RAM(Random Access Memory) 断电，原本的信息将被清空，也就是临时存储

ROM(Read only Memory) 只读存储器，支支持读，不支持写

PageCache 页高速缓存区

DMA(Direct Memory Access, 直接存储器访问)它允许不速度的硬件装置来沟通，而不需要依赖于CPU大量终端负载。否则，CPU需要从来源把每一片段的资料复制到暂存器，然后把他们再次写回到新的地方，在这个事件中，CPU对于其他的工作来说就是无法使用的。

**一． Linux中创建一个进程的过程：**

1. fork()一个父进程，拷贝包括其中的堆，栈，静态区，常量区，代码区。该子进程处理进程号与父进程不同，其他的与父进程均相同。

Fork返回值：负值：创建子进程失败

零：返回到新创建的子进程

正值：返回父母或来电者，该值包含新创建子进程的进程ID.

1. exec（）函数族的作用：重新拷贝一份新的代码到该进程的代码区，从而能执行自己的任务，注意，原代码区的数据均被覆盖。

|  |  |  |
| --- | --- | --- |
| 功能 | 在进程中加载新的程序文件或者脚本，覆盖原有代码，重新运行 | |
| 头文件 | #include<unistd.h> | |
| 原型 | int execl(const  char \*pathname, const char \*arg0,  .../); int execv(const  char  \*pathname, char \*const  \*argv[]);　　int execle(const char \*pathname, const char \*arg(), ....);　　int execve(const char  \*pathname,  char \*const argv[],  char \*const envp[]);  int execlp(const  char \*filename, con) | |
| 参数 | Path | 即将被加载执行的ELF文件或脚本路径 |
| file | 即将被加载执行的ELF文件或脚本名字 |
| arg | 以列表方式罗列的ELF文件或脚本参数 |
| argv | 以数组方式组织的ELF文件或脚本的参数 |
| envp | 用户自定义的环境变量数组 |
| 返回值 | 成功 | 不返回 |
| 失败 | -1 |
| 备注 | 1. 函数名带字母l (如：execl()) 意味着其参数以列表的方式提供。 2. 函数名带字母v (如：execv()） 意味着其参数以矢量（vector）数组的方式提供。 3. 函数名带字母p（如：execlp()） 意味着会利用环境变量PATH来找寻指定的执行文件 4. 函数名带字母e（如：execle(); execve()） 意味着用户梯控自定义的环境变量 | |

Tab.1 函数族exec()的接口规范

代码示例：

#include<unistd.h>

#include<stdio.h>

int main(){

    printf("[%d]\n",\_\_LINE\_\_);

    pid\_t pid = fork();

    if(pid > 0){

        printf("(Parents) PID:%d PPID:%d\n",getpid(),getppid());

    }else if(pid == 0){

        char \*arg[] = {"./bin/demo",NULL};

        execv("./bin/demo",arg);  //后两个表示参数

    }

}

**二．缓冲（cache）和缓存(buffer)的区别**

用户进程通过系统调用访问系统资源的时候，需要切换到**内核态**，而这对应一些特殊的堆栈和内存环境，必须在系统调用前建立好。而在系统调用结束后，CPU会从核心模式切回到用户模式，而堆栈又必须恢复成用户进程的上下文。而这种切换就会有大量的耗时。

**用户进程缓冲区(buffer)：**

你看一些程序在读取文件时，会先申请一块内存数组，称为buffer，然后每次调用read，读取设定字节长度的数据，写入buffer。（用较小的次数填满buffer）。之后的程序都是从buffer中获取数据，当buffer使用完后，在进行下一次调用，填充buffer。

所以说：**用户缓冲区的目的是为了减少系统调用次数，从而降低操作系统在用户态与核心态切换所耗费的时间。除了在进程中设计缓冲区，内核也有自己的缓冲区（buffer）**。

**内核缓冲区（buffer）：**

当一个用户进程要从磁盘读取数据时，内核一般不直接读磁盘，而是将内核缓冲区中的数据复制到进程缓冲区中。但若是内核缓冲区中没有数据，内核会把对数据块的请求，加入到请求队列，然后把进程挂起，为其它进程提供服务。等到数据已经读取到内核缓冲区时，把内核缓冲区中的数据读取到用户进程中，才会通知进程，当然不同的io模型，在调度和使用内核缓冲区的方式上有所不同。

你可以认为，read是把数据从内核缓冲区复制到进程缓冲区。write是把进程缓冲区复制到内核缓冲区。当然，write并不一定导致内核的写动作，比如os可能会把内核缓冲区的数据积累到一定量后，再一次写入。这也就是为什么断电有时会导致数据丢失。所以说内核缓冲区，是为了在OS级别，提高磁盘IO效率，优化磁盘写操作。

**内核缓存区（cache）：**cache叫做高速缓冲存储器，是介于中央处理器和主存储器之间的高速小容量存储器，他的实现主要利用了程序的时间局部性和空间局部性。

**时间局部性：**如果执行了程序中的某条指令，那么不久后这条指令很有可能再次执行；如果某个数据被访问过，不久之后该数据很可能再次被访问。（程序中存在大量的循环）

**空间局部性：**一旦程序访问了某个存储单元，在不久之后，其附近的存储单元也很有可能被访问（因为横夺数据在内存中都是连续存放的，并且程序的指令也是顺序的在内存中存放的）

注意**内核缓存区**（cache）和**内核缓冲区**（buffer）的区别：内核缓存区（cache）是位于CPU与内存之间的临时存储器，因为CPU的计算速度要比内存的读写速度块很多，而把这些可能会被重复访问到的数据存储与CPU缓存中，就会提高读取速度。而内核缓冲区任然是存在内存上的。

**虚拟内存（cache）**

虚拟内存是计算机系统内存管理的一种技术。它使得应用程序认为它拥有连续可用的内存（一个连续完整的地址空间），而实际上，它通常是被分隔成多个物理内存碎片，还有部分暂时存储在外部磁盘存储器上，在需要时进行数据交换。

**一级页表：**

（以32位操作系统为例）比如对于4G的空间，4K页, 那需要2^20 == 1M个页标项

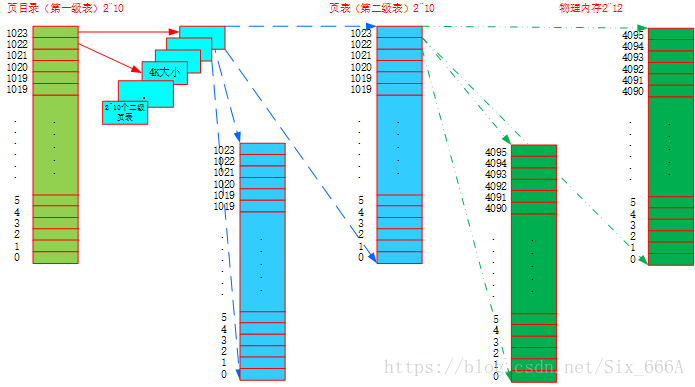


Fig.1 二级页表

1M ＊ 4 ＝＝ 4M，相当于每个进程都要4M的连续内存。

**二级页表：**

每个进程最多需要一个页目录（4K） ＋ 4K\*2^10个页表=4M+4K

每个进程最少需要一个页目录（4K） ＋ 4K\*2^0个页表=4K+4K=8K

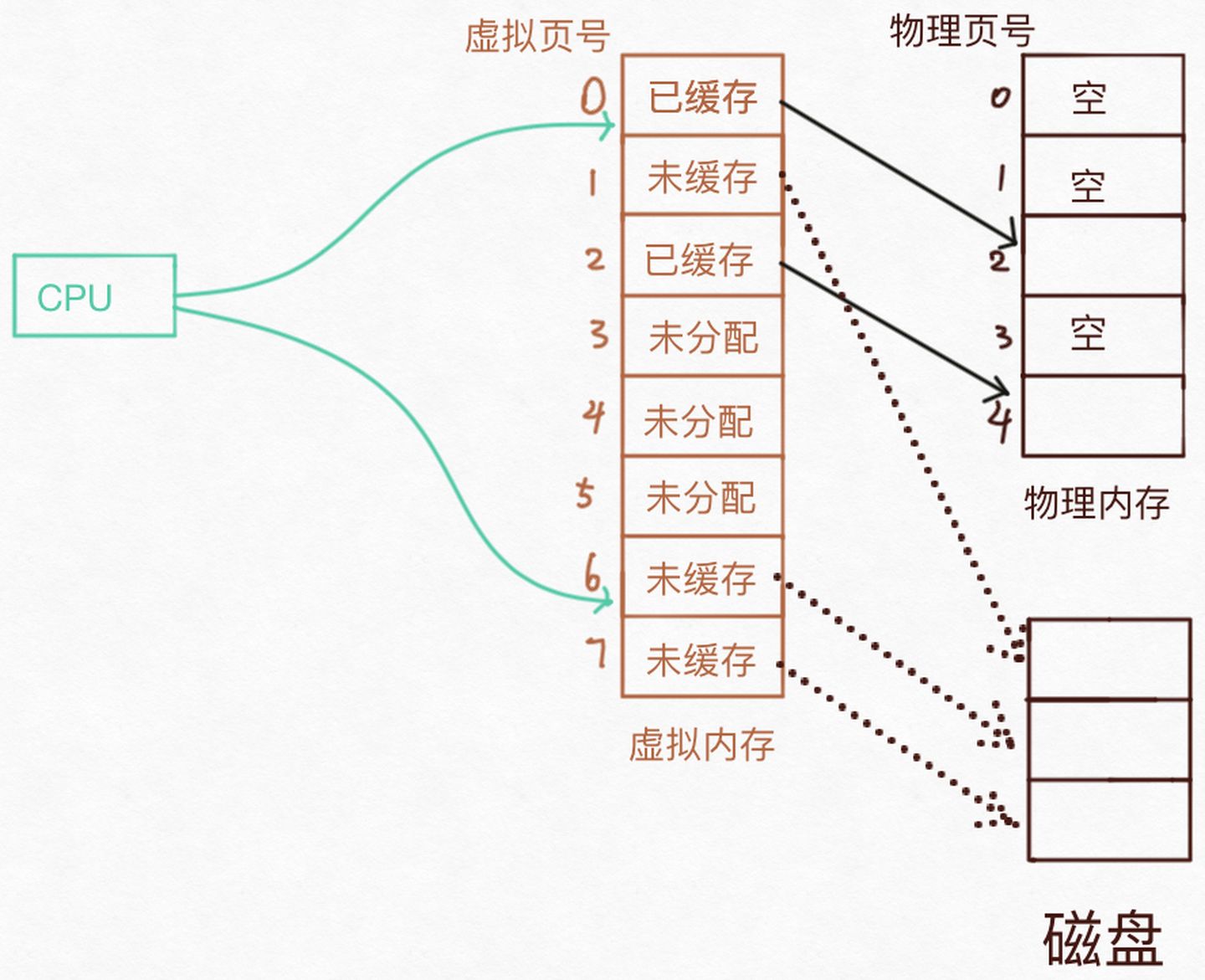


Fig.2 虚拟内存映射

**一级页表的两个问题：**

1.页表在内存分配是必须连续，否则无法查表。

2.每个进程都必须实实在在占用4M，必须全部分配，进程才可以使用。

**二级页表的优势：**

1.允许页表被分散在内存的各个页面中，不需要连续的4M内存块；

2.并不需要为不存在的或线性地址空间未使用部分分配二级页表；

3.可以在虚拟内存中存放二级页表。

**虚拟地址映射**

分页就是把整个虚拟内存和物理内存分割成大小固定的块，以一个页作为映射的最小单位。运行时，CPU使用虚拟地址向内存寻址，通过专用的内存管理单元（MMU）硬件把虚拟地址转换为真实的物理地址（地址翻译），操作系统负责把虚拟地址和物理地址的映射关系维护在页表之中。

当CPU寻址的时候，这个映射会有三种可能。

* 未分配：虚拟地址所在的那一页并未被分配，代表没有数据和他们关联，这部分也不会占用内存。
* 未缓存：虚拟地址所在的那一页被分配了，但并不在内存中。
* 已缓存：虚拟地址所在的那一页就在内存中。

当访问一个未缓存的区域时，系统将产生缺页中断，然后进程被阻塞，等待操作系统将缺失的那一页从磁盘复制到内存。当复制完成后，CPU继续执行导致缺页中断的那条指令，此时就会正常执行了。这种仅在需要的时候将页面拷贝到内存的策略叫做按需调度页面。 可以想象当程序被装入内存的时候，开始时仅有有很小的一部分内容被放入内存。程序在运行中不断缺页，不断的把需要的部分拷贝进内存。



Fig.3 简化的进程虚拟内存

从进程的视角来看，我的数据和代码被存放在一个连续的空间之中，每个区域分别有着不同的功能。典型的如存放代码的区域和存放数据的区域。但是，代码和数据中的地址都是一个虚拟地址，还需要经过地址翻译才能得到真正的物理地址。

Malloc()和mmap（）等内存分配函数，在分配时只是建立了进程虚拟地址空间，并没有分配虚拟内存对应的物理内存。当进程访问这些没有建立映射关系的虚拟内存时，处理器自动触发一个缺页中断，会导致内核实际去分配物理内存，并更新页表映射关系。

**进程在内存中的各个区域：**

1. 栈区（stack）: 存放函数的参数值，局部变量的值，由编译器自动分配释放。
2. 堆区（heap）: 一般由程序员分配释放， 若程序员不释放，程序结束时可能由OS回收。
3. 全局区（静态区）（static）: 用来存放全局变量和静态变量，存在于程序的整个生命周期。
4. 程序代码区：存放程序的而精致代码。
5. 文字常量区：常量字符串就放在这里。

C++中实现信号量

#include <semaphore.h>

sem\_t sem;

// 初始化

//sem\_init() 成功返回0，失败返回1

//pshared = 0 线程间共享 pshared = 1进程间共享 value 信号量的初始值

int sem\_init(sem\_t \*sem,int pshared,unsigned int value);

//P操作，信号量sem减1，若sem=0，该线程被挂起到阻塞列表

sem\_wait(&sem)

//V操作，信号量sem加1，并唤醒阻塞列表中的线程

sem\_post(&sem)