问速度和 CPU 一样。我们用 SRAM 做了三层，为什么呢，因为同样的技术做大了以后访问速度就会慢下来。再有就是用不同的存储介质来做的东西，他们本身差距就在，因为便宜本身又做的比较大，更慢了。整个情况就是往上越快越小，往下越大越慢。我们想达到的目标就是访问速度接近上面，容量接近下面。

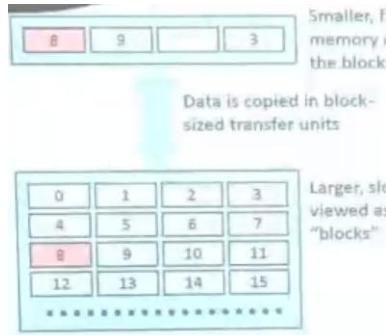
在实现的时候，每相邻的两层，我们就去实现一个 **cache** 技术。



上下两层都要分块，上下的块大小是一样的。只是下面的 **block** 多一点，上面的 **block** 少一点。上下会有以块为单位的数据的传递，数据一开始都存在下面，我们访问的时候会把下面的数据往上传。会不会有访问的数据一半在块 13，一半在块 14 中呢？是有可能的，但是我们之前做的数据对齐尽可能避免了这种情况。如果要访问块 14，并且 14 存在 cache 中，这就叫做 **hit**（访问命中），那么访问速度就比较快。如果要访问块 13,13 不存在于 cache 中，这就叫做 **miss**，只能往下找找到 13，把 13 拿上去。为什么要把 13 拿上去呢？因为有可能这个数据本身就要被反复地用或者被访问数据的相邻访问，要使用到之前的局部性原理。以后再访问就比较容易命中。13 要上去，要替换上面（已满）的数据，决定替换谁的过程叫做 **placement policy** 和 **replacement policy**。替代的策略取决于 cache 的实现方式，可能不一样。

Cache Miss

Cache miss 分为两类。1.cold miss 2.capacity miss (上面已经满了，下面的数据还要往上来)



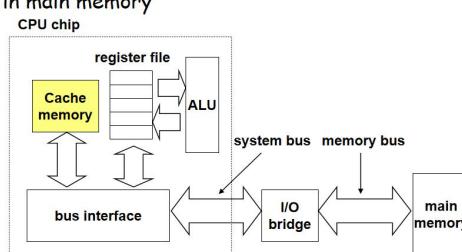
此时要访问 13，因为 cache 策略，虽然有空位，但是要放在 9 的位置替换掉 9.如果 9 和 13 交替访问的话，9 和 13 就会在两层之间交替地放，这就叫做 conflict miss。

Cache Type	What is Cached?	Where is it Cached?	Latency (cycles)	Managed By
Registers	4-8 bytes words	CPU core	0	Compiler
TLB	Address translations	On-Chip TLB	0	Hardware
L1 cache	64-bytes block	On-Chip L1	4	Hardware
L2 cache	64-bytes block	On-Chip L2	10	Hardware
L3 cache	64-bytes block	On-Chip L3	50	Hardware
Virtual Memory	4-KB page	Main memory	200	Hardware + OS
Buffer cache	Parts of files	Main memory	200	OS
Disk cache	Disk sectors	Disk controller	100,000	Disk firmware
Network buffer cache	Parts of files	Local disk	10,000,000	AFS/NFS client
Browser cache	Web pages	Local disk	10,000,000	Web browser
Web cache	Web pages	Remote server disks	1,000,000,000	Web proxy server

Cache Memory

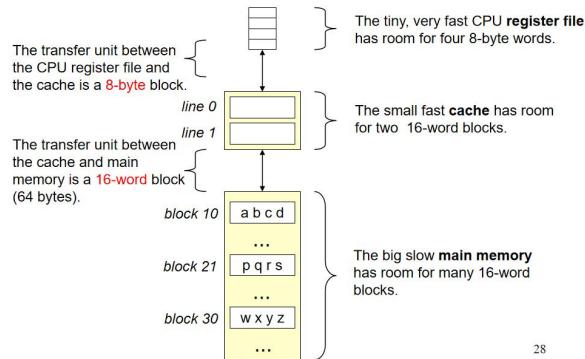
下面我们来讲 L1~L3，叫做 cache memory。

- Hold frequently accessed blocks of main memory in caches
- CPU looks first for data in L1, then in L2, then in L3, then in main memory



27

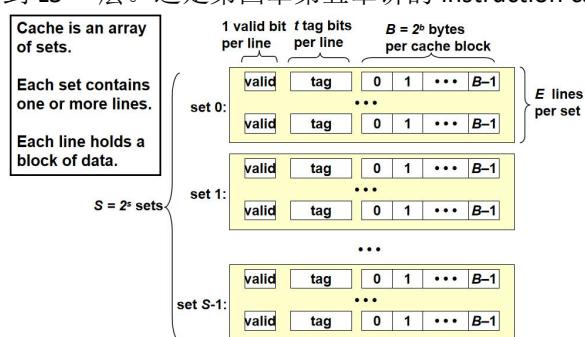
Cache 都是放在 CPU 里头的。



28

上头这一层是 CPU 的寄存器。寄存器和 L1 是相邻的两层，这个划块就是用寄存器的 8 个 byte 去划分。下面这个 L1 和 L2 的两层，块的大小就是 64 个 byte。

这个划块和 cache 基本上都是硬件做的。任意两层上面一层是下面一层的 cache。Cache memory 是特指的 L1 到 L3 一层。这是第四章第五章讲的 instruction cache 和 data cache。



Cache memory 组织起来是从 0 开始编号的若干个集合，每个集合一样大。Line 的格式都是一样的，每个集合中 line 的个数也是确定的。Valid bit 每个 line 都有一个。Tag 是有多少个 bit。最后就是 B 个 byte， B 一定是 2 的幂次。集合的数目也是 2 的幂次。

Derived quantities	
Parameters	Descriptions
$M=2^m$	Maximum number of unique memory address
$s=\log_2(S)$	Number of set index bits
$b=\log_2(B)$	Number of block offset bits
$t=m-(s+b)$	Number of tag bits
$C=B \times E \times S$	Cache size (bytes) not including overhead such as the valid and tag bits

小 m 就是地址的位数。

我们要访问数据，就要拿到地址，注意这里是实地址。拿地址访问内存之前，会先拿物理地址去 cache 中。先要确定在哪个集合，然后确定是哪个 line，然后确定是哪一个数据。这样我们就把地址分成了三部分，中间这部分是有 s 位（决定找哪个集合）， b 是决定找哪个 bit，剩下的就是 tag 位数。所有 tag 位数取决于有几个集合以及每个集合里存了几个 byte。根据 line 的个数，有不同的实现方式。

1. $e=1$ ，一个集合里只有一个 line，称作 direct-mapped cache（直接映射）

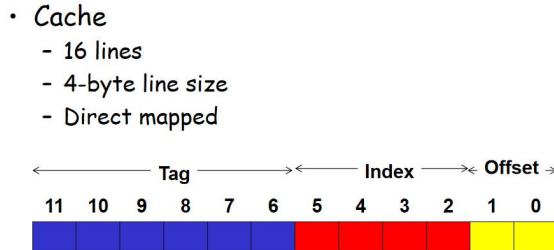
拿到地址访问 cache 分为以下部分，1.set selection，从中间的 s 位确定集合。2.line match，数据不在 cache 之中。换句话说 line 本身得有效。所以我们得到集合编号 i 以后，我们要调用 selected set(i)来看这个 line 是不是从内存中来的，还是冷启动中初始化为空的。一块里

的数据比我实际的要多，最后就是 word extraction，把我要的数据提取出来。

下面举个例子，我有一个 cache，这个 cache 有 16 个 line。每个 line 里头存了 4 个 byte。我们使用直接映射 cache。我们下面来算一下参数，因为是直接映射，有 16 个集合。

$$S = 2^s = 16, \quad B = 2^b = 4$$

$s = 4 \quad b = 2$



可以看到有些地方的 cache line 里面 value 是 0

顺序：拿 index 去找集合，拿 tag 比较。最后拿 offset 取数据。

Cache 的基本原理 <https://zhuanlan.zhihu.com/p/102293437>

2021/4/21

5月12号晚上期中考试，另外一门不考。考第四第五章以及 6.1，剩下的部分就是期末考。建议空的时候看一下历年考题。

上节课提到了两个 float 向量对应元素相乘，再加起来。也就是向量点积。

假设第一个数组从 0 开始，第二个数组从 32 开始。另外有一个 cache，是一个 line 有 16 个 byte，总共有 2 个 line，因为直接映射也就是 2 个集合。Offset 是 4 个 bit，集合 1 个 bit，其余的都是 tag。

前面 4 个元素进 cache 的时候，都是进到 cache0 的。从 x[4] 开始，10000,10100,11000,11100，这四个元素都是在集合 1 里头。对于 y，就变成了 100000,100100,101000,101100，我们看倒数第五位，我们发现在第 0 个集合。

我们要做 $x[0] * y[0]$ ，一开始 $x[0]$ 是 code miss，cache0 的 16 个 byte 就会放 x_0, x_1, x_2, x_3 。找 $y[0]$ 的时候，发现 tag 是 1，和 cache0 中的 tag 为 0 不一致。这时候把 y_0, y_1, y_2, y_3 放进 cache0 中。

我们找 $x[1]*y[1]$ 同理，永远会 miss，这就是 conflict miss。

解决这个问题的方法就是把 $y[0]$ 放在 48 的位置上，这样 $y[0] \sim y[3]$ 就会在 cache1 中。这样 cache0 中有 $x[0] \sim x[3]$ 。所以在 $x[1], y[1]$ 的时候都会命中。

我们把 $x[8]$ 变成 $x[12]$ ， $y[0]$ 自然就放到 48 了。

访问 cache 是拿物理地址分成三段去访问，最后一段叫做 offset，指 cache line 中第几个 byte，这个位置是不能动的，一移动到其他位置，数据的存放就不连续了。为什么先放 tag 再放集合？

Address bits				
Address (decimal)	Index bits (s=2)	Tag bits (t=1)	Offset bits (b=1)	Set number (decimal)
0	00	0	0	0
1	00	0	1	0
2	00	1	0	0
3	00	1	1	0
4	01	0	0	1
5	01	0	1	1
6	01	1	0	1
7	01	1	1	1
8	10	0	0	2
9	10	0	1	2
10	10	1	0	2
11	10	1	1	2
12	11	0	0	3
13	11	0	1	3
14	11	1	0	3
15	11	1	1	3

如果我们用中间的 bit 来排序。在这种情况下，前八个在不同的 cache line 中，后八个在不同的 cache line。一般的访问是连续访问，我们访问 01 的时候在一个 cache line 中，访问 23 的时候，进新的集合的 cache line 中。连续的时候，进了不同的 cache line，这个 locality 就比较好。

Address bits				
Address (decimal)	Index bits (s=2)	Tag bits (t=1)	Offset bits (b=1)	Set number (decimal)
0	00	0	0	0
1	00	0	1	0
2	00	1	0	0
3	00	1	1	0
4	01	0	0	1
5	01	0	1	1
6	01	1	0	1
7	01	1	1	1
8	10	0	0	2
9	10	0	1	2
10	10	1	0	2
11	10	1	1	2
12	11	0	0	3
13	11	0	1	3
14	11	1	0	3
15	11	1	1	3

高位的情况，变化比较慢，要等 4 个才变一个集合。访问 23 的时候，01 就会被淘汰掉，这会导致局部性的利用比较差一点。

Set-associated cache

在直接映射下，放置和替换基本上没有太大差别。往一个地方放，如果被人占了就替换掉。这种对 cache 的使用不是特别好。找数据的出发点都是从物理地址开始，用中间的 s 位找集合。第二步是 line match，现在有几个 line，我们同时作比较，必须先 valid bit 是 1。然后比较 tag，只要相同，就把对应的 line 拿下来。然后根据 offset 的多少，取出相应的数据。数据取多少是根据指令来定的，比如 movq, movl, movb。当然要强调一下 movq (A) %rax，A 是虚地址，而 cache 的地址是物理的实地址。

```

Example          t=2 s=1 b=1
M=16 byte addresses
B=2 bytes/block, S=2 sets, E=2 entry/set

Address trace (reads):
0-[0000] 1-[0001] 13-[1101] 8-[1000] 0-[0000]
miss       hit     miss     miss

(4) 8 [1000] (miss)? LRU
set v tag data
0 1 00 m[0]m[1]
0 1 11 m[12]m[13]
1 1
1 0

```

80

8 进来一定要换掉集合 0 中的一个 line，这就是 replacement policy。这叫做 Least recent use (最远访问)，cache0 中 1213 是刚刚访问的，01 的访问时间远一点。所以我们淘汰 01。

淘汰策略，多个 line 的情况淘汰谁？用的比较多的有两种。

- LFU (least-frequently-used)
 - Replace the line that has been referenced the fewest times over some past time window
- LRU (least-recently-used)
 - Replace the line that was last accessed the furthest in the past
- All of these policies require additional time and hardware

这需要花费额外的时间，因为每一次访问都要打上一个时间戳。

Fully-associated cache

最后一种方法就是 fully associated caches，不需要集合了。现在只有一个集合，只有 tag+offset。选 line 的方法就是，行 valid bit=1 的前提下，逐行比较 tag。但是 line 数量大的时候，比 tag 就比较慢。

我们还有写的问题，一旦写命中了，我们发现数据在 cache 中，如果只写 cache 这一块，memory 中对应的地方如果不写，就违反了 cache 是 memory 的子集。但是如果都写的话，保持了数据的一致性，但是效率就变慢了，这叫做 write through (写透)。

后来改进为 write back 策略，就改 cache 而不改 memory。对于单核来说问题不大，因为要访问数据一定会访问到 cache。当 cache line 被淘汰掉 (evict) 的时候，如果 cache 被歇过了，那么要写回到内存里。为了效率，我们要判断 cache 是否被写过了，我们加一位叫做 dirty bit。如果被写过了，那么 dirty bit = 1。

Write miss 的情况，

1. write-allocate 是拿上来，写在 cache 里。
2. no-write-allocate，直接去修改 memory。

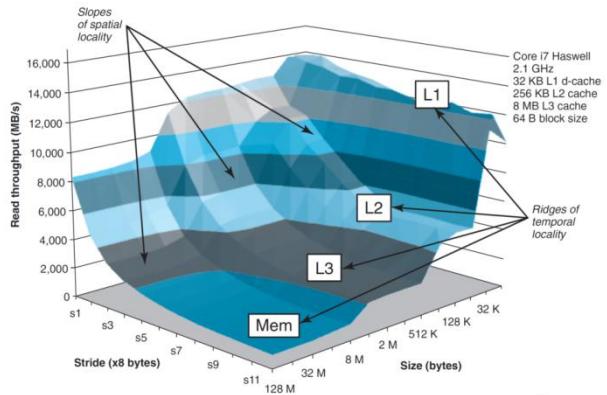
常见的组合：write through + no-write-allocate 或者 write back + write-allocate。

我们来看一下现在的多核 CPU 里，cache 是怎么组成的。Cache 分三层，第一第二层 (L1 和 L2) 是在每个核里头，叫做 private cache。L3 是大家共享的 share cache。在 L1 cache 中 data

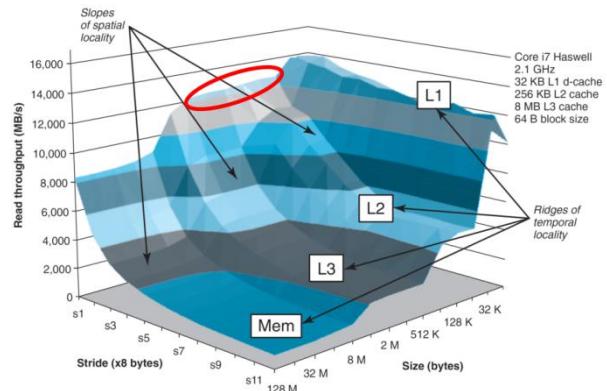
cache 和 instruction cache 是分开的。L2 和 L3 是放在一起的，这就叫做 unified cache。访问 L1 用 4 个 cycle，L2 用 10 个 cycle，L3 用 40~75 个 cycle。L1 的 miss 率是 3% 到 10%，L2 就小于 1% 了。Penalty，如果不在 cache 中，访问 memory 的话，有比较长的时间。

矩阵乘法，矩阵大到一行就会超过 cache 的大小。编译器会把循环内部的变量放到寄存器里，内存访问中只有 abc 这些数组。

2021/4/28



这是不同数据大小和步长下，系统的吞吐量。连续读（小补偿）优于大步长，因为空间局部性。数据越小，可以放入速度更快的 cache 中，吞吐量越高。



这里比较平是因为 prefetch。这张图怎么画出来的呢？

```
/* Run test (elems, stride) and return read throughput (MB/s) */
double run (int size, int stride, double Mhz)
{
    double cycles;
    int elems = size / sizeof(long);

    test (elems, stride); /* warm up the cache */
    cycles = fcyc2(test, elems, stride, 0); /* call test (elems,stride) */
    return (size / stride) / (cycles / Mhz); /* convert cycles to MB/s */
}

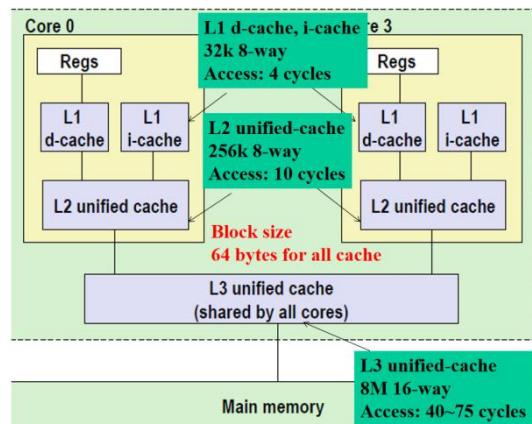
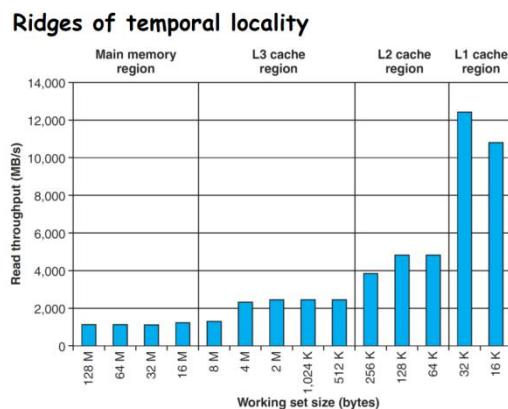
/* mountain.c - Generate the memory mountain. */
#define MINBYTES (1 << 14) /* Working set size ranges from 16 KB */
#define MAXBYTES (1 << 27) /* ... up to 128 MB */
#define MAXSTRIDE 12 /* Strides range from 1 to 12 */
#define MAXELEMS MAXBYTES/sizeof(long)
long data[MAXELEMS]; /* The array we'll be traversing */
```

```

for (i = 0; i < limit; i += sx4) {
    acc0 += data[i];
    acc1 += data[i+stride];
    acc2 += data[i+sx2];
    acc3 += data[i+sx3];
}
for (; i < length; i+=stride)
    acc0 += data[i]
return ((acc0+acc1)+(acc2+acc3));
}

```

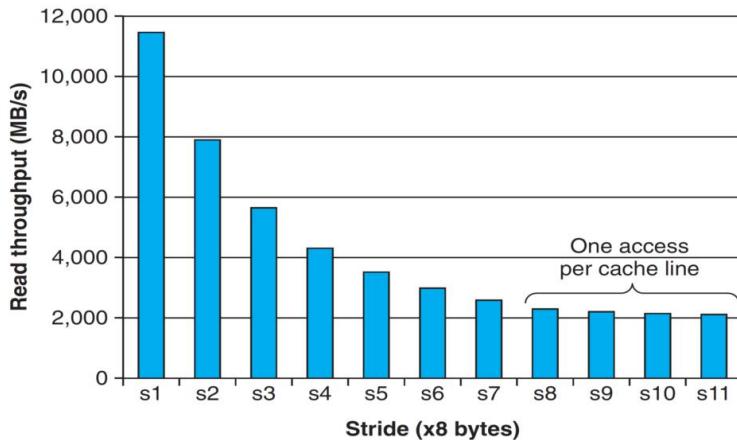
Run 返回访问的数据量处于我们的时间，就是每秒多少 M。fcyc2 函数就是一个估算，跑一个 test 函数，后面两个是 test 的参数，所以 fcyc2 是一个可变参的函数，估算出用了多少 cycle。Test 做了两次，目的是 warm up the cache，防止有很多 cold miss。在 test 中循环展开提高性能，尽量把其他的因素排除掉，只留下读取内存的耗时。



在步长等于 8 的时候，我们取一个剖面，等于一次访问一个 cache line。16K 的地方比 32K 比较慢，这有各种各样的因素，比如我们测试的一些东西也会对它造成干扰，至少量级差不多。

256K 的数据，要放到 L2 cache 应该是正好放进去，但是因为 L2 和 L3 都是 Unified cache，程序运行的时候，L2 和 L3 还要放指令，这样就会淘汰掉一部分数据。所以 256K 和 8M 的性能都相对比较差。

A slope of spatial locality

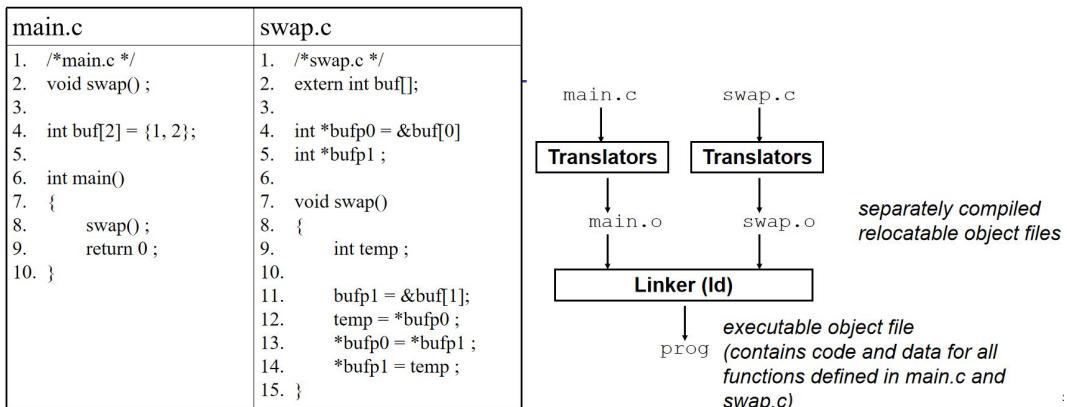


我们固定 4M 的数据，用不同的步长去访问这个数据，我们可以看到它们的吞吐明显不一样。到 8 以后变得一样了，说明了 cache line 里面一个数据的空间局部性已经没有了。在 stride1 的时候，因为有 prefetch，所以比较平。

第七章：链接

第七章是书上写的不是特别好的，解释的不够详细，大家还要配合着 PPT 来看一下。我们先讲什么是一个链接，传统上来说，我们写一个程序只有一个文件的 main.c，然后编译成可执行的。但是这会有两个问题：

1. 效率问题：比如程序里有一百万行代码，修改了一行就要重新编译几十分钟。
2. 模块化困难，难以共享函数。我们要控制复杂性，让程序的结构更加合理。



这个例子是让大家看一下，一个程序由两个代码编译出来的。main.c 里面有一个数组全局变量。main 函数里面还有一个 swap 函数，它要干什么 main.c 里是不清楚的。Swap.c 里面定义了 swap 函数，并且用到了 main.c 里面的全局变量，extern 指的是其他文件里头定义的变量，我这里不分配空间，但是需要使用它。我们最后要变成一个可执行函数，所以我们要分别编译成.o 文件，最后 linker 把所有 relocatable object file 合成一个可执行文件（可以被加载到文件中，并且执行）。

2、relocatable object file

编译的结果就是relocatable object file：hello_world.o文件。一个203个字节的hello_world.c源文件为何变成了1208个字节的hello_world.o文件？还给它起了个relocatable object file的名字？本质上，编译的结果是为了链接，也就是说，hello_world.o文件必须包含下一步链接需要的信息：

(1) 执行代码。源代码虽好，但是只是适合人类阅读，机器阅读还是不适合的。ARM处理器有自己的规则，因此在编译之后，c代码逻辑变成了机器码，可以被处理器解释执行。这些就是.text section

(2) 数据。程序的本质是逻辑（控制流）加上数据（数据流），逻辑由.text section提供，而数据部分稍微复杂一些，函数内部的临时变量位于stack上，不在此列，这里的数据是只全局数据，又分成已经初始化的数据 section和未初始化的.bss section以及只读数据.rodata section。

(3) 定义符号和引用符号的信息。hello_world.c这个模块定义了若干的符号，这使得让广大人民群众（其他模块）都知道，这样，linker在进行symbol resolution的时候才知道其他模块引用的符号是否是未定义的。同理，hello_world.c这个模块也要对外宣布，我需要引用哪些符号，你linker要帮忙解析一下，看看其否其他模块有定义该符号。因此，在relocatable object中存在符号表，即.symtab section。当然，symbol的name的字符串保存在了其他的section，即.strtab中。

(4) 由于还没有形成进程的映像，因此relocatable object file中的代码和数据地址都是从0开始的。例如hello_world这个符号（本质上是一个函数符号）就是位于0地址的，而hello_world_data也是位于0地址，如果不对这些符号进行relocation，那么程序是不可能执行起来的。因此，linker需要把若干个relocatable object file（当然还要有库文件）组织成一个可以被加载的image并分配正确的地址给各个符号。为了帮助linker做这件事情，.o文件必须提供relocation information，也就是.rel.text、.rel.data、.....

(5) 编译器是非常了解目标平台的，相反linker其实没有那么知道target的信息，因此，.o文件也会内嵌这些平台相关信息给linker，以便linker可以更好的工作。

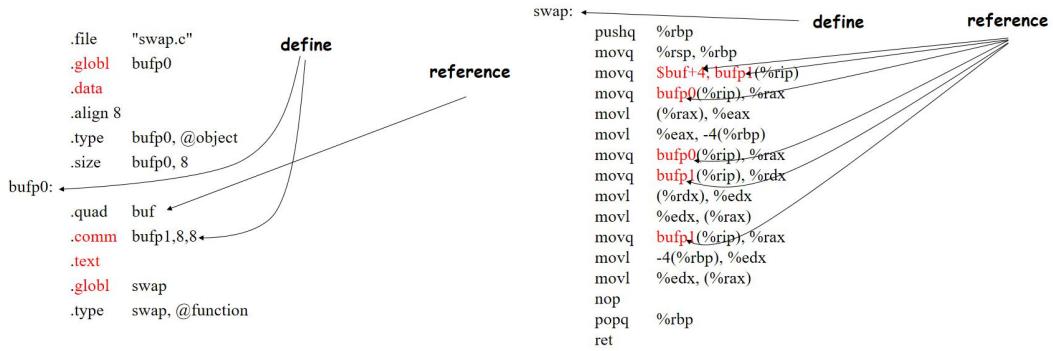
链接可以在编译的时候，在load的时候，在运行的时候进行。在load的时候只链接一部分，在加载的时候会把link全部做完。还可以在应用程序执行的时候再去链接，但是这个需要特殊的系统调用。

编译的过程

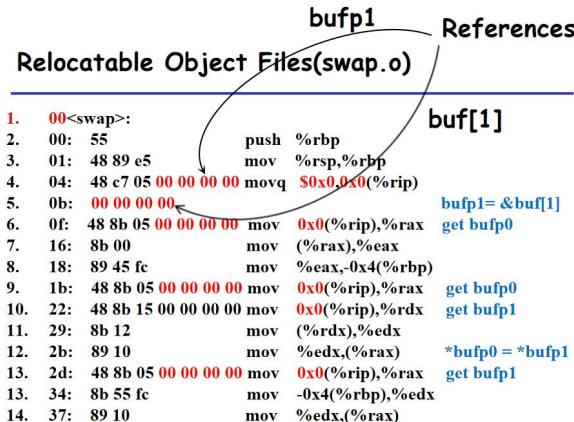
我们先复习一下编译的过程，叫做compiler driver。预处理（去掉//和#define等）->编译（高级语言变成汇编）->汇编器（把汇编程序变成二进制.o文件）->linker（合在一起变成可执行文件）。命令行如下：

```
unix> gcc -Og -o prog main.c swap.c
cpp [other args] main.c /tmp/main.i
cc1 /tmp/main.i main.c -Og [other args] -o /tmp/main.s
as [other args] -o /tmp/main.o /tmp/main.s
<similar process for swap.c>
ld -o prog [system obj files] /tmp/main.o /tmp/swap.o
unix>
```

第一步 main.c->main.s->main.o。其中 cc1 这一步是非常复杂的事情，下学期系统软件方向会去做这个编译器。



`bufp0` 是全局可见的，在另一个文件中也是能看到的。因为 `bufp0` 是一个指针，所以需要 8 对齐。因为 `bufp0` 的值就是 `buf`，所以有`.quad buf`, `.comm` 是没有赋初值的全局变量（8 对齐+长度为 8），连位置都没有给。在程序运行的时候，才会分配位置。`.text` 以后就是代码段，后面有一个`.swap`。具体的 `swap` 在右图中。右图中的汇编代码不太一样，我们`%rsp` 和`%rbp` 都作为帧的栈顶指针。我们关系的是 `buf`, `bufp0`, `bufp1` 这三个全局变量是以什么形式存在的。`bufp1(%rip)`，说明 `bufp1` 是一个相对地址，要加上 pc 的值才是真正的值。上面这个代码是编译器生成的。我们还要把这个汇编代码变成 relocatable object file。



这些中代码不知道的东西全为 0，信息全部丢失了。我们需要有额外的地方来保留这些信息。这就是我们汇编器要干的事情 1. 汇编指令变成二进制。在 x86 里头，很多符号看不到，我们不能像 y86 里面一样简单的扫描一遍就填入所有的地方了。既然 swap.o 里看不到 main.o 里的信息，那为什么 `bufp0` 和 `bufp1` 也填 0 了呢？Bufp0 和 bufp1 在 data 段，但是 data 段的地址需要 code 段和 data 段合并了以后才知道，所以 `bufp0` 和 `bufp1` 的地方也得填 0，因为地址我们现在也不知道。所以汇编器需要把这些额外的信息全部用一个数据结构记住，最后实施回填。

链接器的作用

链接器干什么呢？我们现在有了一堆.o，但是里面很多重要的信息都是 0，还没有回填。虽然 main.o 里面地址不知道，但是 buf 的初值我们知道。Swap.o 里，`bufp0` 是有初值的，`bufp1` 是没有初值的。

1 00 <buf>:	2 00:	01 00 00 00 02 00 00 00
------------------	------------	-------------------------

1 00<bufp0>:

2 00: 00 00 00 00 00 00 00 00

1 00<bufp1>:

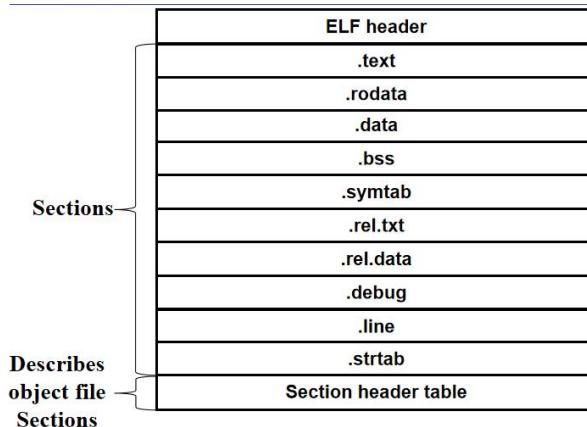
在链接完以后，先分配代码段，所以 main 函数连着就是 swap 函数，然后再分配数据段，把 buf 和 bufp0 和 bufp1 放在各自的位置。这样的话，我们就合并起来把每个符号分配一个地址。根据需要，填入绝对地址，或者是 pc relocate 的相对地址。

- Relocates symbols
 - from their relative locations in the .o files
 - to new absolute positions in the executable
 - main in the 0 pos of the .txt section in main.o to 4004d6 in prog
 - buf in the 0 pos of the .data section in main.o to 601030 in prog
 - swap in the 0 pos of the .txt section in swap.o to 4004eb in prog
 - bufp0 in the 0 pos of the .data section in swap.o to 601038 in prog
 - bufp1 in the 0 pos of the .bss section in swap.o to 601048 in prog

26

这就是我们链接的过程。

.o 分成了很多 section。00 代表在这个 section 里最开始的位置，包括了代码的 section，赋过初值的 section 和没赋过初值的 section。还有一些数据结构也是一些 section。所以整个 relocatable object file。Linking 的过程就是要把符号解析出一个地址来。我们要看到 relocatable object file（汇编出来的东西有些符号不知道），可执行文件（符号都知道了，可以直接进内存执行了，因为是虚地址，进了内存就在这个地址上），还有一种叫 shared object file



ELF 表

在 Linux 里面有一个统一的格式，叫做 ELF (executable and linkable format)，分成三个部分：header (放最重要的信息)，很多 section，section header table (根据头可以找到这张表，通过这张表可以找到中间的每一个 section)。

ELF Header

```
typedef struct{
    unsigned char          e_ident[16]; /* ELF identification */
    Elf64_Half            e_type;      /* Object file type */
    Elf64_Half            e_machine;   /* Machine type */
    Elf64_Word            e_version;   /* Object file version */
    Elf64_Addr            e_entry;     /* Entry point address */
    Elf64_Off              e_phoff;     /* Program header offset */
    Elf64_Off              e_shoff;     /* Section header offset */
    Elf64_Word            e_flags;     /* Processor-specific flags */
    Elf64_Half            e_ehsize;    /* ELF header size */
    Elf64_Half            e_phentsize; /* Size of program header entry */
    Elf64_Half            e_phnum;     /* Number of program header entries */
    Elf64_Half            e_shentsize; /* Size of section header entry */
    Elf64_Half            e_shnum;     /* Number of section header entries */
    Elf64_Half            e_shstrndx;  /* Section name string table index */
} Elf64_Ehdr;
```

32

```
/*
 * ELF definitions common to all 64-bit architectures.
 */

typedef uint64_t      Elf64_Addr;
typedef uint16_t       Elf64_Half;
typedef uint64_t       Elf64_Off;
typedef int32_t        Elf64_Sword;
typedef int64_t         Elf64_Xsword;
typedef uint32_t       Elf64_Word;
typedef uint64_t       Elf64_Lword;
typedef uint64_t       Elf64_Xword;
```

因为我们是文件来表示数据，第一部分一定是最重要的，这里面放了这样的一个数据结构。反正无非就是一些 32 位、64 位的一些数据结构。

e_type	ET_REL(1), ET_EXEC(2), ET_DYN(3)(shared)
e_machine	EM_386(3), EM_IA_64(50)
e_shoff	
e_shentsize	
e_shnum	
e_ident[0-3]	'0x7f' 'E' 'L' 'F' magic number
e_ident[4]	1(32-bit) / 2(64-bit)
e_ident[5]	1(little) / 2(big)
e_ident[15]	size of e_ident[]

E_ident[16]，书上叫做 **magic number**，是一个 16 个字符的字符串。第四个开始告诉你这个 ELF 是 32 位还是 64 位，是 little endian 还是 big endian。

E_type 就是告诉你是哪一类，relocatable, executable 还是 shared。

E_machine 也是一个 16 位整数，记录了是哪个机器，和我们相关的是 EM_386(3)

E_shnum 是一个表里有多少个 element，每个 element 有多大。

ELF Section Header Table

```
typedef struct {
    Elf64_Word  sh_name; /* Section name (index into the
                           section header string table). */
    Elf64_Word  sh_type; /* Section type. */
    Elf64_Xword sh_flags; /* Section flags. */
    Elf64_Addr  sh_addr; /* Address in memory image. */
    Elf64_Off   sh_offset; /* Offset in file. */
    Elf64_Xword sh_size; /* Size in bytes. */
    Elf64_Word  sh_link; /* Index of a related section. */
    Elf64_Word  sh_info; /* Depends on section type. */
    Elf64_Xword sh_addralign; /* Alignment in bytes. */
    Elf64_Xword sh_entsize; /* Size of each entry in section. */
} Elf64_Shdr;
```

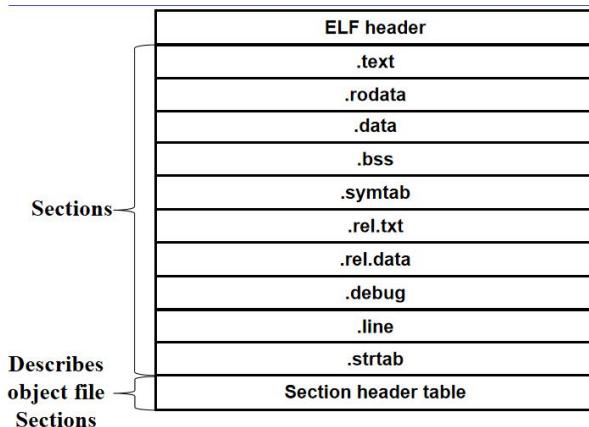
38

然后我们来看一下这个 Section header table。每个 section 一定是在 section header table 里有一项去描述它。在这个 table 中，描述了这个 section 的属性。

2021/5/8

linking 属于编译的环节，实验里面先从模拟器模拟执行，接下来做了一个汇编器。现在的高级指令需要有编译的过程。

ELF 是现在用的比较多的一种文件。.o,.so 里面放的都是二进制代码，目标都是一个可执行的。在升级之后，程序都不需要重新编译。因为程序是在运行时完成一些编译的事情，编译其实就是让两个程序互相能够找到对方的一些变量和函数。它的基础就是 ELF 文件，我们除了 code 和 data 还要放什么东西进去。我们今天还要介绍编译阶段的 bug。



ELF header 介绍了文件里有什么信息，section header table 描述了每个 section。反汇编就是把 ELF 文件通过 Objdump 反汇编成汇编文件。Section 都是独立、连续的。这些 section 都是没有 overlap 的，里面也会有一些废弃的部分。

ELF Section Header Table

```

typedef struct {
    Elf64_Word sh_name; /* Section name (index into the
                           section header string table). */
    Elf64_Word sh_type; /* Section type. */
    Elf64_Xword sh_flags; /* Section flags. */
    Elf64_Addr sh_addr; /* Address in memory image. */
    Elf64_Off sh_offset; /* Offset in file. */
    Elf64_Xword sh_size; /* Size in bytes. */
    Elf64_Word sh_link; /* Index of a related section. */
    Elf64_Word sh_info; /* Depends on section type. */
    Elf64_Xword sh_addralign; /* Alignment in bytes. */
    Elf64_Xword sh_entsize; /* Size of each entry in section. */
} Elf64_Shdr;
  
```

sh_name 比如.data, .text 等。比如要加载到内存的时候，就需要在文件中的位置 sh_offset，也制定了拷贝到内存中的位置 sh_addr，保证不管什么机器来执行，都保证了拷贝到内存中的同一个位置，还有一些内存的对齐信息。Sh_name 是一个比较难存的，因为 string 可长可短，怎么来解决字符串的格式呢？它采用了把所有字符串 group 一起（首尾相接放在一块），所以 sh_name 实际上是一个指针或者偏移量。这就解决了 string 变长这个问题。Sh_type 代表了不同 section 的类型，SYMTAB（符号表，带名字的我们叫做 symbol，比如函数的名字，变量的名字）。Sh_addr，不是所有 section 都要拷贝到内存的，有些只是在 debug 时候用的，比如汇编指令出错了怎么映射到高级指令的某一行代码。为什么在一开始的时候就能知道往哪里拷贝呢？因为每个程序都有完全相同的内存空间，最下面是 code, data, 最上面是 stack。根据约定 ELF 文件也会知道往哪里拷贝。

addralign 是对齐，涉及到 bss 优化，bss 段是未初始化的数据。为了节省空间，如果变量有值 ELF 就要放这个值，到时候直接拷贝的内存里。如果没有初始化，内存需要开辟出这块空间，但是 ELF 中不需要放这块东西。所以内存中有的，ELF 中不一定有（未初始化的全局变量）；ELF 中有的，内存中不一定有。这种就需要 sh_addralign 进行对齐。

sh_entsize，有的 section 是规规整整的，每个 symbol 有一个 entry，可以直接计算得出 section 的大小。

ELF 字符串表

在这个表里，字符串被安排为首位相接的，原来放字符串的地方就记录了一个 offset，

比如 1 (name.) ,7 (Variable) 等。

- The first byte and last byte
 - are defined to hold a null character
- e_shstrndx
 - In the ELF header
 - Index to the Section Header Table

Index	+0	+1	+2	+3	+4	+5	+6	+7	+8	+9
0	\0	n	a	m	e	.	\0	v	a	r
10	i	a	b	l	e	\0	a	b	l	e
20	\0	\0	x	x	\0					

43

.debug section 放了一些局部变量，因为局部变量等在 stack 上，理应在内存中才有局部变量的位置，那么怎么在 debug 模式下用到这些信息呢？可以直接认为把源代码塞到了这个 section 里面，调试的时候可以提供一些辅助。

.line 信息，汇编出错的时候，我们并不知道是哪个 C 的行出错了，这里保留了汇编和 C 的映射关系，在 debug 的时候会给出对应 C 的出错信息。

.text 就是我们的 code，函数、变量本身，我们关注的这些都在.text（编译成二进制，存放在.text）中，程序在运行的时候会把.text 段拷贝到内存中。

下面三个都是关于数据的，我们把数据分成了几类进行处理，分成了.rodata(READ ONLY data),.data,.bss（未初始化的变量）。内存访问的时候如果需要设置权限，我们不可能对每个 bit 设置一个权限，所以要求把 read only 的放在一块，有读写权限的放在一块，拷贝到内存后一整块设置一个权限。

.data 段中包含了初始化的全局变量和静态 C 变量。

.bss 段中包含了未初始化的全局、静态 C 变量，以及那些初始化为 0 的全局、静态 C 变量。它有段头（section header），但是不占据任何磁盘空间。在程序运行过程中，这些变量在内存中分配初始零值。

ELF 中的 Symbol Table

我们重点讲 ELF 中的 symbol table。中间这三个.symtab 和.rel.txt 和.rel.data 是干什么的呢？后面两个是在 relocate 阶段，填入别的函数的位置，换句话说，当一个函数的位置确定了 0x8000 以后，需要告诉很多那些调用这个函数的地方，那些地方都要放入 0x8000。那么.symtab 是什么关系呢？Sym 是要填的地址，代表了哪些别的地方需要用到的东西。比如函数 a, bcd 都调用了 a。在.symtab 中就记录了 a 的信息。我们先要考虑什么东西有资格放入.symtab，无非就是函数和变量，但是不是所有都有资格放进去。

```

1. void foo(void)
2.
3. int main()
4. {
5.     foo();
6.     return 0;
7. }
Unix> gcc -Wall -O2 -o linkerror linkerror.c
/tmp/ccSz5uti.o: In function `main':
/tmp/ccSz5uti.o (.text+0x7): undefined reference to `foo'
collect2: ld return 1 exit status

```

Main、foo 都是一个 symbol

Symbol 分成三类，

1. 定义的全局符号（已经实现了，别的模块可以使用）
2. 引用的全局符号（引用别的模块的函数）
3. 局部的符号（只在文件内可见）

引用符号：调用了别的文件的函数，这里编译的时候留下空间，需要知道这个函数的长相（参数、返回值）等。

Local symbol:一般情况下是没有资格进 symbol table 的，在单个模块编译的时候应该就已经确定了，local 的时候可能有一些 static 的 attribute，这个 symbol 只有这个文件本身会使用。Local symbol 是在单个模块生成编译的时候就会用到。

最后，那些 local nonstatic 程序变量是不会出现在符号表中的，在 ELF 文件中根本就不会存在这种。

<pre> 1. extern int a; 2. int f() 3. { 4. static int x=1; //x.1 5. int b = 2; 6. return x+b; 7. } 8. 9. int g() 10. { 11. static int x = 1; //x.2 12. return x + a; 13. } </pre>	Define Reference Local Not a symbol
--	--

两个函数 f 和 g 定义在这个文件中，可以被别的文件调用，称作 global define。a 就是跨模块调用的函数和变量。static 在函数中是可以同名的，但是互相是不干扰的，所以在 symbol table 的时候，会自动加上别名 x.1 和 x.2。

Local static 会通过是否初始化，放在.data 和.bss 中。

Symbol table 中，可以认为是一个 entry 的数组。每个 symbol 对应一个 entry。在编译的时候，要找到所有的 symbol，define 找出来告诉别人，reference 找出来，知道别的 ELF 的 symbol table 怎么填到我们这里来。建立 symbol table 的时候不需要别的文件，给/填的时候就需要互相使用。

```

void swap()
{
    extern int buf[];
    int *bufp0 = &buf[0];
    static int *bufp1;
    int temp;
    incr();
    bufp1 = &buf[1];
    temp = *bufp0;
    *bufp0 = *bufp1;
    *bufp1 = temp;
}

static void incr()
{
    static int count=0;
    count++;
}

```

.text
.globl swap
.type swap, @function

这个文件 swap.o 自己编译得到以下：

```

.file "swap.c"
.globl bufp0
.type bufp0, @object
.size bufp0, 8
bufp0:
.long buf
.local bufp1
.comm bufp1,8,8

```

.text
.type incr, @function
incr:
pushq %rbp
movq %rsp, %rbp
movl count.1231, %eax
addl \$1, %eax
movl %eax, count.1231
popq %rbp
ret
.size incr, .-incr
.local count.1231
.comm count.1231,4,4

我们可以看到 bufp0 是一个 @object，大小为 8

Bufp0:开始是.data 段，.long 就是需要留下 8 位来填充。.local 定义了 bufp1，.comm bufp1, 8, 8，这就是 local 和 global 不一样的地方。

我们还有一个 main.o

```

main.c
1. /*main.c */
2. void swap();
3.
4. int buf[2] = {1, 2};
5.
6. int main()
7. {
8.     swap();
9.     return 0;
10. }

```

现在我们把 main.o 和 swap.o 合在一起编译。

只有 temp 是 swap 中的局部非静态变量，不会出现在 symbol 里面。

buf 的类型是 `extern (reference)`，在 main.o 中定义，初始化过，所以放在.data 中。

Bufp0 在 swap.o 中定义，初始化过，放在.data 中。

所有 static 的变量都是 local symbol，所以 bufp1 是 swap.o 中的 local symbol，在.bss 中。

Swap 是一个函数，且没加 static，所以是 global，在代码段.text 中。

Incr 是一个 static 函数，所以是 local 的，在代码段.text 中。

count 是静态的，所以是 local 变量，因为初始化为 0，所以其在.bss 中。

Symbol	symtab entry?	Symbol type	Module where defined	Section
buf		extern	main.o	.data
bufp0		global	swap.o	.data
bufp1		local	swap.o	.bss
swap		global	swap.o	.txt
temp	x			
incr		local	swap.o	.txt
count		local	swap.o	.bss

每一个 symbol 都有一个 entry 对应来包装：

```
typedef struct {
    int    name ;          /* string table offset */
    char   type:4;         /* function or data (4 bits) */
    binding:4;             /* local or global (4 bits) */
    char   reserved ;      /* unused */
    short  section ;       /* section header index */
    long   value ;         /* section offset, or abs address */
    long   size ;          /* Object size in bytes */
} Elf64_Symbol ;
```

对于 relocatable module 来说，里面所谓的 symbol 的地址还没有填。

Type 可以

Section 说明这个对象在哪个 section 里面，加了三个额外的 ABS (不应该重分配的符号，这个变量是临时的，或者是假的)， UNDEF (没有定义在这个文件之中，当多个文件一起定义的时候，这个儿就会被消掉)， COMMON (未初始化的数据对象)。COMMON 和.bss 很像，在 symbol 重名的时候会有略微的区别。

- For COMMON symbols
 - value gives the alignment requirement
 - size gives the minimum size
- The difference between COMMON and .bss
 - COMMON
 - Uninitialized global variables
 - .bss
 - Uninitialized static variables,
 - global or static variables that are initialized to zero

大家只要记住，一个未初始化的对象可能是 COMMON 或者.bss。

我们回到上面的例子，查看 main.o 和 swap.o 中的 entry。

GNU readelf tool

main.o

Num:	Value	Size	Type	Bind	Vis	Ndx	Name
8:	0000000000000000	8	OBJECT	GLOBAL	DEFAULT	3	buf
9:	0000000000000000	21	FUNC	GLOBAL	DEFAULT	1	main
10:	0000000000000000	0	NOTYPE	GLOBAL	DEFAULT	UND	swap

Vis: visibility

may be default, protected, hidden or internal

it is defined in the reserved part

swap.o

Num:	Value	Size	Type	Bind	Vis	Ndx	Name
8:	0000000000000000	8	OBJECT	GLOBAL	DEFAULT	3	bufp0
9:	0000000000000000	0	NOTYPE	GLOBAL	DEFAULT	UND	buf
10:	0000000000000008	8	OBJECT	GLOBAL	DEFAULT	COMM	bufp1
11:	0000000000000000	60	FUNC	GLOBAL	DEFAULT	1	swap

Ndx: 3 代表.data, 1 代表.text。

这个 swap 定义了 UND(undefined), 不在这个文件中的任何一个 section 李曼, 是引用的另一个文件中的 section。

这个 buf 中是在 main.o 中, 所以在 swap.o 中的 buf 是 UND。Bufp1 是未初始化的全局变量, 本来是放在.bss 中, 但是我们为了区分 local 和 global, 把它放在临时的 COMM 中, 合并在一起处理完了以后会放在.bss 中。

这里红色的, 对于 COMM, 到编译结束都没有地址, 所以这里地址复用用来放 alignment,

即: .comm bufp1,8,8 。所以标红的这个 8, 其实是一个 alignment。

强名和弱名

现在两个文件编译完了以后, 都有各自的 symbol table。当有多个文件的时候, 我们要用到各自的 symbol table, 来帮助互相找到对方。会出现定义冲突的情况, 比如两个文件都出现了相同的 symbol, 该怎么去处理。

symbol 分成强名和弱名。

强名: 已初始化的全局变量、函数, 只允许有一个, 不允许冲突。两个强名会报错。

弱名: 没有初始化的全局变量, 没有 ELF 空间, 冲突小一点。

规则:

- Multiple strong symbols are not allowed
- Given a strong symbol and multiple weak symbols, choose the strong symbol
- Given multiple weak symbols, choose any of the weak symbol

所以弱名全部提取在了 COMM 中, 当多个文件合并完解决了 symbol 冲突以后, COMM 就没有用了, 全部变为了.bss。

强名和强名冲突:

```
1. /*foo1.c*/
2. int main()
3. {
4.     return 0;
5. }

1. /*foo2.c*/
2. int x=15213;
3.
4. int main()
5. {
6.     return 0;
7. }

1. /*bar1.c*/
2. int main()
3. {
4.     return 0;
5. }

1. /*bar2.c*/
2. int x=15213;
3.
4. void f()
5. {
6. }
```

强名和弱名冲突:

```
1. /*foo3.c*/
2. #include <stdio.h>
3. void f();
4.
5. int x=15213;
6.
7. int main()
8. {
9.     f();
10.    printf("x=%d\n",x)
11.    return 0;
12. }
```

```
1. /*bar3.c*/
2. int x;
3.
4. void f()
5. {
6.     x = 15212 ;
7. }
```

编译通过，但会 warning，任何引用全部指向强名。

弱名和弱名冲突:

```
1. /*foo4.c*/
2. #include <stdio.h>
3. void f();
4.
5. int x;
6.
7. int main()
8. {
9.     x=15213
10.    f();
11.    printf("x=%d\n",x)
12.    return 0;
13. }
```

```
1. /*bar4.c*/
2. int x ;
3.
4. void f()
5. {
6.     x = 15212 ;
7. }
```

到底填的是谁到这个时候还不知道，根据编译器自己的行为决定。

弱名和弱名冲突:

```
1. /*foo5.c*/
2. #include <stdio.h>
3. void f();
4.
5. int x=15213;
6. int y=15212;
7.
8. int main()
9. {
10.    f();
11.    printf("x=%d y=%d\n",
12.           x, y);
13.    return 0;
14. }
```

```
1. /*bar5.c*/
2. double x ;
3.
4. void f()
5. {
6.     x = -0.0 ;
7. }
```

gcc -fno-common

symbol 没有 type 信息，但是有大小信息。Int x 和 y 占了连续的 8 个 byte。通过了编译，但是 f 中对 x 写了 -0.0 以后，foo5.c 中的 x 和 y 一起被擦掉了。

我们来看一下编译这个过程，每一个文件先编译成自己的，然后 group 在一块生成自己的程序，如果用了很多别人的文件，这个时候怎么进行编译。

Naive 方法 1：一个一个输出.o，但是要经常修改而且很麻烦。

方法 2：把一个函数都放在单个源文件中，对于编译来说最简单，但是把大量的无关代码放到了编译的文件中，写了 helloworld 都要编译很久。

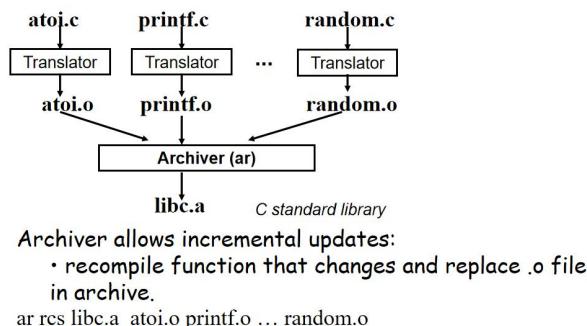
方法 3：每个函数单独写在一个文件中，编起来很麻烦。

所以现在采取了这种方法，把一些相关的函数放在一个文件中。不要的可以再踢掉。

2021/5/12

静态链接

上节课我们讲到 printf 这种库函数怎么处理。要么我们把 printf.o, scanf.o 一个个链接进来（过于繁琐），或者是把所有东西都放在一起 libc.o（过于臃肿）。所以我们就用一个库，把一些函数放在一起.o。所以我们把东西都放在库文件中 (.a archive 文件)，即 libm.a, libc.a 等。



Archive 实际上就是一些.o 文件的集合

Eg:

```
(a) addvec.o
int addcnt = 0;

void addvec(int *x, int *y, int *z, int n)
{
    int i;

    addcnt++;

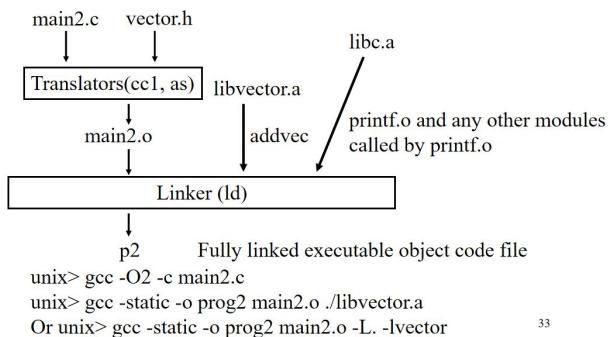
    for (i = 0; i < n; i++)
        z[i] = x[i] + y[i];
}
```

(b) multvec.o

```
int multcnt = 0 ;  
void multvec(int *x, int *y, int *z, int n)  
{  
    int i;  
    multcnt++;  
    for (i = 0; i < n; i++)  
        z[i] = x[i] * y[i];  
}  
unix> gcc -c addvec.c multvec.c  
unix> ar rcs libvector.a addvec.o multvec.o  
/* main2.c */  
#include <stdio.h>  
#include "vector.h"
```

```
int x[2] = {1, 2};  
int y[2] = {3, 4};  
int z[2];
```

```
int main()  
{  
    addvec(x, y, z, 2);  
    printf("z = [%d %d]\n", z[0], z[1]);  
    return 0;  
}
```



33

静态链接一定要是显示的，**libvector**因为是我们构造的，所以也要指明。会从 **libvector.a** 去找 **addvector**，从 **libc.a** 中去找 **printf**。

-L 是库文件的路径。

- E:
 - relocatable object files that will be merged to form the executable
- U:
 - Unresolved symbols
- D:
 - Symbols that have been defined in previous input files
- Initially all are empty

E 就是一些.o 文件的集合，比如上面这个例子中就是 main.o，我们会有两个集合非解决的符号和已经定义过的符号。一开始这三个集合都是空的。

Or unix> gcc -static -o prog2 main2.o -L. -lvector

在一个 command line 中，我们从左向右按照顺序来查找，可能碰到两种情况。

1. 碰到了.o 文件，我们就把.o 这个文件加到 E 中，因为新加的.o 会 define 一些符号，也会引入一些未定义的 symbol（比如 printf 和 addvector），所以我们把 main 加入到 D，printf 和 addvector 加入到 U 中。接下来，我们读到了.a 库文件，但是要加进去的都应该是.o 文件。在 libvector.a 中，包含 addvec.o 和 mulvec.o。而此时在 U 中是 printf 和 addvec，我们需要找 U 里的 symbol 有没有在库文件中被定义，比如 addvec 在这里是找到了。接下来，我们把 addvec.o 加到 E 中，然后去修改 U 和 D。此时 U 里头就只剩一个 printf 了，而 D 里头就有 main 和 addvec。然后我们继续找到了 printf.o 所在的库文件，我们把 printf.o 加入 E 中。这个时候就链接完了。

如果扫描完后，U 中不为空，就会报错。因为我们是按顺序扫描，顺序很重要。如果是 Gcc-static -o progs -L. -lvector main2.o，先会找 lvector，这样最后不会把 addvec 放到 E 中。

.rel.text 讲的是代码段中引用了一个 symbol，用到一次就有一个数组项。所以我们要用两个数据结构来记录，具体记录的是：

1. symbol 被引用的位置

```
5 09: e8 00 00 00 00    callq e <main+0xe>
A swap R_X86_64_PC32 -4
```

Symbol 被引用的位置（此处是在 main.o 地址为 09+1 = A 处）、符号名字（swap）、绝对地址还是相对地址、修正的常数 (-4，也就是下一行 PC 到 A 这个地址差的 4 个 byte) 因为 swap 应该填入一个 PC relative 的地址。

所以每一个数据项中，都该有这四项。

.rel.data:

In swap.o

```
1 00<bufp0>:
2 00:          00 00 00 00 00 00 00 00
               0 buf  R_X86_64_64 0
1 00<bufp1>:
```

所以在 elf 中的.rel.text 和.rel.data，这里面就是来看 text 段和 data 段里有哪些符号的引用。

• Relocation Entry

```
typedef struct {  
    long offset;  
    long type:32,  
           symbol:32;  
    long addend;  
} Elf64_Rela;
```

每一项都要有上述的四个东西，offset（section 中的相对位置）、type（什么引用方式，是 PC-relative 还是绝对地址）、symbol 符号

编译器里经常使用 bit-field，它是不能取地址的。最后 addend 就是一个相对的修正常数。注意 type 和 symbol 的都是一个 32 位的一个整数，拿到了这个整数后，我们可以去查这个 symbol table，从而知道符号是谁。

```
9  e8 00 00 00 00  callq  e<main+0xe>  swap0;  
There is a relocation entry in rel.text  
offset      symbol      type      addend  
a          swap        R_X86_64_PC32 -4
```

Why PC32?

gcc default mode is the X86_64 small code model
which assumes that the total size of the code and data in
the executable object file is smaller than 2GB and thus
can be accessed at run-time using 32-bit PC-relative
addresses

为什么是 PC32?gcc 中缺省的模式是 small code model，数据段和代码段都不会超过 2GB，所以偏移在 32 位之内就可以表示了。

```
int *bufp0 = &buf[0] ;  
00000000 <bufp0>:  
0: 00 00 00 00 00 00 00 00
```

There is a relocation entry in rel.data

offset	symbol	type	addend
0	buf	R_X86_64_64	0

我们把几个.o 合起来的时候，是对应的 section 要合并。各自的.txt 合到一起，各自的.data 合到一起。

有了这两个东西之后，我们就可以对每个函数分配地址，

Eg:

```

9   e8 00 00 00 00  callq  e<main+0xe> swap();
    a: R_X86_64_PC32 swap relocation entry
r.offset = 0xa
r.symbol = swap
r.type = R_X86_64_PC32
r.addend = -4
ADDR(main)=ADDR(.text) = 0x4004d6
ADDR(swap)=0x4004eb
refaddr = ADDR(main)+r.offset = 0x4004e0
ADDR(r.symbol)=ADDR(swap)=0x4004eb
*refptr = (unsigned) (ADDR(r.symbol) + r.addend - refaddr)
= (unsigned) (0x4004eb + (-4) - 0x4004e0)
= (unsigned) 0x7

```

12

PC-relative

知道了地址以后，我们要回填第一行的这种 00 00 00 00 占位的情况。

最终我们计算出一个 7，应该把 07 00 00 00 填进去。这里就讲了 PC-relocate 怎么填。

```

00000000 <bufp0>:
 0: 00 00 00 00 00 00 00 00 int *bufp0 = &buf[0];
 0: R_X86_64 buf relocation entry
r.offset = 0x0
r.symbol = buf
r.type = R_X86_64_64
r.addend = 0
ADDR(r.symbol) = ADDR(buf) = 0x601030
*refptr = (unsigned) (ADDR(r.symbol)+ r.addend)
= (unsigned) (0x601030)
0x601038 <bufp0>:
 0x601038: 30 10 60 00 00 00 00 00

```

这里就直接填入绝对地址就好了，因为 bufp0 的地址合并完我们是知道的。

Operating Systems: Four Easy Pieces

```

foreach section s {
    foreach relocation entry r {
        refptr = s + r.offset ; /* ptr to reference to be relocated */

        /* relocate a PC-relative reference */
        if (r.type == R_X86_64_PC32) {
            refaddr = ADDR(s) + r.offset ; /* ref's runtime address */
            *refptr=(unsigned) (ADDR(r.symbol)+r.addend-refaddr) ;
        }

        /* relocate an absolute reference */
        if ( r.type == R_X86_64_64 || r.type == R_X86_64_32||... )
            *refptr = (unsigned) (ADDR(r.symbol) + r.addend) ;
    }
}

```

14

对于每一个 rel 中的符号，我们根据 type 的不同，采用了两种不同的 relocate 地址。

1. 00<swap>:
2. 00: 55 push %rbp
3. 01: 48 89 e5 mov %rsp,%rbp
4. 04: 48 c7 05 00 00 00 00 movq \$0x0,0(%rip) get bufp1
7: R_X86_64_PC32 bufp1 - 8
5. 0b: 00 00 00 00 get &buf[1]
b: R_X86_64_32S buf + 4
6. 0f: 48 8b 05 00 00 00 00 mov 0x0(%rip),%rax get bufp0
12: R_X86_64_PC32 bufp0 - 4
7. 16: 8b 00 mov (%rax),%eax
8. 18: 89 45 fc mov %eax,-0x4(%rbp)
9. 1b: 48 8b 05 00 00 00 00 mov 0x0(%rip),%rax get bufp0
1e: R_X86_64_PC32 bufp0 - 4

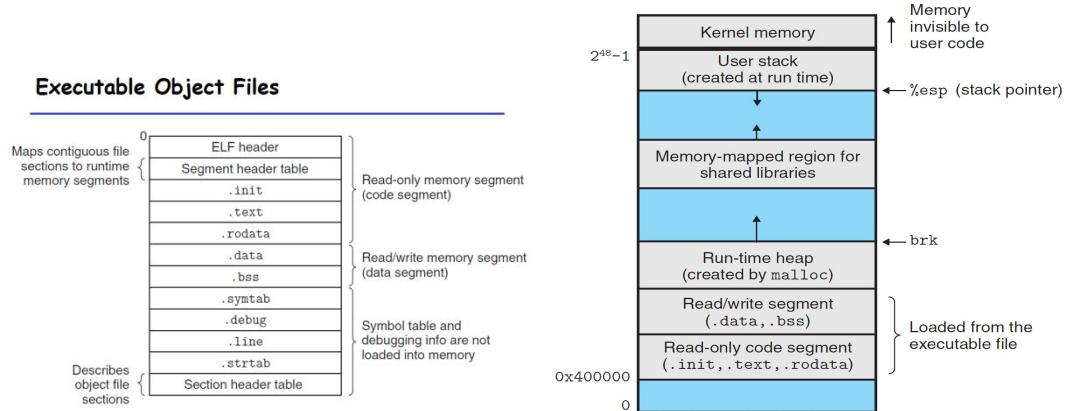
注意为了填入 b，我们的 addend 填入了 buf + 4，故有时候这个值很挺大的。

有一共有六个引用，就会有六个 entry。

```
.rel.text
7: R_X86_64_PC32 bufp1 - 8
b: R_X86_64_32S buf + 4
12: R_X86_64_PC32 bufp0 - 4
1e: R_X86_64_PC32 bufp0 - 4
25: R_X86_64_PC32 bufp1 - 4
30: R_X86_64_PC32 bufp1 - 4
```

ADDR(swapp)=0x4004eb ADDR(buf) = 0x601030
ADDR(bufp0)=0x601038 ADDR(bufp1)=0x601048

Executable Object File



有多个 segment, 所以还有了一共 segment header table。里面有一部分的 segment(.data .bss) 直接 copy 进内存的 Read/write segement 中。

在 ELFheader 中,

```
Elf64_Half      e_phentsize; /* Size of program header entry */
Elf64_Half      e_phnum;    /* Number of program header entries */
```

描述了 entry 的数量以及每个 entry 有多大。

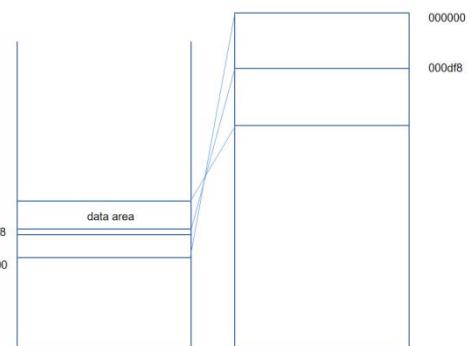
为什么我们拷贝到了 0x400000, 但是没有一条 main 函数是从 0x400000 开始运行的, 因为为了管理的方便, 我们还拷贝了 ELF header 和 Segment header table 和.init 进内存。.init 是 gcc 为每个程序添加的初始化程序。

ELF format

Read only code segment			
LOAD off	0x0000000000000000	vaddr	0x000000000000400000
paddr	0x000000000000400000	align	2**21
filesz	0x0000000000000069c	memsz	0x0000000000000069c
flags	r-x		

Read/write data segment			
LOAD off	0x0000000000000df8	vaddr	0x00000000000600df8
paddr	0x0000000000000df8	align	2**21
filesz	0x0000000000000228	memsz	0x00000000000000230
flags	rw		

- Difference between filesz and memsz means the uninitialized data in .bss
- .init section contains a small function _init called by program's initialization code



进了内存以后是 228->230, 因为有.bss。ELF 中还多了一个.init(整个程序初始化的代码),

在这一段代码之中，有一个点是程序的 entry point，会调用 `_start`。

在 shell 中，我们要执行一个可执行文件，就用 `./<executable file>`，也就是 `fork` 出一个子进程，然后用 `execve` 进行执行，大概就是把 ELF 文件中对应的 segment 拷贝进内存中，当然之后我们会讲到有优化的过程，不是一股脑直接拷贝进内存的。

动态链接

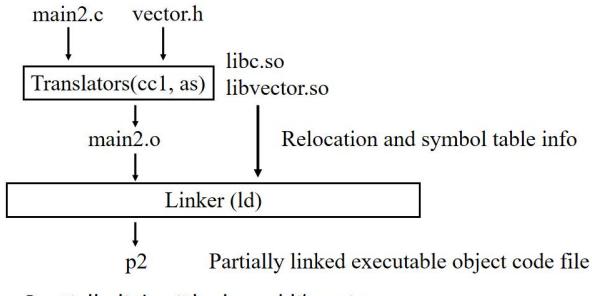
就比如说库函数，`libc` 很大，我们发现一个库函数出现了漏洞，写错了，那么我们要更新库文件。如果我们的程序已经静态编译好了（使用了有漏洞的库函数），那么这个漏洞该怎么修复呢？再编译一次就会很麻烦。同时，静态库可能被许多 C 程序在内存中复制了很多份。为了解决这些问题，就引入了共享库（.so，**shared libraries**；在 windows 中就叫 dll，动态链接库）。一个特定的库只有一个.so，是不会被 copy 的。

• Generate the shared libraries

Unix> `gcc -shared -fPIC -o libvector.so addvec.c multvec.c`

-shared: creating a shared object

-fPIC: creating the position independent code

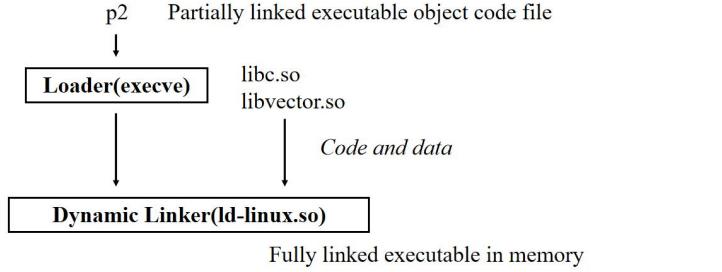


Partially link with shared libraries

Unix> `gcc -o p2 main2.c ./libvector.so`

6

动态链接是怎么链的？首先我们是在链接的时候，使用了 `libc.so` 和 `libvector.so`。得到的是 partial linked executable object code file，`printf` 和 `addvector` 的代码并没有 copy 进来，我们只是把两个.so 的 relocation 和 symbol table 信息放进来了。

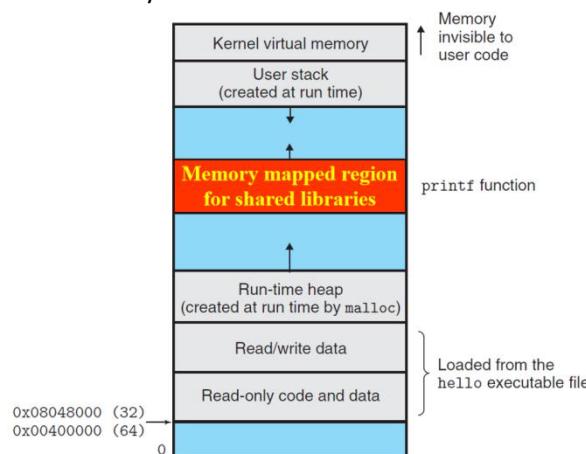


在 `execve` 的时候，才会调用动态链接器。多个进程同时使用的时候，实际上是共享了同一块物理内存。

2021/5/17

上节课讲的动态链接，举的例子就是我们自己写了两个函数，一个叫 `addvec`，另外一个 `mulvec`。我们在一个程序中，`main2.c` 调用向量加和 `printf`，我们在编译这个 `main2.c` 的时候，要把头文件拿进来，有一个 `relocatable object file`。在链接的时候，原来的静态链接是把一些 `.o` 放在一起，而动态链接，`libc` 里头的 `printf` 是共享的 `object file`，在链接的时候，我们并没有把两个相关的库里的函数拿进来变成 `executable`。动态链接的时候，只是修改了相关的 `symbol table`，对应的函数并没有放进去。

所以最后链接的时候是一个部分链接，真正要执行的函数在 `p2` 里是没有的。真正在加载（`execve`）的时候，在执行之前，要做一些额外的工作，调用动态链接器（也不是只调用一次每碰到一个会调用一次），这时候这两个代码就进到内存中，即不在代码段也不在数据段，而是另外有一块 `shared library` 所在的地方。



我们上面说的是在加载的时候做链接，也有在 `runtime` 的时候做链接，在 `loader` 的时候做链接和平常差不多 `gcc -fPIC -o p2 main2.c ./libvector.so`

`gcc -rdynamic -O2 -o p3 dll.c -ldl` 是在加载的时候做链接，有几个对应的函数。

```
1. #include <stdio.h>
2. #include <dlfcn.h>
3.
4. int x[2] = { 1, 2 } ;
5. int y[2] = { 3, 4 } ;
6. int z[2];
7.
8. int main()
9. {
10.     void *handle;
11.     void (*addvec)(int *, int *, int *, int );
12.     char *error ;
13.
14.     /*dynamically load the shared library that contains addvec() */
15.     handle = dlopen("./libvector.so", RTLD_LAZY);
16.     if (!handle) {
17.         fprintf(stderr, "%s\n", dlerror());
18.         exit(1);
19.     }
20.
21.     /*get a pointer to the addvec() function we just loaded */
22.     addvec = dlsym(handle, "addvec");
23.     if ((error = dlerror()) != NULL) {
24.         fprintf(stderr, "%s\n", error);
25.         exit(1);
26.     }
27. }
```

```

28. /* Now we can call addvec() just like any other function */
29. addvec(x, y, z, 2)
30. printf("z=[%d, %d]\n", z[0], z[1]);
31.
32. /* unload the shared library */
33. if (dlclose(handle) < 0) {
34.     fprintf(stderr, "%s\n", dlerror());
35.     exit(1);
36. }
37. return 0;
38. }
```

在里面碰到了两个需要动态链接的函数，`addvec` 和 `printf`，标准库里的总能找得到，关键的是 `addvec` 要在执行的时候去做链接。

所以先要打开共享库（line 15），指定库的文件名和 LAZY（按需）。打开以后库是一个 `handle`（指向这个库的一个指针），然后在库中去找 `addvec`（line22），如果没有错，我们就可以调用这个 `addvec`。调用结束以后就可以关闭这个库。

动态链接库，一种是软件发布，也就是 `shared library` 如果打了新补丁了，我们可以下一个更新，下次我们允许程序的时候，程序会链接新的共享库。另外的一个应用就是动态网页，最基础的是 `CGI`，我们要 `fork` 出一个子进程，然后用 `execve` 执行我们的东西。如果用我们的方式效率会更高一点（生成动态内容的函数打包到一个 `shared library` 中，即 `server` 不再需要 `fork+execve` 了，因为 `execve` 执行的时候要加载很多 `shared library`，这个操作比较慢）。

动态链接主要想克服静态链接中的几个问题

1. 库中有漏洞，怎么打补丁。静态链接中所有应用要重新来一遍。
2. 不要让很多相同的代码占用内存

位置无关代码

共享库的代码进了内存往哪儿放？一个 `naive` 的方法是每个函数预留一个固定的地址，每个库函数都要预留一个地址，但是这样地址的空间利用率不高。

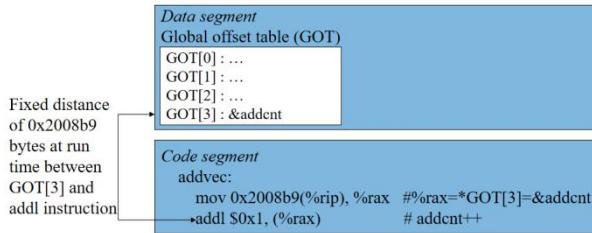
- Better: `load and execute at any address`
 - `Position-independent code (PIC)`
 - `gcc with -fPIC`

`PIC` 代码，我们在第三章里头就看到过一些。比如两个函数（`main`, `swap`）写在一起的时候，`call swap` 放了一个 `0x7`，其实这个 `7` 就是 `PIC`。

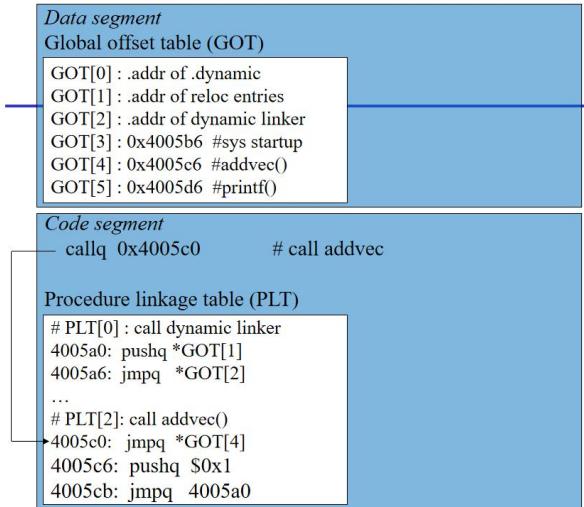
GOT 表

问题比较大的就是外部的，比如 `printf` 和 `addvec` 在链接的时候不知道在哪，只有到了 `loader` 阶段我们才知道。我们采用 `GOT` 表（`global offset table`），它在 `data segment` 的一开始。这个 `GOT` 表是每个程序自有的，不是共享的。`addcnt` 随便放在哪里，我们使用间接的访问方式。

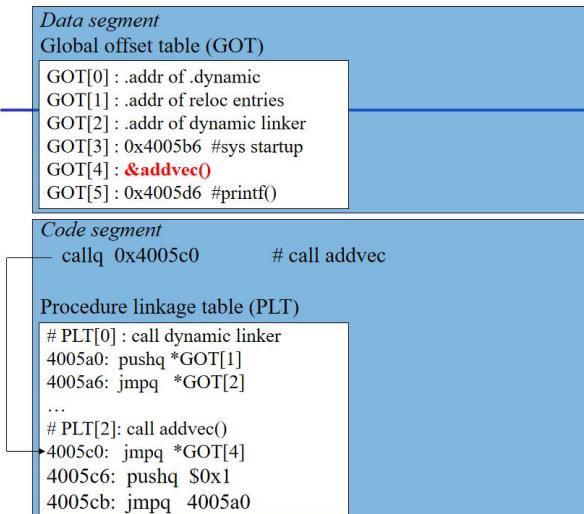
• PIC Data References



在 partially link 的时候，这两个东西都已经有了，可以填进去。在 loader 的时候，addcnt 进到内存后有了一个固定的地址，这个时候再把其地址放到 GOT[3] 中，在代码中，我们可以看到，是先把 GOT[3] 的地址放到 rax 中，再把 GOT[3] 的内容作为地址，加一，也就是两次间接访问。



对于代码，有一个更加复杂的过程（PLT，procedure linkage table），比如说我要调用一个 addvec。这个 PLT 是以表的形式出现在代码段中，这个是一段段代码，代码有入口 PLT[1]、PLT[2] 等。要调用动态链接的函数，就是调用这个表。比如我们 call addvec，可以看到表中有调用 *GOT[4]，跳转到 *GOT[4] 后，存的就是紧接着的下一条指令，然后再跳转到 4005a0，回到了 PLT[0]，它首先把 relocation entries 压栈，然后跳到 *GOT[2]，再跳转到动态链接器的地址，执行的结果就是把 addvec 的地址填进去了。



第二次在执行的时候，就能直接执行 addvec 这个函数。这就是 lazy binding，要用的时候采

取绑定。

动态内存分配

然后到了这本书的最后一章，第九章。我们先来讲 9.9（动态内存分配），这个 lab 容易出 bug，里面都是指针的指针。

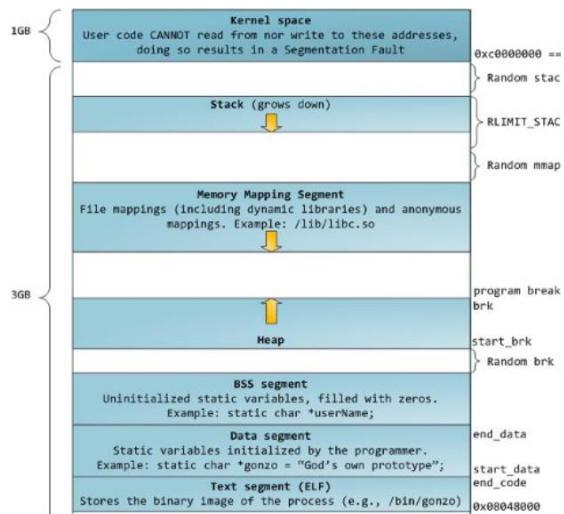
大部分程序里头，通常是运行的时候根据程序运行的需要，向系统申请一块空间，C 中就是 malloc。比如 Java 申请完就不需要主动释放，因为它存在垃圾回收机制。

我们做了一个抽象，把内存变成 block 的集合，每次申请就是从空的 block 中拿出一点分配。

整个动态内存空间，占的是 runtime heap（动态申请空间的堆）。

malloc 就是申请多少个 byte，会对齐到 8 个 byte。如果内存空间不够了，也会返回 NULL 指针。

free 传入一个指针，但必须是动态 malloc、calloc（分配出来都是 0）、realloc（重新分配，并 copy 原数据）申请出来的指针。



Sbrk 就是可以调整申请堆的大小，返回的是老的 brk。调用 sbrk0 就是可以知道当前堆的值是多少。

堆上是 block 的集合（4byte），每次申请是 8byte 对齐。灰色是应用程序正在使用的，没有人申请的叫做 free block。一开始整个空间都是 free 的，如果要 5 个，因为八对齐，就会多占一个。如果 freeP2，那么中间的六个就会 free 掉，如果再 malloc，可以继续拿出来给别人用。

malloc 约束：

在实现的时候，要能够处理任意这样的请求序列。每来一个请求要立马有反应，这个很重要，否则 malloc 的很多问题就不存在了。第三，就是 malloc 只能使用堆上的空间，不能使用别的地方。第四，对齐要求。第五，一旦块被分配出去了（灰色），那么系统或者 malloc 就不能再使用了。

目标：

给定一个 malloc 和 free 的序列，我们要达到

1. 最大吞吐量

2. 最高内存利用率

这两个目标可能是冲突的。吞吐就是单位时间内能够处理多少次请求。

对于请求序列中的一个请求：

如果是 `malloc(m)`, 实际会从 `free` 的空间中, 给应用至少 m 个 byte (m 称为 payload),

为了方便管理, 拿出来的空间会略多一些。到请求 R_k 为止, 当前 aggregate payload P_k , 到

k 这个地方有效的 aggregate 的和。 k 也有一个大小, 假设当前堆大小为 H_k , 假设 H_k 是单调不下降的。

定义：峰值内存利用率 $U_k = \max_{i \leq k} P_i / H_k$

利用率不高是由于 fragmentation 引起的, 分成两种。

1. **internal fragmentation**, 申请 5 个 int, 因为有对齐的要求, 就会浪费一个。Payload 是不会算进去的, 但是 H_k 就会浪费一个; 系统还会使用一些额外的空间来做管理 (实现的时候的 policy), 这也算 internal fragmentation。其最主要的特征, 是这个根据已知的一些 request, 就可以很清楚地算出来。

2. **external fragmentation**, 比如我们在一系列操作完后, 中间空了一些, 但是我们申请的更多, 那么我们必须往后扩容堆才可以申请到新空间。这种 external fragmentation 是和未来的 request pattern 有关的, 比较难以测量。

我们该怎么找到很多分散的 free block (管理); 选择哪一个 block; 剩下来的小空间该怎么处理; 怎么把连在一起的 free blocks 算成一个。

implicit list 把所有分配的都链起来, 区分是否分配只需要 1 个 bit 就行,

动态分区分配算法

https://blog.csdn.net/weixin_43886592/article/details/107581653

first-fit, 找到第一个符合条件的空白区块。

Next-fit, 上次我 search 的时候在哪里结束的, 从这个地方继续开始找。

Best-fit, 找到一个最小且适合自己需求的。

怎么合并呢?

Free 的时候, 往后看看, 就可以合并了, 那么前面的怎么看呢? 这是一个单向链表。

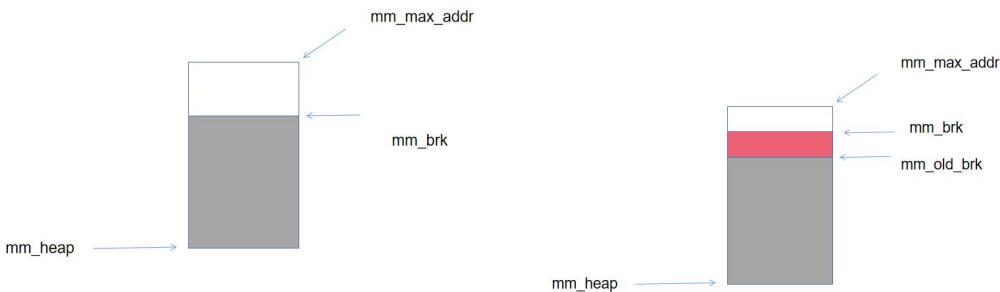
Boundary tags[Knuth73]

2021/5/19

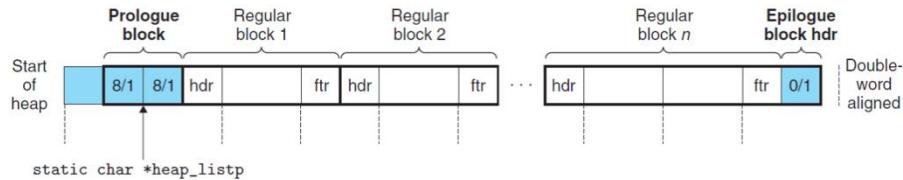
动态内存分配的实现

上节课讲了动态内存分配的一些原则和目标，今天来讲实现。

我们模拟了一个堆，在模拟的堆上使用 `mm_malloc`、`mm_free` 等操作。这个堆是一块连续的内存区域，最底下就是 `mm_heap`，堆的顶上是 `mm_brk`。最大就到 `mm_max_addr`。我们这里 `sbrk` 只能变大，不能变小。给了一个参数 `incr`，把 `brk` 变大。



下面我们来看 `malloc` 和 `free` 怎么做。`Malloc` 要有一个连续的 `block` 的集合，每个 `block` 都是 4 个 byte。



一串块里有 4 个块是比较特殊的，第一个块空在那里整个不用。头上有两个 `prologue block`。最后有一块 `0/1`，标识是尾部。中间的块可以正常变成那些可以 `malloc` 和 `free` 的块。还有一个 `heap_listp`。

```
1 /* Basic constants and macros */
2 #define WSIZE 4 /* word size (bytes) */
3 #define DSIZE 8 /* double word size (bytes) */
4 #define CHUNKSIZE (1<<12) /* Extend heap by this amount (bytes) */
5
6 #define MAX(x, y) ((x) > (y)? (x) : (y))
7
8 /* Pack a size and allocated bit into a word */
9 #define PACK(size, alloc) ((size) | (alloc))
10
11 /* Read and write a word at address p */
12 #define GET(p) (*((unsigned int *)(p)))
13 #define PUT(p, val) (*((unsigned int *) (p)) = (val))
14
15 /* Read the size and allocated fields from address p */
16 #define GET_SIZE(p) (GET(p) & ~0x7)
17 #define GET_ALLOC(p) (GET(p) & 0x1)
18
19 /* Given block ptr bp, compute address of its header and footer */
20 #define HDRP(bp) ((char *)(bp) - WSIZE)
21 #define FTRP(bp) ((char *)(bp) + GET_SIZE(HDRP(bp)) - DSIZE)
22
23 /* Given block ptr bp, compute address of next and previous blocks */
24 #define NEXT_BLKP(bp) ((char *)(bp) + GET_SIZE(((char *)(bp)-WSIZE)))
25 #define PREV_BLKP(bp) ((char *)(bp) - GET_SIZE(((char *)(bp) - DSIZE)))
```

要实现这样一个东西有两页的宏。

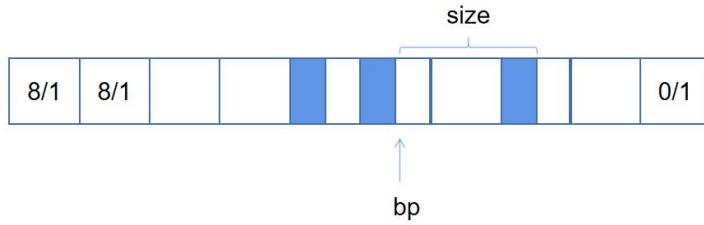
`CHUNKSIZE`：就是堆扩大的最小单位是 4K 个 byte。

`PACK`：`Footer` 和 `header` 是要记录信息的，都是八的倍数，最后三个是 0。可以用一位表示块的状态是 `allocated` 还是 `free`。所以 `size` 和一位的 `alloc` 可以拼起来称为一个复合的编码，即最后三位为 0 的 `size` 和一位的 `alloc` 做了一个 OR。

`PUT,GET`：放好以后可以读出来。

`GET_SIZE,GET_ALLOC`：最后三位清零，或者只读最后一位。

最后，我们要对链进行操作。

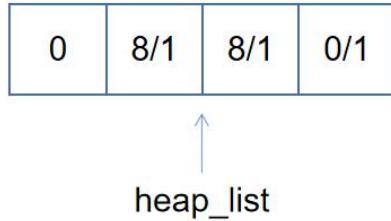


Bp 减 4 就是它的头。FTRP 就是 bp+size - DSIZE 就得到了 foot。

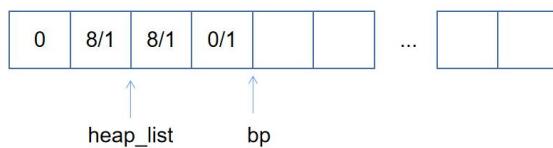
NEXT_BLKP 和 PREV_BLKP 也就是减 8 等。

mm_init()	1 static void *extend_heap(size_t words)
1 int mm_init(void)	2 {
2 {	3 char *bp;
3 /* create the initial empty heap */	4 size_t size;
4 if (heap_listp = mem_sbrk(4*WSIZE)) == (void *) -1	5 /* Allocate an even number of words to maintain alignment */
5 return -1;	6 size = (words % 2) ? (words+1) * WSIZE : words * WSIZE;
6 PUT(heap_listp, 0); /* alignment padding */	7 if ((long)(bp = mem_sbrk(size)) == -1)
7 PUT(heap_listp+(1*WSIZE), PACK(DSIZE, 1)); /* prologue header */	8 return NULL;
8 PUT(heap_listp+(2*WSIZE), PACK(DSIZE, 1)); /* prologue footer */	9 /* Initialize free block header/footer and the epilogue header */
9 PUT(heap_listp+(3*WSIZE), PACK(0, 1)); /* epilogue header */	10 PUT(HDRP(bp), PACK(size, 0)); /* free block header */
10 heap_listp += (2*WSIZE);	11 PUT(FTRP(bp), PACK(size, 0)); /* free block footer */
11 /* Extend the empty heap with a free block of CHUNKSIZE bytes */	12 PUT(HDRP(NEXT_BLKP(bp)), PACK(0, 1)); /* new epilogue header */
13 if (extend_heap(CHUNKSIZE/WSIZE) == NULL)	13 /* Coalesce if the previous block was free */
14 return -1;	14 return coalesce(bp);
15 return 0;	15 }
16 }	16 }

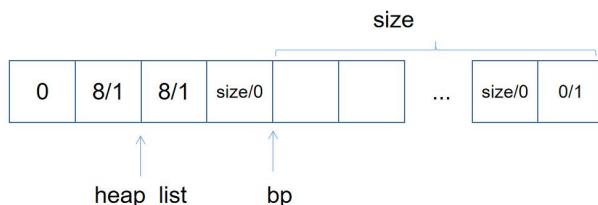
一开始先拿 4 个 word size 出来。初始化成这样：



下面使用 extend_heap，也就是给它 4K 个 byte。传入的时候传的 1000 个 word，然后看是不是 2 的倍数，是否到补成偶数（即是否 byte 是 8 的倍数）。



0/1 就正好可以成为新扩堆的 header，因为前面都借了一个 0/1 作为 header，所以最后的修改为 0/1



扩完了以后，如果前面是个空的 block，它会自动合并起来，进行 Coalesce。

```

1 void mm_free(void *bp)
2 {
3     size_t size = GET_SIZE(HDRP(bp));
4
5     PUT(HDRP(bp), PACK(size, 0));
6     PUT(FTRP(bp), PACK(size, 0));
7     coalesce(bp);
8 }

```

做 free 比较简单，就是把 block 的标识变成 0，我们要做一个 coalesce，也就是前后如果是 free 的，要合并在一起。

```

10 static void *coalesce(void *bp)
11 {
12     size_t prev_alloc = GET_ALLOC(FTRP(PREV_BLKP(bp)));
13     size_t next_alloc = GET_ALLOC(HDRP(NEXT_BLKP(bp)));
14     size_t size = GET_SIZE(HDRP(bp));
15
16     if (prev_alloc && next_alloc) /* Case 1 */
17         return bp;
18     }
19
20     else if (prev_alloc && !next_alloc) /* Case 2 */
21         size += GET_SIZE(HDRP(NEXT_BLKP(bp)));
22         PUT(HDRP(bp), PACK(size, 0)); 要先修改HDRV
23         PUT(FTRP(bp), PACK(size, 0)); 否则FTRP算不对
24     return(bp);
25 }

```

```

27     else if (!prev_alloc && next_alloc) { /* Case 3 */
28         size += GET_SIZE(HDRP(PREV_BLKP(bp)));
29         PUT(FTRP(bp), PACK(size, 0));
30         PUT(HDRP(PREV_BLKP(bp)), PACK(size, 0));
31     }
32 }
33
34 else { /* Case 4 */
35     size += GET_SIZE(HDRP(PREV_BLKP(bp))) +
36             GET_SIZE(FTRP(NEXT_BLKP(bp)));
37     PUT(HDRP(PREV_BLKP(bp)), PACK(size, 0));
38     PUT(FTRP(NEXT_BLKP(bp)), PACK(size, 0));
39 }
40
41 }

```

找到前后的 header 或者 footer 以后，看看最后一位是否为 1，也就是分 00,10,01,11 四种情况。如果确实要合并，就是把合并以后的 size 算出来以后。修改 header 和 footer，注意一定要先修改 header block 的 size，否则算不对 footer 的位置。

```

1 void *mm_malloc(size_t size)
2 {
3     size_t asize; /* adjusted block size */
4     size_t extendsize; /* amount to extend heap if no fit */
5     char *bp;
6
7     /* Ignore spurious requests */
8     if (size <= 0)
9         return NULL;
10
11    /* Adjust block size to include overhead and alignment reqs. */
12    if (size <= DSIZE)
13        asize = DSIZE + DSIZE;
14    else
15        asize = DSIZE * ((size + DSIZE + (DSIZE-1)) / DSIZE); 29 }

```

```

17     /* Search the free list for a fit */
18     if ((bp = find_fit(asize)) != NULL) {
19         place(bp, asize);
20         return bp;
21     }
22
23     /* No fit found. Get more memory and place the block */
24     extendsize = MAX(asize, CHUNKSIZE);
25     if ((bp = extend_heap(extendsize/WSIZE)) == NULL)
26         return NULL;
27     place(bp, asize);
28     return bp;
29 }

```

如果 malloc 的比较小，最小会给 16 个 byte，因为还要头尾的大小。如果超过 8 个 byte 的话，我们把它凑出一个 8 的倍数来，并且还要加上 overhead (header 和 footer)。

如果 free block 不够大，我们就去扩大这个堆，要来更多的空间。如果需求大于 4K，就会直接扩需求；如果需求小于 4K，那么就扩容 4K。

```

1. static void *find_fit(size_t asize)
2. {
3.     void *bp ;
4.
5.     /* first fit search */
6.     for (bp = heap_listp; GET_SIZE(HDRP(bp)) > 0 ; bp = NEXT_BLKP(bp) ) {
7.         if (!GET_ALLOC(HDRP(bp)) && (asize<=GET_SIZE(HDRP(bp)))) {
8.             return bp;
9.         }
10.    }
11.    return NULL; /*no fit */
12. }

```

`find_fit` 就是在堆上一个个 `block` 去找，从 `heap_listp` 开始一个个 `block` 去找。先看是不是 `free` 的，再看 `block size` 是否大于等于我们所需要的。这里实现的是 `first fit`，找到以后直接把第一个 `block` 的指针返回。一直到最后一个 `block`，也就是 `0/1` 的时候，得到 `size` 是 `0`，就结束了。

```

1.     static void place(void *bp, size_t asize)
2.     {
3.         size_t csize = GET_SIZE(HDRP(bp));
4.
5.         if ( (csize - asize) >= (DSIZE + OVERHEAD) ) {
6.             PUT(HDRP(bp), PACK(asize, 1));
7.             PUT(FTRP(bp), PACK(asize, 1));
8.             bp = NEXT_BLKP(bp);
9.             PUT(HDRP(bp), PACK(csize-asize, 0));
10.            PUT(FTRP(bp), PACK(csize-asize, 0));
11.        } else {
12.            PUT(HDRP(bp), PACK(csize, 1));
13.            PUT(FTRP(bp), PACK(csize, 1));
14.        }
15.    }

```

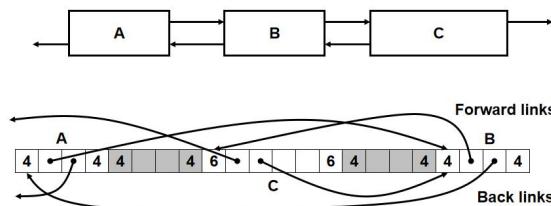
`place` 就是如果 `free` 块比我所需要的大，该怎么去处理。`c_size - a_size` 多下来的东西要大于等于 `16`，才有可能做成一个新的 `block`，否则就直接用。头好了以后，就可以得到 `footer` 的位置，然后可以找到 `next_block` 的位置，然后再去设置第二个 `block` 的 `header` 和 `footer`。

刚才，我们讲了一个简单的算法，我们只实现了功能没有考虑 `throughput`。它会比较慢，因为在 `find_list` 的时候在遍历这个 `block`，我们希望只遍历 `free` 的 `block`。

优化方法：

1. `free block` 自己做成一个双向链表，在 `find_fit` 的时候只需要 `search free blocks`。

Explicit free lists



我们可以看到每个 `block` 有两个指针，每个指针 `8` 个 `byte`。要维护这样一个优化，需要

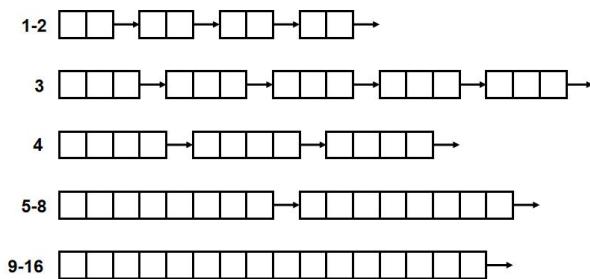
16byte 的 payload。注意到链表的顺序和在堆上的顺序不一定一致，上例中就是 ABC 的顺序。

第一种方式是 LIFO (last-in-first-out) 策略，这也是导致双向链表新的 block 插入在链表的头上，好处就是常数时间的插入，但是 fragmentation 可能不太好。

第二种方式是 address-order 策略，就是按照顺序进行插入。这样插入的时候就不一定是常数复杂度了。

Segregated Storage

2. Segregated Storage 把 free block 按照大小分类，把大小同一类的放在一个链上。



我们可以看到小的时候单独的多一点，大的时候就按照 2 的幂次方进行。

具体实现时有一些方法，比如 simple segregated storage，对每个 size 都有独立的堆的 free list。所以数据进来的时候不会有 split 操作。

- Separate heap and free list for each size class
- No splitting
- To allocate a block of size n:
 - if free list for size n is not empty,
 - allocate first block on list (note, list can be implicit or explicit)
 - if free list is empty,
 - get a new page
 - create new free list from all blocks in page
 - allocate first block on list
 - constant time

如果 free list 不为空，那么就拿出来用；如果 free list 为空，那么我就去扩堆，得到一个新的 page，把 page 切一切，得到一些 free blocks，然后从头上拿走一个。

Segregated Fits

首先每个 size 也有一个独立的堆。

- To allocate a block of size n:
 - search appropriate free list for block of size m > n
 - if an appropriate block is found:
 - split block and place fragment on appropriate list (optional)
 - if no block is found, try next larger class
 - repeat until block is found
 - if no blocks is found in all classes, try more heap memory

如果要 `malloc n`, 如果有也是直接拿出去。如果没有的话, 我们就往下走, 往 $n+1$ 方向去找。如果都没有了, 那才去扩堆。如果得到的 large size, 那么要做 split, 得到小的添加进小的 free list 中 (比如为了 `malloc 9`, 拿到了 16, 那么多出来的 5(16 - 9 - 2)就要放到 5 的 free list 中)。Free 的话要做 coalesce。

Search 的时间比较快。First-fit 策略接近于 best-fit 策略。Coalesce 需要花点时间, 这个可以用 LAZY 的操作, 也就是 deferred coalesce, 此时就需要递归地进行合并。

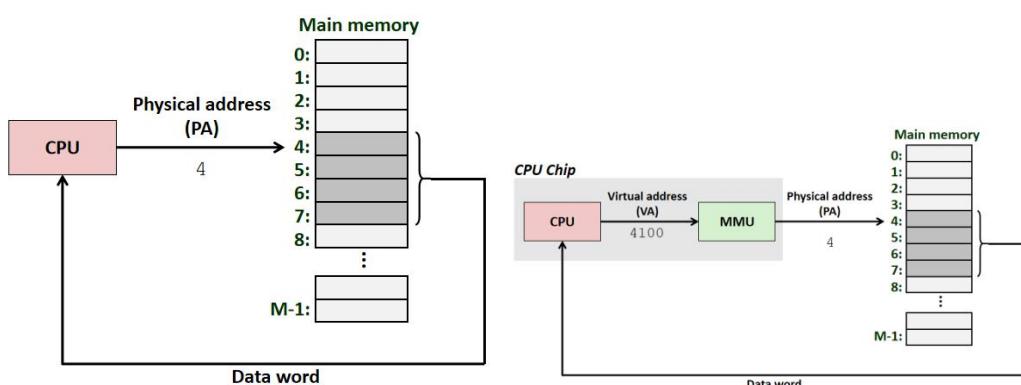
Buddy System

在刚才的地方有一个特殊的, 叫做 Buddy System。每个 block size 都是 2 的幂次。每次要一个 n 的时候, 都去找比 n 大的最小的一个 2 的 k 次方。如果拿到了更大的, 就每次切一半, 按照 2 的幂次分解出新的 blocks。这个在 OS 中用的比较多。

Virtual Memory

物理内存:

最早的 CPU 访问内存就是去拿物理地址去访问的。



真正的数据还是存放在内存里。虚地址怎么到实地址去查呢, 这就需要特殊的硬件 MMU 来进行翻译。翻译的时候实际就是去查表。这个表是操作系统来管的, 所以虚拟地址翻译主要是硬件 MMU, 但是翻译所查的表是操作系统来管的。

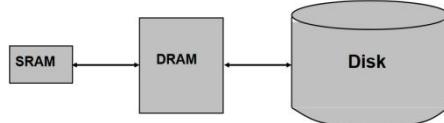
- N-bit Address Space
 - **Virtual address space:**
Set of $N = 2^n$ virtual addresses {0, 1, 2, 3, ..., $N-1$ }
 - **Physical address space:**
Set of $M = 2^m$ physical addresses {0, 1, 2, 3, ..., $M-1$ }
- Clean distinction between
 - data (**bytes**) and their attributes (**addresses**)
- Each object can now have multiple addresses
 - one physical address, one (or more) virtual addresses

对于一个真正要存储的对象, 其在计算机系统中只有一个物理地址, 但它可能有多个虚拟地址, 因为多个不同的进程都可以访问它。最典型的就是我们的共享库, 不同进程中都有不同的虚地址。

- $K=2^{10}$ (Kilo), $M=2^{20}$ (Mega), $G=2^{30}$ (Giga),
- $T=2^{40}$ (Tera), $P=2^{50}$ (Peta), $E=2^{60}$ (Exa)

#virtual address bits (n)	#virtual address (N)	Largest possible virtual address
8	256	255
16	64K	64K-1
32	4G	4G-1
48	256T	256T-1
64	16E	16E-1

为什么要有 virtual memory?

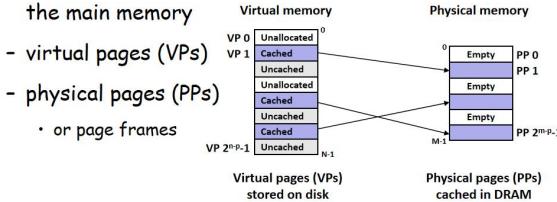


- 虚地址可以把内存作为一个 cache 来用。在第六章 cache 的时候，一个 line 就是 64 个 byte 放 8 个 long。在虚拟内存这个地方，4K、2M、1G 作为一个 page，因为一旦缺页了以后去查硬盘会非常慢。我们通常使用 fully-associated，写策略使用的是 write back 策略（就写内存，淘汰的时候一起写到硬盘中去）。

划分出来的称为 page

Page

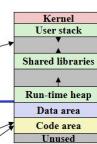
- The data on disk is partitioned into **blocks**
 - Serve as the transfer units between the disk and the main memory
- virtual pages (VPs)
 - VP 0 Unallocated
 - VP 1 Cached
 - VP 2 Uncached
 - VP 3 Unallocated
 - VP 4 Cached
 - VP 5 Uncached
 - VP 6 Unallocated
 - VP 7 Uncached
 - VP 2^{n-p-1} Unallocated
- physical pages (PPs)
 - PP 0 Empty
 - PP 1 PP 0
 - PP 2 Empty
 - PP 3 Empty
 - PP 4 PP 2^{m-p-1}
- or page frames



划分完在 disk 上就有一个个 page。VM 是一个逻辑的划分，实际上不存在这个。在进程栈中，有很多灰色的部分，这些都不能访问。我们把它叫做 unallocated，其看上去是属于这个进程的，但是实际上是不能访问的，因为系统没有分配给进程。剩下的叫做 allocated，又分成是内容在物理内存中 (cached)，或者内容不在物理内存中 (uncached)。如果不在内存中，会产生一次 page fault。

Page Attributes

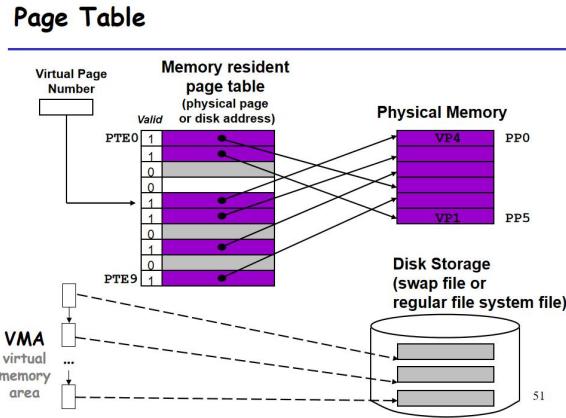
- Unallocated:
 - Pages that have not yet been allocated (or created) by the VM system
 - Do not have any data associated with them
 - Do not occupy any space on disk
- Cached:
 - Allocated pages that are currently cached in physical memory.
- Uncached:
 - Allocated pages that are not cached in physical memory



彩色的一部分在内存里，一部分在硬盘上。

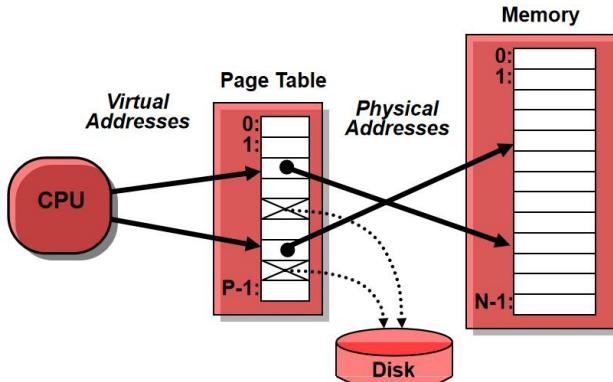
Page Table

虚实地址翻译查的表就是一个 page table。



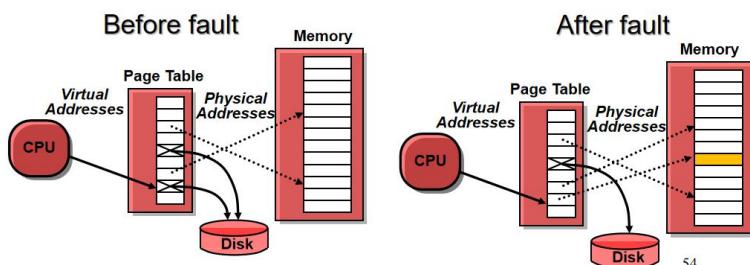
这个表是要在内存当中的。

拿到一个虚拟地址以后，最后十二位不要了，因为这在某一个页里。前面一部分叫做 **virtual page number**，去查这个表，看看这个东西是不是在内存里头。如果存在，靠最后十二位来确定是在内存中的哪个位置。如果查到的是灰色的（valid = 0），分成 unallocated 和 uncached，在 linux 中，有一个 VMA 机制，如果数据在硬盘里，会用一个链去把它链起来。也就是再去查一个 VMA，把硬盘中的数据放到内存里。



Address Translation: Hardware converts *virtual addresses* to *physical addresses* via an OS-managed lookup table (*page table*)

- Swapping or paging
- Swapped out or paged out
- Demand paging



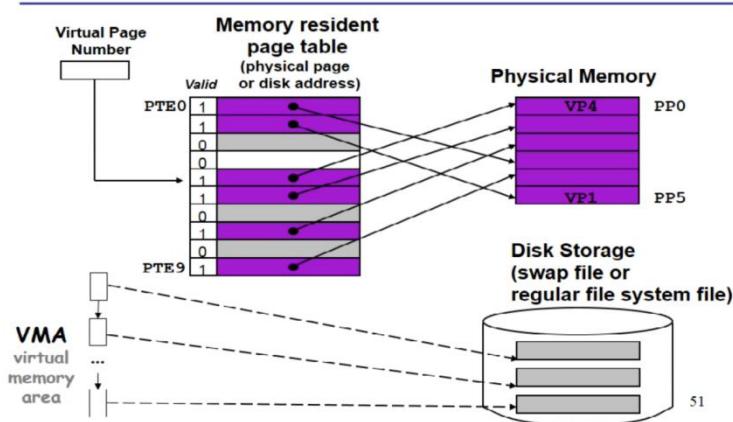
2021/5/26

cached 在内存当中

虚拟地址空间中 array 的状态是未分配、 cached、 未 cached。

一个 bit 只能区分两种状态。

Page Table



虚地址到实地址的映射

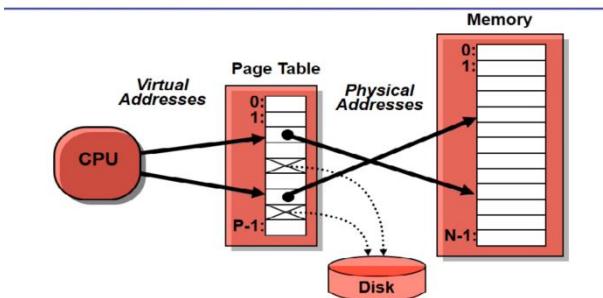
虚地址到实地址的映射是由 MMU 完成的，

MMU 拿到 VA(virtual address)后, 查一张表叫做 PT, 然后拿查出来的 PA(physical address)去找 PM (physical memory)。这张表有一个 valid 位, 置 1 的时候说明内容在 physical memory 中。这张表是以 page 为单位, 可能是 4K, 2M, 1G 为单位。如果是 4K 的, 拿到一个 64 位的虚拟地址, 后面的 12 位代表在页里头是哪一个 byte。要做地址翻译呢, 实际上是把前面的 48 位 (也被称作 virtual page number) 变成 physical page number, 这样就查到了一个物理页, 最后用 12 位来找到位置。页表里就存了一个 physical page number。

上图中 1 表示 cached, 紫色的就是存了一个 physical page number, 可以让我们去访问内存。0 的时候有两种状态, 一种是灰色的 (可以用, 但是内容在硬盘而不在内存中), 白色是不能访问的。Linux 中的处理方式是用了一个链表, 它就是一个 virtual memory array (包含很多个 page), 如果是 0 的话, 就来查这个表, 如果不在这个表里, 那就是 unallocated 否则就是 uncached。

Eg: 访问空指针为什么会报错, 空指针实际上是虚拟地址 0, 空间不能访问。CPU 拿着 0 来查表, 一定查出来是 0, 那到底是 unallocated 还是 uncached 的状态呢? 然后查 VMA, 发现不在里面, 那就是 unallocated, 也就是直接报错。

Page Hits



Address Translation: Hardware converts virtual addresses to physical addresses via an OS-managed lookup table (page table)

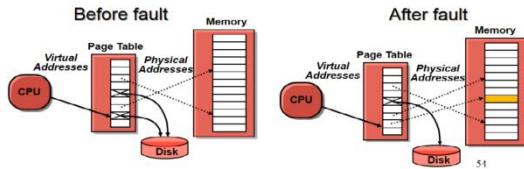
如果是 1，称为 page hit。

Page Faults

- Page table entry indicates virtual address **not in memory**
- OS exception handler invoked to move data from disk into memory
 - current process suspends, others can resume
 - OS has full control over placement, etc.

Page Faults

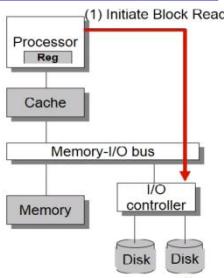
- Swapping or paging
- Swapped out or paged out
- Demand paging



如果是 0，首先称为 page fault。查 VMA 链表，如果在里头，就说明在硬盘里头。如果能访问 (uncached)，那就要把硬盘中的页放到内存里来。放的方法就是 DMA (direct memory access)。要把硬盘中的内容 copy 到内存上去，这件事情是 OS 来做的，OS 做完要去修改 page table 把 0 改为 1，然后需要 reexecute，重新再执行一遍。CPU 再拿这个地址查一次表，就可以正确访问到这个内容了。

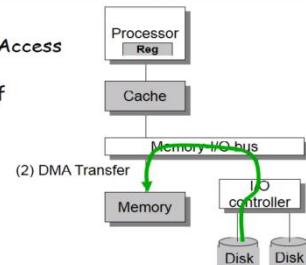
Servicing a Page Fault

- Processor Signals Controller
 - Read block of length P starting at disk address X and store starting at memory address Y



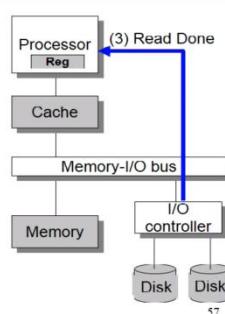
Servicing a Page Fault

- Read Occurs
 - Direct Memory Access (DMA)
 - Under control of I/O controller



Servicing a Page Fault

- I / O Controller Signals Completion
 - Interrupt processor
 - OS resumes suspended process



DMA:

- CPU 发一个命令给 IO controller，需要读硬盘的命令，需要告诉它块号，以及读出来的东

西要放到内存的那块地方。

2. IO controller 把逻辑块转化成物理的 surface、track 等，拿到数据，放到内存中。
3. 中断回来恢复进程。

缺页的时候一定要先认出来 unchached，然后搬到内存中来，搬到内存中的哪里？内存满了怎么办？我们最后一节课会讲替代策略。

Locality to the Rescue Again!

- Virtual memory works because of locality
- At any point in time, programs tend to access a set of active virtual pages called the **working set**
- Programs with better **temporal** locality will have smaller working sets
- If (working set size < main memory size)
 - Good performance for one process after compulsory misses
- If (working set sizes > main memory size)
 - **Thrashing**: Performance meltdown where pages are swapped (copied) in and out continuously

Cached 一定是基于局部性的 (locality)，我们希望花大功夫拿到内存里的数据会被多次访问。当一个数据被访问过以后，在不久的将来会被访问到，相邻的数据也会被访问到。OS 会有工作集 (working) 的概念。工作集：程序访问的虚拟地址空间。如果 working set 小于整个 virtual memory，那么就在内存中放得下，也会全部进来 (fully-associated)。这样性能会比较好。如果 working set 比 memory 大，那么内存里就放不下这些数据，那么数据会被经常淘汰，如果淘汰策略不好，那么会经常出现 Thrashing (颠簸)。

为什么要使用虚拟内存 (VM)

Why Virtual Memory (VM)?

- Uses main memory efficiently
 - Use DRAM as a cache for the parts of a virtual address space
- Simplifies memory management
 - Each process gets the same uniform linear address space
- Isolates address spaces
 - One process can't interfere with another's memory
 - User program cannot access privileged kernel information

VM as a Tool for Memory Management

- Key idea: each process has its own virtual address space

- It can view memory as a simple linear array



Virtual Address Space for Process 2:



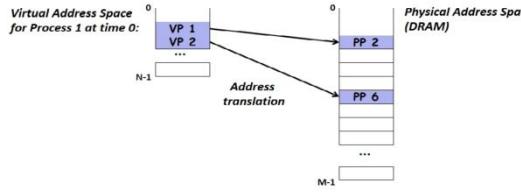
为什么要 VM

1. 提高内存的使用率
2. 简化内存管理，一个程序运行的时候就有独立的逻辑上连续的内存空间。一个进程实际上没有这么大的空间，但是我们认为是有的。

VM as a Tool for Memory Management

- Memory allocation

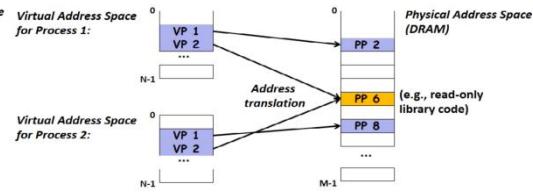
- Each virtual page can be mapped to any physical page
- A virtual page can be stored in different physical pages at different times



VM as a Tool for Memory Management

- Sharing code and data among processes

- Map virtual pages to the same physical page (e.g. PP 6)



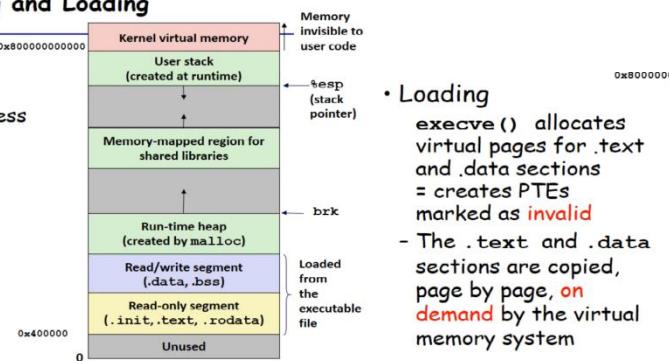
程序运行的时候看到的都是左侧的，真正内容在物理页里，在不同的时间，这两个虚拟页里的东西是可以放到不同的物理内存中。也就是缺页的时候，OS会对物理内存做管理，可能会把空间让出来。

不同的进程可以共享一个页，两个进程映射到同一个物理内存空间，用的最多的是 shared library。

Simplifying Linking and Loading

- Linking

- Each program has similar virtual address space
- Code, stack, and shared libraries always start at the same address



- Loading

- execve() allocates virtual pages for .text and .data sections = creates PTEs marked as invalid
- The .text and .data sections are copied, page by page, on demand by the virtual memory system

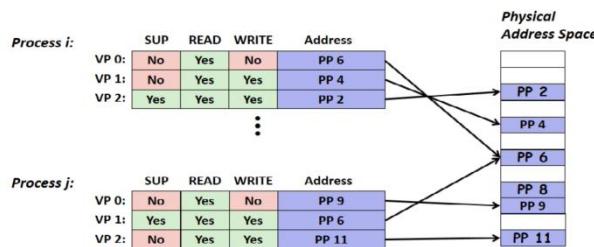
在我们举的例子里头，代码都是 600000 开始的。栈是从 FFFFFFFF 往下的。我们在 loading 的时候，实际上是把一个 ELF 文件。回顾一下 ELF，executable 里有一个 segment，其中有一个属性是 loadable（可以放到内存里去）。在加载的时候，这些 loadable 的东西就是放到内存里去，把它从文件中直接放进来。

3. 做地址空间的隔离。进程不能修改其他进程的内存，要保证两个进程之间不受干扰。用户态的进程不能访问内核态的东西。为了实现这两点，在页表当中有 0 和 1 这一项，有 physical page number。

VM as a Tool for Memory Protection

- Extend PTEs with permission bits

- The same physical page has different permission for different process



我们会额外地增加几个 bit，这里面就会设置访问权限，sup 就是是否需要内核权限才能访问，有些内存是只读的（Read: Yes, Write:No），可读可写（Read: Yes, Write:Yes）等。

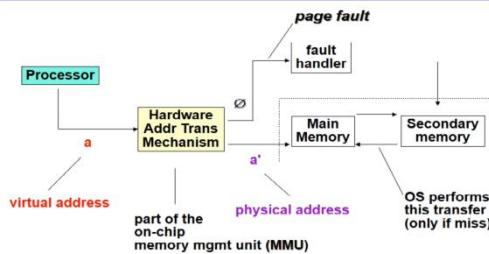
怎么两个进程之间是怎么隔离的呢？也就是每个进程会有一个自己的页表，如果是两个页对应到一个物理地址上且可读可写，就需要额外的办法来保护。否则页表两两之间不会有交集，对应的物理内存块就是进程独有的。

Address Translation

- $V = \{0, 1, \dots, N-1\}$ virtual address space
- $P = \{0, 1, \dots, M-1\}$ physical address space
- $N > M$
- Address Translation
- MAP: $V \rightarrow P \cup \{\emptyset\}$ address mapping function
- $MAP(a) = a'$ if data at virtual address a is present at physical address a' in P
 $= \emptyset$ if data at virtual address a is not present in P (invalid or on disk)

通常情况下是 $N > M$ 的。对于 x86 来说，物理地址最大是 48 个 bit，虚拟地址是 52 个 bit。我们要做的地址转换，就是拿到一个虚地址要映射回物理地址。

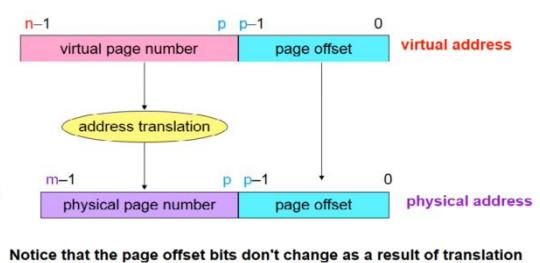
Address Translation



程序在执行的过程中就会访问到虚地址，比如 move 指令，拿到这个虚地址以后就会查 MMU，如果查到了就直接去访问内存。如果没查到，触发 page fault 这个 exception 进到 OS 里面，OS 会进一步区分出 unallocated 和 uncached。如果是 unallocated 就会直接报错。

Address Translation

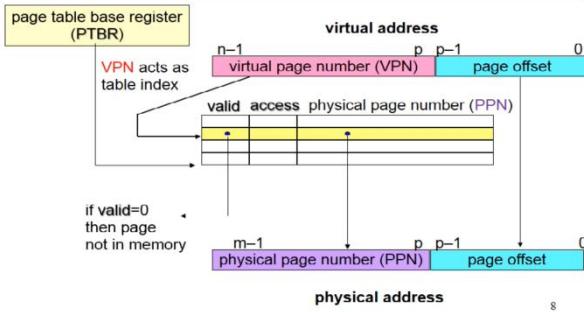
- Basic Parameters
 - $N = 2^n$ = Virtual address limit
 - $M = 2^m$ = Physical address limit
 - $P = 2^p$ = page size (bytes).
- Components of the virtual address (VA)
 - VPO: Virtual page offset
 - VPN: Virtual page number
- Components of the physical address (PA)
 - PPO: Physical page offset (same as VPO)
 - PPN: Physical page number



那么拿到了 a 以后怎么翻译成 a' 呢，因为内存都是地址去访问的， n 位的一个地址表示的就是 2^n 。在内存和硬盘当中，中间需要划开，就是通过 page 来划分的，每个 page 有一个固定的 page size 2^p ，所以也可以用 2 的幂次来表示。

要对地址翻译呢，实际上就是把 VA 和 PA 切开，后面 p 位是不变的，实际上就是把前面的 VPN 翻译成 PPN。

Address Translation via Page Table



通过首地址和下标（VPN 相当于下标），就可以访问表的条目（entry）。表的地址一定要能找得到页表的地址，x86 中表的地址存在一个特殊的寄存器 CR3 里，也就是 page table base register。

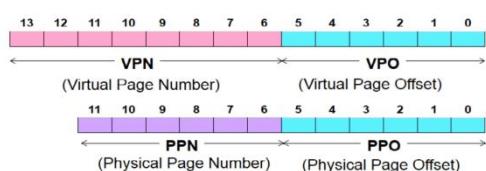
问题：这个寄存器里放的是虚地址还是实地址呢？页表是在内存当中，一定是物理内存，如果这里放的是虚地址。因为是虚地址，要翻译，又要去查页表的基地址。查到的又是一个虚地址，那么循环往复就一直查不到基地址。

进程上下文会包括一些进程的寄存器。Regs, pc, page table base register, 当进程切换的时候，就需要把这些东西记录下来，也就是 PCB（进程控制块）。进程切出去的时候要放进去存起来，进程回来的时候很重要的事情就是把页表这个东西放到 CR3 里去。拿了这个物理地址以后，再拿 VPN 就可以查这个页表了。查的时候先看 valid bit，然后再去看有没有权限（access），如果有权限，就把 PPA 和 VPO 拼在一起，这就是翻译的过程。如果是 0 的话，没翻译成功。

下面就举一个例子

Simple Memory System Example

- Addressing
 - 14-bit virtual addresses
 - 12-bit physical address
 - Page size = 64 bits (6-bit)



Simple Memory System Page Table

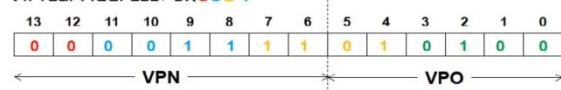
- Only show first 16 entries

PPN	Valid
00	28
01	-
02	33
03	02
04	-
05	16
06	-
07	-

PPN	Valid
08	13
09	17
0A	09
0B	-
0C	-
0D	2D
0E	11
0F	0B

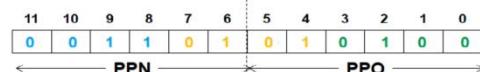
14 位的虚地址空间，12 位的物理地址空间，有一个 64bit 的 page size。最后六位是 page offset。表就是 $2^8=256$ 项。我们给出前 16 项。

Virtual Address: 0x03D4



VPN: 0x0f VPO: 0x14

Page Fault? No



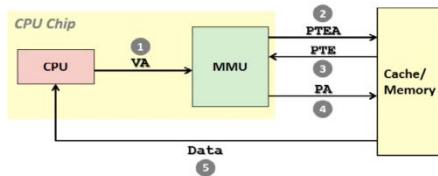
PPN: 0x0d VPO: 0x14

PA: 0x354

一下翻译成功的叫做 page hit，拿到 VM 通过 MMU 查链表。

Page Hit

VA: virtual address
PTEA: page table entry address
PTE: page table entry
PA: physical address



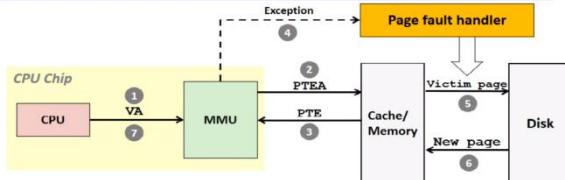
- 1) Processor sends virtual address to MMU
- 2-3) MMU fetches PTE from page table in memory
- 4) MMU sends physical address to cache/memory
- 5) Cache/memory sends data word to processor

16

有了虚拟地址以后，我们访问性能降低了。本来我们只需要访问一次内存，现在为了地址翻译，我们先要访问页表再去访问数据。

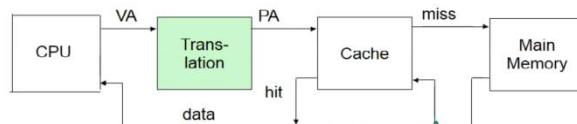
Page Fault

Page Faults



- 1) Processor sends virtual address to MMU
- 2-3) MMU fetches PTE from page table in memory
- 4) Valid bit is zero, so MMU triggers page fault exception
- 5) Handler identifies victim (and, if dirty, pages it out to disk)
- 6) Handler pages in new page and updates PTE in memory
- 7) Handler returns to original process, restarting faulting instruction

装载回来以后，我们要继续修改页表。要写入 valid 位和 PPN。做好这件事以后，OS 会尝试重新执行一遍，然后就是 page hit，成功拿到一个物理地址。



拿到一个物理地址以后，我们会先去访问 cache，如果 cache 命中了，直接从 cache 里拿，否则再到内存中去拿。

Integrating Caches and VM

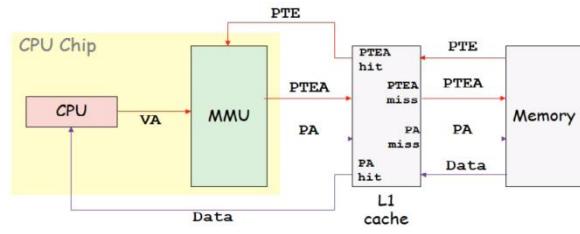
- Most Caches "Physically Addressed"
 - Accessed by physical addresses
 - Allows multiple processes to have blocks in cache at same time
 - Allows multiple processes to share pages
 - Cache doesn't need to be concerned with protection issues
 - Access rights checked as part of address translation

Integrating Caches and VM

- Perform Address Translation Before Cache Lookup
 - But this could involve a memory access itself (of the PTE)
 - Of course, page table entries can also become cached

小的 ARM 会用虚地址来访问 cache。大部分 cache 是用物理地址去访问的。

Integrating Caches and VM



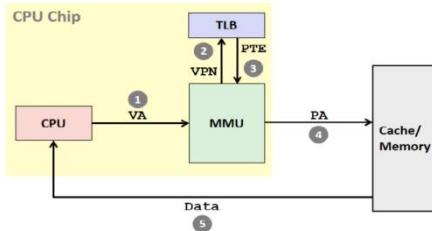
硬件 TLB

查页表的时候又多花了 4 个 cycle，怎么把这个省下来，就需要硬件 TLB 帮忙。

Speeding up Translation with a TLB

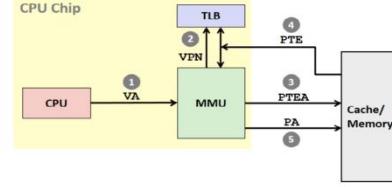
- “Translation Lookaside Buffer” (TLB)
 - Small hardware cache in MMU
 - Maps virtual page numbers to physical page numbers

TLB Hit



A TLB hit eliminates a memory access

TLB Miss

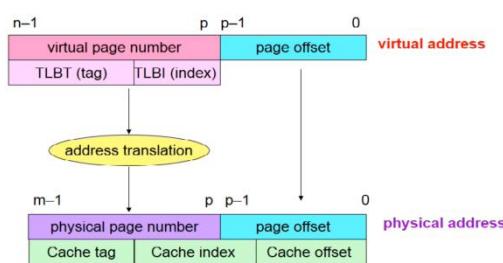


A TLB miss incurs an additional memory access (PTE)

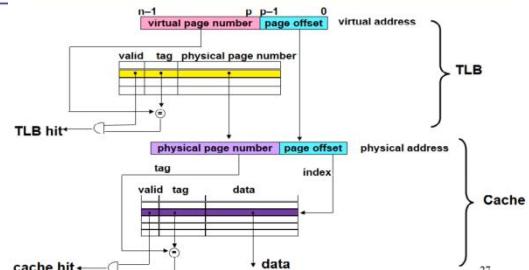
21

TLB 把地址翻译的过程做完了，如果 TLB 没查到，那只能到内存中去查，查到的这一项，还是要写回 TLB 里去。

Speeding up Translation with a TLB

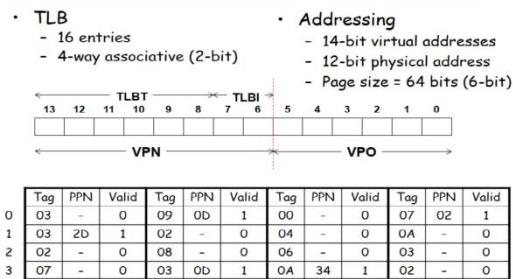


Speeding up Translation with a TLB



在 TLB 里头只有一个页表项，没有 offset。它在访问的时候需要把 VPN 切成 tag 和 index。因为 TLB 拿 index 去找第几项。

Simple Memory System TLB



2021/5/31

多级页表

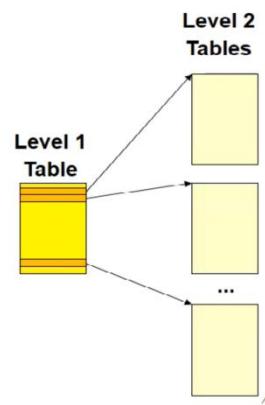
首先我们讲多级页表，也就是看看地址有多少位，比如说 2^{32} ，而页表只有 4K。那么就要有 2^{20} 个页表项。

在 64 位中， $2^{48} / 2^{12} = 2^{36}$ ，是 512G。一个页表就要 512G，实际上我们的内存没有这么大的。如果这样存放的话，我们的页表会占非常大的内存空间。为了解决这个问题，我们用到了多级列表。

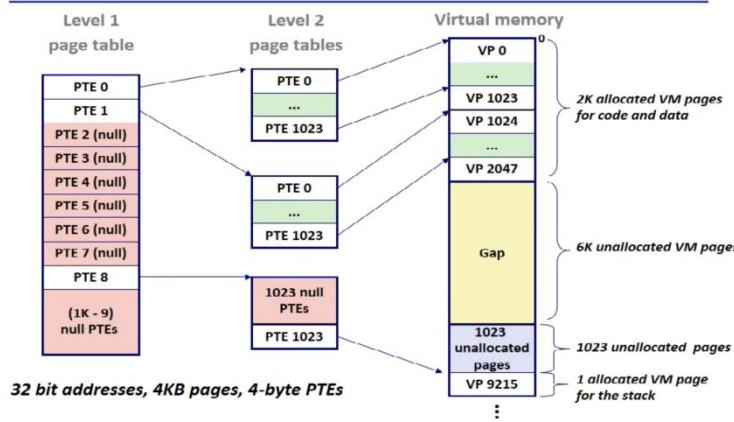
我们可以分成两级，这样看好像没什么区别。但是在实际的虚地址当中有很多地址是不能用的。栈那种用的都是很少的空间。

- Common solution

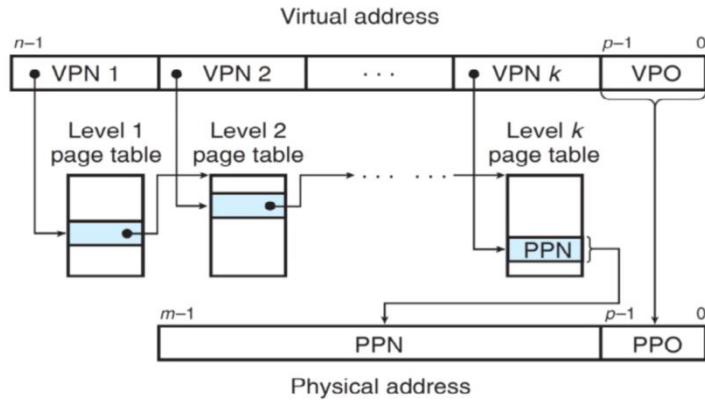
- multi-level page tables
- e.g., 2-level page table
 - Level 1 table: 1024 entries, each of which points to a Level 2 page table.
 - Level 2 table: 1024 entries, each of which points to a page



Multi-Level Page Tables



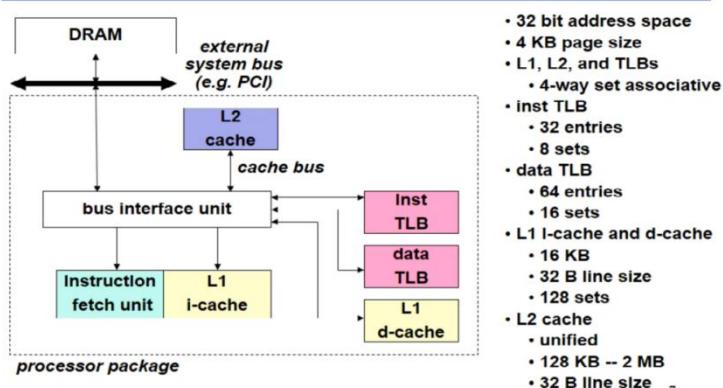
中间这些红色的都是虚拟空间不能用的。如果是一级页表，需要一个一个的取值。二级列表可以用一个项来少掉了 1000 个。二级页表出来的只有三个可以用的，只有 0 和 1 用了。8 只用了一点。页表每个是 4k，我们用 4 个页表就把整个空间都放下了。基于的原理就是在虚拟地址中有大量的连续地址是不能被访问的。



用多级页表当然可以省空间，但是查找的效率会低一些，因为我们要一级一级地去查。本来拿一个下标找到就找到了。现在是先拿虚拟地址的第一段一段 VPN1，去查 1 级页表，查到的是 2 级页表的地址，然后根据下标去查 3 级页表的地址，查到最后，才是 PPN，拼在一起得到一个物理地址。

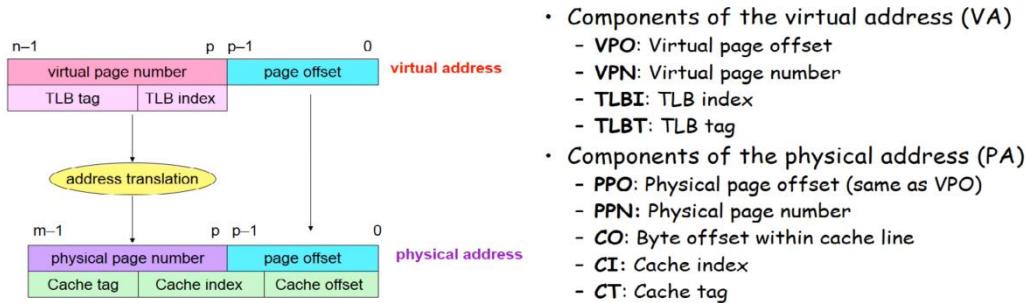
我们希望提供 TLB 来减轻这个查询的压力，因为 TLB 可以直接查到这个东西。

P6 Memory System



在 x86-32 里头，有两个 L1 i-cache 和 d-cache，还有一个 unified cache (L2)。数据 TLB 和指

令 TLB 是分开的。每个页是 4K, PPO 和 VPO 都是 12 位的。每个 cache 和 TLB 都是 4 路组关联的。指令是 32 个 entry, 也就是 8 个集合。数据是 16 个集合。



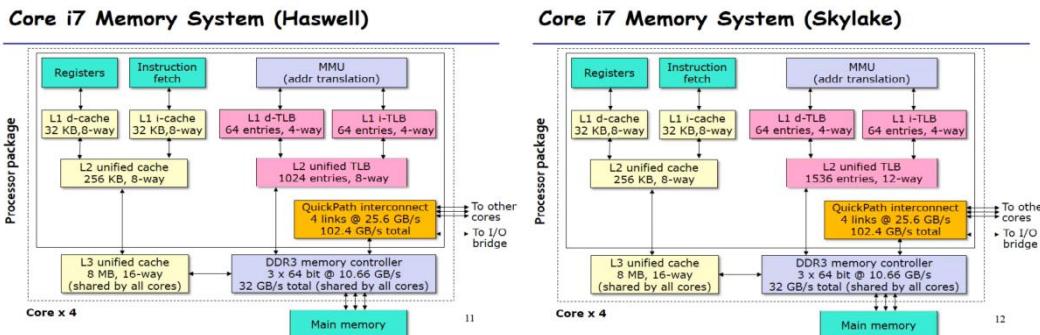
9

在这个地方要去查页表, 先去查 TLB, TLB 查不到再通过页表来查。查到了以后先去访问 cache, 如果 miss 再去访问内存。

Intel Core i7

- **Core i7**
 - The 64-bit Nehalem microarchitecture
 - Current support 48-bit (256 TB) virtual address space and 52-bit (4 PB) physical address space
 - Compatibility mode support 32-bit (4 GB) virtual and physical address spaces

我们重点介绍 intel。它是一个 64 位的微体结构。

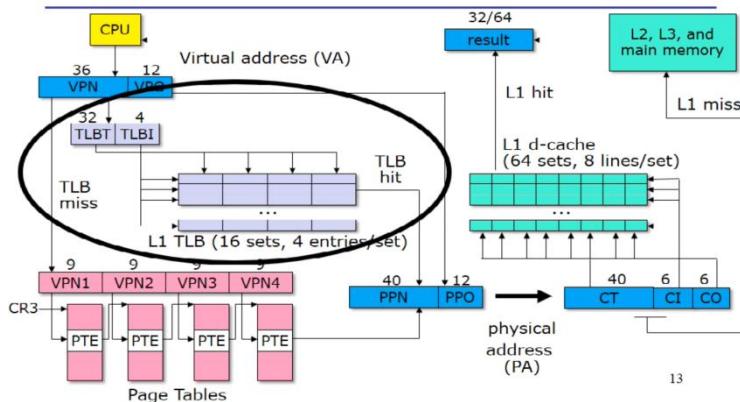


11

12

这是 x86-64 的结构图, 黄色的在 cache 的时候已经介绍过了。我们关注的是红色的区域, 这里 1 级 TLB 都是 64 个 entry, 2 级 TLB 是 8 路 1024 个 entry。

Core i7 Address Translation



拿到 36 位的 VPN 以后，首先去查 TLB，应该是 16 个集合， $\text{index} = 4$ ，剩下的 32 位就是一个 tag，拿这个 4 位的 index 去查，查到行了，就比较同一行的 4 个 tag，如果 tag 相同且 valid=1，那么就是命中，否则我们需要继续到 2 级去看。2 级的时候，就分成 $\text{TLBT}=27, \text{TLBI}=9$ 去查 2 级 TLB。如果还不命中，我们就要去查页表了。在 x86-64 中，我们是 4 级页表。第一级页表的物理地址是在 CR3 里头，查到最后一级中，存放的就是物理页。PPN 和 PPO 合在一起是 52 位。

查找 Cache 是拿物理地址去查找的，物理地址的最低 12 位，虚地址和实地址是一样的，这是特意设置的。这样的好处，我们可以拿虚拟的 12 位开始查找 cache 了，因为最开始有用的是后 12 位来查集合，等待翻译完再查比较慢。翻译的时候，至少可以从集合中把每一行拿出来，等待翻译完了以后就可以直接开始比较 tag。

PTE 作为页表的每一项，到底有几位，存了哪些内容。它应该是 64 位的，前 3 级页表，里面存的格式都是一样的，因为它们的目的就是告诉你下一级页表的地址，充当目录的作用。但是最后这一级存的是真正翻译出来的物理地址，存的内容有一些区别。

Level 1, Level 2 and Level 3 Page Table Entry

63	62	52	51	12	11	9	8	7	6	5	4	3	2	1	0
XD	Unused	Page table physical base addr	Unused	G	PS	A	CD	WT	U/S	R/W	P=1				
Available for OS (page table location in secondary storage)															
XD	Disable or enable instruction fetches														
Base addr	40 most significant bits of base address of child page table (forces page tables to be 4KB aligned)														
G	global page (don't evict from TLB on task switch)														
PS	Page size either 4K or 2M or 1G (may set to 1 only for Level 2 or level 3 PTEs)														
A	Reference bit (set by MMU on reads and writes, cleared by software)														
CD	Cache disabled(1) or enabled(0) for child page table														
WT	Write-through or write-back cache policy														
U/S	User or supervisor(kernel) mode access permission														
R/W	Read-only or read-write access permission														
P	Child page table present in memory(1) or not(0)														16

上图中白色的没有用到，我们用到的是红色的和蓝色的，中间蓝色的 40 位+后头补 12 个 0，就是下一级页表的地址。Valid = 1 的时候，后面的东西才有意义，否则 MMU 就不看它了。

G 叫做 global page，所有竞争都可以用的 page，在进程切换的时候，因为进程的页表是每个进程独有的，换了另外一个进程来，CR3 要换。页表的是缓存是放在 TLB 中，进程一切换 TLB 就要被刷掉，global 就是告诉你这一页不用被刷掉。PS 就是 page size，可以支持 3 个大小，最小的是 4k(2^{12})，然后是 2M(2^{21})，最后是 1G(2^{30})，我们看见相邻 3 个都是差 9。如果是 30，那么 PPO 只有 18 了，也就是 2 级页表；如果是 21，那么 PPO 是 27，也就是 3 级页

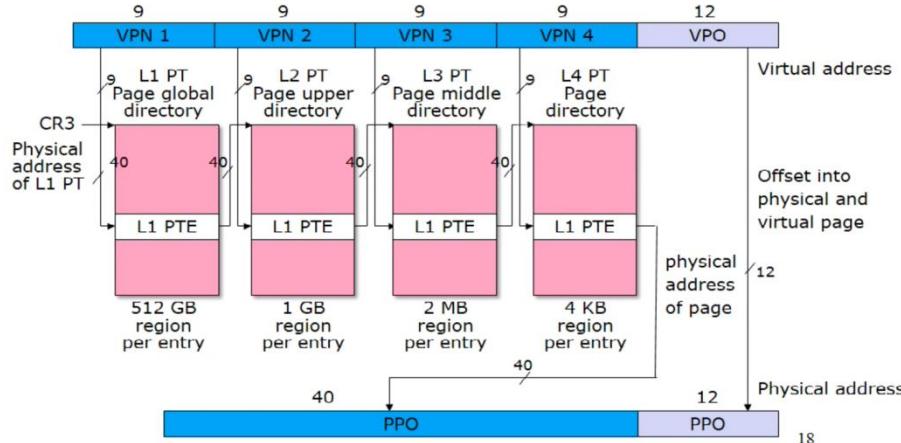
表。

A bit (access bit) 这个关系到页面淘汰，我们已经学过了 LRU。这一页一旦被访问过（读写过）了，就会被 MMU 置上。OS 会来使用这一位。

Level 4 Page Table Entry

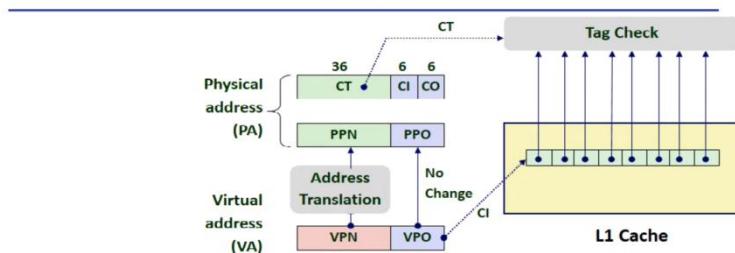
	63 62	52 51	12 11	9 8	7	6	5	4	3	2	1	0
XD	Unused	Page table physical base addr	Unused	G	D	A	CD	WT	U/S	R/W	P=1	
Available for OS (page table location in secondary storage)												
P=0												
XD	Disable or enable instruction fetches											
Base addr	40 most significant bits of base address of child page table (forces page tables to be 4KB aligned)											
G	global page (don't evict from TLB on task switch)											
D	Dirty bit (Set by MMU on writes, cleared by software)											
A	Reference bit (set by MMU on reads and writes, cleared by software)											
CD	Cache disabled(1) or enabled(0) for child page table											
WT	Write-through or write-back cache policy											
U/S	User or supervisor(kernel) mode access permission											
R/W	Read-only or read-write access permission											
P	Child page table present in memory(1) or not(0)											

第四级就有一点点区别：PS 没有了，多了一个 D (dirty bit)。在 cache 的时候，淘汰出来的时候，要把 cache 的内容写到内存里。



只能在 VPN2、VPN3 的 PS 置成 1，说明其中是存的物理地址。PS 只要为 1，就不往后走了，如果前面都是 0，那么就是一个 4 级页表。

Cute Trick for Speeding Up L1 Access

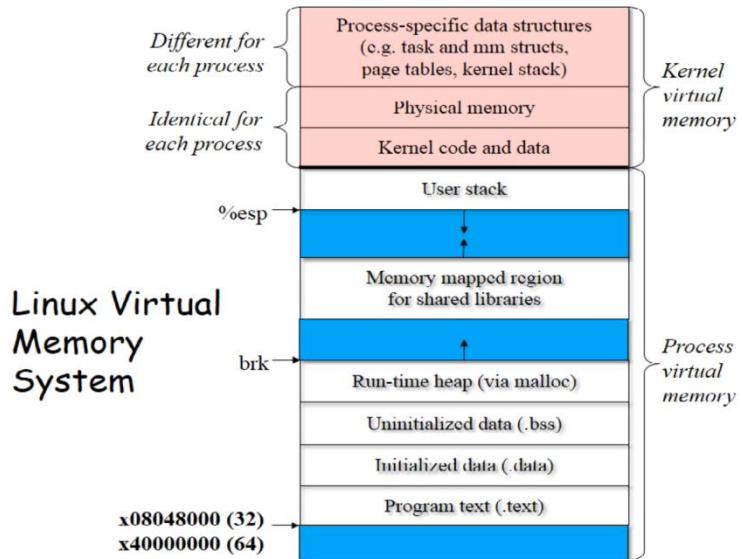


- Observation

- Bits that determine CI identical in virtual and physical address
- Can index into cache while address translation taking place
- Generally we hit in TLB, so PPN bits (CT bits) available next
- "Virtually indexed, physically tagged"
- Cache carefully sized to make this possible

拿到物理地址以后，先去查 cache，至少最后 12 位 VPO 和 PPO 是一样的，我们去查 VPO 的 12 位，拿 CI 就可以来查。查到了这个集合以后，我们知道在 4 项里头，但是不知道是谁，所以我们拿到 CT 以后再去和 Tag 去比。我们叫做 Virtually indexed, physically tagged。虚地

址查集合，物理地址比 tag。

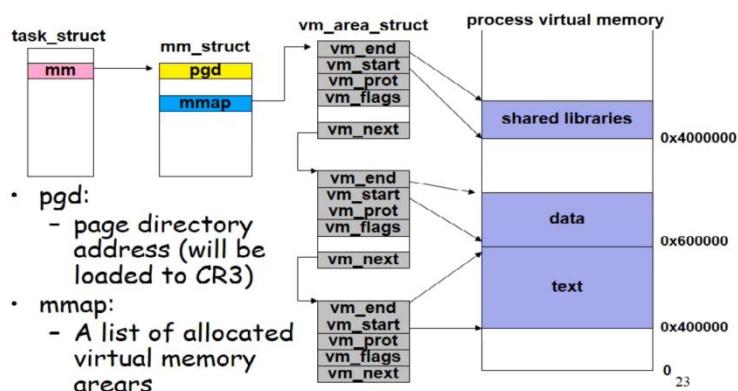


蓝色区域是大量的，我们可以节省空间。这张图里的所有东西都应该比较熟悉了。

Linux 里对虚拟地址的支持

一个 array 是连续的虚拟地址空间，每个 array 里都有一些特定的用途。

Linux organizes VM as a collection of “areas”



每一个 array 有一个对应的项。每个 array 都有这样一个数据结构，标志着哪里开始，哪里结束。Page directory address 需要在进程切换的时候放到 CR3 里去。`Vm_prot` 是这个 array 的 read write protection。`Vm_flags` 是 shared (几个进程共享，比如共享库) 还是 private。

我们再回顾一下之前提的问题，c 语言访问空指针是怎么报错的。访问时第一次一定会缺页，缺页以后进入操作系统，首先会看你这个是不是白色的那些不能访问的（是否落到紫色区域里头），如果属于不能访问的就会报 signal segmentation violation。第二种是可以访问的，但是访问权限不对，比如尝试写一个 `read only` 的 array，会报 protection violation。只有地址对+权限对的情况下，这时候才会把内容从文件中拷贝到物理页上，让这条指令重新执行。

注意在上图的 `vm_area_struct` 数据结构中，存的都是虚地址。这些值原始的都存在文件上。每个 array 对应了一个硬盘上的 object 初值，vma 要从硬盘上获取初值。最常见的是 regular file，比如在第七章介绍的 ELF。在 load 以后，把.data 和.text 对应到数据段和代码段。

1. VMA 的初值是和 regular file 的一部分对应起来了。

Memory mapping

- Area can be backed by
 - regular file on disk (e.g., an executable object file)
 - initial page bytes come from a section of a file
 - anonymous file (e.g. nothing)
 - First fault will allocate a physical page full of 0's (demand-zero page)
 - Once the page is written to (dirtied), it is like any other page

2. 匿名文件，比如 page 拿过来以后初值都是 0，比如.bss 并不在 ELF 文件中，是需要操作系统给它配上去的。

Memory mapping

- Dirty pages are copied back and forth between memory and a **swap file**.
- Swap file
 - Maintained by the kernel
 - Also known as swap space or the swap area
 - Bounds the total amount of virtual pages that can be allocated by the currently running processes

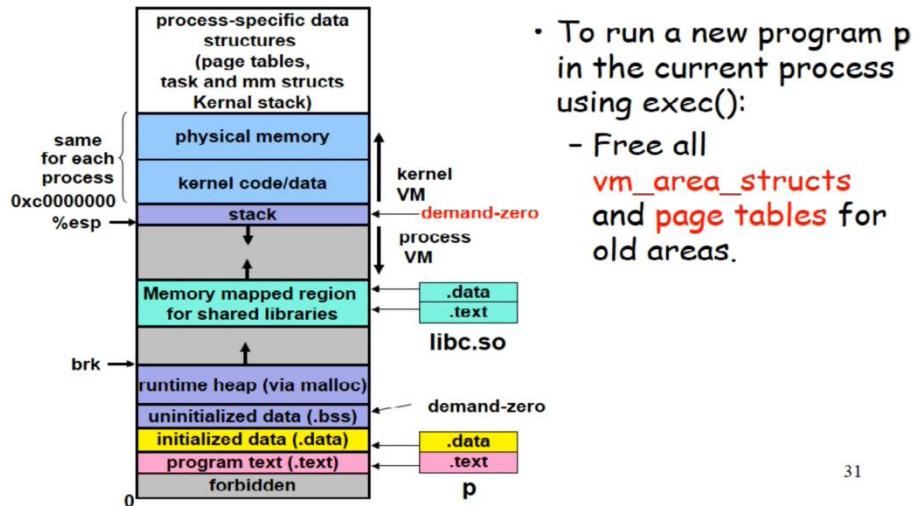
前面的这两个如果被写过了以后，就变成了 dirty page，把它淘汰出物理内存的时候，不能去修改原来的文件，要生成 swap file。所以一个 virtual memory array 是要和一个文件对应的，这叫做 memory map，它会和三种文件对应：1.regular file。2.匿名文件 3.swap file。

Demand Paging

- **Key point:** no virtual pages are copied into physical memory until they are referenced!
 - known as "**demand paging**"
 - crucial for time and space efficiency

真正用到的时候，才会把物理页放到内存中，把虚页和实页做一个对应。

Exec() revisited



Exec 它是在一个进程中被执行。

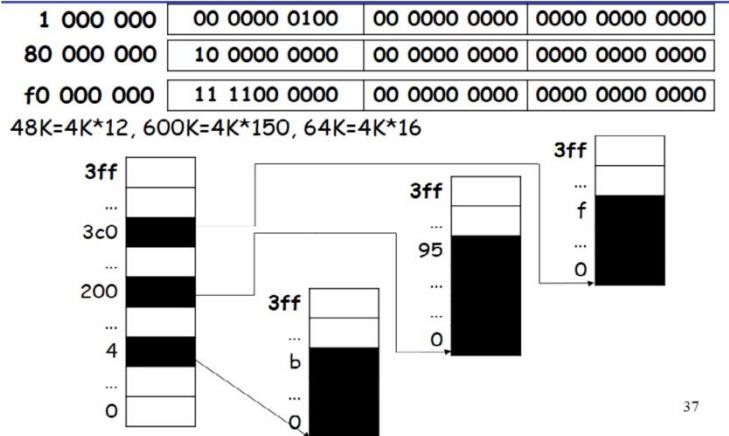
1. 把原来进程中的 vma 和 page table 全部清除。
2. Create 新的 vma 和 page tables。Stack, bss, data, text, shared libs。要做 map，把 text、data 和 ELF 做 map，bss、stack 和匿名文件做 map，shared lib 是共享 areas。
3. 把 entry point 赋给 PC。此时 virtual memory space 就准备好了，切换到这个进程的时候，第一次会报 page fault，第一页的指令就进去了。在执行过程中，就按需做 page，物理地址就一点点和虚地址对应上了。

Example

- A 32-bit machine with 2-level page table of 10-10-12 structure
- What is the skeleton of the page table for a process that has
 - a code segment of 48K starting at address 0x1000000,
 - a data segment of 600K starting at address 0x80000000
 - and a stack segment of 64K starting at address 0xf0000000 and growing upward (towards higher addresses)?

32 位的 2 级页表，分成 10-10-12 的结构。

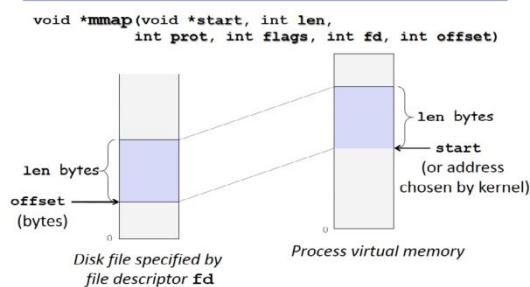
Example



37

我们一开始根据 ELF 的结构，决定了内存有多大。在执行的过程中，VMA 可能会发生变化，比如我们使用 `sbrk`，一开始堆的一个大小，映射一个匿名文件。我们发现堆不够的时候，使用 `sbrk` 调整大小后，相应的页表项也会发生变化。在执行的过程中，我们也可以在不能用的地址中，强行找一块。

User-level memory mapping



User-level memory mapping

```
void *mmap(void *start, int len, int prot,
           int flags, int fd, int offset)
```

Map `len` bytes starting at offset `offset` of the file specified by file description `fd`, preferably at address `start` (usually 0 for don't care).

- `prot`: `MAP_READ`, `MAP_WRITE`
- `flags`: `MAP_PRIVATE`, `MAP_SHARED`

Return a pointer to the mapped area

需要传入一些 area 的参数。

mmap() example: fast file copy

```
/*
 * mmapcopy - uses mmap to copy file fd to stdout
 */
void mmapcopy(int fd, int size)
{
    char *bufp;
    /* map the file to a new VM area */
    bufp = mmap(0, size, PROT_READ,
               MAP_PRIVATE, fd, 0);
    /* write the VM area to stdout */
    write(1, bufp, size);
    return ;
}
```

```
int main(int argc, char **argv)
{
    struct stat stat;
    /* check for required command line argument */
    if (argc != 2) {
        printf("usage: %s <filename>\n", argv[0]);
        exit(0);
    }
    /* open the file and get its size*/
    fd = open(argv[1], O_RDONLY, 0);
    fstat(fd, &stat);
    mmapcopy(fd, stat.st_size);
}
```

42

43

这样就把一个文件映射到了内存当中。真正使用的时候，要去判两个参数。我们把整个文件映射到内存里去，并且输出到屏幕上来。

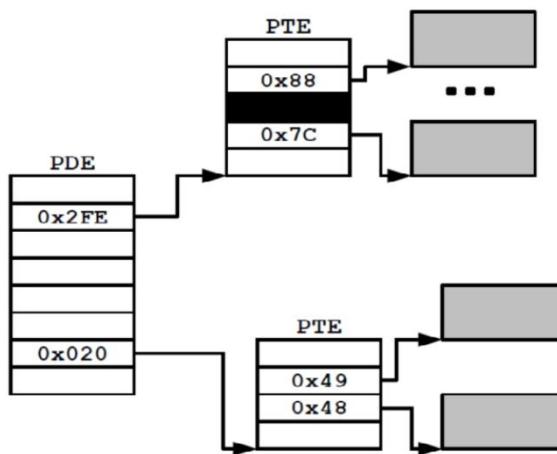
Example

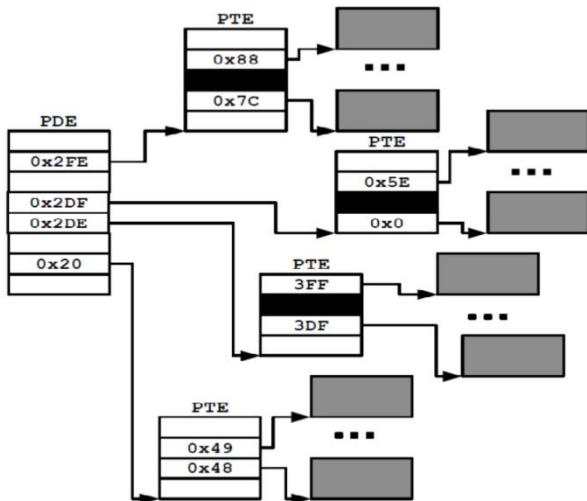
- 下列程序运行在Pentium/Linux存储系统下

```
#define PAGE_SIZE 4 * 1024
int main(void) {
    char *p = NULL;
    int i;
A:   p = mmap(0, 128 * PAGE_SIZE, PROT_READ |
               PROT_WRITE, MAP_PRIVATE | MAP_ANON, 0, 0);
    printf("buffer start: %p\n", p);
    for(i = 0; i < BUF_SIZE; i += PAGE_SIZE)
        p[i] = 1;
B:   munmap(p, BUF_SIZE);
    return;
}
```

- 当程序到达标号A时，页表如下。块中的数字表示PDE/PTE的偏移值。函数printf的输出值是“**buffer start: 0xb7bdf000**”。
- 请画出程序到达标号B时页表的形状。
 - 注意：没有数字的白色块意味着空的PDE/PTE，填写了数字的白色块表示PDE/PTE被使用，黑色块表示连续多个PDE/PTEs被使用，灰色块表示一个有数据/代码的页）

最初是这样的：

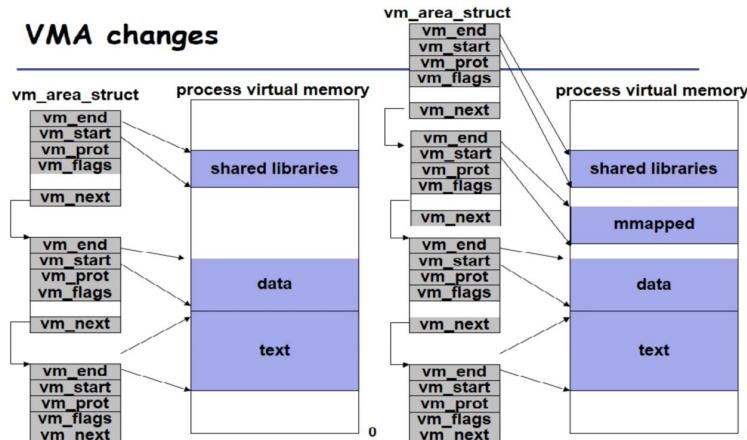




跨了两个页表，就会出两项

如果一个 array 的时候，需要用多个 2 级页表的时候，应该怎么变。

2021/6/2



上节课讲了 linux 中支持虚存的比较重要的数据结构 VMA，每个 array 包含一些信息。在程序的运行过程中，这个数据结构可能发生一些变化，比如通过 mmap 增加一块，或者通过堆的 sbrk 可以放大缩小，至少 vm_end 会发生变化。

Example

- A 32-bit machine with 2-level page table of 10-10-12 structure
- What is the skeleton of the page table for a process that has
 - a code segment of 48K starting at address 0x1000000,
 - a data segment of 600K starting at address 0x80000000
 - and a stack segment of 64K starting at address 0xf0000000 and growing upward (towards higher addresses)?

进程刚开始，如果这个数据结构确定了，我们的页表怎么构造出来呢？我们根据起始地

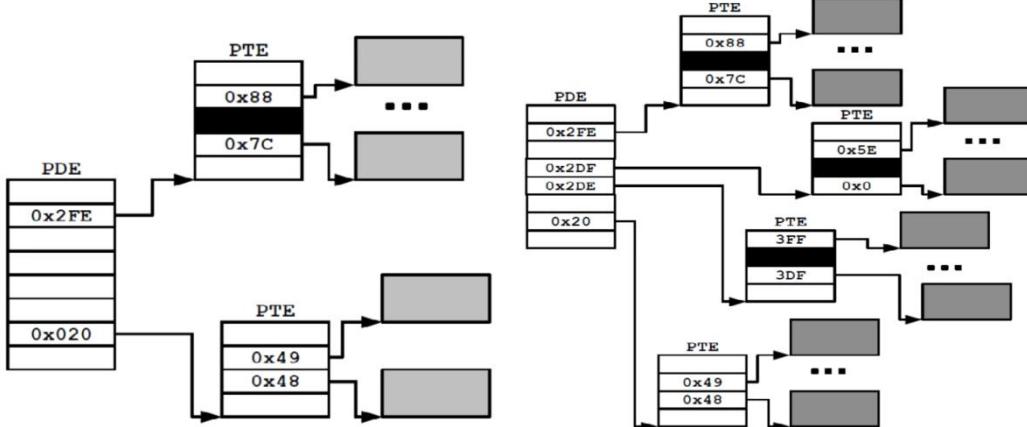
址和每个 array 的大小，可以构造出这样一个页表。

Example

- 下列程序运行在Pentium/Linux存储系统下

```
#define PAGE_SIZE 4 * 1024
int main(void) {
    char *p = NULL;
    int i;
A: p = mmap(0, 128 * PAGE_SIZE, PROT_READ |
           PROT_WRITE, MAP_PRIVATE | MAP_ANON, 0, 0);
    printf("buffer start: %p\n", p);
    for(i = 0; i < BUF_SIZE; i += PAGE_SIZE)
        p[i] = 1;
B: munmap(p, BUF_SIZE);
    return;
}
```

- 当程序到达标号 A 时，页表如下。块中的数字表示 PDE/PTE 的偏移值。函数 printf 的输出值是 “**buffer start: 0xb7bf000**”。
- 请画出程序到达标号 B 时页表的形状。
注意：没有数字的白色块意味着空的 PDE/PTE，填写了数字的白色块表示 PDE/PTE 被使用，黑色块表示连续多个 PDE/PTEs 被使用，灰色块表示一个有数据/代码的页）
- 0xb7bf000 1011011110 1111011111 0000000000000000
2de 3df



下边我们来看这个例子，进程本身有这个页表。在运行过程中，我们用 mmap 增加了 128 个 page，它是 private+可读可写的，我们使用匿名文件，说明这一段虚拟内存初始的时候都是 0。我们给每一个 page 写一个值，访问一次。

我们把这个 16 进制变成 2 进制，也就是我们把 32 位分成 10+10+12, 2de 就是一级页表当中的下标，而 3df 是二级页表中的下标。我们从 3de 开始放 128 个 page 在一个二级目录里放不下，二级页表最高是 3FF。3FF 到 3DE 是 0x21=33 个 page，我们总共是 128 个 page，还有 95 个页面，我们需要再来一个二级页面，所以我们从 2DE 之后加一个页面到 2DF，这样再放 95 个。

Fork() revisited

- To create a new process using **fork**:
 - make copies of the old process's **mm_struct**, **vm_area_struct**, and **page tables**
 - Two processes are sharing all of their pages (At this point)
 - How to get separate spaces without copying all the virtual pages from one space to another?
 - “**Copy On Write**” (COW) technique.

Copy-On-Write (写时复制)

第二部分是另一个概念，copy on write。我们要看这个 fork，fork 这件事情是说这边有一个进程 A，它在某一点执行 fork 的时候，它会执行子进程 c。在 fork 的这一点上，除了进

程号，子进程和父进程长的一模一样。因为它们都一样，我们能不能不拷贝，比如代码是 **read-only** 的，我们能可以直接 **share**。但是对于数据段（**data** 段），在这个点虽然是完全相同的，两个进程在这个点之后应该各自修改自己的 **data**，如果父进程 **a** 和子进程 **c** 都去修改同一个数据段，那么就有问题了。

所以这个技术 **copy on write**，写的时候就要 **copy** 了。OS 写的时候是按照 **page** 为单位来控制的，一个 **array** 控制好几个页。当我要写 **data** 段的时候，我就要从 **a** 拷贝一份到 **c** 来。

Shared Object

- Shared Object

- An object which is mapped into an area of virtual memory of a process
- Any writes that the process makes to that area are visible to any other processes that have also mapped the shared object into their virtual memory
- The changes are also reflected in the original object on disk

- Shared Area

- A virtual memory area that a shared object is mapped

要介绍写时复制，我们还要搞清楚 **shared object**。**Shared object** 相当于一个 **file** 或者一个 **file** 的一部分，也就是 **vm** 中的内容是需要一个文件来支撑的。可以映射到多个进程的不同虚地址上，这在 **shared library** 是很常见的。我们还要求一个进程对虚地址上的 **share array** 上的写，要能被 **share** 的其他进程看到，并且写要能反映到对应的文件上。

Copy-on-Write

- A private object begins life in exactly the same way as a shared object, with only one copy of the private object stored in physical memory.

另一种就是 **private** 状态。第三种状态就是 **copy on write** 状态。在 **fork** 之前，有很多 **private** 的 **object**。一开始我们对 **read-only** 物理页可以映射到子进程的虚地址中，就不需要 **copy** 了。我们这么区分写的怎么 **copy**，我们把 **array** 写成 **copy on write object**，我们把它称作 **copy on write object**。在 **vma** 里头有 **flags**，我们除了 **share** 和 **private**，增加一个 **copy-on-write** 状态。

Copy-on-Write

- For each process that maps the private object
 - The page table entries for the corresponding private area are flagged as **read-only**
 - The area struct is flagged as **private copy-on-write**
 - So long as neither process attempts to write to its respective private area, they continue to share a single copy of the object in physical memory.

Copy-on-Write

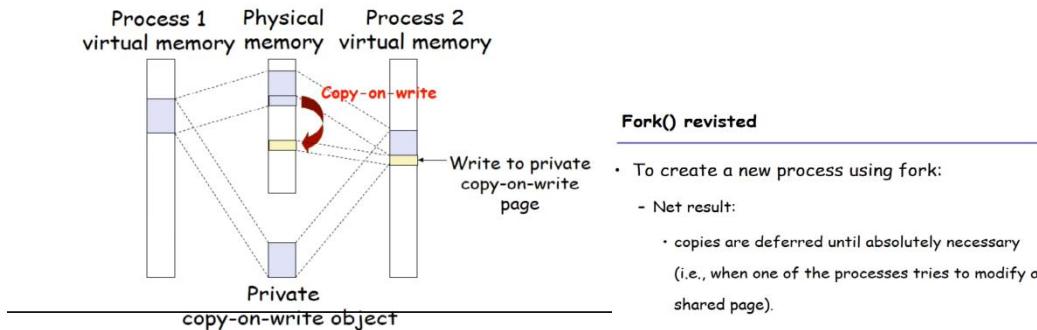
- For each process that maps the private object
 - The **fault handler**
 - Creates a new copy of the page in physical memory
 - Updates the page table entry to point to the new copy
 - Restores write permissions to the page

我们在页表上置的是 **read-only**，在 **VMA** 中置的是 **private copy-on-write**

一旦发生写了，因为我们写的这个页在 **MMU** 在管，一旦我们写了就会发生出 **fault**，**mmu** 一看是 **read-only**，**mmu** 就会抛一个 **exception** 出来，就会进入 **fault handler**，OS 接管拿这个地址去查 **VMA**，看到 **flags** 是 **private copy on write**，它就新做一个部件出来，把原来的部分 **copy** 出来，在新的页当中去写。所以在写的时候我们才去做 **copy**，主要提高的是 **fork**

的效率，不要把所有的东西都直接在 `fork` 的时候 `copy`，这个操作把必须要做的操作推迟到写的时候再进行。

Sharing Revisited: Private COW Objects



我们这本书还有一点，指针和垃圾回收。我们今天先把新的那本书的一点东西讲掉。我们在系统软件已经上过这本书了。

Operating System Three Easy Pieces

最优替换策略

和我们虚拟内存相关的，在那本书上是替换策略（replacement policy）。ICS 基本上覆盖了这本书的前两个 pieces，比如虚存和锁、信号量的实现。

The Optimal Replacement Policy

- The best possible replacement policy**
 - developed by Belady many years ago
 - replacing the page that will be accessed furthest in the future
 - hard to implement
 - incredibly useful as a comparison point in simulation or other studies

最优替换策略：实际上是实现不了的，为什么我们讲呢？因为这是任何一个实际替换策略的上限，它是拿来作比较用的。我们知道上限以后就可以来看实际的和它差了多少。

Example

- Trace: 0, 1, 2, 0, 1, 3, 0, 3, 1, 2, 1

Access	Hit/Miss?	Evict	Resulting Cache State
0	Miss		0
1	Miss		0, 1
2	Miss		0, 1, 2
0	Hit		0, 1, 2
1	Hit		0, 1, 2
3	Miss	2	0, 1, 3
0	Hit		0, 1, 3
3	Hit		0, 1, 3
1	Hit		0, 1, 3
2	Miss	3	0, 1, 2
1	Hit		0, 1, 2

我们只有 3 个物理页，012 先是 3 个 miss，然后和物理页对应上。接下来访问 3 的时候，我们要先淘汰掉 2，多出一个物理页给 3，我们从 3 开始往前看访问序列，我们发现 2 是最

后被访问，所以我们淘汰掉最晚被访问掉的那个。

我们实际上不能预知未来得到这个 best 的策略，我们只能看历史数据来预测将来会怎么样。

1. 先进先出策略

First-In, First-Out Replacement Policy

- Hit rate = $4/11 = 36.4\%$

Access	Hit/Miss?	Evict	Resulting Cache State
0	Miss		First-in → 0
1	Miss		First-in → 0, 1
2	Miss		First-in → 0, 1, 2
0	Hit		First-in → 0, 1, 2
1	Hit		First-in → 0, 1, 2
3	Miss	0	First-in → 1, 2, 3
0	Miss	1	First-in → 2, 3, 0
3	Hit		First-in → 2, 3, 0
1	Miss	2	First-in → 3, 0, 1
2	Miss	3	First-in → 0, 1, 2
1	Hit		First-in → 0, 1, 2

这个策略命中率比较低，还有一个现象 Belady's Anomaly，所谓 Belady 现象是指：采用 FIFO 算法时，如果对一个进程未分配它所要求的全部页面，有时就会出现分配的页面数增多但缺页率反而提高的异常现象。

Belady's Anomaly

- The reference trace is 0,1,2,3,0,1,4,0,1,2,3,4

Time	1	2	3	4	5	6	7	8	9	10	11	12	
reference string	0	1	2	3	0	1	4	0	1	2	3	4	
primary device	-	0	0	0	1	2	3	0	0	0	1	4	
contents	-	-	1	1	2	3	0	1	1	1	4	2	pages brought in
remove	-	-	-	2	3	0	1	4	4	4	2	3	
bring in	0	1	2	3	0	1	4	-	-	2	3	-	9

Belady's Anomaly

- The reference trace is 0,1,2,3,0,1,4,0,1,2,3,4

Time	1	2	3	4	5	6	7	8	9	10	11	12	
reference string	0	1	2	3	0	1	4	0	1	2	3	4	
primary device	-	0	0	0	0	0	0	0	1	2	3	4	0
contents	-	-	1	1	2	3	0	1	1	1	2	3	1
remove	-	-	-	2	3	0	1	4	4	4	0	1	2
bring in	0	1	2	3	0	1	4	-	-	2	3	4	10

因为淘汰策略的问题，导致 cache 变大，命中率反而降低了。

2. 随机选择策略

Random Replacement Policy

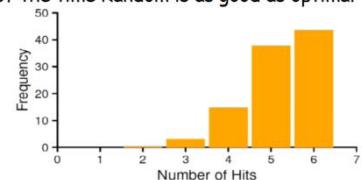
- Hit rate = $5/11 = 45.5\%$

Access	Hit/Miss?	Evict	Resulting Cache State
0	Miss		0
1	Miss		0, 1
2	Miss		0, 1, 2
0	Hit		0, 1, 2
1	Hit		0, 1, 2
3	Miss	0	1, 2, 3
0	Miss	1	2, 3, 0
3	Hit		2, 3, 0
1	Miss	3	2, 0, 1
2	Hit		2, 0, 1
1	Hit		2, 0, 1

Random Replacement Policy

- How many hits Random achieves

- over 10,000 trials
- each with a different random seed
- over 40% of the time Random is as good as optimal



大概率会比 FIFO 好。

3. LRU 替换策略

LRU Replacement Policy

- Lean on the past and use *history* as our guide
 - Frequency
 - if a page has been accessed many times, perhaps it should not be replaced as it clearly has some value
 - Least-Frequently-Used (LFU)
 - Replace the least-frequently used page
 - Recency of access
 - the more recently a page has been accessed, perhaps the more likely it will be accessed again
 - Least-Recently-Used (LRU)
 - Replace the least-recently-used page

LFS 是淘汰掉使用频率最低的。

LRU 的 Stack 算法

Stack Algorithms

- Subset property
 - For the optimal policy, at all times
 - the pages it keeps in the 3-page memory
 - is a subset of that it keeps in the 4-page memory
- No Belady's anomaly if subset property holds
 - At all times and
 - For every possible capacity of primary device
 - It creates a total ordering for pages at a given time

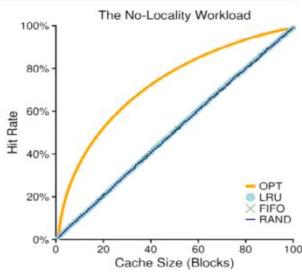
Stack Algorithms for LRU

Time	1	2	3	4	5	6	7	8	9	10	11	12
reference string	0	1	2	3	0	1	4	0	1	2	3	4
stack contents after reference	0	1	2	3	0	1	4	0	1	2	3	4
number of moves in	-	-	-	-	-	-	-	-	-	-	-	-
size 1 in/out	0/-	1/0	2/1	3/2	0/3	1/0	4/1	0/4	1/0	2/1	3/2	4/3
size 2 in/out	0/-	1/-	2/0	3/1	0/2	1/3	4/0	0/1	1/4	2/0	3/1	4/2
size 3 in/out	0/-	1/-	2/-	3/0	0/1	1/2	4/3	-/	-/	2/4	3/0	4/1
size 4 in/out	0/-	1/-	2/-	3/-	-/	-/	4/2	-/	-/	2/3	3/4	4/0
size 5 in/out	0/-	1/-	2/-	3/-	-/	-/	4/2	-/	-/	-/	-/	5

k 一定是 k+1 的一个子集，序列 12、12、10、8、5 一定是单调不增的。

Random Access Workload

- the workload
 - accesses 100 unique pages over time
 - choosing the next page to refer to at random
 - overall 10,000 pages are accessed



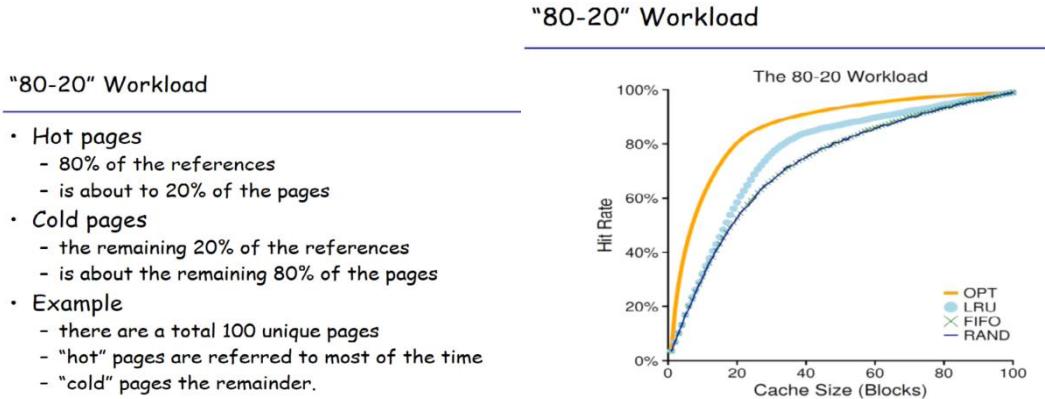
Random Access Workload

- It doesn't matter much which realistic policy you are using
 - LRU, FIFO, and Random all perform the same
 - the hit rate exactly determined by the size of the cache
- When the cache is large enough to fit the entire workload
 - it also doesn't matter which policy you use
 - all policies (even Random) converge to a 100% hit rate when all the referenced blocks fit in cache

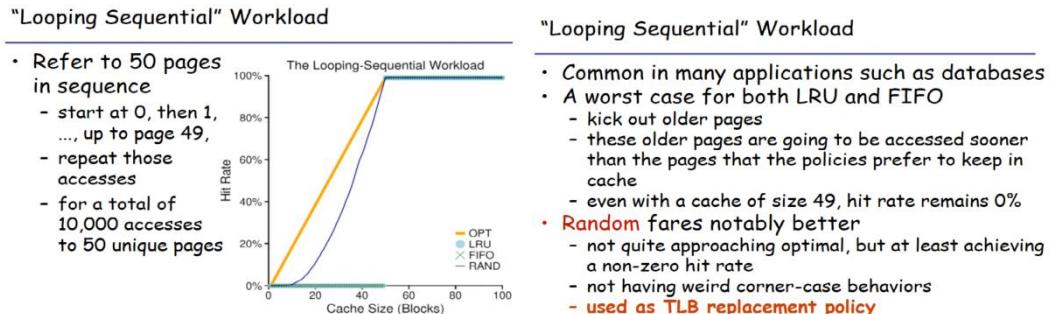
我们已经介绍了四种算法，我们看看这几种淘汰策略在 100 个虚拟页上、随机数据访问情况下。物理页超过 100 页是没什么区别的，除了第一次会 miss。所以每种淘汰策略在物理页

为 100 页时，命中率接近 100%。因为 optimal 可以偷看到未来，所以比起所有策略都比较好。

但是这没什么意义，因为所有程序都是符合局部性原理的。



在 80-20 workload 中，20 个页是经常被访问的（占 80% 的访问），叫做 hot page，还有 80 个是 cold page。此时我们就发现所有策略都比随机策略要好，明显的是 LRU 胜出了。



那么随机策略有什么用呢？我们看上图这个反复访问 50 个页面，当物理页面超过 50 的时候，都是 100%，但是哪怕在 49 个物理页的时候，FIFO 和 LRU 都很糟糕，因为他们总是把即将访问的页面淘汰出去。所以在 TLB 中我们就使用随机的替换策略。

How to implement LRU?

- Hardware should put a time stamp on each physical page whenever it is accessed
- When evicting, it should sort the pages by their time stamps
- It may be too many pages to be sorted
 - One million pages for 4GB memory with 4KB page-size

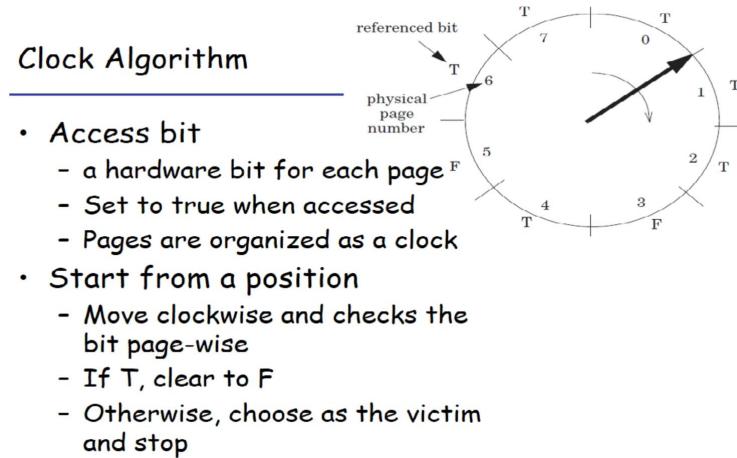
LRU 是否真的可实现呢？我们需要知道哪个虚页是最近被访问的。

2021/6/9

上节课讲了 LRU，在大部分情况下是比较好的淘汰策略，那么怎么实现 LRU 呢？访问页面的顺序应该是虚页，因为虚页才会有对应的物理页不够，才会被淘汰出去，原来的虚页 VP 和 PP 有对应关系，淘汰完就是 VP 对应一个空。我们需要淘汰的时候，比如要访问 VP2，产生了一个缺页异常，OS 要从 PP1 到 PPN 中选一个，把对应关系去掉，同时把 swap 文件写进硬盘，然后再把 VP2 对应上。这时候就要求我们的物理页上有一个时间戳（最近一次访问是什么时候）。需要淘汰的时候，我们需要根据时间戳来排序，最早访问的物理页的对应关系就被淘汰掉。

如果我是 4G 的内存，一个页 4K 的话，那么我们有一百万个物理页，每次 page fault 的时候，需要对一百万个元素进行排序。为了实现 LRU，硬件要来打一个时间戳，软件要来排序，但是这两个代价太大了。在真正地在 OS 中，就没有严格意义上去实现 LRU。那么我们用的是近似的时钟算法。

时钟算法（LRU 的近似实现）

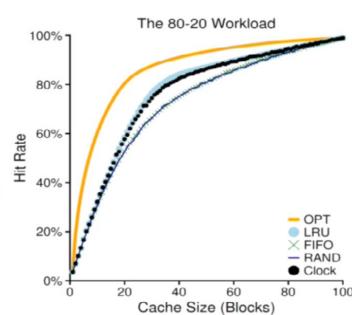


我们把页做成一个环形的页表，每个页上有一个标记着访问与否的 access bit。发生 page fault 的时候，从上一次开始的地方顺时针扫描，看到 true 代表从上次到现在这个时间段里头被访问过的，它一边扫描会把 true 变成 false，扫描到有 false 的地方，我们认为这就是最近一次没有被访问过的页，我们就停在这个地方，下次就从这个地方开始。

对于上节课说的 80-20 workload，我们有一百个虚页，随机访问一万人次，每个平均访问 100 次，在 80-20 workload 下，有 20% 的页在被访问了 8000 次 (hot page)，剩下 80% 的页被访问了 2000 次 (cold page)。这种情况下 LRU 执行的是比较好的。

Clock Algorithm

- It doesn't do quite as well as perfect LRU
- It does better than approaches that don't consider history at all
- It can use dirty bit to reduce disk writing further



现在我们用了时钟近似算法，发现曲线是基本上吻合的。这时候我们讲的淘汰策略有这几个，最优的（黄色的）是不存在的。

Page Selection Policies

- When to bring pages into memory?
 - Demand paging
 - Prefetching
 - the OS could
 - guess a page is about to be used
 - and bring it in ahead of time
 - Clustering (grouping)
 - Set two numbers as low/high water mark
 - when available page number is less than the low water mark,
 - OS begins to evict pages
 - until the number of available pages reaches the high water mark

Prefetching, OS 猜测下一次会取哪个页，要的时候已经取好了，猜错就不太合算了。用的比较多的是 clustering，在实现的时候设一个 high/low water mark，这个就是说比如我们有 50 万 high water 10 万 low water mark，当我自由的物理页低于 low water mark，我们再来做页面的淘汰，一淘汰就淘汰掉一批（和虚页的关系断开），直到自由的物理页（没有对于虚拟页的物理页）的数量变成 high water mark。

Control Thrashing

- admission control
 - not to run some processes
 - hope to reduce working sets (the pages that they are using actively) to fit in memory
- out-of-memory killer
 - A daemon
 - choose a memory intensive process and kill it
 - thus reducing memory
 - it kills the X server and thus renders any applications requiring the display unusable

因为我们进程特别多，每个进程都需要有物理页，总是缺页一直在发生淘汰，总是缺页需要我们想一点办法，比如有些进程把它停下来不跑了，这些进程停了以后占用的物理页释放出来，我们希望剩下的进程这些物理页够它们使用。我们有一个 out-of-memory killer，这是一个后台进程，我们有一些耗费很多内存的进程，out-of-memory killer 会把它选出来杀死。

Level 1, Level 2 and Level 3 Page Table Entry

63	62	52	51	12	11	9	8	7	6	5	4	3	2	1	0				
XD	Unused	Page table physical base addr				Unused	G	PS	A	CD	WT	U/S	R/W	P=1					
Available for OS (page table location in secondary storage)														P=0					
XD	Disable or enable instruction fetches																		
Base addr	40 most significant bits of base address of child page table (forces page tables to be 4KB aligned)																		
G	global page (don't evict from TLB on task switch)																		
PS	Page size either 4K or 2M or 1G(may set to 1 only for Level 2 or level 3 PTEs)																		
A	Reference bit (set by MMU on reads and writes, cleared by software)																		
CD	Cache disabled(1) or enabled(0) for child page table																		
WT	Write-through or write-back cache policy																		
U/S	User or supervisor(kernel) mode access permission																		
R/W	Read-only or read-write access permission																		
P	Child page table present in memory(1) or not(0)														27				

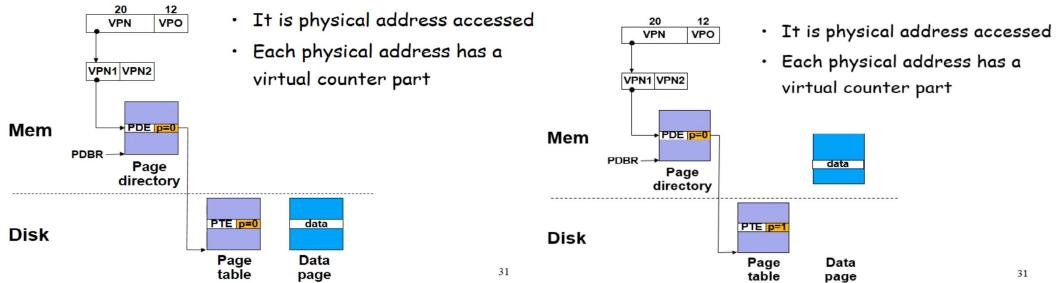
我们继续回到时钟算法，每个页面需要打一个 access bit，每次去访问页面的时候才会打上这个 bit，这个 bit 我们有没有呢，我们再来复习一下页表。页表上有一项为 A bit，这个 bit 我们称为 reference bit，这个页每次去读和写的时候 MMU 就会置 1，软件来清零，这个位就是来给我们做时钟算法的，OS 在淘汰的时候看到 true 就清成 false，看到是 false 就

把这个对应的页淘汰掉。页表项每次访问的时候就会把它置上，缺页的时候 OS 就会访问这些位来做淘汰。

时钟算法还有一个页表，页表是对应虚拟页的，访问的 A bit 是置到虚拟页上的，淘汰的时候应该是淘汰物理页。我们要找到物理页到对应的虚拟页的映射，物理页对应的虚拟页可能有多个，要去看看对应的多个虚拟页上有没有 A bit，如果有说明它已经被访问过一次了。

最后还有一个问题，DMA 的时候是把硬盘上的一块东西搬到内存里去，MMU 是不会置 A bit 的，因为是 disk 直接搬运的，这时候我们怎么去设置 A bit 呢，OS 会对应的置上 A bit。

How to Evict Page Table



最后再提一下 32 位中用的很多的，页表也有可能被淘汰出去。我们发现二级页表不在内存里。我们希望淘汰的时候不要出现右图的情况，如果要淘汰就是左图的情况（包括数据一起淘汰掉）。一级页表一定要在内存里。

我们再回到这本书，第九章还有一些东西。

Outline

- Garbage Collection
- Common Memory-Related Bugs in C Programs
- Suggested reading: 9.10, 9.11

垃圾回收

垃圾回收，很多语言里头都会有应用程序动态申请空间，申请完的空间要不要释放，不同的语言有不同的选择，C 和 C++ 中要求程序员自己回收掉。

Implicit Memory Management

- *Garbage collection*
 - automatic reclamation of heap-allocated storage
 - application never has to free

```
void foo() {  
    int *p = malloc(128);  
    return; /* p block is now garbage */  
}
```

如果只有这一段代码，空间被占用了但是没有人能用，我们就称为这是一个垃圾，内存

就被浪费掉了，如果这段程序被反复调用，就会产生内存泄露。内存泄露以后，程序最终就被报 out of memory。

还有一些语言，比如 java 或者一些 functional language，用到的就是垃圾回收机制。用户只管申请，系统会来负责回收用户产生的垃圾。

Garbage Collection

- How does the memory manager know when memory can be freed?
 - In general we cannot know what is going to be used in the future since it depends on conditionals
 - But we can tell that certain blocks cannot be used if there are no pointers to them

我们难以预测程序的行为来回答这块内存将来会不会被用到。但是我们可以根据是否有指向这一块的指针，我们要确认这个东西是一个垃圾。

垃圾回收的三个条件：

1. 我们首先要能区分出指针和非指针；
2. 所有的指针都指向块的起始地方；
3. 指针不能是一个隐藏指针(C 和 C++可以把指针藏起来，也就是指针可以赋值给 Long 强制类型转换，这时候我们没有办法区分指针和非指针)。

Example

```
typedef struct list_ {
    struct list_ *link;
    int key;
} *list;

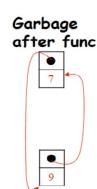
typedef struct tree_ {
    int key;
    struct tree_ *left;
    struct tree_ *right;
} *tree;

tree maketree();
void showtree();
```

第一个是单向链表，第二个是一个二叉树。

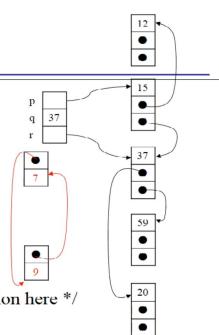
Example

```
void func()
{
    list x = malloc(sizeof(struct list_));
    list y = malloc(sizeof(struct list_));
    x->link = y; x->key = 7;
    y->link = x; y->key = 9;
}
```



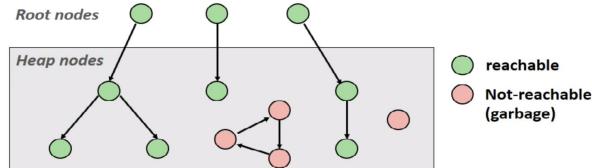
Example

```
main()
{
    tree p, r;
    int q;
    func();
    p = maketree();
    r = p.right;
    q = r.key;
    showtree(r); /* garbage collection here */
}
```



调用 `func` 就产生了一个垃圾。比如说最后一句发现内存不够了，我们需要调用垃圾回收机制来会受到一些内存。我们怎么判断出垃圾呢，因为这里我们有 `p` 和 `r` 这两个指针，从这两个指针出发可以访问到这棵二叉树的节点。而我们不知道这个链表的地址，访问不到，我们就认为是垃圾回收掉。

Memory as a Graph



A node (block) is **reachable** if there is a path from any root to that node.

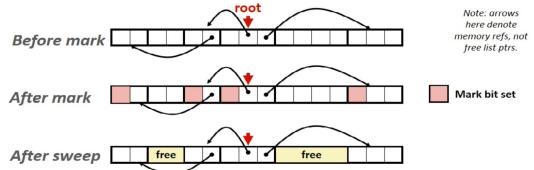
Non-reachable nodes are **garbage** (cannot be needed by the application)

我们就得到了一张图，有点有边，从任何一个 `root` 出发，我们看一下可达性。不可达的点就属于垃圾。

Mark and Sweep Collecting

- Can build on top of malloc/free package
 - Allocate using `malloc` until you "run out of space"
- When out of space:
 - Use extra **mark bit** in the head of each block
 - Mark:** Start at roots and set mark bit on each reachable block
 - Sweep:** Scan all blocks and free blocks that are not marked

Mark and Sweep Collecting



我们用的最简单的方法就叫做 `mark and sweep` 收集。下学期我们学编译，对于垃圾回收就会有更多的算法。我们从根节点出发能够达到的这些点，我们置一个 `mark bit` (变成绿色的)，再去扫描 `heap node`，如果被标记成绿色就留着，剩下哪些红色的就可以回收掉。

Assumptions For a Simple Implementation

- Application
 - `new(n)`: returns pointer to new block with all locations cleared
 - `read(b, i)`: read location `i` of block `b` into register
 - `write(b, i, v)`: write `v` into location `i` of block `b`
- Each block will have a header word
 - addressed as `b[-1]`, for a block `b`
 - Used for different purposes in different collectors

Mark and Sweep (cont.)

- Mark using depth-first traversal of the memory graph

```
ptr mark(ptr p)
{
    if (!is_ptr(p)) return; // do nothing if not pointer
    if (markBitSet(p)) return; // check if already marked
    setMarkBit(p);
    for (i=0; i < length(p); i++)
        mark(p[i]); // call mark on all words
    return;
}
```

Mark and Sweep (cont.)

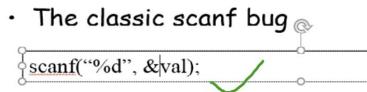
- Sweep using lengths to find next block

```
ptr sweep(ptr p, ptr end)
{
    while (p < end) {
        if (markBitSet(p))
            clearMarkBit();
        else if (allocateBitSet(p))
            free(p);
        p += length(p);
    }
}
```

书上写了怎么判断哪些东西是 `heap node`，有些基本的函数。`Mark` 就是是不是一个

pointer。如果节点被置上了就回去，否则就把它置上，从这个结点出发继续往下走，这就是一个 dfs。对于 sweep，如果置上了就 free 掉。

书上还有一个是常见的内存相关的 bug。‘



如果不加地址是没有用的。

Reading uninitialized memory

• Assuming that heap data is initialized to zero

```
/* return y - Ax */
int *matvec(int **A, int *x)
{
    int *y = malloc(N*sizeof(int));
    int i, j;

    for (i=0; i<N; i++)
        for (j=0; j<N; j++)
            y[i] += A[i][j]*x[j];
    return y;
}
```

19

malloc 是本身不带初值的，y 这个向量没有初始化。

Overwriting memory

• Allocating the (possibly) wrong sized object

```
int i, **p;

p = (int **)malloc(N*sizeof(int));

for (i=0; i<N; i++) {
    p[i] = malloc(M*sizeof(int));
}
```

申请 N 个整数，这在 32 位上是成立的，地址长度等于 int 长度，但是在 64 位就不成立了。这就越界了，会把 memory manager 里的一些信息破坏掉。

• Stack Buffer Overflow

```
void bufoverflow()
{
    char buf[64];

    gets(buf); /* Here is the stack buffer overflow bug */
    return;
}
```

它不会去判断是否越界，因为 64 没有作为参数传进去。

Overwriting memory

- Referencing a pointer instead of the object it points to

```
int *binheapDelete(int **binheap, int *size)
{
    int *packet;
    packet = binheap[0];
    binheap[0] = binheap[*size - 1];
    *size--;
    heapify(binheap, *size, 0);
    return(packet);
}
```

23

Size 是一个指针，想把 size 的值减一。如果 15 级优先级比较清楚的话，*size--代表的是*(size--) 而不是我们期望的(*size)--

考试的话 这部分不考，这部分之前不考

- Garbage Collection
- Common Memory-Related Bugs in C Programs
- Suggested reading: 9.10, 9.11

从 6.2, 7, 9, 9 的话是 9.1 到 9.9

还有一个补充的 replacement policy。这个是我们要考的，范围就是这个样子。期末的内容不是特别多。去除 7 的考点就是 memory 相关的。