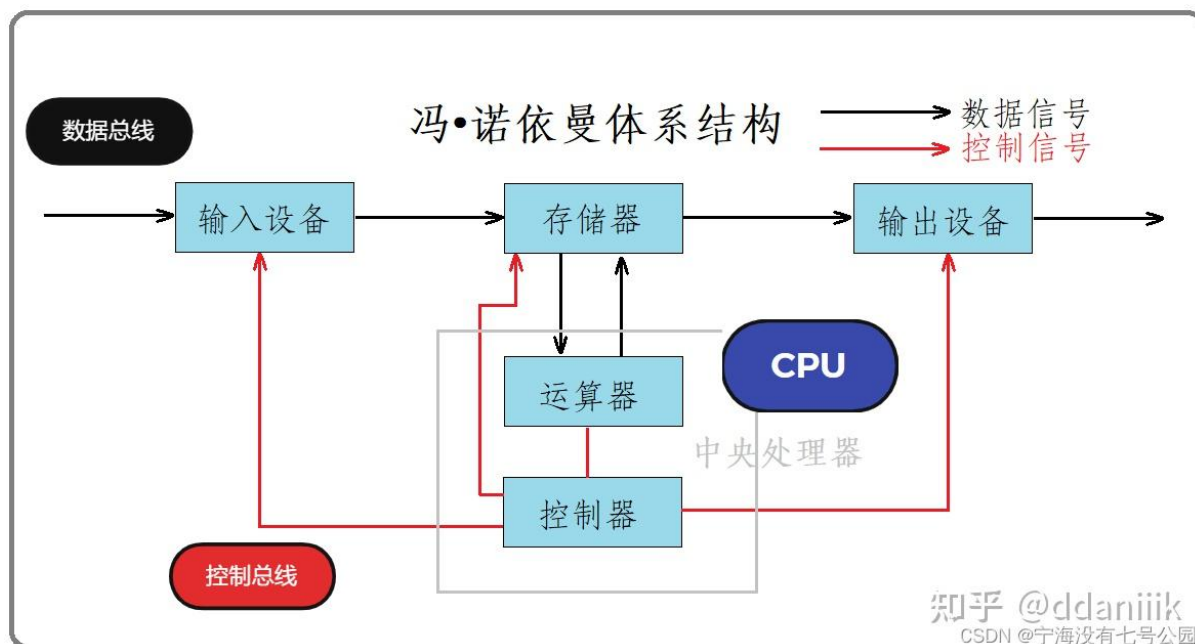


进程概念

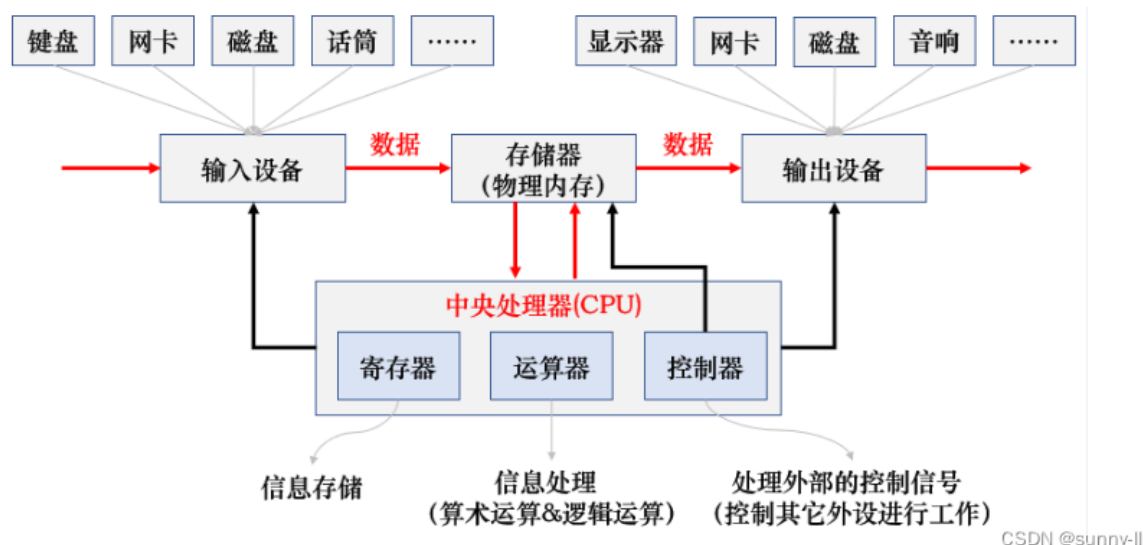
1.冯·诺依曼体系结构

我们常见的计算机，如笔记本。我们不常见的计算机，如服务器，大部分都遵守冯诺依曼体系。



1.1冯·诺依曼体系结构的5大部件

在冯诺依曼体系结构主要由五种设备组成，分别是：输入设备、存储器、运算器、控制器和输出设备，它们各司其职，都做着它们各自的工作。



★输入和输出设备

首先要来讲的就是我们能直接接触到的东西，也就是**两个输入、输出设备**

- **【输入设备】**：向计算机输入数据和信息的设备，是计算机与用户或其他设备通信的桥梁，例：键盘、话筒、摄像头、**网卡、磁盘**
- **【输出设备】**：是计算机硬件系统的终端设备，用于接收计算机数据的输出显示、打印、声音、控制外围设备操作，例：显示器、声卡、**网卡、磁盘**

对于输入输入设备和输出设备，我们统称为**外围设备**，对于**外围设备**而言，都比较慢，比如说**【磁盘】**，不过虽然它比较慢，但是价格并不贵，三五百块钱就可以买到一块512G的硬盘，贵一点的话可能像固态硬盘价格在900 ~ 1200不等，但是它们都有一个优点就是：**存储容量大、可以长久保存数据不丢失！**

★存储器

可以看到，对于上面所列举的输入和输出设备中，**同时出现的就是网卡和磁盘**这两样，我们主要来说说**磁盘**这个东西。因为我们要通过输入设备将输入都输入到计算机中，那计算机肯定要对这些数据去进行一些处理，此时这些数据肯定是要一直存放在计算机中的，所以肯定要有东西将我们输入到计算机中的数据都保存起来，这个时候就需要使用到一些存储器了，此时我们就要来讲讲存储器了，它分为**内存和外存**，不过一般我们都称之为**内存**

- **【内存】**：用于存放电脑运行中的原始数据、中间结果以及指示电脑工作的程序，断电后会丢失，**容量小，速度快**
- **【外存】**：用来存放一些需要长期保存的程序或数据，断电后也不会丢失，**容量比较大，但存取速度慢**

注意：这里存储器只是内存，不包括外存。

★中央处理器（CPU）

既然可以存储我们输入进去的数据了，那要如何去处理这些数据呢？此时就需要使用到**中央处理器（CPU）**

- **【运算器】**：计算机中执行各种算术和逻辑运算操作的部件。运算器的基本操作包括加、减、乘、除四则运算等等
- 当运算器运算完成之后，就会经过输出设备将处理后的结果交给内存，再由内存展现给用户，这就是用户与计算机之间的交互过程

不过上面这些从输入到存储到计算，再到输出的过程，计算机如何去执行的呢？靠什么去控制这种种行为？此时就要使用到**控制器**了

【控制器】：控制器也是一个硬件，虽然**外设和中央处理器**在数据上没有交互，但并不代表它俩就没有交互。前面得知**输入设备**会把数据预装载到**内存**，从而和**cpu**进行

交互，但是你怎么知道所有数据都被预装载了呢，针对没被预装载的数据，中央处理器就要和外设进行交互协商，而这个操作就是由控制器完成的，从而将数据尽可能加载到内存，或把数据从内存加载到外设。

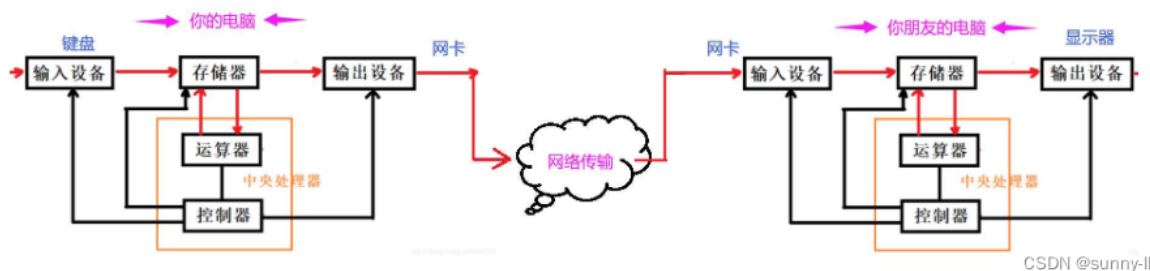
对于上面的运算器和控制器，我们将其合称为【中央处理器】，即 **cpu**。它是计算机的大脑、也是核心部分，很多控制信号都要经过CPU才能执行

1.2冯·诺依曼体系结构的应用

理解数据在网络中的流动

现在你在上网，使用QQ向你的朋友发送了一句“在吗”，那此时这个数据在网络中进行流动的呢？假设你们的电脑都是基于冯诺依曼体系结构，需要有这么一个输入、处理、输出这么一个流程。

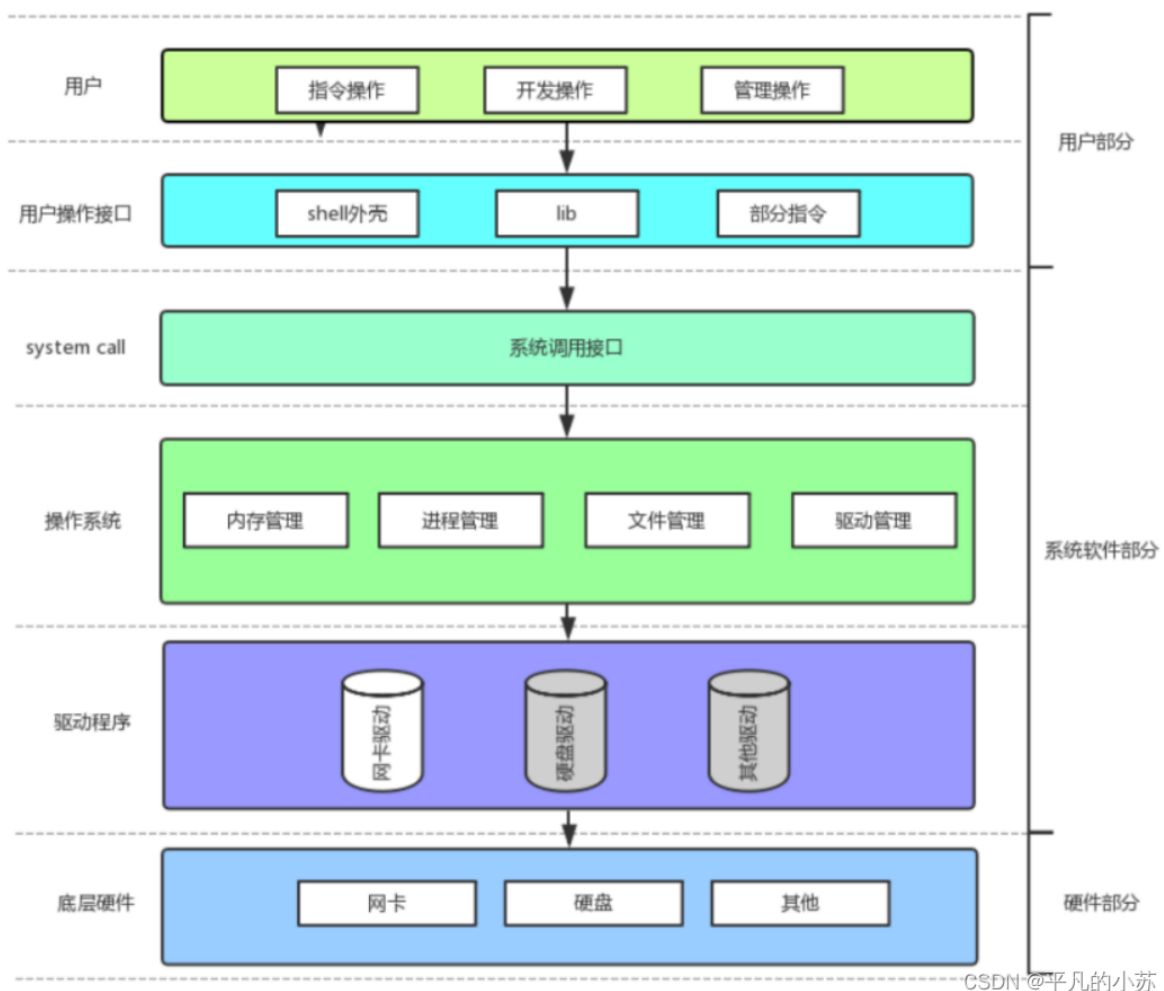
流程图如下：



2.操作系统

1.为什么要有操作系统

- 与硬件交互，管理所有的软硬件资源
- 为用户程序（应用程序）提供一个良好的执行环境



2.操作如何提供给用户服务

- 操作系统里面，里面会有各种数据，可是，操作系统不信任任何用户！
操作系统为了保证自己的数据安全，也为了保证给用户提供服务，操作系统以接口的方式给用户提供调用的入口，来获取操作系统内部的数据！

3.进程

3.1进程概念

- 课本概念：程序的一个执行实例，正在执行的程序等
- 内核观点：担当分配系统资源（CPU时间，内存）的实体。

3.2内核分类

- 标示符:** 描述本进程的唯一标示符，用来区别其他进程。
- 状态:** 任务状态，退出代码，退出信号等。

- **优先级:** 相对于其他进程的优先级。
- **程序计数器:** 程序中即将被执行的下一条指令的地址。
- **内存指针:** 包括程序代码和进程相关数据的指针，还有和其他进程共享的内存块的指针
- **上下文数据:** 进程执行时处理器的寄存器中的数据[休学例子，要加图CPU，寄存器]。
- **I/O状态信息:** 包括显示的I/O请求,分配给进程的I/O设备和被进程使用的文件列表。
- **记账信息:** 可能包括处理器时间总和，使用的时钟数总和，时间限制，记账号等。
- **其他信息**

3.3查看进程

- 进程的信息可以通过 ls /proc 系统文件夹查看

```
[junmoxiao@VM-24-6-centos ~]$ ls /proc
1 106 15847 2 25428 26701 29 35 48 652 8904 consoles filesystems key-users modules self timer_stats
10 11 1592 20 25465 26711 291 36 49 655 9 cpuinfo fs kmsg mounts slabinfo tty
1005 1180 1593 200 26 27 29104 37 51 662 9278 crypto interrupts kpagecount mtrr softirqs
1006 1181 15947 21 261 27169 29112 38 52 673 941 devices iomem kpageflags net stat version
1011 12 16 22 262 27472 29113 387 533 677 acpi diskstats ioports loadavg pagetypeinfo swaps vmallocinfo
1032 13 1631 22469 263 276 29136 4 6 7 buddyinfo dma irq locks partitions sys vmstat
1036 13365 18 23 264 28 292 403 628 8 bus driver kallsyms mdstat sched_debug sysrq-trigger xpmem
1039 14 18052 24 265 28619 29285 406 65 8885 cgroups execdomains kcore meminfo schedstat sysvipc zoneinfo
1043 1584 19 25 266 28926 293 46 651 8893 cmdline fb keys misc scsi timer_list
```

- 通过ps命令查看进程

```
[junmoxiao@VM-24-6-centos ~]$ ps ajx
PPID  PID  PGID  SID  TTY  TPGID  STAT  UID  TIME  COMMAND
0  1  0  0  ?  -1 Ss  0  31:48 /usr/lib/systemd/systemd --switched-root --system --deserialize 22
0  2  0  0  ?  -1 S  0  0:05 [kthreadd]
2  4  0  0  ?  -1 S<  0  0:00 [kworker/0:0H]
2  6  0  0  ?  -1 S  0  6:24 [ksoftirqd/0]
2  7  0  0  ?  -1 S  0  0:59 [migration/0]
2  8  0  0  ?  -1 S  0  0:00 [rcu_bh]
2  9  0  0  ?  -1 S  0  56:31 [rcu_sched]
2  10  0  0  ?  -1 S<  0  0:00 [lru-add-drain]
2  11  0  0  ?  -1 S  0  0:39 [watchdog/0]
2  12  0  0  ?  -1 S  0  0:33 [watchdog/1]
2  13  0  0  ?  -1 S  0  0:59 [migration/1]
2  14  0  0  ?  -1 S  0  5:37 [ksoftirqd/1]
2  16  0  0  ?  -1 S<  0  0:00 [kworker/1:0H]
2  18  0  0  ?  -1 S  0  0:00 [kdevtmpfs]
2  19  0  0  ?  -1 S<  0  0:00 [netns]
2  20  0  0  ?  -1 S  0  0:02 [khungtaskd]
2  21  0  0  ?  -1 S<  0  0:00 [writeback]
2  22  0  0  ?  -1 S<  0  0:00 [kintegrityd]
2  23  0  0  ?  -1 S<  0  0:00 [bioset]
2  24  0  0  ?  -1 S<  0  0:00 [bioset]
2  25  0  0  ?  -1 S<  0  0:00 [bioset]
2  26  0  0  ?  -1 S<  0  0:00 [kblockd]
```

- 查看当前运行的进程

```
ps ajx | head -1 && ps ajx | grep mytest
while :;do ps ajx | head -1 && ps ajx | grep mytest | grep -v
```

解析:

1. `ps ajx` —— 查看当前系统中所有进程
2. `head -1` —— 获取第一行
3. `grep mytest` —— 过滤只带【mytest】的进程

3.4系统调用获取进程标识符

- 进程id (PID)
- 父进程id (PPID)

```
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
int main()
{
    printf("我是子进程, pid: %d\n", getpid());
    printf("我是父进程, ppid: %d\n", getppid());
    return 0;
}
```

```
[yeyushengfan@VM-24-6-centos pro1]$ make
gcc -o pro pro.c
[yeyushengfan@VM-24-6-centos pro1]$ ./pro
我是子进程, pid: 3010
我是父进程, ppid: 1245
```

3.5通过系统调用初识fork

3.5.1通过man手册了解fork

fork的返回值是一个int，fork是给父进程返回子进程的pid，而子进程则是返回0，要是fork失败的话那就是返回-1。

RETURN VALUE

On success, the PID of the child process is returned in the parent, and 0 is returned in the child. On failure, -1 is returned in the parent, no child process is created, and `errno` is set appropriately.

函数说明：

- 通过复制调用进程创建一个新进程。
- `fork` 有两个返回值。
- 父子进程代码共享，数据各自私有一份（采用写时拷贝）。

3.5.2为什么父进程返回子pid，子进程返回0？

这是为了区分不同的执行流，执行不同的代码块

那有同学说：你这不说了跟没说一样嘛，要区分的话当然得不同了，那为什么父进程得到的是子进程的 `PID`，但是子进程却是 `0` 呢，为什么不可以倒过来？

这位同学，你问到点子上了，确实这是它们最大的区别，不过呢这样的返回值还是有原因的。读者可以这么来理解：一个父亲可以有多个孩子👨‍👩‍👧，但是呢一个孩子却只能有一个父亲👨‍👩‍👧 父亲所获取到的返回值是子进程的PID是由于他要靠不同的PID值来区分不同的孩子；但子进程的返回值都是0的原因在于他一定只对应着某一个父进程，只需让父进程知道它被成功创建出来了即可。

3.5.3函数是如何返回两次的，如何理解？

在我们创建子进程的时候，父子之间的代码是共享的，数据会独自开辟空间，当我们想要使用数据时，会发生**写时拷贝**。

```
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>

int main()
{
    printf("begin: 我是一个进程, pid: %d, ppid: %d\n", getpid())

    sleep(5);
    pid_t id = fork();

    //printf("我是后续的代码\n");
    //sleep(1);
    if(id == 0)
    {
```

```
        // 子进程
        while(1)
        {
            printf("我是子进程, pid: %d, ppid: %d\n", getpid(),
                sleep(1);
        }
    }
    else if(id > 0)
    {
        //父进程
        while(1)
        {
            printf("我是父进程, pid: %d, ppid: %d\n", getpid(),
                sleep(1);
        }
    }
    else
    {
        //error
    }
    return 0;
}
```


一个则是子进程（新的分支）

所以现在我们可以得出创建进程的两种方式：

1. `./运行我们的程序` -- 指令层面-----（bash）
2. `fork()` -- 代码层面

4.进程状态

为了弄明白正在运行的进程是什么意思，我们需要知道进程的不同状态。一个进程可以有几个状态（在Linux内核里，进程有时候也叫做任务）

4.1R 运行状态

并不意味着进程一定在运行中，它表明进程要么是在运行中要么在运行队列里。

```
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>

int main()
{
    while(1)
    {
        ;
    }
}
```

```
[yeyushengfan@VM-24-6-centos prol]$ ps ajx|head -1&& ps ajx|grep mytest
PPID  PID  PGID  SID  TTY      TPGID STAT   UID    TIME COMMAND
21179 28988 28987 21179 pts/1    28987 R+      1003    0:00 grep --color=auto mytest
```

4.2S 睡眠状态

意味着进程在等待事件完成（这里的睡眠有时候也叫做**可中断睡眠**（interruptible sleep））。

```
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>

int main()
```

```
{
    while(1)
    {
        printf("i am a process: pid: %d\n", getpid());
        sleep(1);
    }
}
```

```
[yeyushengfan@VM-24-6-centos prol]$ ps ajx|head -1&& ps ajx|grep mytest
PPID  PID  PGID  SID TTY      TPGID STAT  UID   TIME COMMAND
21179 30306 30305 21179 pts/1    30305 S+    1003   0:00 grep --color=auto mytest
```

4.3D 磁盘休眠状态

有时候也叫**不可中断睡眠状态**（uninterruptible sleep），在这个状态的进程通常会等待IO的结束。

4.4T 停止状态

可以通过发送 SIGSTOP 信号给进程来停止（T）进程。这个被暂停的进程可以通过发送 SIGCONT 信号让进程继续运行。

4.5X 死亡状态

这个状态只是一个返回状态，你不会在任务列表里看到这个状态

4.6Z 僵尸状态

- 僵尸状态（Zombies）是一个比较特殊的状态。当进程退出并且父进程（使用 wait()系统调用）没有读取到子进程**退出的返回代码时就会产生僵死(尸)进程**
- 僵尸进程会以终止状态保持在进程表中，并且会一直在等待父进程读取退出状态代码。
- 所以，只要子进程退出，父进程还在运行，但父进程没有读取子进程状态，子进程进入Z状态

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>

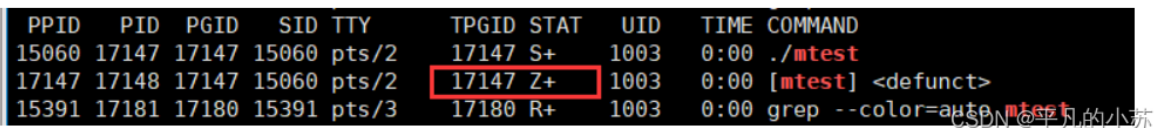
int main()
{
```

```

pid_t id = fork();
if(id == 0)
{
    //child
    int cnt = 500;
    while(cnt)
    {
        printf("i am child, pid: %d, ppid: %d, cnt: %d\n",
            cnt--);
        sleep(1);
    }
    exit(0);
}
else
{
    int cnt = 5;
    //father
    while(cnt--)
    {
        printf("i am father, pid: %d, ppid: %d\n", getpid(),
            sleep(1));
    }

    //父进程目前并没有针对子进程干任何事事情
}
}

```



PPID	PID	PGID	SID	TTY	TPGID	STAT	UID	TIME	COMMAND
15060	17147	17147	15060	pts/2	17147	S+	1003	0:00	./mtest
17147	17148	17147	15060	pts/2	17147	Z+	1003	0:00	[mtest] <defunct>
15391	17181	17180	15391	pts/3	17180	R+	1003	0:00	grep --color=auto mtest

僵尸进程的危害

- 进程的退出状态必须被维持下去，因为他要告诉关心它的进程（父进程），你交给我的任务，我办的怎么样了。可父进程如果一直不读取，那子进程就一直处于Z状态？是的！
- 维护退出状态本身就是要用数据维护，也属于进程基本信息，所以保存在task_struct(PCB)中，换句话说，Z状态一直不退出，PCB一直都要维护？是的！

- 那一个父进程创建了很多子进程，就是不回收，是不是就会造成内存资源的浪费？是的！因为数据结构对象本身就要占用内存，想想C中定义一个结构体变量（对象），是要在内存的某个位置进行开辟空间！

4.7孤儿进程

- 父进程如果提前退出，那么子进程后退出，进入Z之后，那该如何处理呢？
- 父进程先退出，子进程就称之为“孤儿进程”
- 那一个父进程创建了很多子进程，就是不回收，是不是就会造成内存资源的浪费？是的！因为数据结构对象本身就要占用内存，想想C中定义一个结构体变量（对象），是要在内存的某个位置进行开辟空间！

5.进程优先级

概念

- cpu资源分配的先后顺序，就是指**进程的优先权**（priority）。
- 优先权高的进程有优先执行权利。配置进程优先权对多任务环境的linux很有用，**可能**改善系统性能。
- 还可以把进程运行到指定的CPU上，这样一来，把不重要的进程安排到某个CPU，可以大大改善系统整体性能

查看进程优先级命令

```
ps -al
```

```
[yeyushengfan@VM-24-6-centos ~]$ ps al
F  UID  PID  PPID  PRI  NI   VSZ  RSS  WCHAN  STAT  TTY      TIME  COM
MAND
4    0  1180    1   20   0 110208   820  n_tty_  Ss+   tty1      0:00  /sb
4    0  1181    1   20   0 110208   832  n_tty_  Ss+   ttyS0     0:00  /sb
0  1003 20814 31659  20   0 153328  1528  -      R+    pts/0     0:00  ps
```

我们很容易注意到其中的几个重要信息，有下：

UID：代表执行者的身份

PID：代表这个进程的代号

PPID：代表这个进程是由哪个进程发展衍生而来的，亦即父进程的代号

PRI：代表这个进程可被执行的优先级，其值越小越早被执行

NI：代表这个进程的nice值

PRI VS NI

- PRI也还是比较好理解的，即进程的优先级，或者通俗点说就是程序被CPU执行的先后顺序，此值越小进程的优先级别越高
- 那NI呢?就是我们所要说的nice值了，其表示进程可被执行的优先级的修正数值
- PRI值越小越快被执行，那么加入nice值后，将会使得PRI变为：
 $PRI(new)=PRI(old)+nice$
- 这样，当nice值为负值的时候，那么该程序将会优先级值将变小，即其优先级会变高，则其越快被执行
- 所以，调整进程优先级，在Linux下，就是调整进程nice值
- nice其取值范围是-20至19，一共40个级别

更改进程优先级命令

- top命令
- 进入top后按“r”→输入进程PID→输入nice值

6.环境变量

基本概念

- 环境变量一般是指在操作系统中用来**指定操作系统运行环境**的一些参数
- 如：我们在编写C/C++代码的时候，在链接的时候，从来不知道我们的所**链接的动态静态库在哪里**，但是照样可以链接成功，生成可执行程序，原因就是有相关环境变量帮助编译器进行查找。
- 环境变量通常具有某些特殊用途，还有在系统当中通常**具有全局特性**

常见的环境变量

- PATH：指定命令的搜索路径
- HOME：指定用户的主工作目录(即用户登陆到Linux系统中时,默认的目录)
- SHELL：当前Shell,它的值通常是/bin/bash。

注意：为什么我们执行系统命令不用带路径，而执行可执行程序需要带./呢？这是因为环境变量的因素

```
[sqy@hecs-354086 lesson8.8]$ echo $PATH
/usr/local/bin:/usr/bin:/usr/local/sbin:/usr/sbin:/home/sqy/.local/bin:/home/sqy/bin
[sqy@hecs-354086 lesson8.8]$
```

CSDN @平凡的小苏

系统命令大多都在红色方框的路径中，所以在执行指令时候，shell会在PATH中寻找，找到就会执行指令了。如果我们把上面的路径放入环境变量，我们在执行可执行程序的时候也可以不带./了

系统命令大多都在红色方框的路径中，所以在执行指令时候，shell会在PATH中寻找，找到就会执行指令了。如果我们把上面的路径放入环境变量，我们在执行可执行程序的时候也可以不带./了

添加环境变量命令

`PATH=$PATH: + 路径`

环境变量的相关命令

- `echo`: 显示某个环境变量值
- `export`: 设置一个新的环境变量
- `env`: 显示所有环境变量
- `unset`: 清除环境变量
- `set`: 显示本地定义的shell变量和环境变量

通过系统调用获取环境变量

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>

int main()
{
    printf("%s\n",getenv("PATH"));
    return 0;
}
```

```
[yeyushengfan@VM-24-6-centos pro2]$ ./mytest
/usr/local/bin:/usr/bin:/usr/local/sbin:/usr/sbin:/home/yeyushengfan/.local/bin:/home/yeyushengfan/bin
```

命令行的三个参数

```
int main(int argc,char *argv[],char *env[])
```

先来看前面两个参数的作用

```
#include <stdio.h>
#include <string.h>

int main(int argc, char *argv[], char *env[])
{
    if(argc != 2)
    {
        printf("Usage: %s -[a|b|c|d]\n", argv[0]);
        return 0;
    }
    if(strcmp(argv[1], "--help")==0)
    {
        printf("Usage: %s -[a|b|c|d]\n", argv[0]);
    }
    else if(strcmp(argv[1], "-a") == 0)
    {
        printf("功能1\n");
    }
    else if(strcmp(argv[1], "-b") == 0)
    {
        printf("功能2\n");
    }
    else if(strcmp(argv[1], "-c") == 0)
    {
        printf("功能3\n");
    }
    else if(strcmp(argv[1], "-d") == 0)
    {
        printf("功能4\n");
    }
    else
    {
        printf("default功能\n");
    }
    return 0;
}
```



```
[yeyushengfan@VM-24-6-centos pro2]$ ./mytest
Usage: ./mytest -[a|b|c|d]
[yeyushengfan@VM-24-6-centos pro2]$ ./mytest -a
功能1
[yeyushengfan@VM-24-6-centos pro2]$ ./mytest -b
功能2
[yeyushengfan@VM-24-6-centos pro2]$ ./mytest -c
功能3
[yeyushengfan@VM-24-6-centos pro2]$ ./mytest -d
功能4
```

第三个参数：获取系统中全部的环境变量

```
#include <stdio.h>
#include <string.h>

int main(int argc, char *argv[], char *env[])
{
    int i = 0;
    for(; env[i]; i++)
    {
        printf("%s\n", env[i]);
    }

    return 0;
}
```

```
[yeyushengfan@VM-24-6-centos pro2]$ make
gcc -o mytest mytest.c
[yeyushengfan@VM-24-6-centos pro2]$ ./mytest
XDG_SESSION_ID=291688
HOSTNAME=VM-24-6-centos
TERM=xterm
SHELL=/bin/bash
HISTSIZE=3000
SSH_CLIENT=42.92.206.207 7922 22
SSH_TTY=/dev/pts/3
USER=yeyushengfan
LD_LIBRARY_PATH=/home/yeyushengfan/.VimForCpp/vim/bundle/YCM.so/el7.x86_64
LS_COLORS=rs=0:di=01;34:ln=01;36:mh=00:pi=40;33:so=01;35:do=01;35:bd=40;33:cd=40;33:or=40;31:01:mi=01;05;37;41:su=37;41:sg=30;43:ca=30;41:tw=30;42:ow=34;42:st
=37;44:ex=01;32:*tar=01;31:*tgz=01;31:*arc=01;31:*arj=01;31:*taz=01;31:*lha=01;31:*lzd=01;31:*lzh=01;31:*lzm=01;31:*tlz=01;31:*tzo=01;31:*t
7z=01;31:*zip=01;31:*z=01;31:*Z=01;31:*dz=01;31:*gz=01;31:*lrz=01;31:*lz=01;31:*lzo=01;31:*xz=01;31:*bz2=01;31:*bz=01;31:*tbz2=01;31:*tbz=01
;31:*deb=01;31:*rpm=01;31:*jar=01;31:*war=01;31:*ear=01;31:*sar=01;31:*rar=01;31:*alz=01;31:*ace=01;31:*zoo=01;31:*cpio=01;31:*7z=01;31:*rz=01;31:*cab
=01;31:*jpg=01;35:*jpeg=01;35:*gif=01;35:*bmp=01;35:*pbm=01;35:*pgm=01;35:*ppm=01;35:*tga=01;35:*xbm=01;35:*xpm=01;35:*tif=01;35:*tiff=01;35:*png=01;35
:*svg=01;35:*svgz=01;35:*mng=01;35:*pcx=01;35:*mov=01;35:*mpg=01;35:*mpeg=01;35:*m2v=01;35:*mkv=01;35:*webm=01;35:*ogm=01;35:*mp4=01;35:*m4v=01;35:*mp
4v=01;35:*vob=01;35:*qt=01;35:*nuv=01;35:*wmv=01;35:*asf=01;35:*rm=01;35:*rmvb=01;35:*flc=01;35:*avi=01;35:*fli=01;35:*flv=01;35:*gl=01;35:*dl=01;35:*
xcf=01;35:*xwd=01;35:*yuv=01;35:*cgm=01;35:*emf=01;35:*axv=01;35:*anx=01;35:*ogv=01;35:*ogx=01;35:*aac=01;36:*au=01;36:*flac=01;36:*mid=01;36:*midi=01;
36:*mka=01;36:*mp3=01;36:*mpc=01;36:*ogg=01;36:*ra=01;36:*wav=01;36:*axa=01;36:*oga=01;36:*spx=01;36:*xspf=01;36:
MAIL=/var/spool/mail/yeyushengfan
PATH=/usr/local/bin:/usr/bin:/usr/local/sbin:/usr/sbin:/home/yeyushengfan/.local/bin:/home/yeyushengfan/bin
PWD=/home/yeyushengfan/pro2
LANG=en_US.utf8
SHLVL=1
HOME=/home/yeyushengfan
LOGNAME=yeyushengfan
SSH_CONNECTION=42.92.206.207 7922 10.0.24.6 22
LESSOPEN=||/usr/bin/lesspipe.sh %s
PROMPT_COMMAND=history -a; history -a; printf "\033]0;%s@%s:%s\007" "${USER}" "${HOSTNAME%.*}" "${PWD/#$HOME/~}"
XDG_RUNTIME_DIR=/run/user/1003
HISTTIMEFORMAT=%F %T
_=./mytest
OLDPWD=/home/yeyushengfan
```

7.进程地址空间

7.1虚拟地址

先看一段父子进程共存的程序，由于进程对全局变量grobal_val进行修改：

```
#include <stdio.h>
#include <unistd.h>
int grobal_val=10;
int main()
{
    pid_t id=fork();
    if(id==0)
    {
        int cnt=0;
        while(1)
        {
            printf("子进程:pid=%d,ppid=%d | grobal_val=%d,&gro
sleep(1);
            ++cnt;
            if(cnt==10)
            {
                grobal_val=200;
                printf("子进程已更改全局变量grobal_val\n");
            }
        }
    }
}
```

```
else if(id>0)
{
    while(1)
    {
        printf("父进程:pid=%d,ppid=%d | global_val=%d,&gro
        sleep(1);
    }
}
else
{
    printf("fork error\n");
    return 1;
}
return 0;
}
```

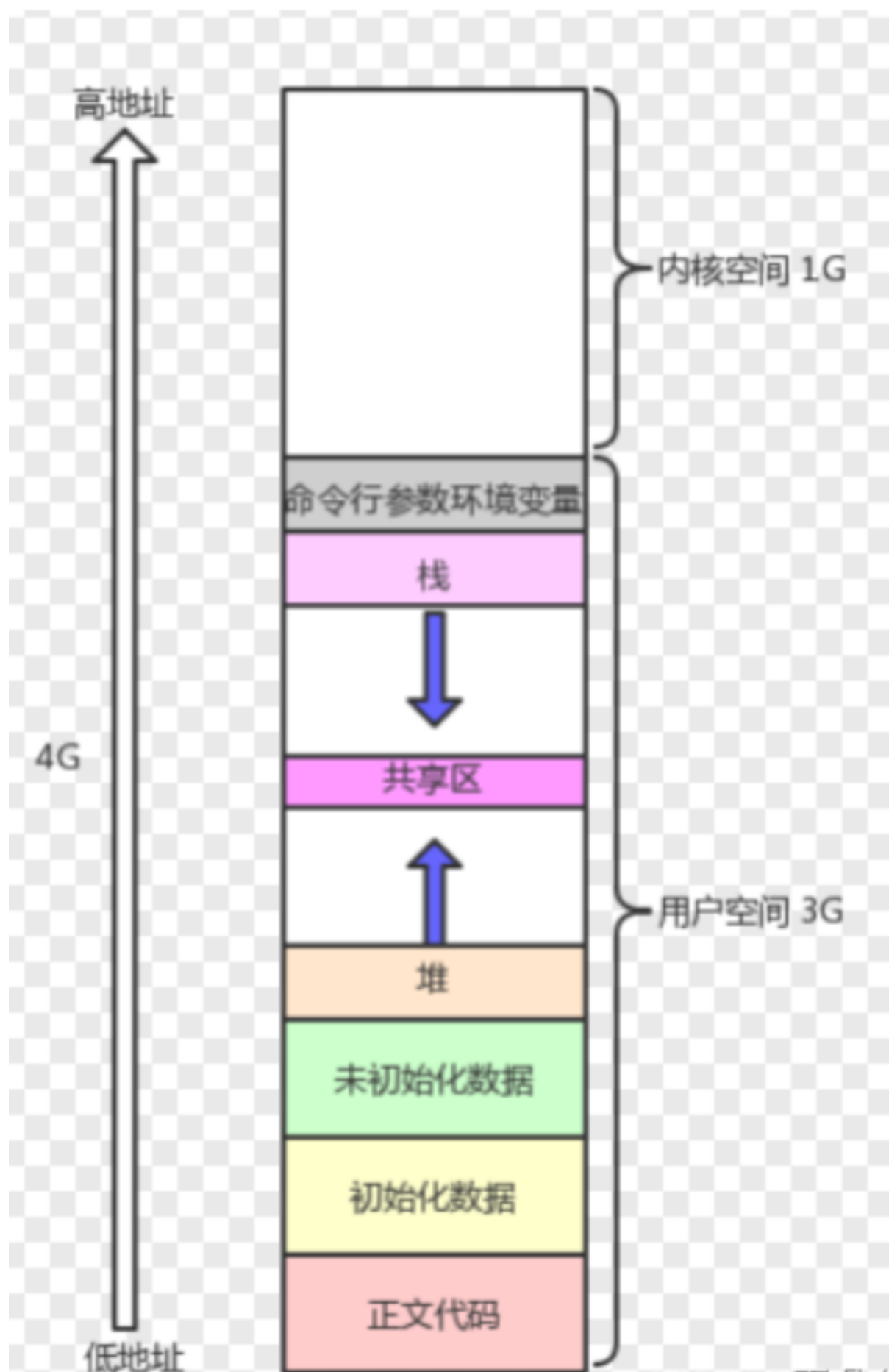
```
[yeyushengfan@VM-24-6-centos pro2]$ vim mytest.c
[yeyushengfan@VM-24-6-centos pro2]$ make
gcc -o mytest mytest.c
[yeyushengfan@VM-24-6-centos pro2]$ ./mytest
父进程:pid=22479,ppid=32764 | global_val=10,&global_val=0x40405c
子进程:pid=22480,ppid=22479 | global_val=10,&global_val=0x40405c
父进程:pid=22479,ppid=32764 | global_val=10,&global_val=0x40405c
子进程:pid=22480,ppid=22479 | global_val=10,&global_val=0x40405c
父进程:pid=22479,ppid=32764 | global_val=10,&global_val=0x40405c
子进程:pid=22480,ppid=22479 | global_val=10,&global_val=0x40405c
父进程:pid=22479,ppid=32764 | global_val=10,&global_val=0x40405c
子进程:pid=22480,ppid=22479 | global_val=10,&global_val=0x40405c
父进程:pid=22479,ppid=32764 | global_val=10,&global_val=0x40405c
子进程:pid=22480,ppid=22479 | global_val=10,&global_val=0x40405c
父进程:pid=22479,ppid=32764 | global_val=10,&global_val=0x40405c
子进程:pid=22480,ppid=22479 | global_val=10,&global_val=0x40405c
父进程:pid=22479,ppid=32764 | global_val=10,&global_val=0x40405c
子进程:pid=22480,ppid=22479 | global_val=10,&global_val=0x40405c
父进程:pid=22479,ppid=32764 | global_val=10,&global_val=0x40405c
子进程:pid=22480,ppid=22479 | global_val=10,&global_val=0x40405c
父进程:pid=22479,ppid=32764 | global_val=10,&global_val=0x40405c
子进程:pid=22480,ppid=22479 | global_val=10,&global_val=0x40405c
父进程:pid=22479,ppid=32764 | global_val=10,&global_val=0x40405c
子进程:pid=22480,ppid=22479 | global_val=10,&global_val=0x40405c
父进程:pid=22479,ppid=32764 | global_val=10,&global_val=0x40405c
子进程已更改全局变量global_val
子进程:pid=22480,ppid=22479 | global_val=200,&global_val=0x40405c
父进程:pid=22479,ppid=32764 | global_val=10,&global_val=0x40405c
子进程:pid=22480,ppid=22479 | global_val=200,&global_val=0x40405c
父进程:pid=22479,ppid=32764 | global_val=10,&global_val=0x40405c
子进程:pid=22480,ppid=22479 | global_val=200,&global_val=0x40405c
```

父子进程谁先执行不确定，由系统进行调度。

当子进程将全局变量 `grobal_val` 由10改为200，我们可以看到，父子进程的 `grobal_val` 的地址相同，但是父子进程从这个地址中获取的值却并不相同！

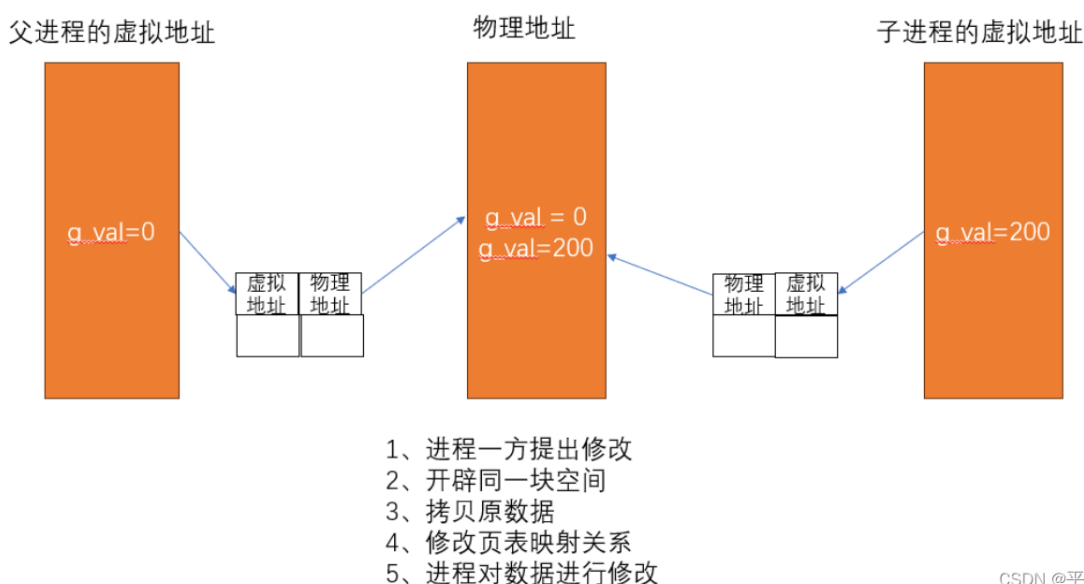
从同一块物理地址中取出的值是相同的，所以这个程序取出的地址（指针）并不是物理地址，而是虚拟地址（线性地址、逻辑地址）。注：逻辑地址指可执行程序编译完成后内部函数、变量的地址。逻辑地址有两种表示方法，一种是各个区域地址递增，另一种是每个区域的地址都从零偏移量开始（这种是比较老的表示方式）。

在 Linux 中的逻辑地址是第一种表示方式，所以 Linux 中逻辑地址就是虚拟地址。



之前学习的 `C/C++` 内存区域，是一块虚拟内存空间，每个进程有它自己的虚拟内存空间，即进程地址空间。所以上面的代码用 `fork` 创建子进程，因为子进程是父进程的拷贝，父子进程的 `global_val` 虽然虚拟地址一样，但会被映射到不同的物理地址上。

当 `global_val` 未被改变时，父子进程映射同一块 `global_val` 的物理地址，一旦父子进程的一方对共享数据进行修改，由于**进程的独立性**，操作系统会在物理内存中**再开辟一块空间**，并**拷贝原数据**，提出修改的**进程的页表映射关系将会被改变**，然后再让**进程对数据进行修改**，所以我们看到父子进程的数据并不一样。这种技术称为**写时拷贝**，对不同进程的数据进行分离。



7.2进程地址空间的理解

- 进程它自己会认为它独占CPU资源，但其实并不是。因为进程以时间片轮转的形式占用CPU资源，时间一到，马上从运行状态进入休眠状态，实质上是**通过虚拟地址空间**，让进程认为它独占CPU资源。
- 进程地址空间是操作系统给进程开辟的一块虚拟内存空间，这块空间用内核的一种数据结构来描述、组织。
- 操作系统给每个进程一块4GB的虚拟内存，进程每次想使用，按需申请即可，但不会全部给进程。（注意这里给的是虚拟内存，就像老板给员工画饼一样）

对Linux操作系统中进程的理解中提到过，进程使用进程控制块`task_struct`结构体进行管理，同样的，每个进程地址空间也需要被管理，管理进程地址空间的结构体叫`mm_struct`，`task_struct`中有一个指针指向自己的`mm_struct`。

`mm_struct`伪代码：

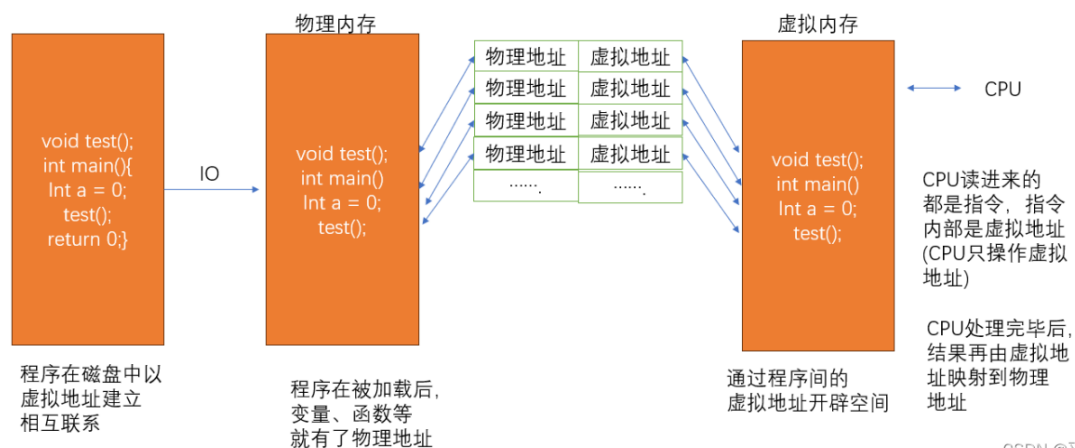
```

struct mm_struct
{
    uint32_t code_start,code_end;
    uint32_t data_start,data_end;
    uint32_t heap_start,heap_end;
    uint32_t stack_start,stack_end;
    .....//存储进程地址空间各区域的起始位置
};

```

7.3为什么要通过虚拟地址映射的方式访问物理地址

- 直接访问物理内存是非常不安全的，例如越界操作、恶意进程读取等。
- 页表会拦截不合理的请求，可以保护物理内存，防止恶意进程的访问。所以写代码出现野指针、内存越界等情况并不会造成操作系统的崩溃。
- 进程地址空间的存在，可以让进程和进程间的代码进行解耦（互不干扰），保证了进程独立性的特征。
- 进程和编译器均遵守进程地址空间这一套规则，编完即可使用。



CSDN @平凡的小苏

编译器也遵守进程地址空间这一套规则：

我们的代码在磁盘时，程序的函数、变量等通过虚拟地址建立联系，满足程序间的互相跳转；

当程序由磁盘被加载到内存中时，就具备了物理地址。函数、变量等通过页表映射至虚拟地址。

根据可执行程序虚拟地址初始化mm_struct结构体中每个虚拟内存中的边界。

当程序在CPU中跑起来时，CPU根据虚拟地址运行完程序后，通过页表映射至物理地址。