计算机系统期末：李昊部分

- 李昊老师部分
  - 1. 代码优化
    - 1.1. 不依赖于机器水平的优化

      两个优化障碍：函数调用，内存别名
      - 1.1.1. 函数调用级的优化：省得循环一次调用一次

```
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');
}
```

      - 1.1.2. 内存别名(一内存位置可以被两个名称访问)

```
/* 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;
    }
}
```

        1. 按道理来说b[i]在最内层循环，其应该储存在寄存器当中；但实际上b[i]跑到内存当中去了，原因在于b[i]实质是用的指针
        2. 极端情况下如果b[],a[]在内存同一片地方，运行就会对a[]产生副作用，正是这一种情况导致编译器不把b[]放入寄存器
        3. 解决方案是把b[i]变成临时变量(在栈上，不会有此问题)
    - 1.2. 依赖于机器结构的优化：整长数组累加/累乘函数为例

Method	Integer		Double FP	
Operation	Add	Mult	Add	Mult
Combine4	1.27	3.01	3.01	5.01
Unroll 2x1	1.01	3.01	3.01	5.01
Unroll 2x1a	1.01	1.51	1.51	2.51
Unroll 2x2	0.81	1.51	1.51	2.51
Latency Bound	*1.00*	*3.00*	*3.00*	*5.00*
Throughput Bound	*0.50*	*1.00*	*1.00*	*0.50*

- 1.2.1. 基础优化

```
void combine1(vec_ptr v, data_t *dest)
{
    long int i;
    *dest = IDENT;
    for (i = 0; i < vec_length(v); i++) {
       data_t val;
       get_vec_element(v, i, &val);
       *dest = *dest OP val;
    }
}
```

```
void combine4(vec_ptr v, data_t *dest)
{
  long i;
  long length = vec_length(v);
  data_t *d = get_vec_start(v);
  data_t t = IDENT;
  for (i = 0; i < length; i++)
    t = t OP d[i];
  *dest = t;
}
```

1. 把函数调用 `i < vec_length(v)` 和 `get_vec_element(v, i, &val);` 放在循环外
2. 用临时变量代替每次循环中改变的变量

- 1.2.2. 循环展开(减小循环次数)优化：2x1，2x1a，2x2

```
void unroll2a_combine(vec_ptr v, data_t *dest)
{
    long length = vec_length(v);
    long limit = length-1;
    data_t *d = get_vec_start(v);
    data_t x = IDENT;
    long i;
    /* Combine 2 elements at a time */
    for (i = 0; i < limit; i+=2) {
       x = (x OP d[i]) OP d[i+1];
    }
    /* Finish any remaining elements */
    for (; i < length; i++) {
       x = x OP d[i];
    }
    *dest = x;
}
```

```
void unroll2aa_combine(vec_ptr v, data_t *dest)
{
    long length = vec_length(v);
    long limit = length-1;
    data_t *d = get_vec_start(v);
    data_t x = IDENT;
    long i;
    /* Combine 2 elements at a time */
    for (i = 0; i < limit; i+=2) {
       x = x OP (d[i] OP d[i+1]);
    }
    /* Finish any remaining elemen
    for (; i < length; i++) {
       x = x OP d[i];
    }
    *dest = x;
}
```

```
void unroll2a_combine(vec_ptr v, data_t *dest)
{
    long length = vec_length(v);
    long limit = length-1;
    data_t *d = get_vec_start(v);
    data_t x0 = IDENT;
    data_t x1 = IDENT;
    long i;
    /* Combine 2 elements at a time */
    for (i = 0; i < limit; i+=2) {
       x0 = x0 OP d[i];
       x1 = x1 OP d[i+1];
    }
    /* Finish any remaining el
    for (; i < length; i++) {
       x0 = x0 OP d[i];
    }
    *dest = x0 OP x1;
}
```

2x1(循环二阶展开)

1. 每次循环进行两个操作，步长为2，而且最后附带一个额外的循环解决尾巴的问题
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
3. 仅在每次循环任务不重时候有效

2x1a(减小数据依赖)

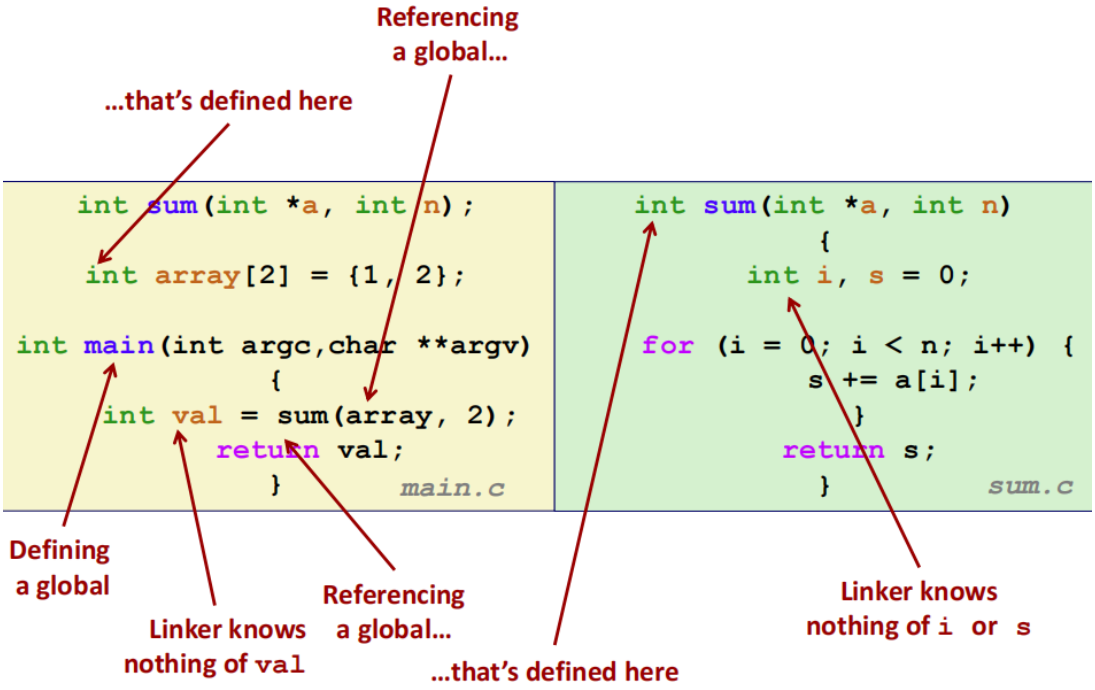
1. 标红部分做出运算顺序的改变，这一操作是带有风险的
2. 性能提升的原因在于：`x = (x OP1 d[i]) OP2 d[i+1]` 中要等OP1完成才能进行OP2，`x = x OP1 (d[i] OP2 d[i+1])` 中两个OP可并行
3. OP为加法时没用

2x2(步长为2，操作掰成两半)

1. x0专门负责累加d[0/2/4....]，x1负责d[1/3/5....]
2. 原理还是数据依赖去除

- 2. 链接

- 2.1. Symbol Resolution(符号解析)：找到不同文件中定义的符号的地址



上图的解析：
array是全局变量，塞进符号表；main是一个函数的名字，也是全局变量；sum也是全局函数
linker不关心局部变量，所以对于val/i/s一无所知

- 2.2. Symbol Identification(符号标识)：哪些变量会进符号表？

symbols.c:

```
int time;

int foo(int a) {
  int b = a + 1;
  return b;
}

int main(int argc,
         char* argv[]) {
  printf("%d\n", foo(5));
  return 0;
}
```

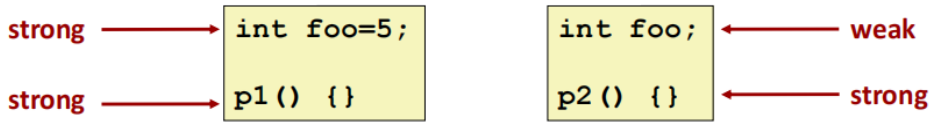
Names:

- time
- foo
- a
- argc
- argv
- b
- main
- printf
  - "%d\n"

能够被外界所看见的变量就是全局变量，就可以进符号表，示例如上(红色进符号表)

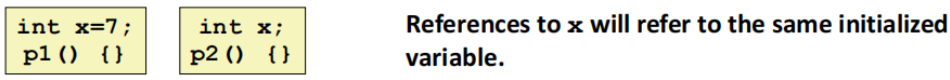
- 2.3. 链接器如何解析重复的符号定义？

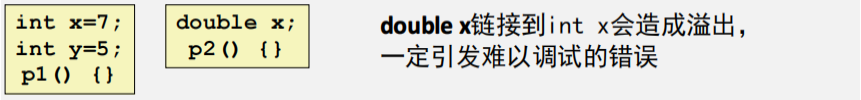
- 2.3.1. 强符号(函数+定义了的全局变量)与弱符号(未定义全局变量)



- 2.3.2. 基于强弱符号的链接规则
  - 注意一点：如果x1链接到x2，那么所有对x2的操作都会变成对x1的操作
  - 2.3.2.1. 多强无弱，强符号不允许重名，否则报错

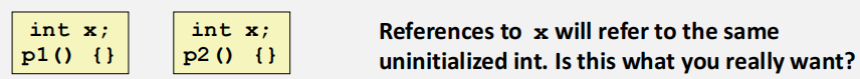
  - 2.3.2.2. 一强多弱，弱的链接到(指向)强的(linker不进行符号类型查询)

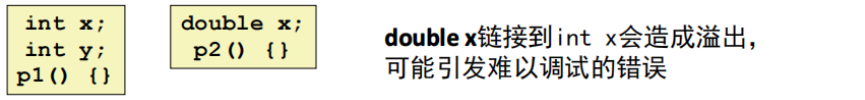




右边文件所有对x的操作，都会转化为对左边文件x地址的操作，但二者操作方法不一样，所以冲突

  - 2.3.2.3. 无强多弱，随机选择一个弱符号，让所有弱符号指向他





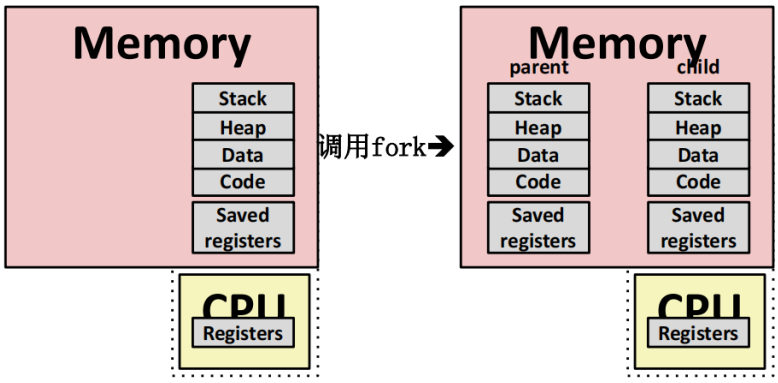
- 3. 进程与多任务
  - 3.1. 父子进&创建进程

1. 建立进程：只能通过让父进程调用函数fork，创建子进程
2. 父子进程几乎一样：子进程获得与父进程相同的虚拟地址空间独立副本，子进程获得父进程打开文件的副本，但二者PID不一样

3. fork完之后先执行子进程还是父进程是随机的，千万不要想当然假设先执行哪个

- 3.2. fork函数概述
  - 3.2.1. 调用一次，返回两次

    1. 在父进程中返回返回子进程的PID(标识进程的唯一ID)
    2. **在子进程中返回0**
    3. 所以我们可以在fork以后，利用返回值约束下一段代码是子进程还是父进程中运行
  - 3.2.2. fork函数原理的可视化(完全复制执行状态→指定谁父谁子→恢复执行)



  - 3.2.3. fork函数示例

```c
int main(int argc, char** argv)
{
    pid_t pid;
    int x = 1;

    pid = Fork();
    if (pid == 0) {  /* Child */
        printf("child : x=%d\n", ++x);
        printf("child : x=%d\n", ++x);
        return 0;
    }

    /* Parent */
    printf("parent: x=%d\n", --x);
    printf("parent: x=%d\n", --x);
    return 0;
}
```

```
linux> ./fork2
parent: x=0
parent: x=-1
child : x=2
child : x=3
```

```
linux> ./fork2
child : x=2
child : x=3
parent: x=0
parent: x=-1
```

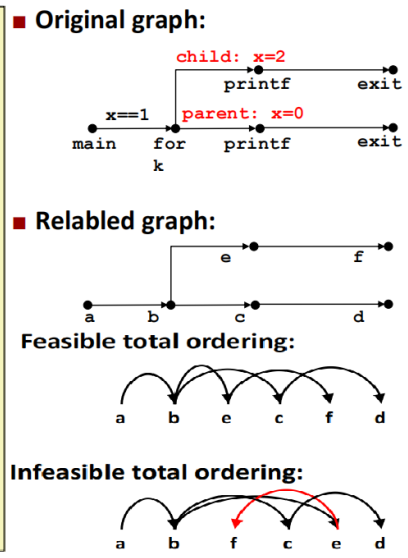
    右边是两种可能的运行结果，这个例子说明：①子进程和父进程中的x相互独立②子进程和父进程执行顺序随机

- 3.3. 基于进程图的fork可视化建模
  - 3.3.1. 进程图：点代一个语句的执行+边代表先后顺序

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

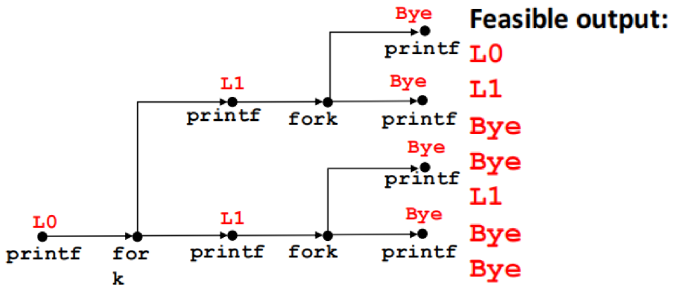
- 3.3.2. 进程图示例1

```
int main(int argc, char** argv)
{
    pid_t pid;
    int x = 1;

    pid = Fork();
    if (pid == 0) {  /* Child */
        printf("child : x=%d\n", ++x);
        printf("child : x=%d\n", ++x);
        return 0;
    }

    /* Parent */
    printf("parent: x=%d\n", --x);
    printf("parent: x=%d\n", --x);
    return 0;
}
```



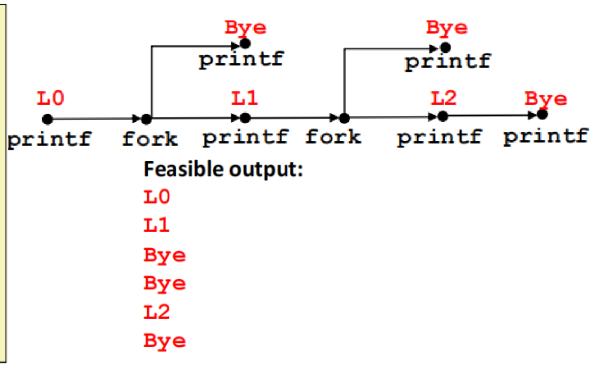
- 3.3.3. 进程图示例2：连续两个分叉

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



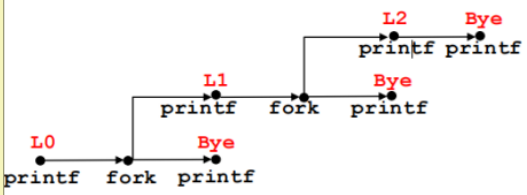
- 3.3.4. 进程图示例3：父进程嵌套分叉

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



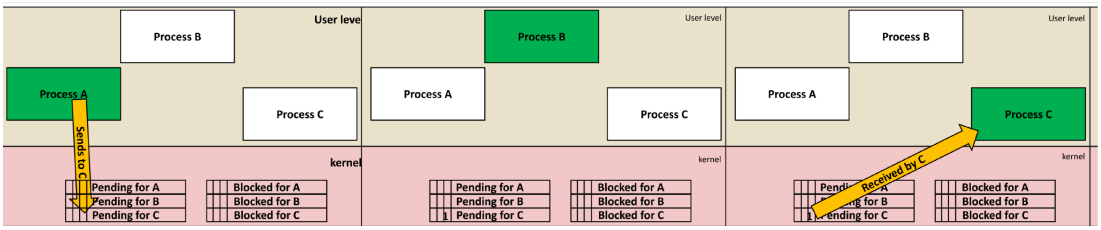
- 3.3.5. 进程图示例4：子进程嵌套分叉

```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

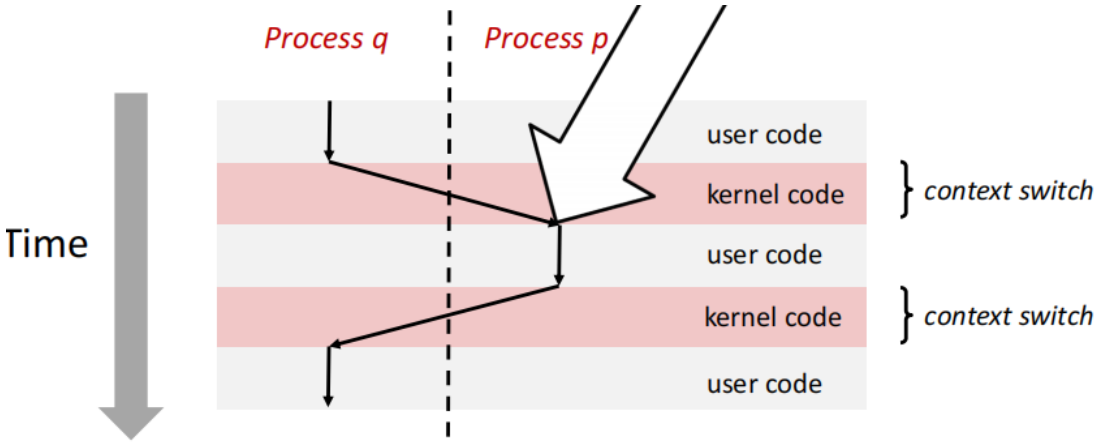
- 4. 异常控制流：信号，使两个进程互相通信
  - 4.1. 信号的发送



kernel
存放了一个signal表，用来表示哪个进程收到了信号与否，或者哪些信号是被封锁了的
有一位1/0表示：信号pending/反之

上图过程：A进程给C进程发送信息的过程
1. 来到A进程，A进程进入kernel修改C进程信号表(Pending C改变)， 然后锁定C进程信号(Blocked C置1)，但进程C不会立即收到信号
2. 来到B进程，由于锁定，B进程无法改变kernel中C进程信号表
3. 来到C进程，解除kernel中C进程信号的封锁(Blocked C置0)，接收到A进程发送的信息，C进程发生相应改变

  - 4.2. 信号的接收：异步



在进程q进行的时候，发送信号
在context switch的时候，检测kernel：有信号进入进程q处理，没有信号就回来

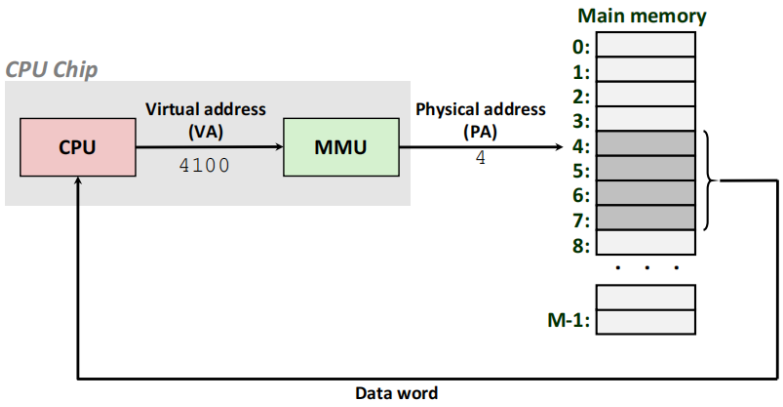
- 4.3. 经典示例

```
volatile int ccount = 0;
void child_handler(int sig) {
    int olderrno = errno;
    pid_t pid;
    if ((pid = wait(NULL)) < 0)
        Sio_error("wait error");
    ccount--;
    Sio_puts("Handler reaped child ");
    Sio_putl((long)pid);
    Sio_puts(" \n");
    sleep(1);
    errno = olderrno;
}

void fork14() {
    pid_t pid[N];
    int i;
    ccount = N;
    Signal(SIGCHLD, child_handler);

    for (i = 0; i < N; i++) {
        if ((pid[i] = Fork()) == 0) {
            Sleep(1);
            exit(0);  /* Child exits */
        }
    }
    while (ccount > 0) /* Parent spins */
        ;
}
```

child_handler 是一个信号处理程序。当一个子进程结束时，它会向其父进程发送SIGCHLD信号

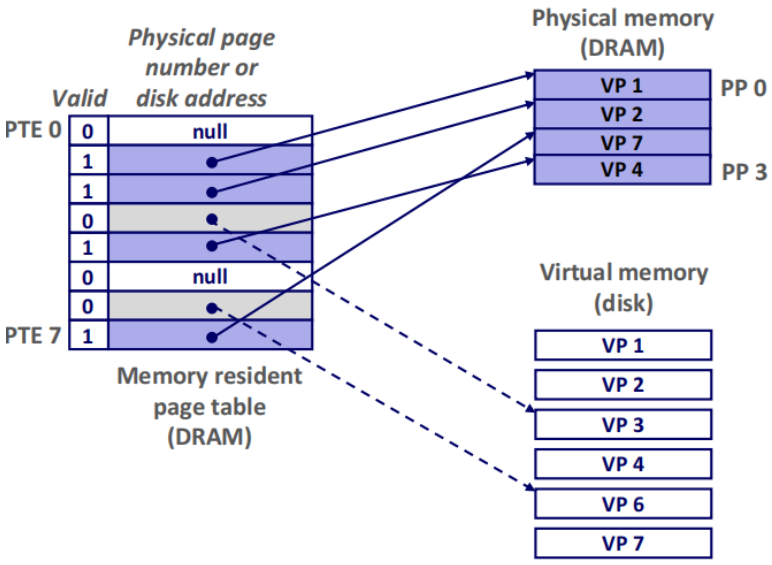
fork14 函数包括了创建子进程和处理子进程结束的主要逻辑
①ccount=N，表示创建N个子进程
②调用Signal函数将 SIGCHLD 信号的处理程序设置为child_handler。当一个子进程结束时，child_handler 就会被调用。
③进入循环，创建N个子进程。每个子进程休眠1秒然后结束。
④父进程进入一个无限循环，即所有的子进程都已经结束，信号处理程序都已经被调用。

- 5. 虚拟内存
  - 5.1. 使用虚拟寻址的系统：优化内存管理+多进程(隔离)



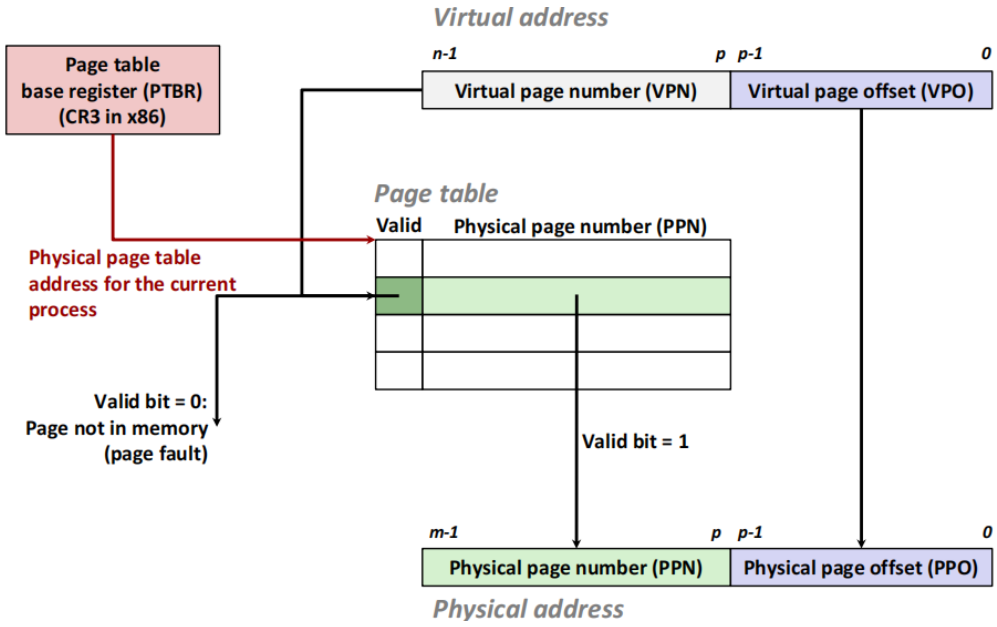
1. 在进程中看到的所有地址都是虚拟地址
2. 通过CPU内的MMU将虚拟地址转化为物理地址，然后访问
3. DRAM：虚拟内存系统缓存，在主存中缓存虚拟页

- 5.2. 实现VM的DS：页表——页表条目的数组，将虚拟页映射到物理页



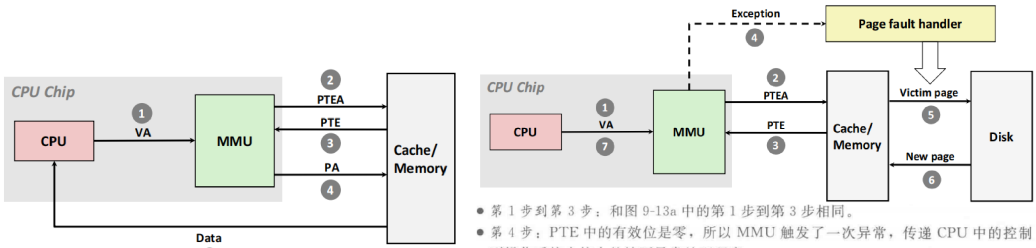
1. 内存分页：把内存分为好多页，实现一堆地址到一堆地址的映射
2. 页表原理：VM中每一页都有个固定PTE在页表，有效位为1表示虚拟页被存在DRAM/磁盘中
3. 如上示例：8虚拟页/4物理页，四个虚拟页在DRAM中，两个未分配，两个分配了但未缓存

- 5.3. 基于页表的地址转换
  - 5.3.1. 虚拟地址转化为物理地址



1. 原理补充：PTRB指向当前页表，MMU利用VPN来选择PTE然后得到PPN，VPO和PPO是相同的
2. 过程当有一个虚拟地址时，用PTBR找到对应页表，然后把VPN扔进去由此找到PPN，再和VPO拼起来

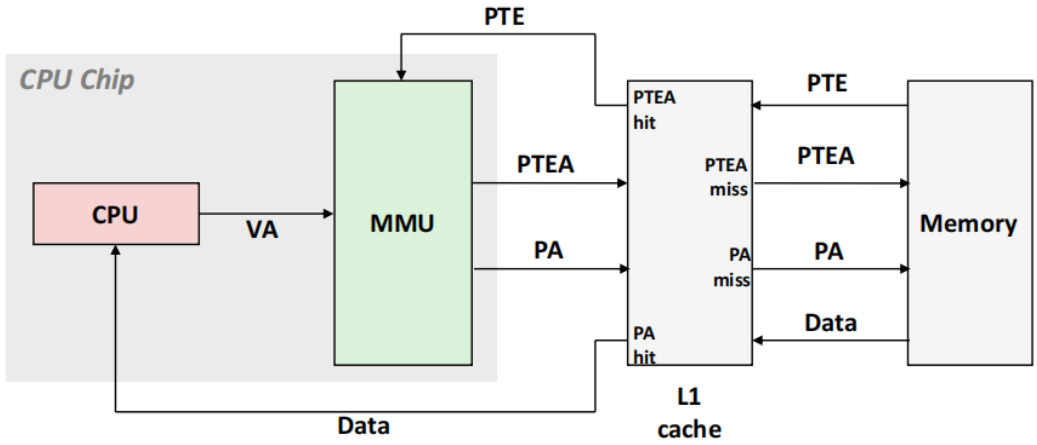
- 5.3.2. 页命中(左)and不命中(右)

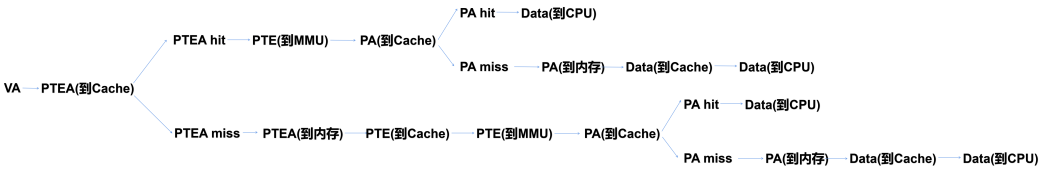


- 第 1 步：处理器生成一个虚拟地址，并把它传送给 MMU。
- 第 2 步：MMU 生成 PTE 地址，并从高速缓存/主存请求得到它。
- 第 3 步：高速缓存/主存向 MMU 返回 PTE。
- 第 4 步：MMU 构造物理地址，并把它传送给高速缓存/主存。
- 第 5 步：高速缓存/主存返回所请求的数据字给处理器。

- 第 1 步到第 3 步：和图 9-13a 中的第 1 步到第 3 步相同。
- 第 4 步：PTE 中的有效位是零，所以 MMU 触发了一次异常，传递 CPU 中的控制到操作系统内核中的缺页异常处理程序。
- 第 5 步：缺页处理程序确定出物理内存中的牺牲页，如果这个页面已经被修改了，则把它换出到磁盘。
- 第 6 步：缺页处理程序页面调入新的页面，并更新内存中的 PTE。
- 第 7 步：缺页处理程序返回到原来的进程，再次执行导致缺页的指令。CPU 将引起缺页的虚拟地址重新发送给 MMU。因为虚拟页面现在缓存在物理内存中，所以就会命中，在 MMU 执行了图 9-13b 中的步骤之后，主存就会将所请求字返回给处理器。

- 5.3.3. 结合高速缓存和虚拟内存(看图不难理解)



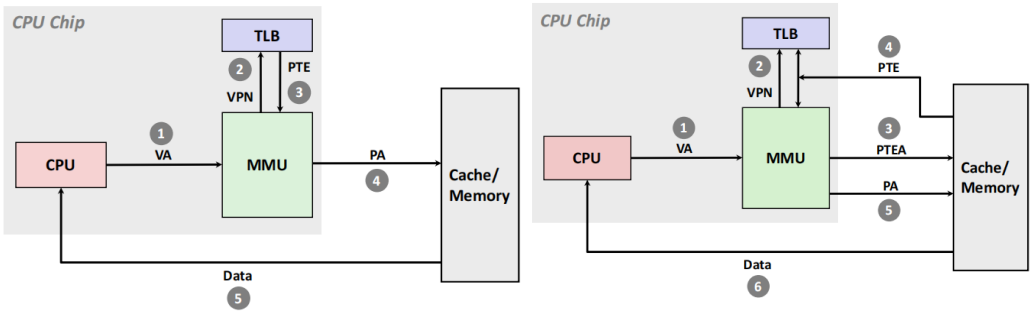


VA:虚拟地址。PTEA:页表条目地址。PTE:页表条目。PA:物理地址。

- 5.4. 基于TLB(翻译后备缓冲器)的内存转化加速
  - 5.4.1. TLB是什么

在CPU中的一小块超高速缓存，专么用来缓存PTE
以上图是TLB一行的结构
若TLB有T=2^t个组，那么TLB索引(TLBI)是由VPN的t个最低位组成的，而TLB标记(TLBT)是由VPN中剩余的位组成的

  - 5.4.2. TLB命中(左)/不命中(右)：TLB不命中时再向Cache请求PTE



以上内容整理于 幕布文档