CSAPP期末复习

期末不考范围

第一章计算机系统漫游

2.4浮点数的表示与运算

3.1汇编代码的历史

3.11浮点代码

4 处理器体系结构

6.1存储器计数

6.5编写高速缓存友好代码

6.6综合：高速缓存对程序性能的影响

开始复习

第一节——比特、字节（信息的基础表示）

进制的转换

进制的转换	B 二进制	O 八进制	D 十进制	H 十六进制
Binary	-	从小数点开始每三位二进制数为一组转换为一位八进制数	十进制数等于按位乘幂	四位二进制数为一组转换为一位十六进制数
O 八进制	一位八进制数变为三位二进制数	-	按位乘幂	先转二再转十六
D 十进制	降幂法	先转二再转八	-	先转二再转十六
H 十六进制	一位十六进制数变为四位二进制数	先转二再转八	按位乘幂	-

位移算符

位移算符包括左移"<<"和右移">>"，右移包括逻辑右移和算术右移

左移<<:

x << y 表示将数据x按位左移y位，丢弃高位，末尾补0

右移>>:

逻辑右移	不管符号位，左侧补0	1011->0101
算术右移	新加入位复制符号位	1011->1101

Argument x	01100010
<< 3	00010000
Log. >> 2	00011000
Arith. >> 2	00011000

Argument x	10100010
<< 3	00010000
Log. >> 2	00101000
Arith. >> 2	11101000

整型

无符号数与有符号数

1、无符号数与有符号数仅作用于计算某个二进制数的值时，不影响原有二进制表示。对一个变量声明为有符号类型/无符号类型仅在解释这个变量的值时生效，不改变这个变量原来存储的二进制数据。

2、无符号数的表示：所有位均表示数值位，最高位无符号意义

eg：当有5位数据10110

解释为无符号数据时值为：16+0+4+2+0=22

3、有符号数的表示：最高位解释为符号位+数值位的组合，“0”表示正数，“1”表示负数

eg：有5位数据10110

解释为有符号数					
16	8	4	2	1	结果

解释为有符号数					
1	0	1	1	0	=-16+0+4+2+0=-10
0	1	0	1	1	=0+8+2+1=11
解释为无符号数					
16	**8**	**4**	**2**	**1**	**结果**
1	0	1	1	0	=16+0+4+2+0=22

4、有符号数与无符号数的取值范围

类型	**取值范围**
无符号数	0——2^w-1
有符号数	-2^(w-1)——2^(w-1)-1

类型转换

1、在C语言中当表达式中混有有符号数和无符号数时，**有符号数会隐式转换为 无符号数**

2、有符号数转换为无符号数时，将首位解释为正数即可

3、无符号数转换为有符号数时，当首位为1时进行重新计算

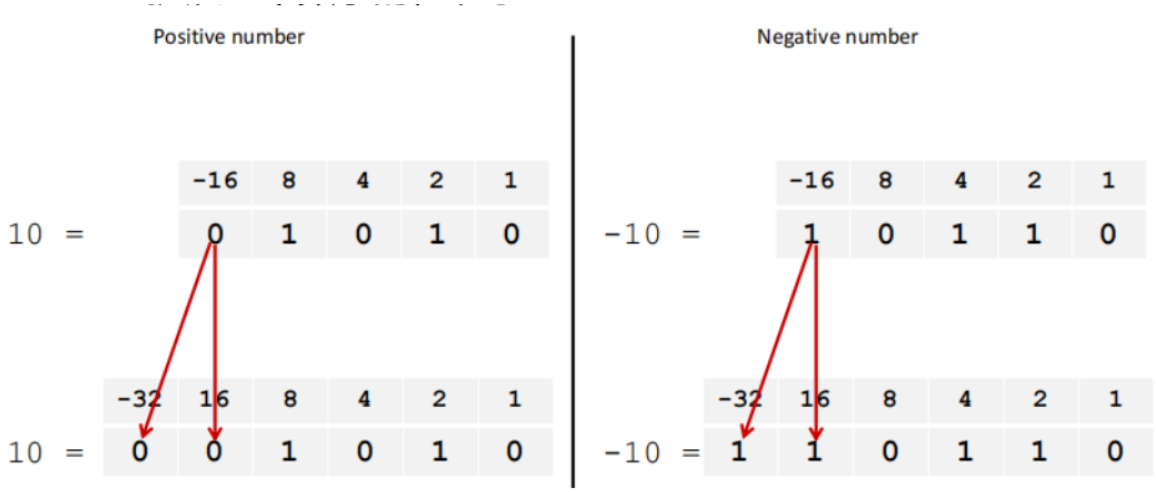
拓展一个数字的位表示

扩展无符号数

扩展k位，在高位上加k个0

扩展有符号数

扩展k位，将原最高位复制给所有拓展的高位



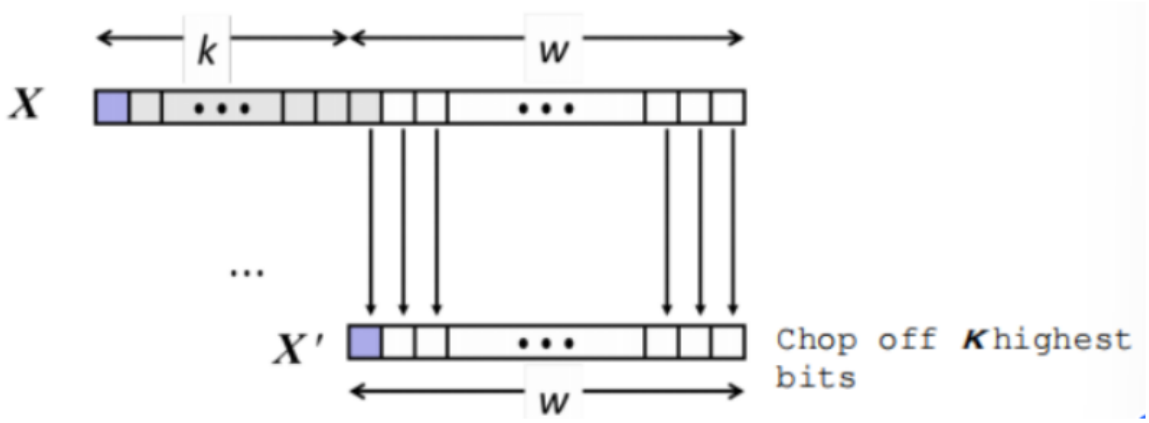
截断数字

截断无符号数

直接丢弃高位，只保留低位

截断有符号数

将原首位视作符号位，截断k位时实际上丢弃了前k+1位数据，然后在剩下的位的首位补上原符号位数据



加法与乘法

1、无符号加法，产生进位（溢出）时取模，丢掉进位

2、有符号数加法，溢出时回绕

3、乘法：产生进位时直接截断，丢掉所有进位

内存表与字节序

C语言中部分类型的字长

C语言类型	64位中字长
char	1
int	4
short	2
long	8
float	4
double	8
void*(pointer)指针	8

字节序：小端与大端

1、小端序：低位字节存储于低位地址（低位字节存储于地址的左侧）(速记：**小端字节顺序颠倒，字内不变**)

2、大端序：低位字节存储于高位地址（高位地址存储于地址的左侧）

目标地址：0x01234567	0x100	0x101	0x102	0x103
小端排序：	67	45	23	01
大端排序：	01	23	45	67

第二节——机器代码，汇编语言

操作数大小后缀

后缀名	表示操作数大小
q	8字节（64位数据）/四字
l	4字节（32位数据）/双字
w	2字节（16位数据）/字
b	1字节（8位数据）/字节

字和字节是两个东西，字来源于早期十六位操作系统，一个字（Word）表示十六位数据

常用汇编指令

movq——数据移动指令

1、操作数类型：

立即数：加上$前缀：$0x400

寄存器：加上%前缀：%rdi

内存：通过括号（寄存器）表示将这个**寄存器中存储的数据看作指针**，取这个指针指向的地址中的值：(%rax)　*注：括号可用于计算偏移量：见下文

2、语法格式：

"movq　源，目标"————————————将前者移动给后者

eg：

movq %rax, %rbx	将%rax寄存器中的值赋值给%rbx
movq $0x100, %rax	将立即数（常数）赋值给%rax寄存器

寻址模式

1、通过寄存器简单寻址：取寄存器指向的内存值：movq（%rcx)，%rax

2、通过偏移量计算寻址：通过括号 + 系数表示基址 + 偏移量：

eg：8(%rax)表示取%rax中的值 + 8的地址处的值

3、复杂寻址结构：s基址+索引+比例

D(R1,R2,S)：表示取地址中

寄存器1%r1的值 + 寄存器2%r2的值 * 比例系数S + 偏移量D

的值

eg：8(%rax，%rcx，3)表示：取地址为 **%rax的值+%rcx的值*3+8** 位置处的地址中存放的数据

算数与逻辑运算指令

格式为：操作指令 操作源，操作目标

1、**leaq Src, Dst**

用于将某个寄存器中的值通过复杂寻址结构的计算逻辑赋给另一个寄存器

2、其它汇编指令可以参考汇编指令文档

条件控制——control flow

1、%rsp寄存器——用于存储栈顶指针

2、%rip寄存器——存于存储下一条指令

3、条件控制码——CF,ZF,SF,OF

其中：

CF进位标志——最近的操作溢出

SF符号标志——最近的操作得到负数

ZF零标志——最近的操作结果为0

OF溢出标志——最近的操作导致补码溢出

4、 ❤️ ❤️ ❤️***cmp指令——后比前***❤️ ❤️ ❤️

cmp src, dst = dst - src

计算后者减前者的数：dst-src然后进行条件判断

5、test指令——计算两者的与

test src, dst = dst&src

大多时候用于判断一个寄存器中存的数是否为真

三种循环的汇编表示

do-while循环——先运行后条件判断

```
long fact(long n){
    long result = 1;
    do{
        result *= n;
        n += 1;
    }while(n > 1)
    return result;
}
```

```
fact_do:
    movl $1, %eax
.L2
    imulq   %rdi,%rax       //传进来的第一个参数存放在%rdi寄存器中
    subq    $1,%rdi
    cmpq    $1,%rdi
    jg      .L2
    rep;ret
```

通过汇编可以看出，do-while语句是先执行循环内的语句，最后进行判断和跳转

while循环——先跳转再判断最后执行

```
long fact(long n){
    long result = 1;
    while(n>1){
        result *= n;
        n--;
    }
    return result;
}
```

```
fact:
    movl    $1,%eax
    jmp     .L5
.L6
    imulq   %rdi,%rax
    subq    $1,%rdi
.L5
    cmpq    $1,%rdi
    jg      .L6
```

由汇编可看出，while循环先确定了判断块和循环语句块，先跳转到判断块再决定是否执行循环语句块

for循环——先转换为while循环再编写汇编

第三节——过程传递

过程控制的机制

控制传递

通过call label指令实现过程的调用：

使用call指令时通过将**当前的下一位地址**压入栈中存储（即**压入栈顶**），并将%rip指令寄存器指向要**跳转的地址**位置的指令

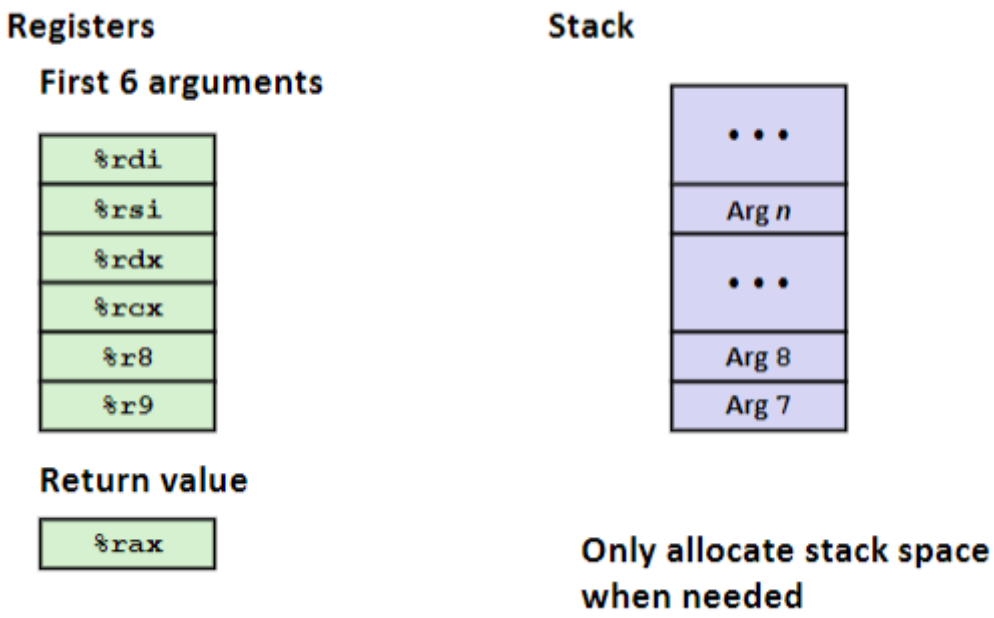
使用return指令时通过**弹出栈顶地址元素**找到原先call指令存储的返回地址，然后跳转到这个返回地址

数据传递

传入的前六个参数保存在前6个寄存器中，传入超过6个参数则存放在栈中

返回值通常存放在寄存器%rax中

如图所示：



栈的结构与特性

生长方向

栈向低地址拓展，%rsp始终指向栈顶元素地址

%rbp——>	**栈底bottom**
高地址	
8字节每单位	
低地址	
%rsp——>	**栈顶Top**
栈顶向下拓展	新插入栈的元素位置

以8字节为单位拓展空间

操作指令

pushq Src：将Src的值压入当前栈顶，然后将%rsp寄存器地址向下拓展8字节：%rsp = %rsp - 8

popq Dest：将Dest的值从当前栈顶读取出来，然后将栈顶指针%rsp从当前位置向上收缩8字节：%rsp = %rsp + 8

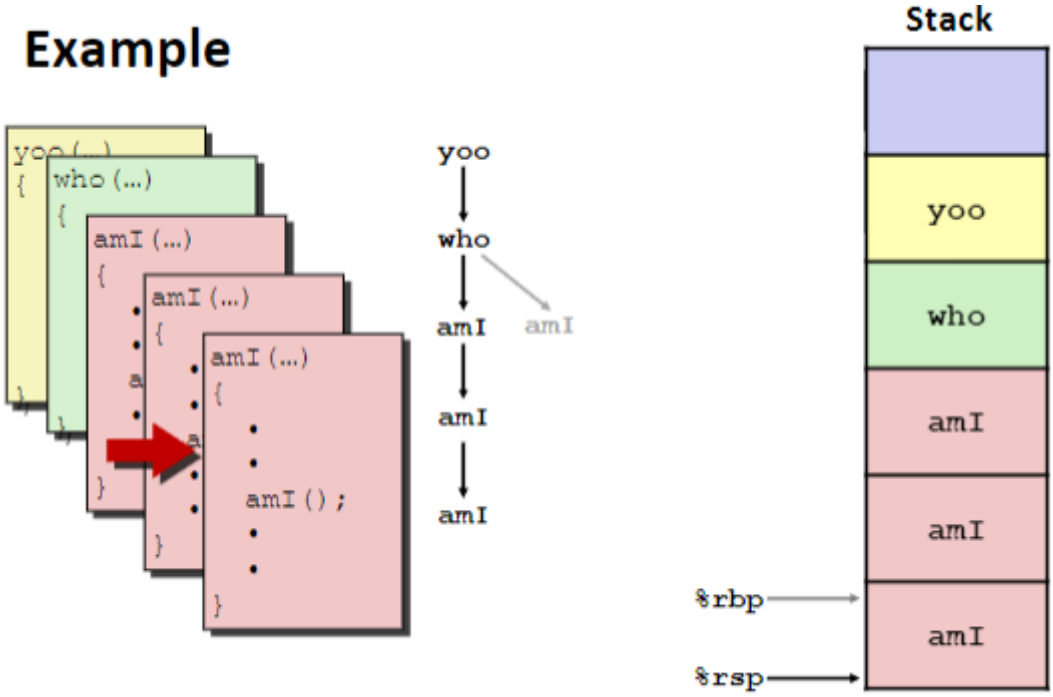
栈帧

定义

栈上的一块连续内存区域，对应了运行一个函数时的运行环境，一块栈帧对应了c语言中的一个运行函数，这个函数运行时存放的数据放在当前函数申请的栈帧中

结构

申请时通过%rbp确定栈底位置，通过%rsp确定栈顶位置，由此确定一块栈内存块



寄存器约定

caller寄存器——调用者保存

速记：后缀为"er"，社会等级高，责任大，需要擦**所有人**屁股

功能：caller寄存器由本体函数声明时，需要在**本体函数中保存自己的值**，传递给子函数时子函数可以**修改**这些值，并且返回原函数时**不用恢复**，由主函数负责恢复或者判断是否需要修改。只能用于临时变量，传递函数参数等

callee寄存器——被调用者保存

速记：由后缀"ee"可知，这些寄存器社会等级低，责任小，只需要擦**自己**屁股

功能：callee寄存器在接收到**主函数传来的参数**时需要在自己的子函数中**保存**这个值，将这个值传回主函数时**需要恢复原值**，可用作存储中间变量，可以跨函数保存数据保护原数据不被修改

第四节——数据的存放

数组

一维数组

1、内存分配：一维数组的内存分配是连续分配Length*sizeof(Type)个空间

eg：int val[5]占用空间为 5*4=20个字节

2、访问方式：数组名等效首元素指针

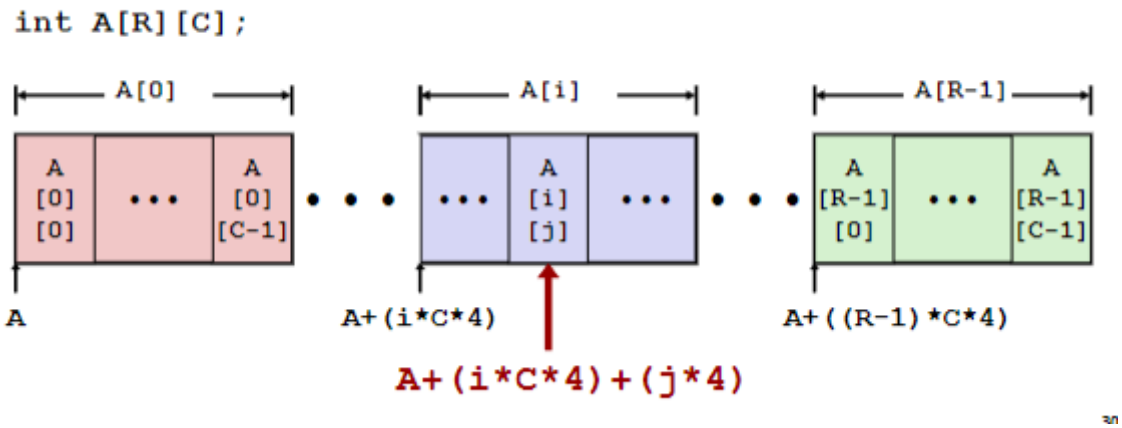
多维数组

1、存储顺序：行优先排列

eg：有数组A [R] [C]

首先根据C，即列数确定第j行的存储单元中有几个块：C*sizeof(Type)

然后再根据行数R，确定第i行的存储位置，每个第i行的元素中有C*sizeof(Type)个空间大小

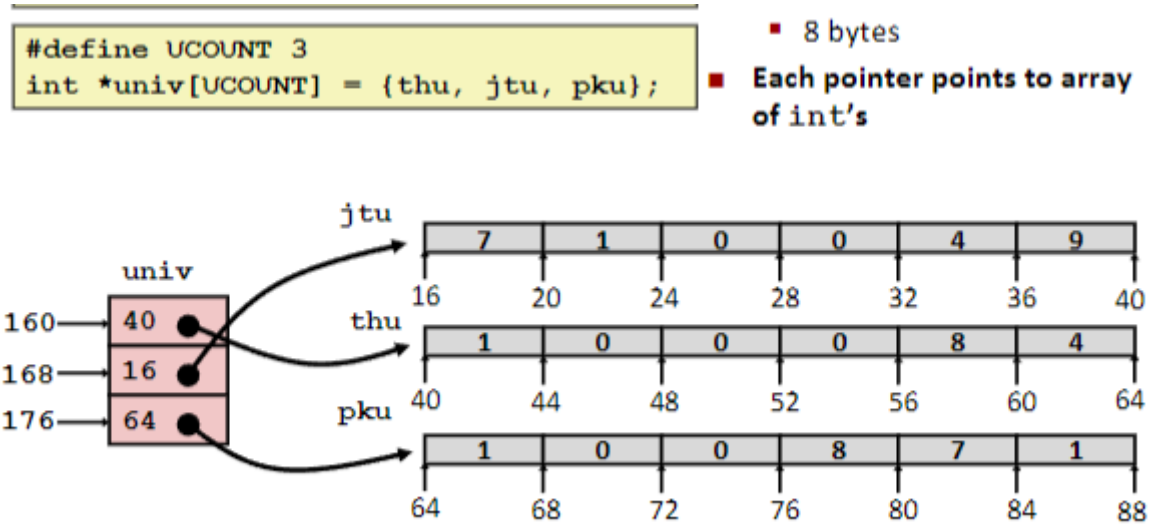


存储地址公式：对于数组int A [R] [C]，中的第[i] [j]个元素

```
address = A + i*(C*sizeof(int)) + j * sizeof(int)
```

多级数组

1、结构特点：每一个数组元素为一个指针，其中每一个指针指向一个数组



2、访问元素

通过两次内存读取访问元素：

eg：

```
int *univ[3];
//通过univ[index][digit]访问第index个指针中的第digit个元素
```

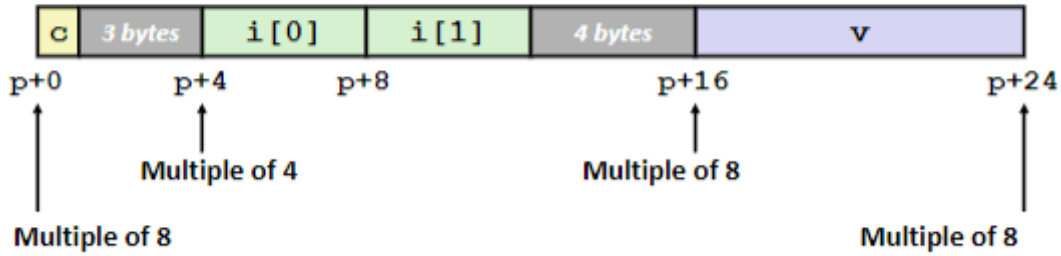
结构体

内存布局

根据声明顺序布局内存

```
struct s1{
    char c;
    int i[2];
    long v;
}*p;
```

内存布局为：



结构体的内存进行分布时需要进行对齐检查

对齐规则

1、结构体内每种类型的字段对齐规则：其地址必须处于其类型大小的整数倍位置处，如上图所示，整型数组大小为4字节，从地址0x4开始，long类型大小为8字节，需要从0x16开始

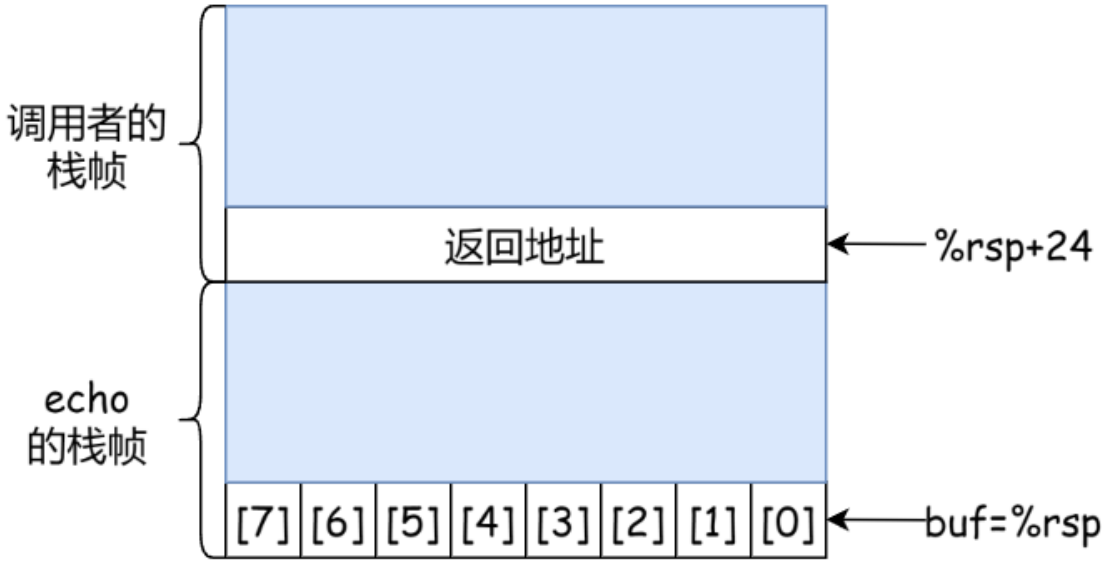
2、结构体自身的内存对齐规则：应为自身所有字段中最大类型字段对齐值的倍数，且能存放结构体中包含的所有类型。图中最大类型字段为8字节，则struct应以24字节对齐

第五节——buffer overflow攻击

常规buffer overflow攻击

原理简介

某些函数(如gets())可以无限制输入字符，这些输入的字符通常存储在栈中，并且从**低地址向高地址存放**。当输入的字符**溢出**了原本申请到的用于存储这个字符串的栈空间，就有可能**改变**栈上原来存放的**返回地址**数据，使得程序执行时返回到特定的地址执行编写好的攻击代码



当输入的字符串长度超过23字节时（末尾会带有“\0”），会改变返回地址中存放的地址

防范措施

1、使用更安全的函数（防止buffer overflow）

2、栈随机化，每次程序运行时的栈都是随机分配地址的

3、使栈中的代码不可执行

4、通过**栈金丝雀**（Stack Canary）：在栈的缓冲区中放置特殊值，通过函数返回前检查这些值是否被改变确定输入是否超界以至于影响到栈中存放的数据

ROP返回导向编程

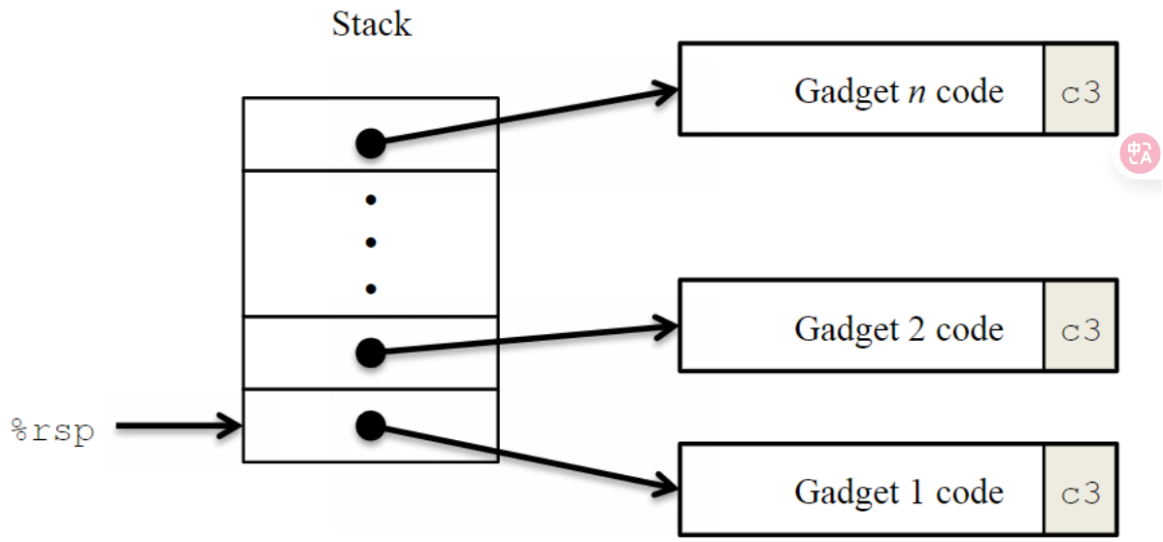
原理简介

1、同样利用输入函数的漏洞，通过输入超过当前存放空间的字符串改变原栈中的数据。

2、overflow的数据存放的是物理地址，这些地址指向的是内存中的文本段空间，不是栈空间，文本段空间**可以执行代码**，并且其**内存地址固定**

3、通过寻找原文本段中以c3（ret指令）结尾的机器代码的地址，将这个地址存放到栈中。在文本段中程序执行到c3时会在栈顶**弹出**下一个保存好的**物理地址**，然后跳转到这个物理地址指向的文本段代码**进行执行**，直到遇到下一个c3，再**跳转到下一个文本段地址执行下一段代码**

4、实际原理是利用已有的程序指令的部分机器代码指令实现ROP。由于机器代码有限，其不同的排列组合方式会对应不同的汇编代码。



存放实际地址　　地址对应的文本段中代码

第六节——Memory Hierarchy内存层次结构

内存层次结构

层次构成

1、寄存器

2、L1缓存（SRAM）

3、L2缓存（SRAM）

4、L3缓存（SRAM）

5、内存（DRAM）

6、磁盘

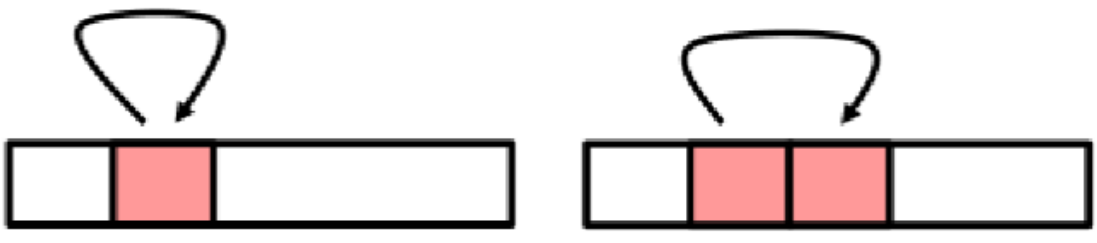
缓存原理

1、时间局部性

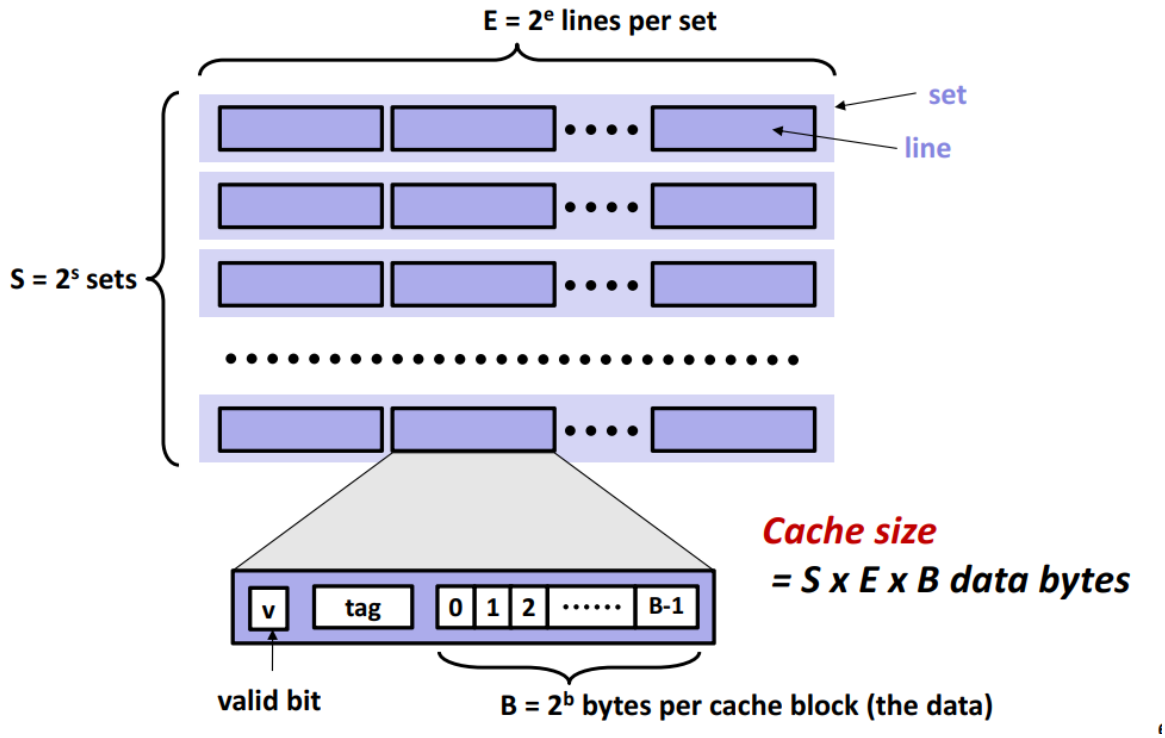
已访问的数据在短期内被再次访问

2、空间局部性

地址相邻的数据更可能被连续访问



高速缓存Cache的结构



❤️ ❤️ ***Cache Line——缓存行*** ❤️ ❤️

定义

Cache Line是Cache中***最小的存储单元***，通常为64字节大小的二进制数据

结构

cache line中存储最基本的cache数据，包括：

数据块Cache Block：实际存储的数据，有b位数据

标签Tag：cache line数据的**高位部分**，用于区分**一个路**中的cache line

有效位Valid Bit：标记这个cache line是否存储有效数据

脏位Dirty Bit：仅在写回策略中标记数据是否被修改

总结：一个cache line的结构大致如下图：

Valid bit（1 bit）	Dirty Bit (1 bit)	Tag (64-2-b bit)	Data Block（b bit）

Set——组

定义

一个组内包含多个cache line

Set		
cache line 1	cache line 2	chche line 3

Way——路

定义

一个组内cache line的数量

cache的关联性

组相联结构

Cache被划分为多个组（Set），每个组N路（即每个组中有N个cache line）

eg：四路组相联

cache line 1_1	**cache line 1_2**	**cache line 1_3**	**cache line 1_4**
cache line 2_1	**cache line 2_2**	**cache line 2_3**	**cache line 2_4**
cache line 3_1	**cache line 3_2**	**cache line 3_3**	**cache line 3_4**
cache line 4_1	**cache line 4_2**	**cache line 4_3**	**cache line 4_4**

全相联结构

整个Cache被划分为一个组Set，所有的cache line属于同一个组

优点：冲突率低，利用率高

缺点：需要并行比较全部标签，成本高

直接映射结构

每一组只有一路，每个组中只有一个cache line

优点：访问速度快，不用并行比较tag

缺点：冲突率高

Cache的访存

CPU对cache进行访存数据时，向cache发送一串二进制数据用于访问不同的cache line和读取数据（位数不确定，根据题目定）

主存数据结构

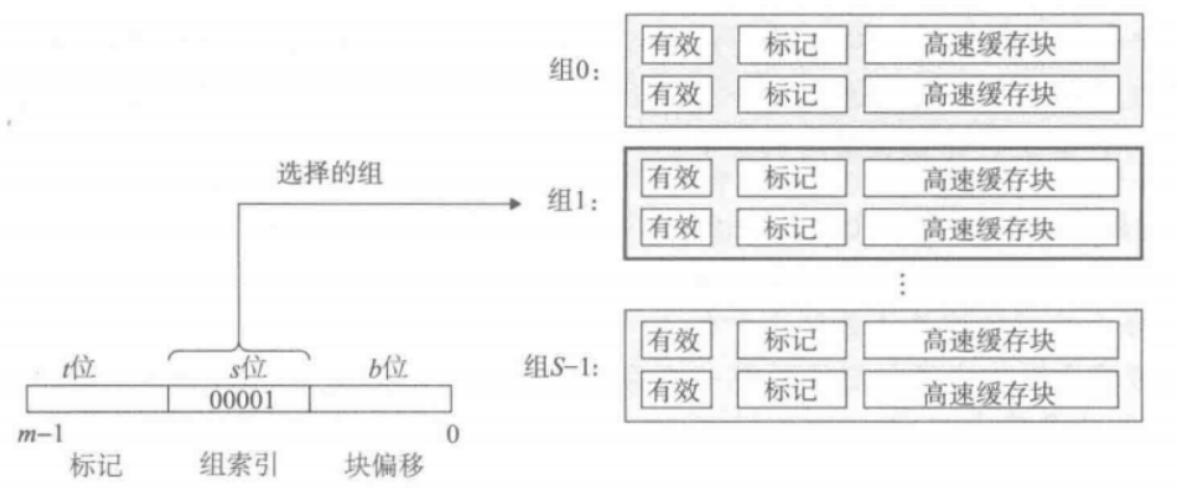
标签 tag	**组索引 set index**	**块内偏移 block offset**

标签 tag

用于与每个cache line中的tag数据标签位进行对比确定在一个组中访存哪一个cache line

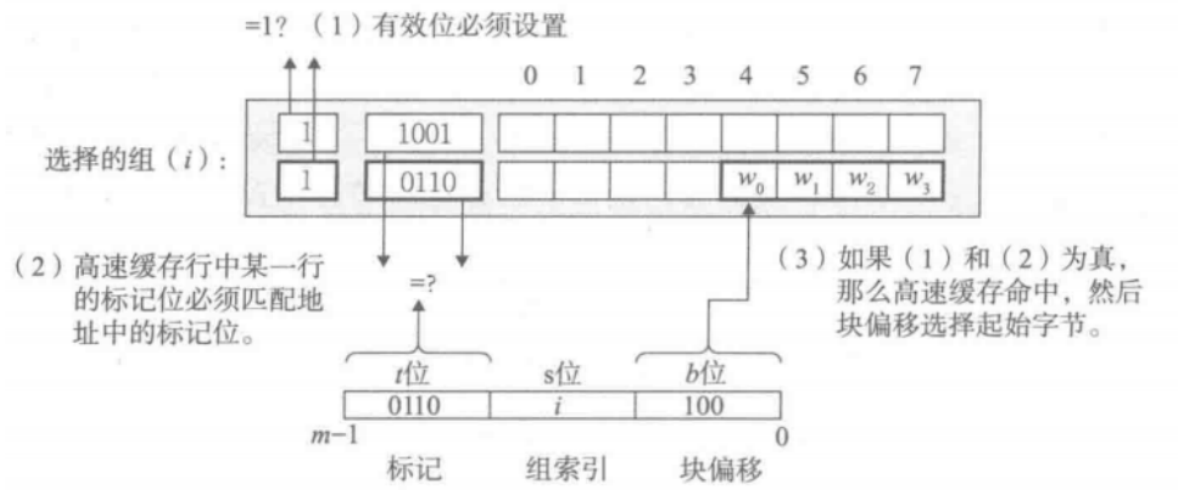
组索引 set index

用于寻找对应的cache的组，在这个组中通过tag寻找对应的cache line



块内偏移 block offset

用于在一个cache line的data block中确定索取数据的起始字节



主存数据结构大小的计算

组索引位数

取决于当前cache有多少个组，若当前cache存在**2^i**个组，则需要**i**位数据

块内偏移位数

取决于每个cache line中数据块data block的大小。若data block有**B=2^b**位，则block offset需要**b**位数据

eg：当块大小（data block）为64字节时，为了能读取到每个字节，需要用6位二进制数据表示每个字节的位置

标签位数

标签位数 = 主存数据位数 — 组索引位数 — 块内偏移位数

Cache大小的计算

若已知**有S个组**，每组的**路数为E**，每个cache line中数据块的大小为**B字节**，则这个Cache的大小为：

$$CacheSize = S * E * B（bytes）字节$$

示例计算：

已知一个cache参数：

地址数据为32位

4路组相联

块大小为64字节

总大小为32KB

解：

（1）根据四路组相联，一个组有四个cache line，则E=4，B=64，根据cache大小计算公式：S * B * E = 32KB = 2^15B

则一共有S=2^7=128组，一共有128 * 4 = 512个cache line

（2）组索引位数：

$$log_2 S = 7$$

所以组索引有7位

（3）block offset位数

由于一个块有64字节大小，为了能准确定位到每一个字节位置，需要一个6位二进制数据——>block offst的位数为6位

（4）tag标签位位数

tag=32-6-7=19位

第七节——代码优化

不依赖机器水平的优化

机器无关优化，包括消除公共子表达式，代码移动，内联函数等

函数调用级的优化

对于要在一个循环中调用一个函数，最好设置一个临时变量

```
void lower(char *s)
{
  size_t i;
  size_t ;
  for (i = 0; i < strlen(s); i++)
    if (s[i] >= 'A' && s[i] <= 'Z')
      s[i] -= ('A' - 'a');
}
```

```
void lower(char *s)
{
  size_t i;
  size_t len = strlen(s);
  for (i = 0; i < len; i++)
    if (s[i] >= 'A' && s[i] <= 'Z')
      s[i] -= ('A' - 'a');
}
```

内存别名

同一个物理内存位置可以被两个名称访问或修改

```
/* Sum rows is of n X n matrix a
   and store in vector b  */
void sum_rows1(double *a, double *b, long n) {
    long i, j;
    for (i = 0; i < n; i++) {
        b[i] = 0;
        for (j = 0; j < n; j++)
            b[i] += a[i*n + j];
    }
}
```

```
/* Sum rows is of n X n matrix a
   and store in vector b  */
void sum_rows1(double *a, double *b, long n) {
    long i, j;
    for (i = 0; i < n; i++) {
        double tmp = 0;
        for (j = 0; j < n; j++)
            temp += a[i*n + j];
        b[i] = tmp;
    }
}
```

依赖机器结构的优化

基础优化——不依赖机器的优化方式

即上文提及的，可以通过：

用临时变量表示函数调用

```
int temp = strlen(a);
for(int i = 0;i<temp;i++)
```

指令集并行优化ILP——循环展开

方法

对一个需要循环n次的循环，对齐进行m阶展开，使得一次循环内可以同步执行m次操作用于减少循环优化次数

原理

2. 优化的原理：每次for循环都是有开销的，打个比方，开一次门放十个作业本肯定比开十次门每次放一个效率高

性能提升

若原指令的CPE为D，则执行m阶展开后的循环理论上**CPE=D/m**

分支预测优化

定义

CPU执行指令时会提前获得后几十个周期的指令，在进行分支时若分支判断错误会损失几十个周期的运行时间，运行错误结果

预测策略与优化方法

启发式预测

向后分支（在汇编中向后跳转的分支一般为循环语句）预测为“取”，向前分支（在汇编中向前跳转的分支一般为if条件判断语句）预测为“不取”

代码转换

1、循环展开，以减少分支次数

2、条件移动

性能优化中的边界

延迟边界

根据指令执行限制的基本CPE，例如整数加法理论上CPE>=1，整数乘法CPE>=3

吞吐量边界

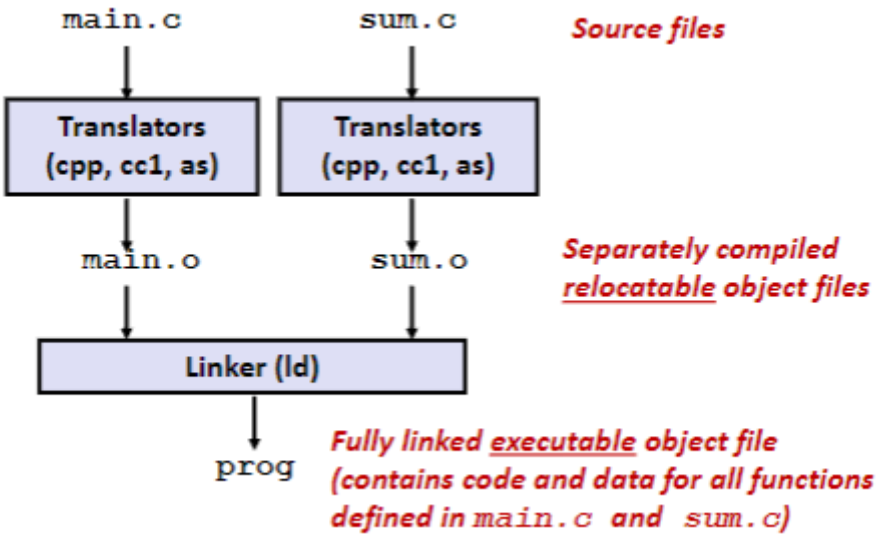
受功能单元限制的最大指令执行速率，例如若CPU含有两个整数加法单元，则其吞吐量边界为CPE>=0.5

第八节——链接 Linker

基本原理

定义

为了提高编程/编译的效率，程序可以拆分为多个源文件（如main.c，sum.c），最终根据linker将众多源文件中共享的地址等相互连接即可形成一个可执行的总文件。



优点

时间效率上的提升

修改单个源文件后只需对单个源文件重新编译和链接，无需重新编译全部的文件，提升了时间效率

空间效率的提升

通过链接在内存中为每个程序提供运行需要的基准代码库，降低了每个程序的大小（每个程序不再需要附带额外的运行代码库，只需要包含程序本体代码即可）节省了空间

链接的基本流程——符号解析+重定位

定义

找到不同文件中定义的符号的地址并存储到**符号表**中。符号包括每个模块的定义和引用符号（函数名，变量名等）

可以进入符号表的符号类型

能被外界（其他文件）看见的符号进符号表

全局符号

在一个文件中声明的**非静态**的C语言**函数**和**全局变量**，可被跨文件调用

外部符号

在本文件中调用的，由其它文件定义声明的全局符号（包括全局变量和非静态函数）

局部符号

用static修饰的**函数**或**全局变量**，仅在本文件中可见

❤️ ❤️ ❤️ ***注：所有的全局/外部/局部都是相对多个文件来说的，全局即所有文件可访问，外部即从其他文件中读取，局部即只有自己的文件可访问*** ❤️ ❤️ ❤️

eg：在文件symbol.c中

```c
int index = 1;                  //index:全局符号，其他文件可调用
static int foo(int a){          //foo:局部符号,其他文件不可调用   a:局部变量不属于任何符号
    int b = a + index;          //b:局部变量不属于任何符号
    return b;
}
int sum(int a,int b){           //sum:全局符号，其他文件可调用
    int summer=0;               //summer:局部变量不属于任何符号
    b+=a;
    return summer;
}
```

在文件main.c中：

```c
int main(){                 //main:全局符号,其他文件可调用
    int m=1;
    int n=2;
    index += sum(n,m);      //index:外部符号 sum:外部符号 从symbol.c中读取的
    printf("%d\n",index);   //printf:外部符号
}
```

符号解析规则

基本知识——强符号与弱符号

强符号：函数和已经定义的全局变量

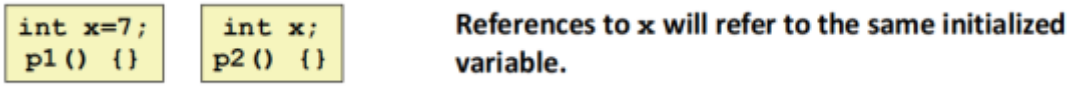
弱符号：未定义的全局变量或使用extern类型声明的符号

规则1——不允许多个强符号

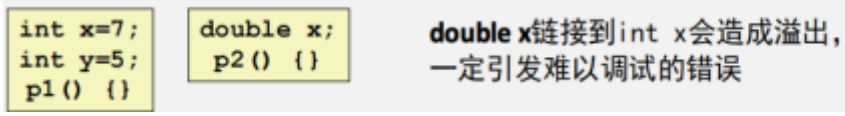
链接时强符号不允许重名

规则2——一强多弱

对于同名的符号，所有同名的弱符号链接到其同名的强符号地址

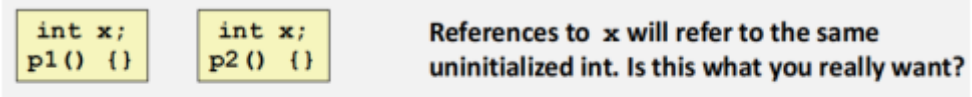


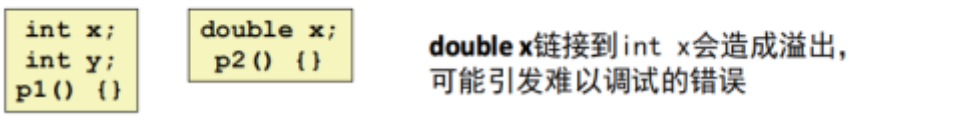
❤️ ❤️ ❤️ ***注：弱符号链接到强符号时必须确保类型匹配，否则会发生溢出*** ❤️ ❤️ ❤️



规则3——多个弱符号任选其一

若一个名称为x的符号全为弱符号，则任选其一进行链接





重定位

合并段与地址分配

将多个可重定位的对象文件的代码和数据段合并，为每个符号分配最终的绝对内存地址

对象文件类型和ELF格式

对象文件类型

类型	拓展名	描述
可重定位对象文件	.o	由单个源文件编译生成，可与其他的.o文件链接形成可执行文件
可执行对象文件	a.out	可直接在内存中执行的二进制文件
共享对象文件	.so	动态链接库，可在加载或运行时动态链接

第九节——进程与多任务

基础知识

进程

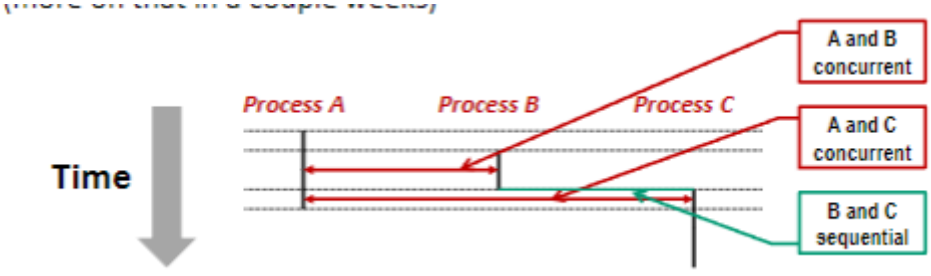
进程是运行程序的示例，进程为程序提供了虚拟内存占用私有地址空间；通过上下文切换实现逻辑控制流，让程序看似独占CPU

多任务

计算机同时运行多个进程，通过时间片轮转实现并发过程

并发过程：两个进程运行时间有重叠

顺序执行过程：两个进程运行时间不重叠



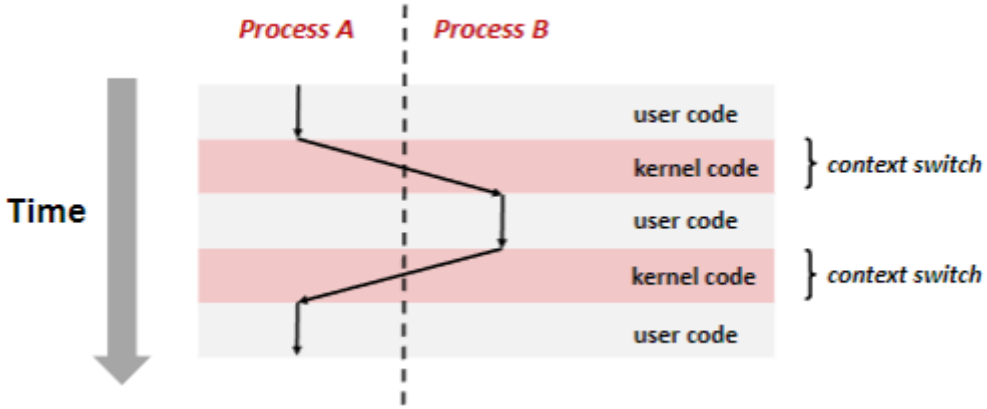
控制流与上下文切换

控制流

CPU从启动到关闭按顺序读取和执行机器指令形成的物理控制流

上下文切换——context switch

通过内核进行管理，内核是驻留在内存中的OS代码，通过内核进行多个程序之间的切换。切换时需要保留原程序切换前寄存器的状态，当完成一次切换循环回到自身时需要恢复寄存器中的值



系统调用

作用

程序在执行自身意外的操作时向内核请求帮助的接口，如open(),fork()等

❤️ ❤️ ❤️ *进程控制* ❤️ ❤️ ❤️

父子进程&创建进程——fork()函数的使用

进程的创建只能通过在**父进程中调用fork()函数**创建子进程

过程

1、通过fork()函数，将父进程的代码和虚拟地址空间副本复制给子进程，

2、随后父进程**返回子进程的PID**

3、**在子进程中返回0**

4、fork()后进入子进程还是父进程是随机的，可以通过**返回值的约束**决定执行的下一段代码位于子进程还是父进程

💯 💯 fork()的进程建模图 💯 💯

点代表一个语句的执行，边代表执行的先后顺序：

1. 代表a先于b执行，点的上下位置相同代表执行同步
2. 可以用变量当前值来标记边，printf顶点可以用输出进行标记
3. 每个图都以一个没有内边的顶点开始
4. 执行一次fork函数会导致一条边叉出来

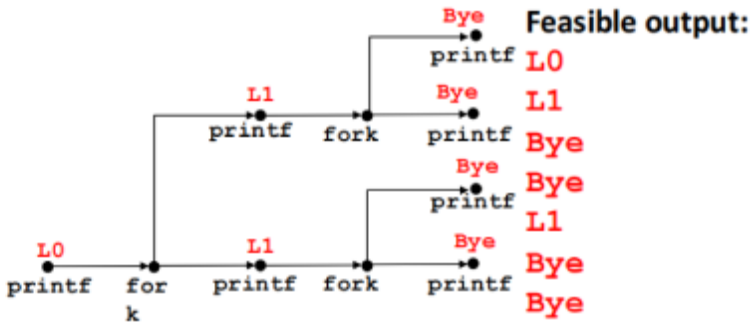
示例：

```c
int main(){
    pid_t pid;
    int x=1;
    pid = fork();
    if(pid == 0){            //fork函数具有两个返回值，返回值pid=0时是子进程
        ptintf("%d\n",x);
        return 0;
    }
    printf("%d\n",++x);      //fork函数的返回值不为0时，进入父进程
    return 0;
}
```

建模图：

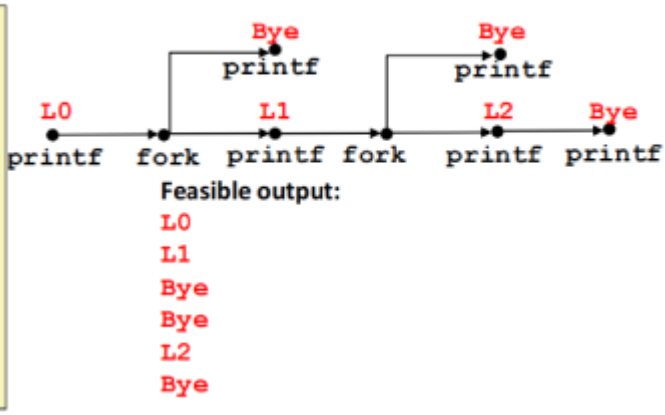
3.3.3. 进程图示例2：连续两个分叉

```c
void fork2()
{
    printf("L0\n");
    fork();
    printf("L1\n");
    fork();
    printf("Bye\n");
}
```
forks.c



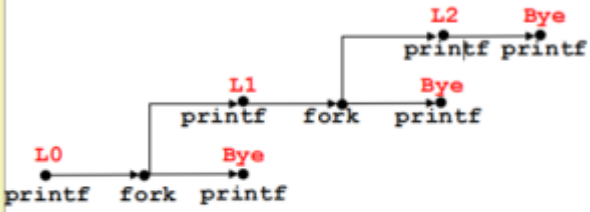
3.3.4. 进程图示例3：父进程嵌套分叉

```c
void fork4()
{
    printf("L0\n");
    if (fork() != 0) {
        printf("L1\n");
        if (fork() != 0) {
            printf("L2\n");
        }
    }
    printf("Bye\n");
}
```
forks.c



3.35进程图示例4：子进程嵌套分叉

```c
void fork5()
{
    printf("L0\n");
    if (fork() == 0) {
        printf("L1\n");
        if (fork() == 0) {
            printf("L2\n");
        }
    }
    printf("Bye\n");
}
```
forks.c



Feasible output:L0
Bye
L1
L2
Bye
Bye

进程ID

每个进程都会被分配一个PID，可以通过两种函数获取这个进程的ID

```c
getpid(void);   //返回当前进程的PID
getppid(void);  //返回父进程的PID
```

进程的状态

Running运行

进程正在CPU中执行指令

Blocked/Sleeping阻塞/睡眠

进程正在等待外部指令激活，无法执行指令

Stopped停止

进程被用户ctrl+z阻止执行

Terminated/Zomble终止/僵尸

进程已经终止，但是父进程没有回收这个子进程

进程的回收

每一个父进程通过fork()函数生成新的子进程时需要负责回收这个子进程。若子进程终止后未被父进程回收会占用系统资源

回收函数

wait()/waitpid()：父进程通过wait()和waitpid()函数获取子进程的终止状态。若父进程没有回收子进程，则由init进程回收

进程的加载

execve()函数

用于加载可执行文件，不返回值，覆盖当前进程的代码，数据和栈。

第十节——异常控制流Exceptional Control Flow

基本概述

异常控制流是响应系统状态变化而改变程序控制流的机制，包括**低层机制**和**高层机制**

Shell程序

shell程序，即命令行终端，功能是读取用户命令，解析命令并执行

低层机制——异常

概念

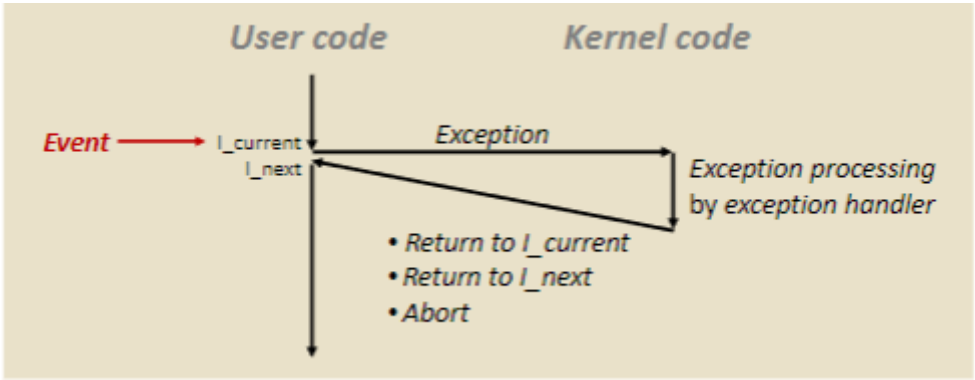
1、响应处理器状态变化，**转移**到内核控制内核的机制，如除零错误，Ctrl-C终止事件等。内核处理这个错误后会**返回错误信息**到主程序中，并存储到异常表中

2、通过**syscall指令**触发

一些syscall指令示例：

Number	Name	Description
0	read	Read file
1	write	Write file
2	open	Open file
3	close	Close file
4	stat	Get info about file
57	fork	Create process
59	execve	Execute a program
60	_exit	Terminate process
62	kill	Send signal to process

机制图：



💯 💯 💯 ***高层异常——信号*** 💯 💯 💯

概念：

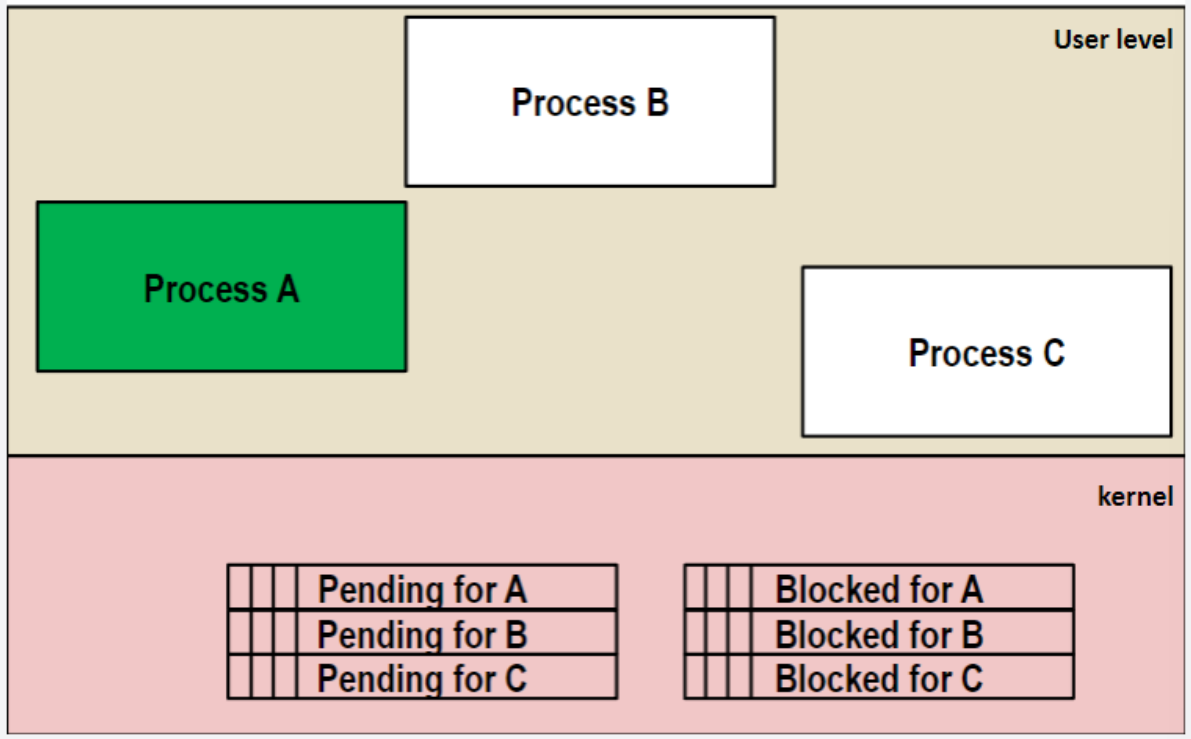
信号是一个用于表示一个进程发生了什么异常事件的标识符，通过1-30的整数表示

例如：

ID	Name	Default Action	Corresponding Event
2	SIGINT	Terminate	User typed ctrl-c
9	SIGKILL	Terminate	Kill program (cannot override or ignore)
11	SIGSEGV	Terminate	Segmentation violation
14	SIGALRM	Terminate	Timer signal
17	SIGCHLD	Ignore	Child stopped or terminated

❤️ ❤️ ❤️ ***信号的发送*** ❤️ ❤️ ❤️

在操作系统内核中存在一个信号发送接收表：



这个表存储两类信号：**Pending信号**和**Blocked信号**

Pending信号表：

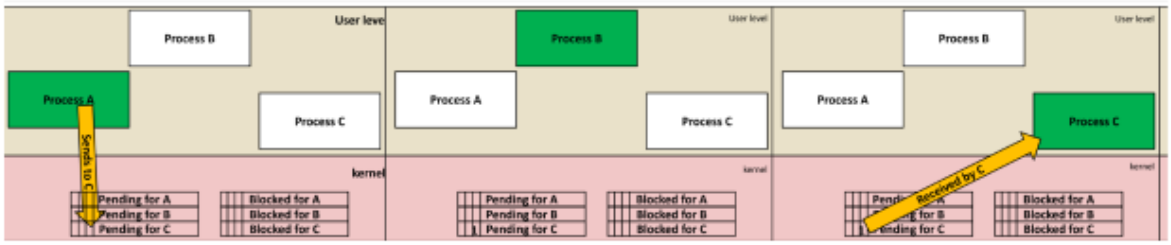
只存储**一位**信息：1/0，用于**标记是否存在其他进程发送来的信号。只能标记一位**（即只能知道其他进程向此进程发送了消息，但不知道谁发送的，不知道是不是多个进程都发送了）

Blocked信号表：

只存储**一位**信息：1/0，用于**标记当前进程是否阻塞接受信号**。1表示当前进程不接受任何信号，0表示可以接收外界信号

注：SIGKILL信号不可阻塞，一旦发生SIGKILL立刻终止进程

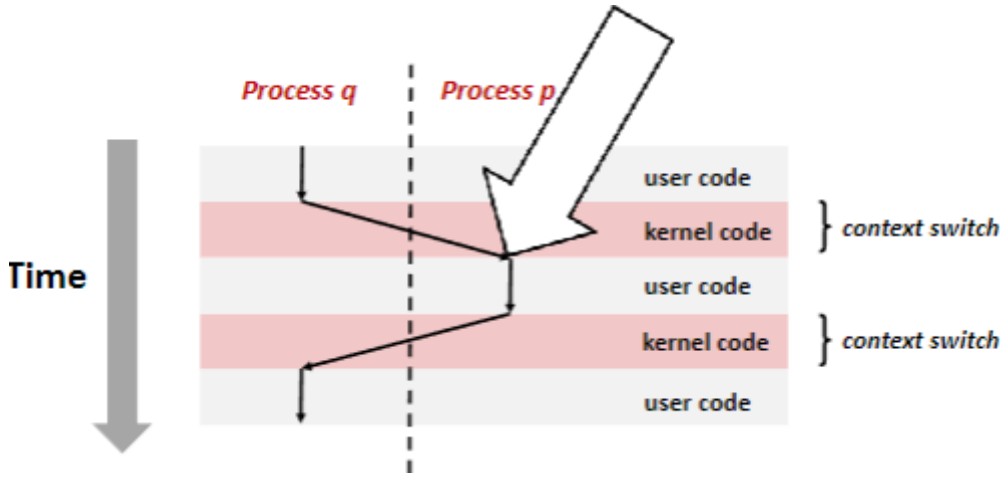
流程图：



信号发送在内存中建立表的作用 😊

操作系统在控制多个进程执行时会让CPU反复跳动执行任意一个进程，而每次跳动都**需要进入一次内核检查信号表——context switch**

示例1：



检查流程：

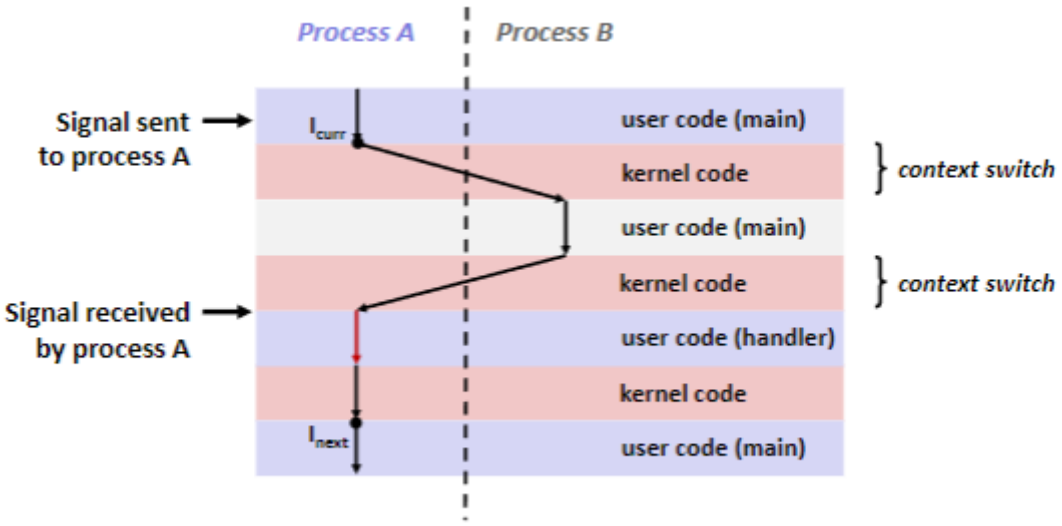
当系统从进程q通过context switch切换到进程p时，首先在内核中进行信号的检查（图中箭头位置）检查进程p是否有未处理的信号：

情况1：进程p存在未处理的信号，内核会优先控制CPU调用信号处理函数处理信号，而不是继续执行进程p的用户代码

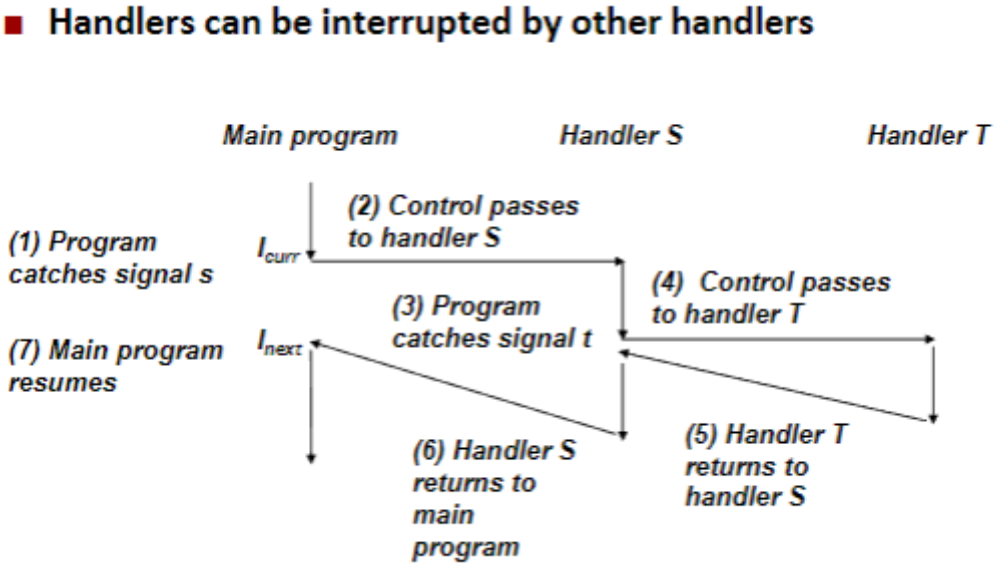
情况2：进程p不存在未处理的信号，则恢复进程p的用户代码执行（如图示情况）

随后再进行context switch跳转回进程q执行相同操作

示例2：在各个进程中编写了信号处理函数（图中紫色模块）



控制流程图：



非局部跳转（类似于时间回溯）

机制

setjmp(jmp_buf j)

设置了一个跳转返回位置，保存这个位置上寄存器的上下文，用于作为longjmp跳转返回的**landmark** 🚩

必须在longjmp()函数之前调用

longjmp(jmp_buf j,val)

用于在程序执行时将进程和代码跳转回到setjmp()设置时的状态（相当于时间回溯 ⏱ ）

必须在设置了setjmp()跳转返回点后调用

功能

用于从深层嵌套函数直接跳转到调用处或者用于错误回复和信号处理

第十一节——虚拟内存

地址空间

线性地址空间

一个有序的连续非负整数地址集合

虚拟地址空间

在64位系统中，理论上每个进程都能分配到一片虚拟地址空间，大小为N = 2^64个字节

物理地址空间

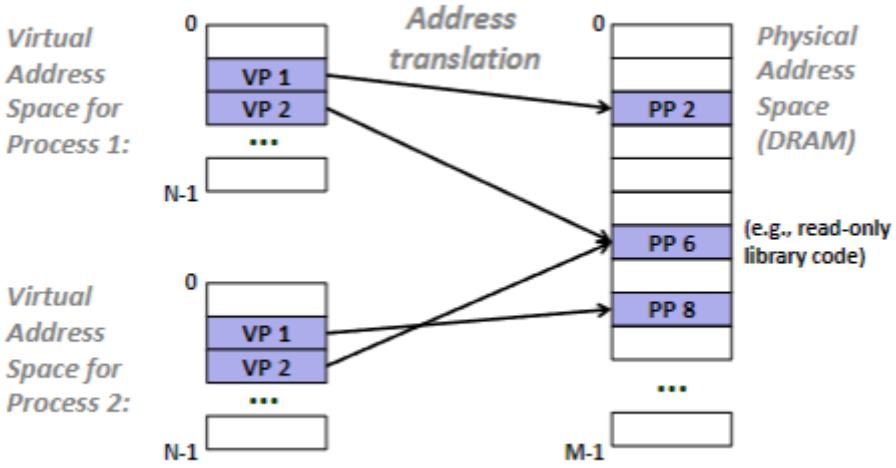
受限于实际物理内存大小，包含M=2^m字节

虚拟内存的角色

作为内存管理工具

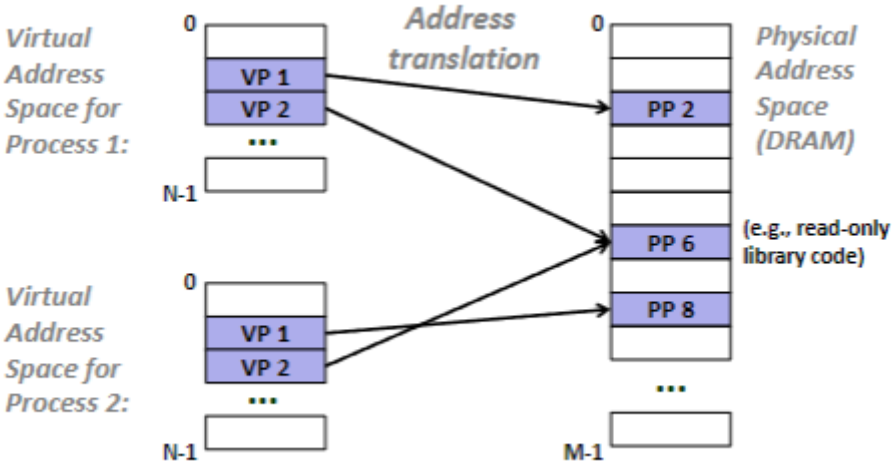
本质作用

为每个进程分配一个完整的独立虚拟内存空间，通过映射函数分散到物理内存地址中



优点

改善了内存中存储数据的空间局部性，简化了链接操作，简化了内存分配，方便内存地址寻址操作，可以实现不同进程将虚拟页映射到同一片物理页共享数据的操作



作为缓存工具

本质

虚拟内存本质上是存储在**硬盘（disk）**上的N字节大小连续数组

内存（DRAM）缓存特点

1、发生miss的惩罚较大

2、全相联结构，只有一组，任意虚拟页可以放入任意物理页

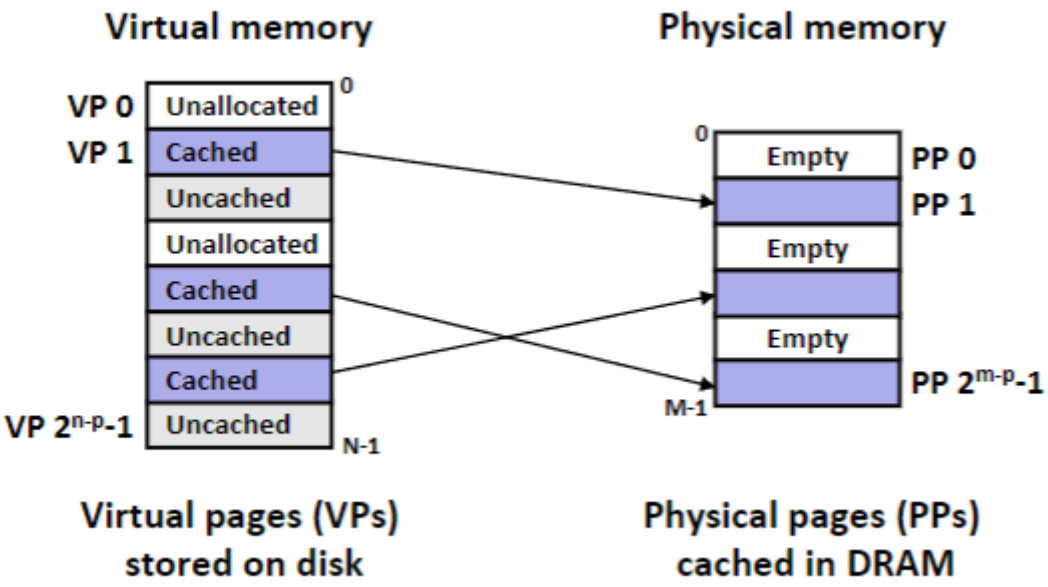
虚拟内存基本结构

页与缓存块

将这个数组的内容按照一个**固定大小划分**后得到的划分块称为**缓存块**，也称为**页**，大小为P=2^p字节，通常为4KB大小

内存缓存模式

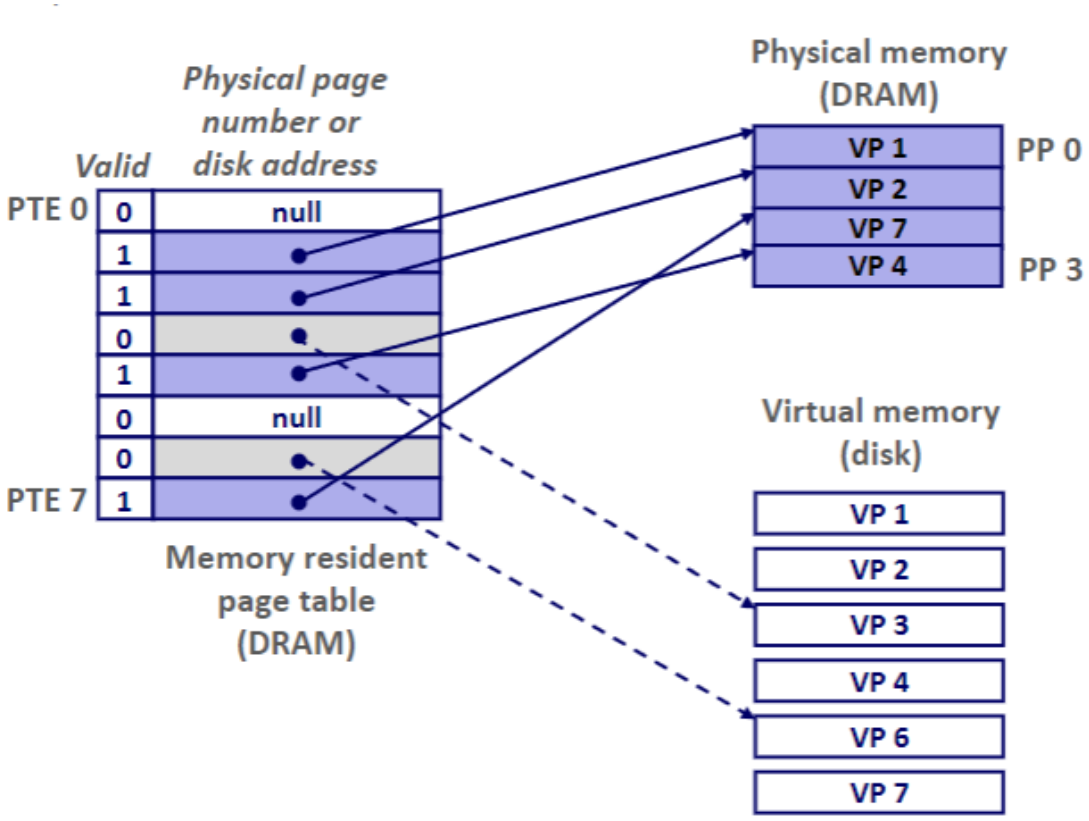
从硬盘缓存数据到内存时，将虚拟内存数组上的某些**页面**拉入物理内存中



图中，每一个分块都是一个页（缓存块）大小为4KB，拉取数据时按照页为基本单位拉取一次性拉取4KB数据存放到内存中

页表

页表是一个存储虚拟页到物理页的映射关系的数组，其中的每个元素为页表项（page table entries (PTE)）存储物理页号，磁盘地址和有效位信息等



图中页表：

1、null表示没有分配对应的虚拟内存给当前的程序

2、灰色PTE表示分配了对应的虚拟内存给当前的程序，但还没有映射到物理内存地址

3、蓝色PTE表示分配了对应的虚拟内存，并且已经映射了确定的物理内存

页表的实现原理

1、每个页表项PTE对应一个虚拟页：第i个页表项对应第i个虚拟页

```
PTE i => VP i
```

2、通过**有效位valid**确定是否为当前PTE中存储的虚拟内存地址映射**物理内存**

0：没有映射

1：映射

3、程序调用虚拟内存时通过访问页表中对应的PTE位置获得实际的物理内存地址

页面的命中与页错误

页面命中

访问的虚拟内存已经缓存在物理内存上了

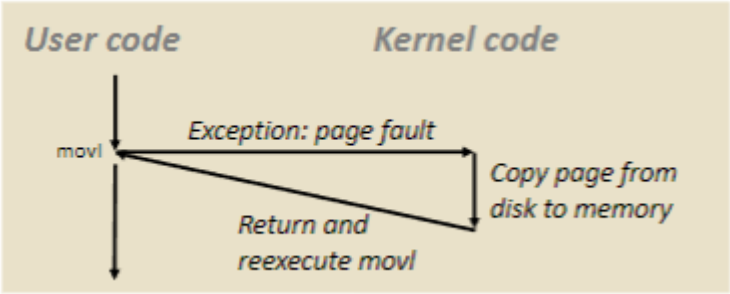
页错误

访问的虚拟内存还没有缓存到物理内存上，触发页错误，需要重新加载对应的虚拟页到物理页上

示例：对于这个函数：

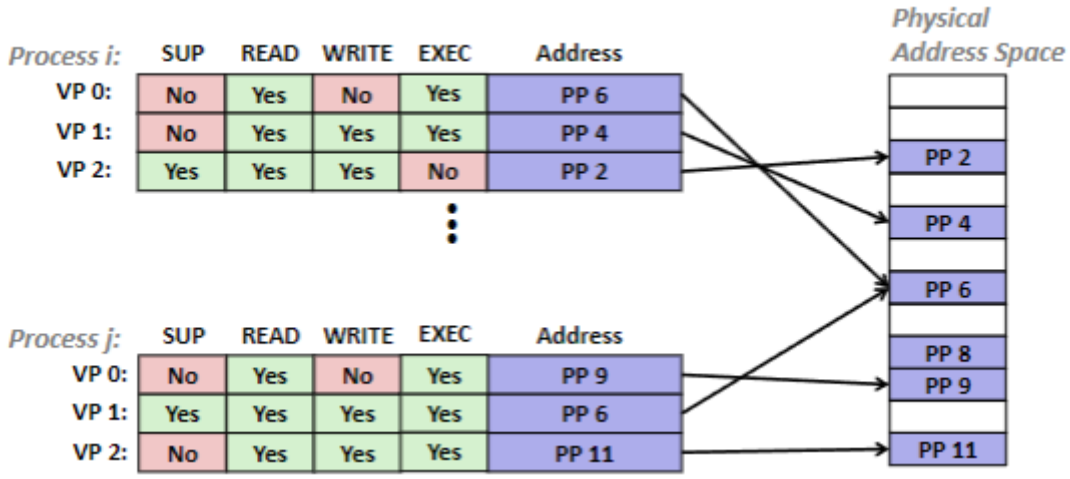
```
int a[1000];
main ()
{
    a[500] = 13;
}
```

处理过程：



虚拟内存用于保护内存数据

可以为虚拟内存在映射时设置条件：可读、可写等，保护实际存储空间中的数据不被修改



虚拟内存的地址转换

地址组成

虚拟地址（VA）

1、**虚拟页号（VPN）**：virtual page number

2、**虚拟页偏移（VPO）**：virtual page offset

虚拟地址实际上是虚拟页号和虚拟页偏移组合而成的：VA=VPN+VPO

物理地址（PA）

1、**物理页号（PPN）**：physical page number

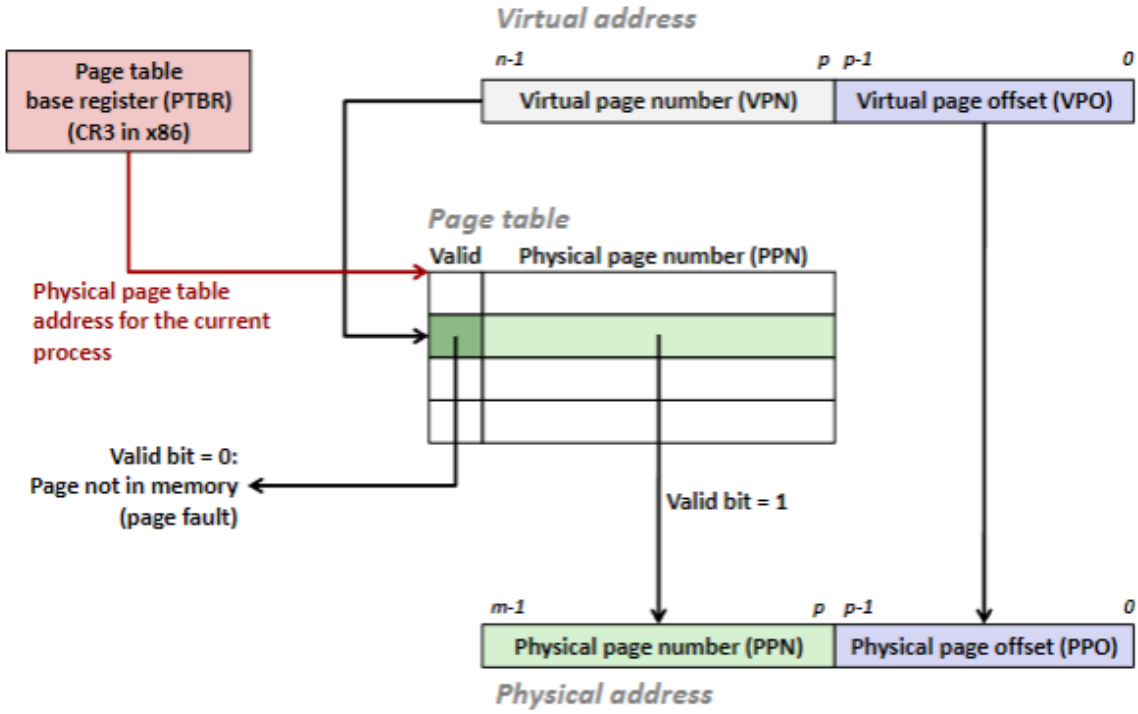
2、**物理页偏移（PPO）**：physical page offset

地址转换原理补充

1、CPU中存在一个页表基址寄存器**PTBR**，通过PTBR可以指向任意一个页表的地址

2、CPU中存在一个内存管理模块**MMU**，用于将CPU发送的虚拟地址转换为物理地址

地址转换原理



1、CPU将需要的虚拟地址发送到MMU上

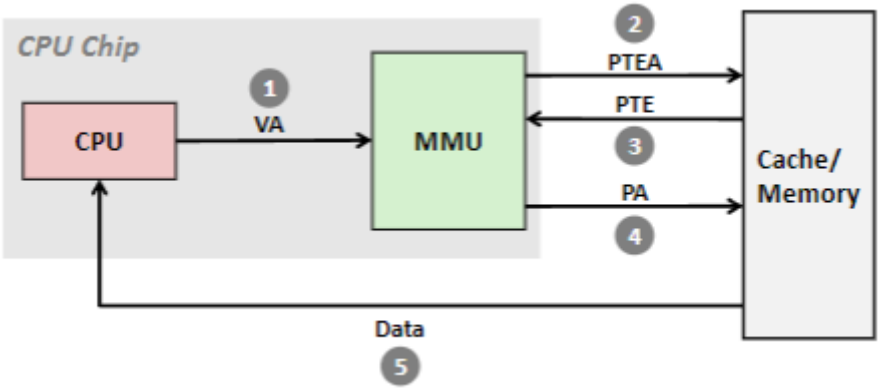
2、MMU中流程如上图，首先根据页表基址寄存器PTBR找到对应页表

3、根据VPN（虚拟页号）找到对应的PTE

4、若对应的PTE有效，则拿出其中存放的PPN（物理页号）和VBO拼起来组合成真正的物理地址

具体示例

页命中：



1、CPU向MMU发送虚拟地址

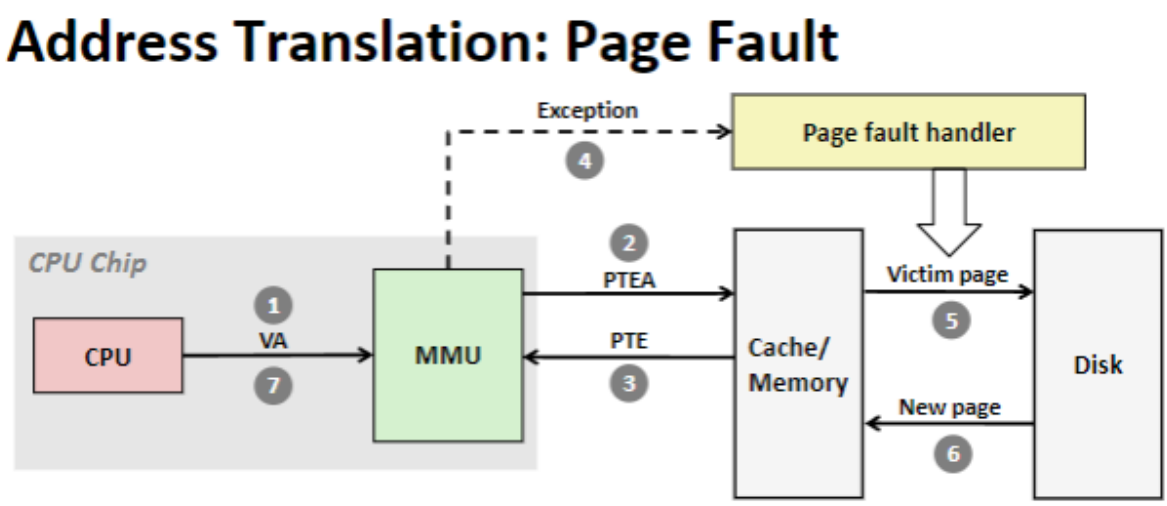
2、MMU生成一个PTE地址，向cache/memory请求取得这个PTE的值

3、这个PTE在cache/memory中命中，向MMU返回PTE的值（即PPN）

4、MMU构造物理地址PA并发送到cache/memory中取值

5、cache/memory将对应PA的值取到后返回给CPU

页错误



1、CPU向MMU发送VA

2、MMU向cache/memory发送PTE地址请求

3、cache/memory向MMU传回的PTE的有效位是0（即页未命中）

4、MMU触发异常，进入缺页异常处理程序

5、缺页异常程序找到物理内存中的一个牺牲页将其丢掉（放回硬盘）

6、缺页处理程序将需要的物理页拉进内存，更新PTE

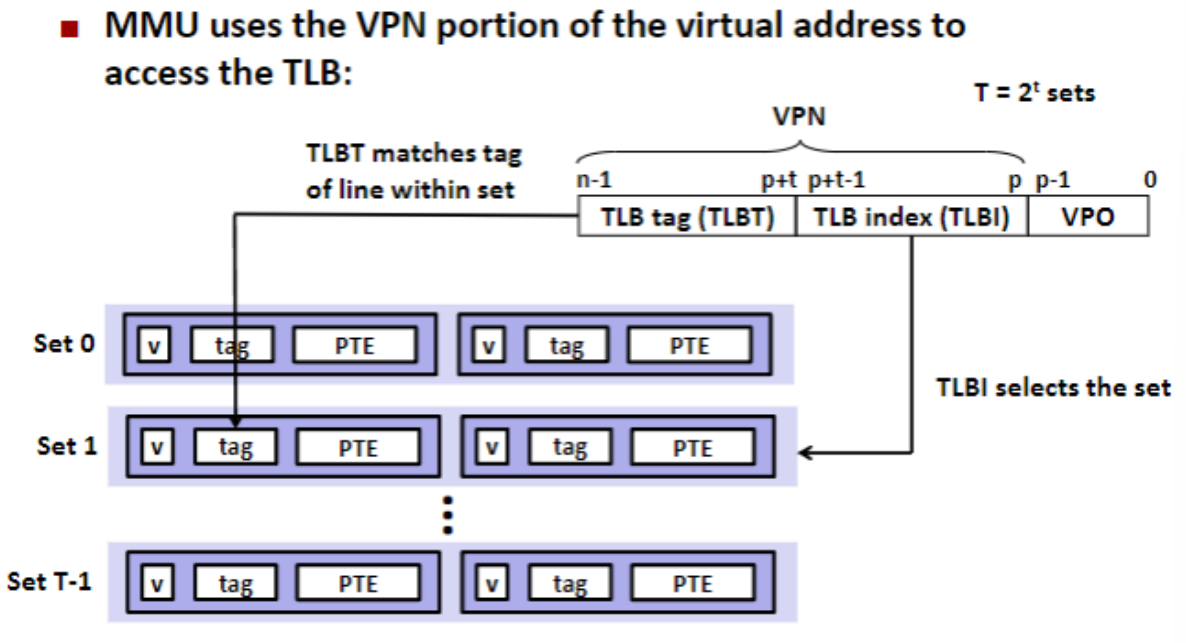
7、缺页处理程序返回原进程，再次执行原指令重新进行一次页命中操作

基于TLB的内存转换加速

TLB是什么

TLB——翻译后备缓冲器，CPU中的一小块超高速缓存，专门用于缓存完整的PTE页表项

结构

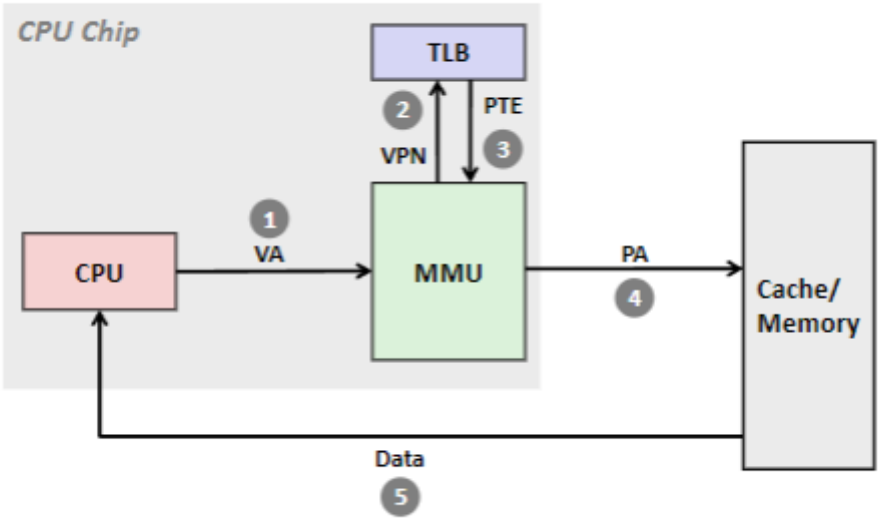


类似于cache，通过对VPN进行划分得到TLB索引TLBI（TLB index，存储组的位置）和TLB标记TLBT（TLB tag，存储标记位tag，用于组内比较tag位确定放在哪一个line中）

若TLB中含有2^t组，则TLBI是由VPN的t个最低位组成的，TLBT由VPN剩余位组成

TLB工作原理

1、TLB命中



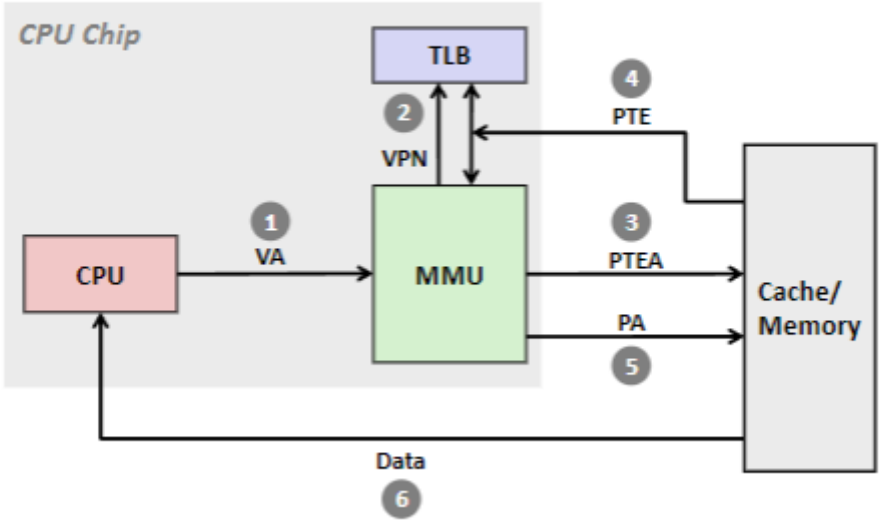
1、TLB中提前存放好了一些PTE，当CPU需要取VA时向MMU发送VA

2、MMU向TLB发送VPN用于寻找对应TLB中是否存放着对应的PTE（此时假设命中）

3、命中后从TLB中快速获取到了PPN，返回到MMU中组合形成PA，向cache/memory发送PA读取数据

4、cache/memory向CPU返回PA的数据

2、TLB miss



1，2步同理

3、由于TLB中VPN没有命中任何一个TLB line（参见cache line miss），MMU触发页表查询，向下级缓存中发送PTE请求

4、cache将PTE数据传回MMU和TLB中

5、MMU得到物理地址PA后向下级缓存发送PA申请

6、下级缓存发送PA对应的数据给CPU

基于多级页表的内存转换加速

使用原因

由于单级页表在较大地址存储空间下占用空间较大：

对于PTE数量：一个32位地址空间的虚拟内存（大小为2^32），若一个虚拟页为4KB=2^12，则需要的PTE数量为2^20（即需要20位数据用于表示PPN）

对于单个PTE大小：单个PTE包含有效位等标志位和PPN（取决于物理内存大小和页面大小）

原理

通过多级页表减少需要的页表空间大小。首页表的PTE中存放指向下一级页表的地址。当某些进程不需要较大虚拟内存地址时，部分低级的页表可以全部置空节省空间（动态分配）

第十二节——动态内存分配

原理详见：[Lab 7: Malloc Lab - Introduction to Computer Systems Spring 2025]

（来不及写笔记了呜呜）

一些这个网站中不包含的计算：

性能指标

吞吐量

单位时间内完成的内存分配malloc和释放free的操作总数

计算方式：

$$\text{吞吐量} = \text{完成的请求总数} / \text{总时间}$$

- **示例**：若 10 秒内完成 5000 次`malloc`和 5000 次`free`，则吞吐量为：

$$\frac{5000+5000}{10} = 1000 \text{ 次/秒}$$

运用隐式空闲链表时malloc时会对空闲块大小进行遍历搜索直到找到合适的空闲块，所以单位时间内能完成的malloc次数较少

空间利用率——内存开销O

用于衡量分配器使用的堆空间中未被程序有效数据利用的比例，反应了内存的浪费程度

计算方式：

$$O = (H_k / max P_i) - 1$$

内存开销=当前堆总大小（已分配块大小 + 空闲块大小）/前k次操作中应用程序有效数据的峰值总和

内存开销越小说明空间利用率越高

内部碎片率

- **内部碎片率**：内部碎片占已分配空间的比例。

$$\text{内部碎片率} = \frac{\sum(\text{块大小} - \text{Payload})}{\sum \text{块大小}}$$