# 深入理解计算机系统

## 一．计算机系统漫游

1. 系统的硬件组成：总线、I/O 设备、主存、处理器
2. 程序的编译：.c -> .i -> .s -> .o -> 可执行文件
3. 抽象（惨痛教训）：

（1）文件：I/O 设备的抽象
（2）虚拟内存：程序存储器的抽象
（3）进程：一个正在运行的程序的抽象
（4）虚拟机：对整个计算机的抽象

# Ⅰ. 程序结构和执行

## 一．信息的表示和处理

1. 十六进制表示法：一般左面不会再填 0
2. 字数据大小：

| C 声明 | | 字节数 | |
|---|---|---|---|
| 有符号 | 无符号 | 32 位 | 64 位 |
| char | unsigned char | 1 | 1 |
| short | unsigned short | 2 | 2 |
| int | unsigned int | 4 | 4 |
| long | unsigned long | 4 | 8 |
| float | | 4 | 4 |
| double | | 8 | 8 |
| 指针 | | 4 | 8 |

3. x = 0x12345678
   小端序（常用）：78 56 34 12
   大端序：12 34 56 78
4. 位级运算：&、|、~、^
5. 逻辑运算：&&、||、!
6. 移位运算：

（1）逻辑移位：右移、左移均填 0
（2）算术移位：右移填最高位，左移填 0

7. 整数表示

（1）有符号数、无符号数
（2）补码：反码 + 1

8. 整数运算

（1）加法：溢出取模
（2）乘法：溢出模 2，截断后与原码运算结果一致
（3）除法：向 0 舍入，补码可加偏量 2^k - 1

9. IEEE 浮点表示：符号（s：1 位）、阶码（E：8 位）、尾数（M：23 位）

| | | | |
|---|---|---|---|
| 规格化 | s | e != 0 && e != 255 | frac |
| 非规格化 | s | e = 0 | frac |
| 无穷 | s | e = 255 | frac = 0 |
| NaN | s | e = 255 | frac != 0 |

10. 舍入：向上、向下、向 0、向偶（最贴近平均水平）
11. 浮点运算：IEEE 指定运算规则，将浮点数视为实数，带舍入

## 二．程序的机器级表示

1. 数据格式：字节（b）、字（w）、双字（l）、四字（q）
2. 寄存器

| 63:0 | 31:0 | 15:0 | 7:0 | 寄存器 |
|---|---|---|---|---|
| %rax | %eax | %ax | %al | 返回值 |
| %rbx | %ebx | %bx | %bl | 被调用者保存 |
| %rcx | %ecx | %cx | %cl | 第 4 个参数 |
| %rdx | %edx | %dx | %dl | 第 3 个参数 |
| %rsi | %esi | %si | %sil | 第 2 个参数 |
| %rdi | %edi | %di | %dil | 第 1 个参数 |
| %rbp | %ebp | %bp | %bpl | 被调用者保存 |
| %rsp | %esp | %sp | %spl | 栈指针 |
| %r8 | %r8d | %r8w | %r8b | 第 5 个参数 |
| %r9 | %r9d | %r9w | %r9b | 第 6 个参数 |
| %r10 | %r10d | %r10w | %r10b | 调用者保存 |
| %r11 | %r11d | %r11w | %r11b | 调用者保存 |
| %r12 | %r12d | %r12w | %r12b | 被调用者保存 |
| %r13 | %r13d | %r13w | %r13b | 被调用者保存 |
| %r14 | %r14d | %r14w | %r14b | 被调用者保存 |
| %r15 | %r15d | %r15w | %r15b | 被调用者保存 |

3. 操作数指示符

（1）立即数：$Imm

（2）寄存器：R[ra]

（3）内存引用：Mb[addr]

| | | |
|---|---|---|
| 存储器 | (ra) | M[R[ra]] |
| 存储器 | Imm(rb) | M[Imm + R[rb]] |
| 存储器 | (rb, ri) | M[R[rb] + R[ri]] |
| 存储器 | Imm(rb, ri) | M[Imm + R[rb] + R[ri]] |
| 存储器 | (, ri, s) | M[R[ri] * s] |
| 存储器 | Imm(, ri, s) | M[Imm + R[ri] * s] |
| 存储器 | (rb, ri, s) | M[R[rb] + R[ri] * s] |
| 存储器 | Imm(rb, ri, s) | M[Imm + R[rb] + R[ri] * s] |

4. 数据传送

（1）movx　a, b　将 a 赋给 b

（2）movzx　a, b　将做了零扩展的 a 赋给 b（传 unsigned 会用到）

（3）movsx　a, b　将做了符号扩展的 a 赋给 b

5. 压入和弹出栈数据

（1）pushq 等价于 %rsp 减 8 后将四字值压入

（2）popq 等价于将四字值传送到 %rax 后 %rsp 加 8

6. 算数和逻辑操作

（1）leaq　a, b　加载有效地址，将&S 赋给 D，通常用来执行简单算术操作

（2）一元操作：INC、DEC、NEG、NOT

（3）二元操作：ADD、SUB、IMUL、XOR、OR、AND（b 为操作数）

（4）移位：SAL（SHL）、SAR（算数）、SHR（逻辑）

7. 条件码

（1）CF 进位、ZF 零、SF 负数、OF 溢出

（2）访问 set：

e 相等（零）、ne 不等（非零）、s 负数、ns 非负、g 大于（有符号）、l 小于（有符号）、a 超过（无符号）、b 低于（无符号）

8. 条件分支

（1）跳转：jmp 直接跳转，否则 j 后跟条件码，可实现条件分支

（2）传送：cmovx　a, b（b 为操作数）

（3）分支预测：错误处罚 Tmp = 2*(Tran - Tok)、预测错误所需时间：Tok + Tmp

9. 循环分支

（1）do while：loop 中增加 jxx　loop

（2）while：jump to middle（循环部分放在中间）、guard to（根据条件选择分支）

（3）for：可翻译成 while，continue 用 go to 解决

10. 开关语句：使用跳转表，参数出现顺序与表中相同

11. 过程

（1）转移控制：callq + <函数名>

（2）数据传送：寄存器不足、存在&符号、变量为数组或结构时可用栈传送

（3）寄存器中的局部存储空间：使用被调用者保存寄存器防止覆盖

12. 数组分配和访问

（1）指针运算：E in %rdx，i in %rcx

| 表达式 | 类型 | 汇编代码 |
|---|---|---|
| E | int * | movq　%rdx, %rax |
| E[0] | int | movl　(%rdx), %rax |
| E[i] | int | movl　(%rdx, %rcx, 4), %rax |
| &E[2] | int * | leaq　8(%rdx), %rax |
| E + i - 1 | int * | leaq　-4(%rdx, %rcx, 4), %rax |
| *(E + i - 3) | int | movl　-12(%rdx, %rcx, 4), %rax |
| &E[i] - E | long | movq　%rcx, %rax |

（2）嵌套数组：&D[i][j] = xD + L(C * i + j)

13. 异质数据结构

（1）结构：struct 中每个字段具有不同的偏移量，从 0 开始

（2）联合：防止浪费字节

```
/**原始版本：32个字节**/
struct node_s {
    struct node_s *left;
    struct node_s *right;
    double data[2];
};
```

```
/**联合版本：16个字节**/
union node_u {
    struct {
        union node_u *left;
        union node_u *right;
    }internal;
    double data[2];
};
```

```
/**升级版本：24个字节**/
typedef enum { N_LEAF, NINTERNAL } nodetype_t;

struct node_t {
    nodetype_t type;//需要填充4个字节
    union node_u {
        struct {
            union node_u *left;
            union node_u *right;
        }internal;
        double data[2];
    }info;
};
```

（3）数据对齐：与结构中最长的字节对齐，按字节降序排列可节省空间

14. 机器控制程序

（1）常见错误：数组越界、缓冲区溢出

（2）对抗缓冲区溢出攻击：栈随机化、栈破坏检测、限制可执行代码区域

（3）支持变长栈帧：使用%rbp 作为帧指针

函数开始：将%rbp 当前值压入栈中，再将%rbp 指向当前栈地址

函数结尾：将栈指针设置为保存%rbp 的值，将该值弹出到%rbp

## 三. 处理器体系结构

1. Y86-64 指令集体系结构

（1）程序员可见状态：CC（条件码）、Stat（状态）、DMEM（内存）、PC（地址）、RF（程序寄存器）

（2）指令（r：寄存器、i：立即数、m：内存）

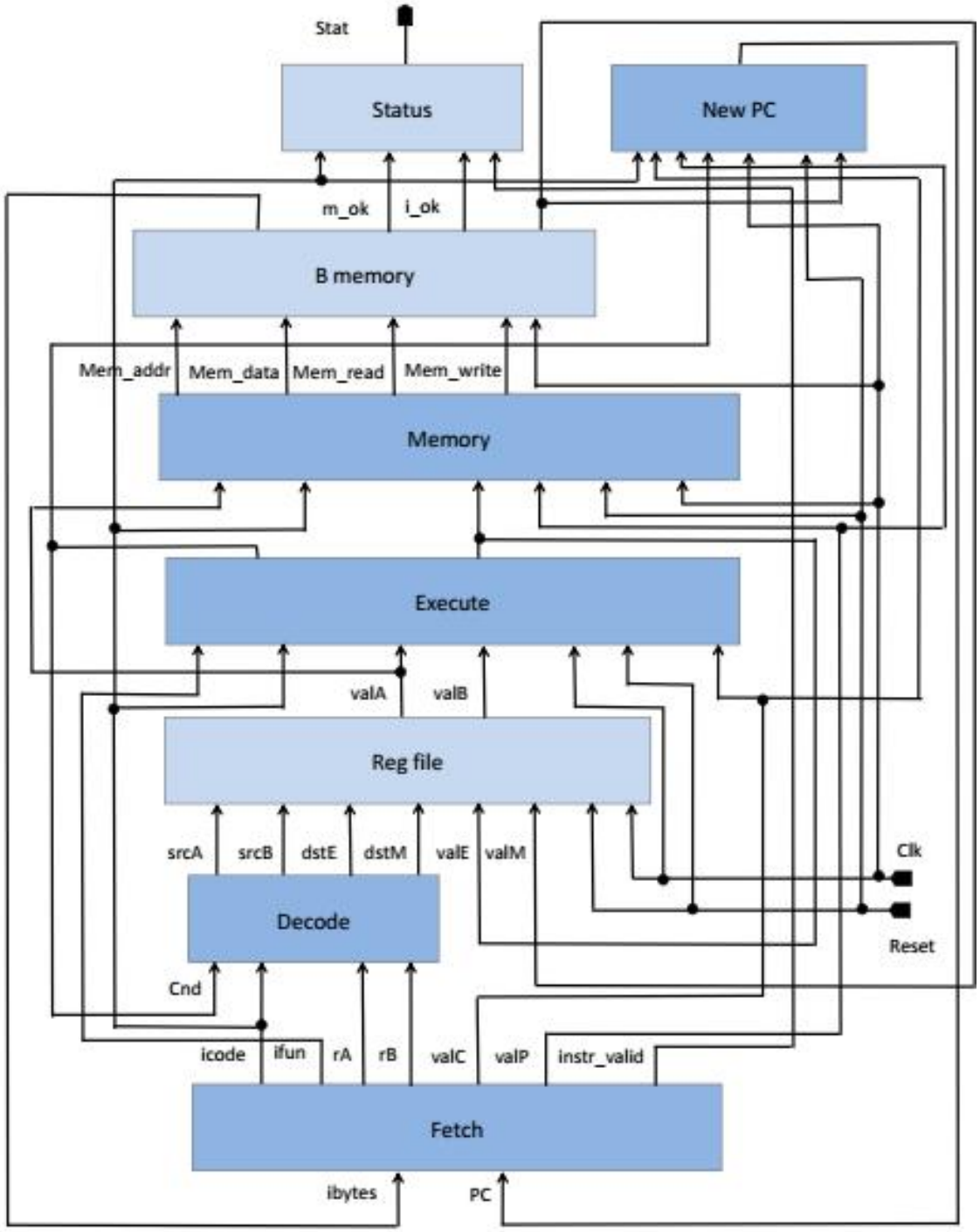
| 字节 | 0 | | 1 | | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| halt | 0 | 0 | | | | | | | | | | |
| nop | 1 | 0 | | | | | | | | | | |
| rrmovq | 2 | 0 | rA | rB | | | | | | | | |
| irmovq | 3 | 0 | F | rB | V：常数值 | | | | | | | |
| rmmovq | 4 | 0 | rA | rB | D | | | | | | | |
| mrmovq | 5 | 0 | rA | rB | D | | | | | | | |
| OPq | 6 | fn | rA | rB | | | | | | | | |

| | | | | | |
|---|---|---|---|---|---|
| jXX | 7 | fn | Dest：绝对地址 | | |
| cmovXX | 2 | fn | rA | rB | |
| call | 8 | 0 | Dest：绝对地址 | | |
| ret | 9 | 0 | | | |
| pushq | A | 0 | rA | F | |
| popq | B | 0 | rA | F | |

Opq：addq 0、subq 1、andq 2、xorq 3
jXX：jmp 0、jle 1、jl 2、je 3、jne 4、jge 5、jg 6
cmovXX：cmovle 1、cmovl 2、cmove 3、cmovne 4、cmovge 5、cmovg 6

| 数字 | 寄存器 | 数字 | 寄存器 |
|---|---|---|---|
| 0 | %rax | 8 | %r8 |
| 1 | %rcx | 9 | %r9 |
| 2 | %rdx | A | %r10 |
| 3 | %rbx | B | %r11 |
| 4 | %rsp | C | %r12 |
| 5 | %rbp | D | %r13 |
| 6 | %rsi | E | %r14 |
| 7 | %rdi | F | 无寄存器 |

字节编码必须有唯一解释，否则为非法指令
（3）异常：AOK 1 正常、HLT 2 执行 halt、ADR 3 非法地址、INS 4 非法指令
（4）程序：
算数指令不能使用立即数
“.”开头的伪指令指明代码或数据的位置
可用 subq 同时设置条件码，testq 非必需
进入循环之前需用 andq 先设置条件码
有些必要的伪指令和代码差不多每个程序都会遇到，套路少不得，反正没什么用就对了
2. 逻辑设计
（1）逻辑门：与 &&、或 ||、非 ！
（2）组合电路：
输入：主输入、某存储单元的输出、某逻辑门的输出
输出：两个或多个逻辑门输出不能连在一起
网必须无环
（3）HCL：只需清楚指令计算过程即可
3. Y86-64 顺序实现
（1）取指 F：将 PC 翻译成 icode、ifun、rA、rB、valC，计算 valP
（2）译码 D：通常读入 rA（F）、rB，得到 valA、valB
（3）执行 E：执行指令的操作、计算内存引用地址，值为 valE，设置条件码
（4）访存 M：向内存写数据，或从内存读出数据 valM
（5）写回 W：最多写两个结果到寄存器文件
（6）更新 PC：从 valP、valC、valM 中选择一个作为下一个地址




4. Y86-64 流水线原理

（1）延迟：以一个阶段中所需时间最长的部分为准

（2）局限：性能提高，吞吐量由于流水线寄存器延迟并不加倍

（3）反馈：（比较危险）

数据相关：下一条指令会用到这一条的计算结果

控制相关：一条指令要确定下一条指令的位置

5. Y86-64 流水线实现：

（1）重新安排计算，将 PC 移到取指下面（SEQ+）
（2）插入流水线寄存器：F、D、E、M、W（PIPE-）
F：取指之前，保存程序计数器的预测值
D：取指和译码之间，保存指令信息
E：译码和执行之间，保存译码指令和寄存器文件读出值
M：执行和访存之间，保存执行指令的结果，分支条件和分支目标信息
W：访存和反馈之间，提供计算出的结果和 PC 选择逻辑的返回地址
（3）对信号重新排列和标号：大写前缀_信号名（流水线寄存器），小写前缀_信号名（流水线阶段）
（4）预测下一个 PC：除 cmovXX 和 ret，皆可预测，预测错误要提供恢复机制
call、jmp：valC
其他指令：valP
cmovXX：AT 策略（60%），BTFNT 策略（%65），NT 策略（40%）
ret：这个指令太强了处理器不敢预测，只能暂停处理新指令直到 ret 写回
6. 流水线冒险：数据冒险、控制冒险
（1）暂停避免数据冒险：插入 bubble 将指令阻塞在译码阶段，等待数据写回
（2）转发避免数据冒险：将要写的值传送到流水线寄存器 E 作为源操作数
（增加旁路路径和转发逻辑的 PIPE-就是最终实现的 PIPE 结构）
（3）加载/使用数据冒险：加载互锁 = 暂停 + 转发
（4）避免控制冒险：无法确定下一条指令的地址引起的冒险
ret：三个 bubble 直到 ret 写回
分支预测错误：在下一周期的译码和执行阶段插入 bubble，取出跳转指令后的指令
7. 异常处理
（1）异常种类：halt 指令、非法指令和功能码组合的指令、取指或数据读写试图访问非法地址
（2）基本原则：由流水线中最深的指令引起的异常优先级最高
（3）细节问题：多个异常指令、分支预测错误取消异常指令、流水线化的处理器在不同阶段更新系统状态的不同部分导致影响系统状态
（4）处理方式：在每个流水线寄存器设置状态码 stat
8. 流水线控制逻辑：处理加载\使用冒险，ret，预测错误的分支，异常
（1）特殊控制情况期望的处理：
禁止执行阶段中的指令设置条件码
向内存阶段插入气泡，禁止数据写入内存
当写回阶段有异常指令时，暂停写回阶段
（2）发现特殊控制条件

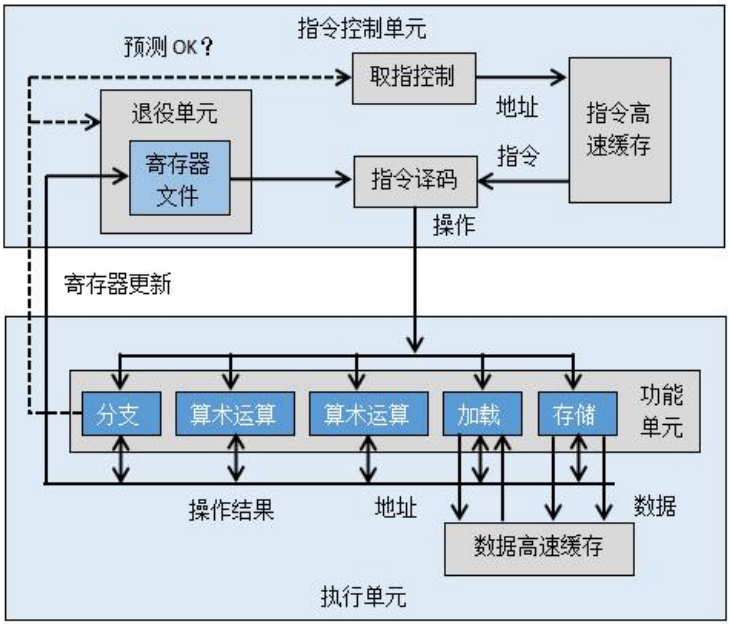
| 条件 | 触发条件 |
| --- | --- |
| 处理 ret | IRET in {D_icode, E_icode, M_icode} |
| 加载/使用冒险 | E_icode in {IMRMOVL, IPOPL} && E_dstM in {d_srcA, dsrB} |
| 预测错误的分支 | E_icode = IJXX && ! e_Cnd |
| 异常 | m_stat in {SADR, SINS, SHLT} \|\| W_stat in {SADR, SINS, SHLT} |

（3）流水线控制机制

| 条件 | 流水线寄存器 | | | | |
|---|---|---|---|---|---|
| | F | D | E | M | W |
| 处理 ret | 暂停 | 气泡 | 正常 | 正常 | 正常 |
| 加载/使用冒险 | 暂停 | 暂停 | 气泡 | 正常 | 正常 |
| 预测错误的分支 | 正常 | 气泡 | 气泡 | 正常 | 正常 |

## 四. 优化程序性能

1. 优化编译器的局限：不同的指针是否指向同一位置
2. 表示程序性能：CPE 每元素的周期数
3. 消除循环的低效率：将需要计算的不变值移到循环外
4. 消除不必要的内存引用：引入一个变量来计算循环结果，最后再写内存
5. 现代处理器（超标量：每个时钟周期执行多个操作，乱序）




（1）整体操作：ICU 从指令高速缓存读取指令，译码，EU 接收操作，执行，读写内存，对操作求值，正确指令退役，错误指令被清空

（2）功能单元的性能

| 运算 | 整数 | | | 浮点数 | | |
|---|---|---|---|---|---|---|
| | 延迟 | 发射 | 容量 | 延迟 | 发射 | 容量 |
| 加法 | 1 | 1 | 4 | 3 | 1 | 1 |
| 乘法 | 3 | 1 | 1 | 5 | 1 | 2 |
| 除法 | 3 ~ 30 | 3 ~ 30 | 1 | 3 ~ 15 | 3 ~ 15 | 1 |

关键路径：延迟最长的操作

Horner 求多项式值：a0 + x * (a1 + x * (a2 + ... + x *(an-1 + x * an) ... ))

6. 循环展开（k * 1）：增大步长，在循环体中增加次数，直至达到延迟界限
7. 提高并行性：

（1）多个累积变量（k * k）：每次并行累积 k 个值，不受延迟界限限制

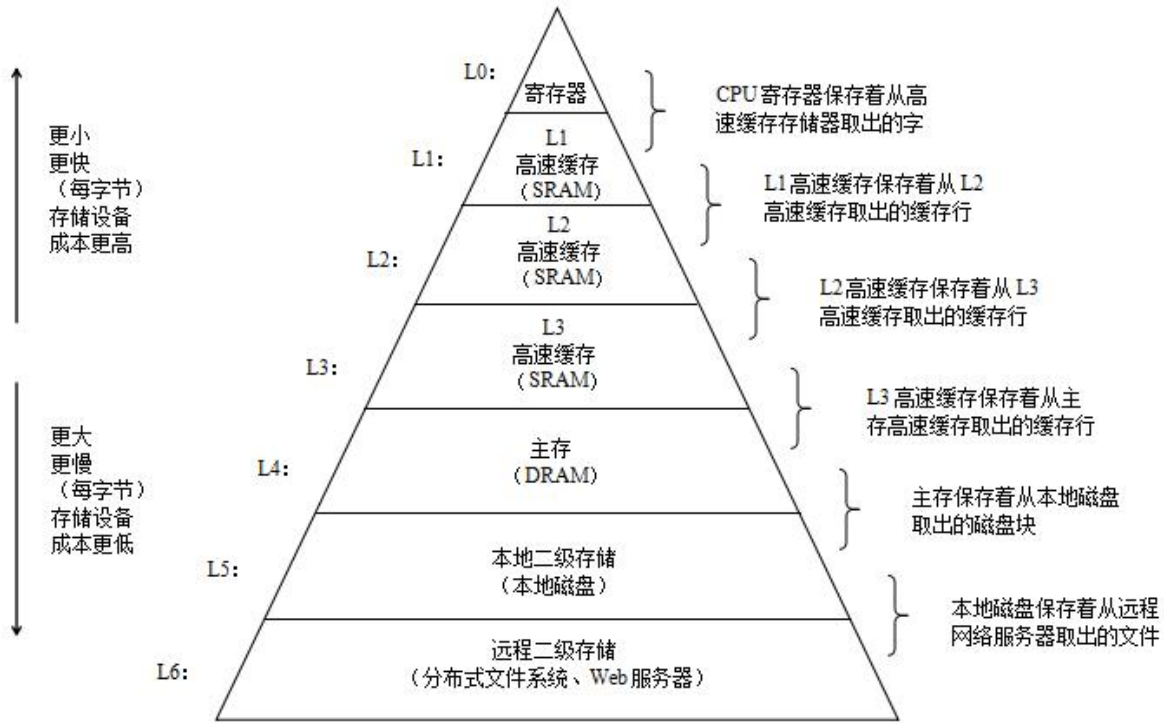
（2）重新结合变换（k * 1a）：

```
val = (val OP data[i]) OP data[i+1];    ->    val = val OP (data[i] OP data[i+1]);
```

8. 限制因素：寄存器溢出（循环展开有度），预测错误处罚（从代码下手）
9. 理解内存性能
（1）加载：依赖于流水线能力和加载单元延迟（不在关键路径上）
（2）存储：可能出现读/写相关导致处理速度下降

## 五．存储器层次结构

1. 存储技术
（1）随机访问存储器：SRAM、DRAM、ROM、闪存、访问主存
（2）磁盘存储
构造：盘片（表面覆盖磁性记录材料，每个磁道划分一个扇区）、主轴
容量：由记录密度、磁道密度、面密度决定
公式：磁盘容量 = 字节数 * 平均扇区数 * 磁道数 * 表面数 * 盘片数（字节）
操作：访问（寻道、旋转、传送），读/写（防止冲撞，密封包装）
访问时间计算：T = T avgseek + 0.5 * 60/RPM + 60/(RPM * (平均扇区数/磁道))  s
逻辑磁盘块：B 个扇区大小的逻辑块序列，编号为 0 ~ B-1，对应唯一标识的（盘面，磁道，扇区）三元组
（3）固态硬盘：基于闪存，读比写快（数据以页为单位读写，擦除整块才能写）
2. 局部性：时间局部性（多次引用）、空间局部性（连续引用）
3. 存储器层次结构




（1）高速缓存：底层部分映射到上层，能不能中听天命
（2）缓存命中（比较有缘）：可从 k 层读到 k+1 层的数据
（3）缓存不命中（算你倒霉）：
冷不命中：k 层缓存为空，执行放置策略
冲突不命中：限制性的放置策略导致（为什么不叫热不命中我也很奇怪）
容量不命中：缓存太小，处理不了工作集
4. 高速缓存存储器

（1）通用结构：M = 2^m 个地址，S = 2^s 个高速缓存组，每组 E = 2^e 个高速缓存行，每行 B = 2^b 个数据块，江湖人称（S，E，B，m），大小为 C = S * E * B

| 地址（m 位）： | t 位：标记 | s 位：组索引 | b 位：块偏移 |
|---|---|---|---|

| | | | | | | |
|---|---|---|---|---|---|---|
| 组 0 | 有效 | 标记 | 0 | 1 | …… | B - 1 |
| | ……每组 E 行 | | | | | |
| | 有效 | 标记 | 0 | 1 | …… | B - 1 |
| 组 1 | 有效 | 标记 | 0 | 1 | …… | B - 1 |
| | ……每组 E 行 | | | | | |
| | 有效 | 标记 | 0 | 1 | …… | B - 1 |
| ……总共 S 组 | | | | | | |
| 组 S - 1 | 有效 | 标记 | 0 | 1 | …… | B - 1 |
| | ……每组 E 行 | | | | | |
| | 有效 | 标记 | 0 | 1 | …… | B - 1 |

（2）直接映射高速缓存：组选择、行匹配（标记位、有效位均匹配）、字选择
冲突不命中（抖动）：在一个数组后填充 B 字节避开相同组
（3）组相连高速缓存：同上，除了每个组里的高速缓存行比较多，需要遍历
不命中时的行替换：有空行直接存，无空行 LRU
（4）全相联高速缓存：只有一组，地址不带组索引，只需行匹配和字选择（贵）
（5）高速缓存写：
写命中：直写（生猛慎用）、写回（增加修改位，拖到被驱逐）
写不命中：非写分配（不经过高速缓存直接写入低层）、写分配（加载相应的块到高速缓存，更新高速缓存）
（6）统一的高速缓存：两个独立高速缓存分别处理指令和数据
（7）性能影响：命中率、命中时间、不命中处罚（总之什么参数都是双刃剑）
5. 高速缓存友好：快（主要看循环）、准（减少不命中）、狠（你真棒）
6. 高速缓存对程序性能的影响
（1）存储器山（图太随心所欲我不画了，看教材封面上那个彩虹披风）
读数据时间：周期数 = 时钟频率 / 吞吐量峰值 * 字节长
（2）重新排列循环可提高空间局部性：矩阵乘法的例子

# II. 在系统上运行程序

## 一. 链接

1. 静态链接：符号解析，重定位
2. 目标文件：可重定位目标文件、可执行目标文件、共享目标文件
3. 可重定位目标文件：可与其他重定位目标文件合并
（1）ELF 头：
16 字节序列：生成该文件的系统的字大小和字节顺序
ELF 头的大小、目标文件的类型、机器类型
节头部表的文件偏移、节头部表中条目的大小和数量
（2）节：

静态变量：static

| | |
|---|---|
| .text | 已编译程序的机器代码 |
| .rodata | 只读数据 |
| .data | 已初始化的全局和静态 C 变量 |
| .bss | 未初始化（初始化为 0）的全局和静态 C 变量 |
| .symtab | 在程序中定义、引用的函数和全局变量，不包含局部变量条目 |
| .rel.text | .text 节中位置的列表（目标文件组合时需修改） |
| .rel.data | 被模块引用或定义的所有全局变量的重定位信息 |
| .debug | 调试符号表，条目为程序中定义的局部变量和类型定义（-g 得到） |
| .line | 原始 C 源程序中的行号和.text 节中机器指令之间的映射 |
| .strtab | 字符串表，包括.symtab 和.debug 中的符号表，节名字，^结尾 |

（3）节头部表

4. 符号和符号表

（1）符号：

本模块定义，并能被其他模块引用的全局符号

其他模块定义，被本模块引用的全局符号

只被本模块定义并引用的符号

（2）符号表：

| 类型 | 名称 | 含义 |
|---|---|---|
| int | name | 字符串表中的字节偏移 |
| char | type | 数据或函数 |
| char | binding | 表示符号是本地还是全局 |
| char | reserved | 未使用（谁给我一个 Unused 的更贴切的翻译……） |
| short | section | 节头部索引（感谢谷歌） |
| long | value | 距定义目标的节的起始位置的偏移 |
| long | size | 目标的大小（字节为单位） |
| 伪节 | ABS | 不该被重定位的符号 |
| 伪节 | UNDEF | 未定义的符号（本模块引用，其他地方定义） |
| 伪节 | COMMON | 未初始化的全局变量 |

2. 符号解析

（1）多重定义的全局符号：链接器通常不表明检测到多个 x 的定义

不允许有多个同名的强符号

如果有强、弱符号同名，选择强符号

多个弱符号同名，任选一个

（2）与静态库链接

静态库：所有相关的目标模块打包形成的单独文件

不用静态库的方法：编译器辨认标准函数调用直接生成代码、将所有标准 C 函数放在一个单独的可重定位目标模块中（浪费空间）

存放格式：存档

（3）使用静态库解析引用：符号解析阶段，链接器从左到右按命令行顺序扫描可重定位目标文件和存档文件，在扫描中维护一个可重定位目标文件的集合 E、

一个未解析的符号集合 U、一个在前面输入文件中已经定义的符号集合 D，初始均为空。

输入文件 f 为目标文件，添加 f 至 E，修改 U 和 D 反映 f 中符号定义和引用

f 是存档文件，尝试匹配 U 中未解析的符号和存档文件成员定义的符号，若成员 m 定义了 U 中引用的符号，添加至 E，修改 U 和 D

完成扫描后如果 U 非空，输出错误并终止

关于库的一般准则是将其放在命令行结尾，除非各个库的成员相互独立

3. 重定位：

（1）重定位节和符号定义：所有相同类型的节合并为同一类型的新聚合节

（2）重定位节中的符号引用：依赖重定位条目修改代码节和数据节中对符号的引用，使其指向正确运行地址

（3）重定位条目：汇编器遇到对最终位置未知的目标引用，生成重定位条目

（4）重定位符号引用：PC 相对引用、绝对引用

```
/**PC相对引用**/
call指令开始于节偏移 0xe 的地方

r.offset = 0xf
r.symbol = sum
r.type = R_X86_64_PC32
r.addend = -4

ADDR(s) = ADDR(.text) = 0x4004d0
ADDR(r.symbol) = ADDR(sum) = 0x4004e8

refaddr = ADDR(s) + r.offset = 0x4004df
*refptr = (unsigned) (ADDR(r.symbol) + r.addend - refaddr)
        = (unsigned) 0x5

PC相对引用值为 0x5
4004de:       e8  05  00  00  00      callq  4004e8 <sum>
/**绝对引用**/
mov指令开始于节偏移 0x9 的地方

r.offset = 0xa
r.symbol = array
r.type = R_X86_64_32
r.addend = 0

ADDR(r.symbol) = ADDR(array) = 0x601018

*refptr = (unsigned) (ADDR(r.symbol) + r.addend)
        = (unsigned) 0x601018

4004d9:  bf  18  10  60  00   mov  $0x601018, %edi
```

7. 可执行目标文件：可被直接复制到内存并执行，ELF 表没有 rel，增加了.init、段头部表，程序入口点，运行用 ./文件名

8. 动态链接共享库：

（1）共享库（共享目标）：可在运行或加载时加载到任意内存地址的目标模块

（2）动态链接：将共享库和一个内存中的程序链接起来的过程

（3）从应用程序加载和链接共享库的例子：分发软件、构建高性能 Web 服务器

9. 位置无关代码：可以加载无需重定位，共享库编译 always 选择

（1）PIC 数据引用：代码段中任何指令和数据段中任何变量间的距离都是运行时常量，可在数据段开始的地方创建 GOT 供所有引用使用
（2）PIC 函数调用：延迟绑定（GOT 和 PLT 共同实现）
10. 库打桩机制：允许截获对共享库函数的调用，用自己的代码取而代之
基本思想：给定一个需要打桩的目标函数，创建一个包装函数，原型与目标函数一致

## 二．异常控制流

1. 异常：控制流中的突变，用来响应处理器状态中的某些变化
（1）异常处理：系统中每个可能的异常都有唯一的非负整数异常号，系统启动时分配并初始化的异常表中，条目 k 包含异常 k 处理程序的地址
（异常处理程序在内核模式下，对所有系统资源有完全的访问权限）
（2）异常类别

| 类别 | 原因 | 异步/同步 | 返回行为 |
|---|---|---|---|
| 中断 | 来自 I/O 设备的信号 | 异步 | 总是返回到下一条指令 |
| 陷阱 | 有意的异常 | 同步 | 总是返回到下一条指令 |
| 故障 | 潜在可恢复的错误 | 同步 | 可能返回到当前指令 |
| 终止 | 不可恢复的错误 | 同步 | 不会返回 |

（3）Linux / x86-64 系统中的异常
故障和终止：除法错误、一般保护故障（神 • 段错误）、缺页、机器检查
系统调用：%rax 包含系统调用号，从系统调用返回时，包含返回值
2. 进程：一个执行中程序的实例
（1）逻辑流：程序运行时一系列 PC 值的序列
（2）并发流：一个逻辑流的执行在时间上与另一个重叠（不存在的，穿插而已）
（3）私有地址空间：相同的通用结构，不同的关联内容，只能被本进程读或写
（4）上下文切换：内核重新启动一个被抢占的进程（调度器处理）
3. 系统调用错误处理：通常返回-1，设置 errno
4. 进程控制
（1）获取进程 ID：pid_t getpid(void); 当前进程　pid_t getppid(void); 父进程
（2）创建和终止进程：
进程的三种状态：运行、停止、终止
创建子进程：fork 函数（调用一次，返回两次，父子并行，共享文件）
停止：收到 SIGSTOP、SIGTSTP、SIGTTIN 或 SIGTTOU 挂起，SIGCONT 解禁
终止：收到终止信号、从主程序返回、调用 exit 函数
（3）回收子进程：pid_t waitpid(pid_t, int *statusp, int options)
init 接盘所有父进程停止的孤儿进程，回收僵死子进程
等待集合成员：pid > 0，为单独的子进程，pid = -1，为父进程所有子进程
修改默认行为：options
WNOHANG：挂起调用进程直到有子进程终止，返回值为 0
WUNTRACED：挂起调用进程直到有进程终止或停止，返回其 PID
WCONTINUED：挂起调用进程直至有进程终止或收到 SIGCONT 重新执行
WNOHANG | WUNTRACED：立即返回，都不终止返回 0，否则返回 PID
检查已回收子进程的退出状态：statusp

WIFEXITED(status)：子进程正常终止，返回真
WEXITSTATUS(status)：返回正常终止的子进程的退出状态
WIFSIGNALED(status)：子进程因未捕获的信号终止，返回真
WTERMSIG(status)：返回导致子进程终止的信号的编号
WIFSTOPPED(status)：返回的子进程当前为停止，返回真
WSTOPSIG(status)：返回导致子进程停止的信号的编号
WIFCONTINUED(status)：子进程收到 SIGCONT 重新启动，返回真

错误条件：
调用进程没有子进程，waitpid 函数返回-1，设置 errno 为 ECHILD
waitpid 函数被一个信号中断，返回-1，设置 errno 为 EINTR

wait 函数：wait(&status) = waitpid(-1, &status, 0)

（4）让进程休眠：unsigned int sleep(unsigned int secs);

（5）加载并运行程序：调用一次从不返回（除非出现错误）

函数：int execve(const char *filename, const char *argv[], const char *envp[]);

启动代码：int main(int argc, char **argv, char **envp);

5. 信号（30 个不想写了……）：

（1）进程组：每个进程都只属于一个进程组，由进程组 ID 标识

pid_t getpgrp(void); 返回当前进程组 ID

int setpgid(pid_t pid, pid_t pgid); 将进程 pid 的进程组改为 pgid，pid 为 0，使用当前进程的 PID，pgid 为 0，用 pid 指定的进程 PID 作为进程组 ID。

（2）发送信号：

/bin/kill：向另外的进程发送任意信号（PID 为负导致信号被发送到进程组 PID 中的每个进程）

从键盘发送信号：Ctrl + C——SIGINT（终止）、Ctrl + Z——SIGTSTP（挂起）

使用 kill 函数：int kill(pid_t pid, int sig);（pid > 0，发送到进程 pid；pid = 0，发送到每个进程；pid < 0，发送到绝对值进程组）

使用 alarm 函数：unsigned int alarm(unsigned int secs); 进程发送给自己，返回前一次闹钟剩的秒数，如没有返回 0

（3）接收信号：sighandler_t signal(int signum, sighandler_t handler);

handler = SIG_IGN，忽略 signum

handler = SIG_DFL，signum 恢复默认行为

否则 handler 为用户定义的信号处理程序地址，改变默认行为

（4）阻塞和解除阻塞信号：隐式（来啥挡啥）、显式（sigprocmask 指定的信号）

int sigprocmask(int how, const sigset_t *set, sigset_t *oldset);

how = SIG_BLOCK：将 set 添加到 blocked

how = SIG_UNBLOCK：从 blocked 中删除 set

how = SIG_SETMASK：block = set

（blocked 位向量中维护着被阻塞的信号集合）

（5）编写信号处理程序：安全、正确、简单

（6）同步流避免并发错误：sigprocmask 同步进程，避免竞争

（7）显式地等待信号：父进程设置 SIGINT 和 SIGCHLD 的处理程序，调用函数 sigsuspend 等待子进程终止，将其非零的 PID 赋给全局 pid 变量，终止循环。

6. 非本地跳转：

（1）int setjmp(jmp_buf env);

返回多次，保存当前调用环境，非零返回值指明错误类型

（2）void longjmp(sigjmp_buf env, int retval);

从不返回，恢复调用环境，允许跳过所有中间调用的特性可能的意外后果

## 三．虚拟内存

1. 虚拟寻址：CPU -> 虚拟地址 -> 地址翻译 -> 物理地址
2. 地址空间：n 位虚拟地址空间（N = 2^n 个虚拟地址），物理地址空间（M 个）
3. 虚拟内存作为缓存工具

（1）虚拟页状态：未分配、未缓存、缓存

（2）页表：每个页在页表中都有一个 PTE（假设：有效位 + n 位地址）

PTE 数量：2^(n - p)（P = 2^p 为页大小）

（3）页命中：根据有效位判断是否在缓存中（类高速缓存）

（4）缺页：缓存不命中，选择牺牲页

4. 虚拟内存作为内存管理工具

（1）多个虚拟页面可映射到同一个共享物理页面上

（2）简化链接：独立地址空间允许每个进程的内存映像使用相同的基本格式

（3）简化加载：为代码和数据分配虚拟页，标记为无效，页表条目指向目标文件中适当位置

（4）简化共享：将不同进程中适当的虚拟页面映射到相同的物理页面

（5）简化内存分配：分配连续的 k 个虚拟页面，映射到不连续的物理页面

5. 虚拟内存作为内存保护工具：PTE 添加许可位（SUP、READ、WRITE）
6. 地址翻译

| 基本参数 | |
|---|---|
| 符号 | 描述 |
| N = 2^n | 虚拟地址空间中的地址数量 |
| M = 2^m（just 为了方便） | 物理地址空间中的地址数量 |
| P = 2^p | 页的大小（字节） |

| 物理地址（PA）的组成部分 | |
|---|---|
| 符号 | 描述 |
| PPO | 物理页面偏移量（字节） |
| PPN | 物理页号 |
| CO | 缓冲块内的字节偏移量 |
| CI | 高速缓存索引 |
| CT | 高速缓存标记 |

| 虚拟地址（VA）的组成部分 | |
|---|---|
| 符号 | 描述 |
| VPO | 虚拟页面偏移量（字节） |
| VPN | 虚拟页号 |
| TLBI | TLB 索引 |
| TLBT | TLB 标记 |

（1）结合高速缓存和虚拟内存：地址翻译在高速缓存查找前
（2）TLB：小的虚拟寻址缓存，每行保存一个单 PTE 块，可加速地址翻译
（3）多级页表（简单来讲有点像反过来的缓存）：
一级页表：每个 PTE 映射一个 4MB 的片（由 1024 个连续页面组成）
二级页表：一级页表片 i 中至少有一个页分配，就指向一个二级页表基址
（一级页表 PTE 为空时，对应二级页表不存在，二级页表可随时创建或调用）
（4）端到端的地址翻译
虚拟地址

| 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| VPN：TLBT | | | | | | VPN：TLBI | | VPO | | | | | |

物理地址：

| 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| PPN：CT | | | | | | PPO：CI | | | | PPO：CO | |

7. 内存映射：将一个虚拟内存区域与一个磁盘上的对象关联起来
（1）映射对象：Linux 文件系统的普通文件、匿名文件（内核创造，全二进制 0）
（2）一个进程将一个共享对象映射到虚拟地址空间的一个区域内，所有操作对映射了同样共享对象的其他进程可见
（3）私有对象：写时复制（两个进程将一个私有对象映射到不同区域，没有进程写它自己的私有区域，就可以共享物理内存中对象的一个单独副本）
（4）创建新的虚拟内存区域（注意不是 mmp）：
void *mmap(void *start, size_t length, int prot, int flags, int fd, off_t offset);
最好从 start 开始，将 fd 指定的对象的一个连续的片映射到这个新区域，片大小 length，偏移量 offset
prot = PROT_EXEC：这个区域内的页面由可被 CPU 执行的指令组成
prot = PROT_READ：这个区域内的页面可读
prot = PROT_WRITE：这个区域内的页面可写
prot = PROT_NONE：这个区域内的页面不能被访问
flag = MAP_ANON：映射匿名对象
flag = MAP_PRIVATE：映射私有对象
flag = MAP_SHARED：映射共享对象
（5）删除虚拟内存的区域：int munmap(void *start, size_t length);
8. 动态内存分配
（1）分配器的要求：处理任意请求序列、立即响应请求、只使用堆、对齐块、不修改已分配的块
（2）分配器的目标：最大化吞吐率、最大化内存利用率
（3）碎片：内部（已分配块大小和有效载荷之差的和）、外部（空闲内存合计可以分配，单独不行）
（4）隐式空闲链表：连续的已分配块和空闲块的序列
块 = 头部（编码大小） + 有效载荷 + 填充
（5）放置已分配的块：首次（从头）、下一次（接上）、最佳（遍历）适配
（6）获取额外的堆内存：调用 sbrk 函数，将额外内存变成大空闲块插入链表

（7）合并空闲块：立即合并、推迟合并

设脚部表示前一个块的状态，根据四种情况分别合并

前后均有、前有后无（后并）、前无后有（前并）、前后均无（两头并）

（8）显式空闲链表：双向链表，可用 LIFO 顺序维护，搭首次适配，也可用地址顺序首次适配，接近最佳适配的内存利用率

（9）分离的空闲链表：

分离存储：每个大小类的空闲链表包含大小相等的块，块大小为大小类中最大元素大小

分离适配：首次适配后将分割出的空闲块插入适当链表，或申请内存

伙伴系统：每个大小类都是 2 的幂，每次分配递归二分

9. 垃圾收集：

（1）根节点、堆节点（堆中的已分配块）

（2）Mark & Sweep 垃圾收集器（标记可达，清除不可达）

10. C 中常见内存相关错误：

（1）间接引用坏指针：scanf 忘加&

（2）读未初始化的内存：一般错误地认为是 0

（3）允许栈缓冲区溢出：串过长

（4）假设指针和他们指向的对象等大：给指针分配内存的时候少加*

（5）错位错误：数组从 0 到 n

（6）引用指针而非对象：指针忘加括号

（7）误解指针运算：单位不一定是字节

（8）引用不存在的变量：返回&变量名

（9）引用空闲堆块中的数据：free 后引用

（10）引起内存泄漏（缓慢杀手）：忘记 free