

# 第二章 核心导言

- 2.1 UNIX操作系统的体系结构

“文件”和“进程”是UNIX系统的两个最基本实体和中心概念，UNIX系统的所有操作都是以这两者为基础的。整个系统核心由以下五个部分组成：

- ① 文件系统：

文件管理和存储空间管理（节点和空间管理）

- ② I/O设备管理：

核心→缓冲→块设备（随机存取设备）

核心→原始设备（raw设备，字符设备，裸设备）

- ③ 进程控制：

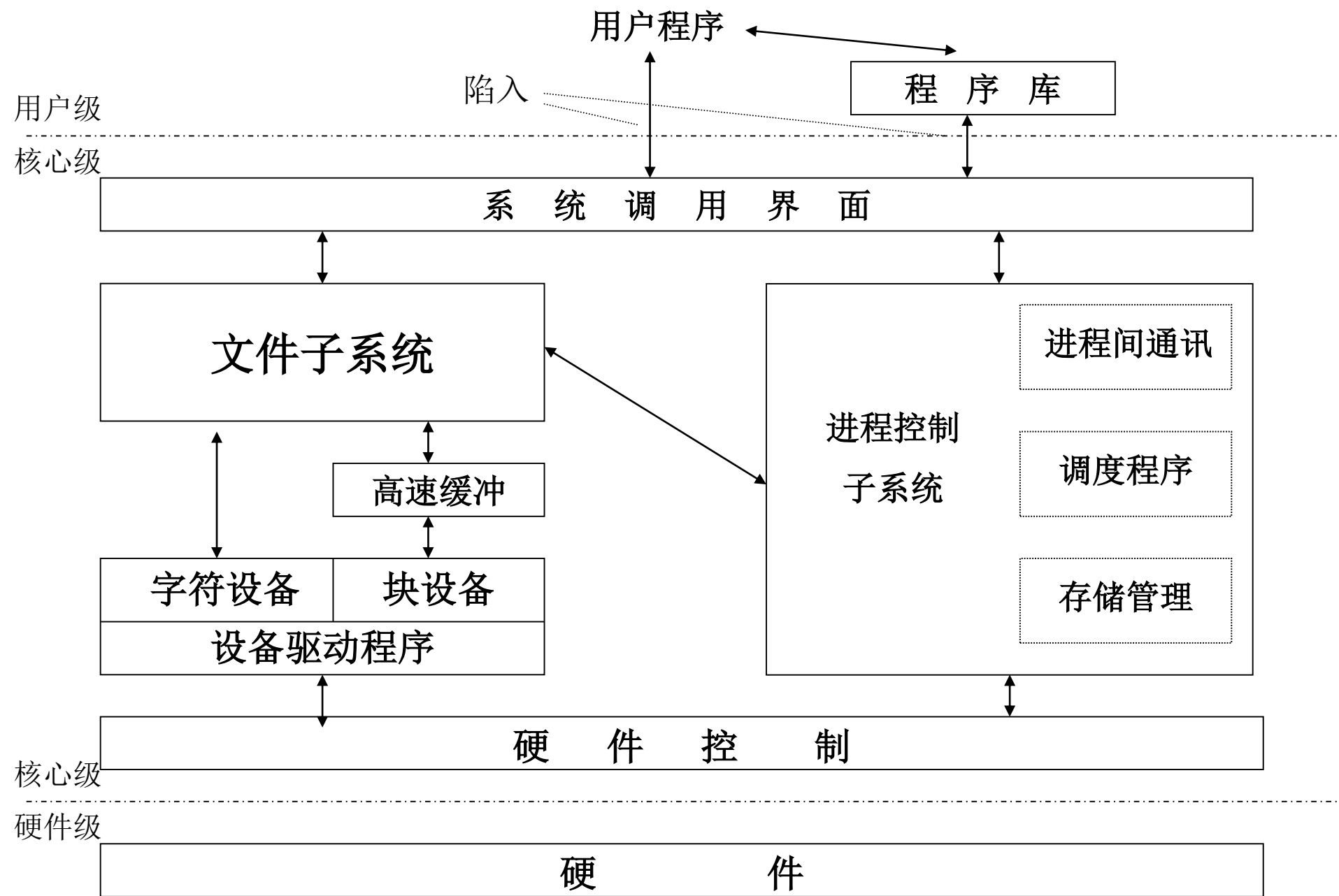
进程的调度、同步和通讯

- ④ 存贮管理：

在主存与二级存储之间对程序进行搬迁

- ⑤ 时钟管理：

把cpu的时间分配给当前最高优先权的进程。



## • 2.2 系统概念

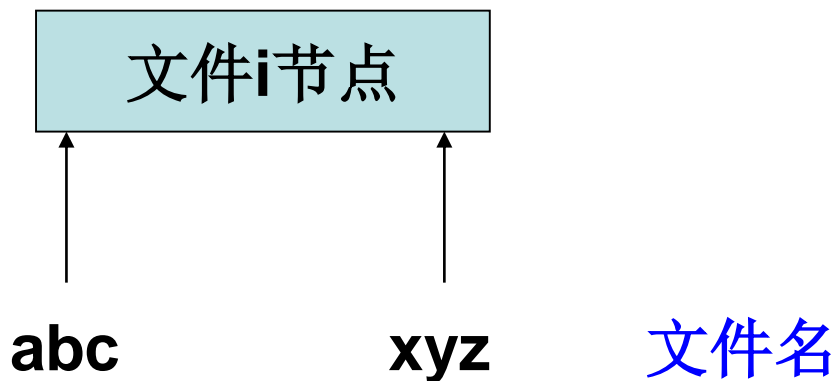
### 2.2.1 文件系统概貌

#### 1. 索引节点 (**index node**——**inode**)

##### **inode**特征:

- 文件的内部名称（或代号），方便机器操作；
- 每个文件都有一个且只有一个**inode**与之对应；
- **inode**存放文件的静态参数：存放地点、所有者、文件类型、存取权限、文件大小等；
- 每个文件都可以有多个名字，但都映射到同一个**inode**上；
- 各**inode**之间以**inode**号相区别；

## 2. 链结 (link) ——对应命令名 ln



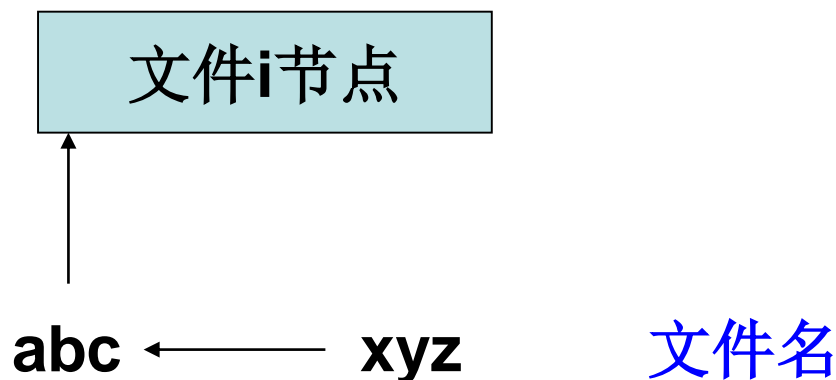
- 一个文件可有多多个名字，多个名字都对应同一个文件i节点，每个名字就是该文件节点的一个链结；
- 一个普通文件的名字个数，就是该文件的链结数；
- 每个链接名可以放在不同的目录下（同一个文件系统下）；
- 删除一个链接名时，文件链接数减一。如链接数不为零，则文件（节点）仍然存在。

## 使用文件链结的目的：

- ①方便用户的使用习惯，如“列目录”，可用**ls**、**dir**、**list**、**lc**等；
- ②误删文件时可补救，又不多占空间。**abc**和**xyz**具有相同的i结点号；
- ③减少移植应用程序时，因使用指定位置的文件，而拷贝该文件到指定位置去的麻烦。


### 3. 符号链结 (symbol link)

——对应命令名 **ln -s**



- 给文件的名称再取一个名字，而不是给文件节点再取一个名字。
- 链接的是“符号”而不是文件，因此“符号”可以是不存在的文件，即无意义的字符串。
- **abc**和**xyz**具有不同的**inode**号，**xyz**的内容是它所指向的文件的字符串，大小是字符串长度为3字节。
- “普通链结”中各名字必须在同一文件系统中，“符号链结”可在不同的文件系统中。

## 4. 活动i节点表（索引节点表）—— **inode**表

在内存中存放当前要使用的文件**inode**的表（或称为活动i节点表），表中的每一个表项对应一个当前正被使用的文件的状态信息。这样要使用（打开）同一个文件的进程不必再到盘上去寻找了，（共享！） 

## 5. 用户打开文件表（或称用户文件描述符表）

在系统中每一个进程都有一个描述该进程的数据结构**user**（类似于描述文件的i节点），在**user**中有一个数组，存放一组指针指向系统打开文件表中该进程打开的文件所对应的表项。

**struct file \*u\_ofile[NOFILE]**

**NOFILE** 为每个进程最多可同时打开的文件数，这与系统中的进程数和内存大小以及交换区大小等有关，一般为**20~100**。

这个**u\_ofile**数组就是该进程的用户打开文件表。

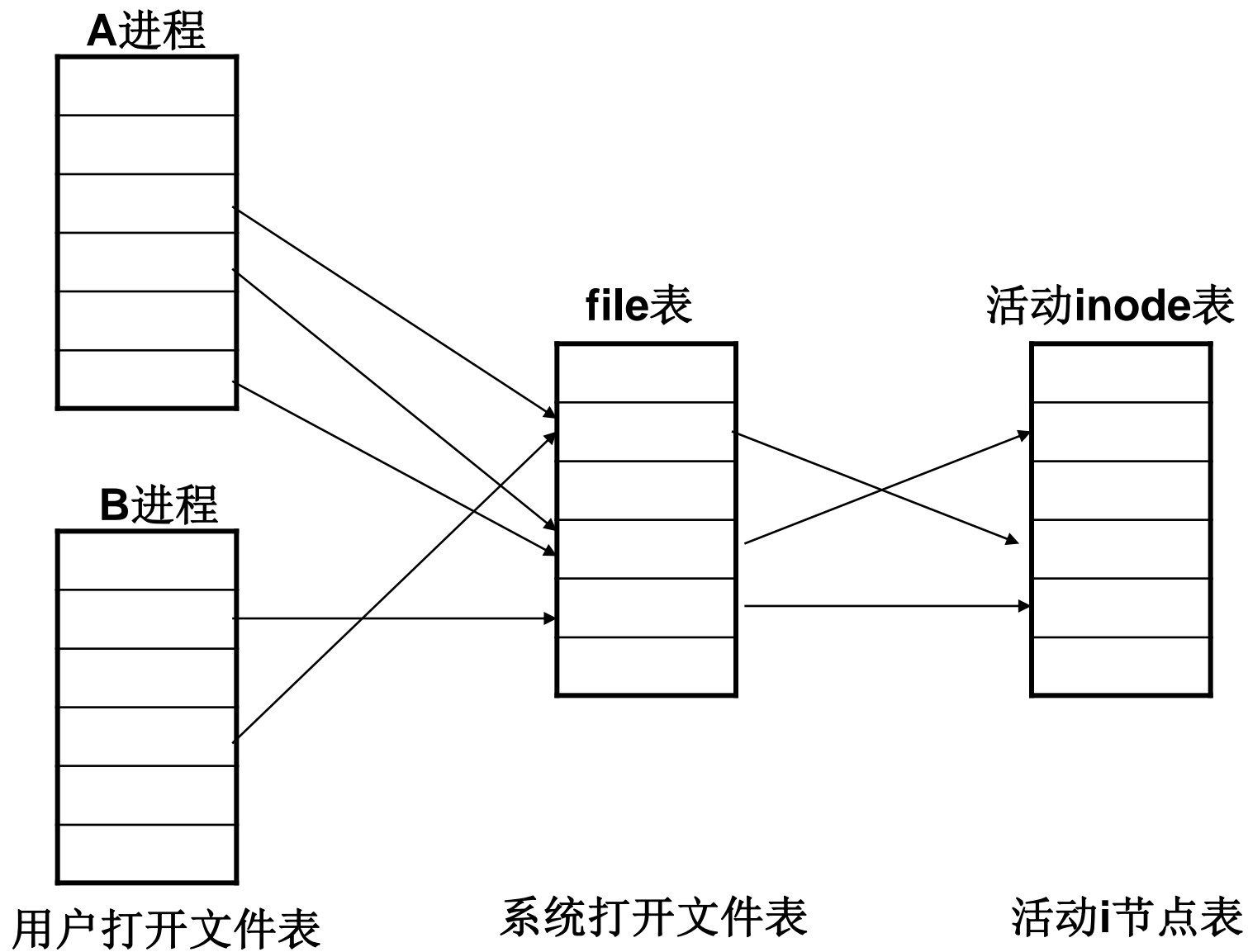


## 6. 系统打开文件表（**file**表）

系统打开文件表主要存放被打开文件的读写指针。

因为一个进程在一个时间片内可能读写不完所需内容，需要在下一个时间片继续从上一个时间片结束时的读写位置开始读写，故在进程生存期间应保持一读写指针。

此外**file**表中还存放被打开文件的动态信息：如文件状态、引用计数（当前使用该文件的进程数）等。



## 为什么要单独设立一个**file**表来存放读写指针呢？

由于可能有多个进程要共享一个被打开文件的**inode**，而每个进程的读写指针都不相同，故不能放在**inode**表中。

另一方面，要使不同进程的打开文件指针（文件描述符）或同一进程的不同打开文件指针能够共享一个打开文件指针（协同操作），就不能把读写指针放进某一个进程的用户打开文件表中。

因此只能在用户打开文件表和活动**inode**表之外再建立一个系统打开文件表（**file**表）来存放读写指针。

# UNIX操作系统中共享活动文件的方法:

在内存中某个活动文件的副本只有一个，不同的进程采用不同的指针指向这文件的副本。由于任一时刻只有一个进程在运行（微观上看），故该文件也只要求内存中有一个副本即可，只是各个进程有自己的读写指针而已。

这是在**UNIX**系统中共享文件（包括用户文件和系统文件）的主要方法。对其它资源的共享采用的是与之相似的另外几种方法。

## 2.2.2 进程

### 相关概念：

#### 映像——

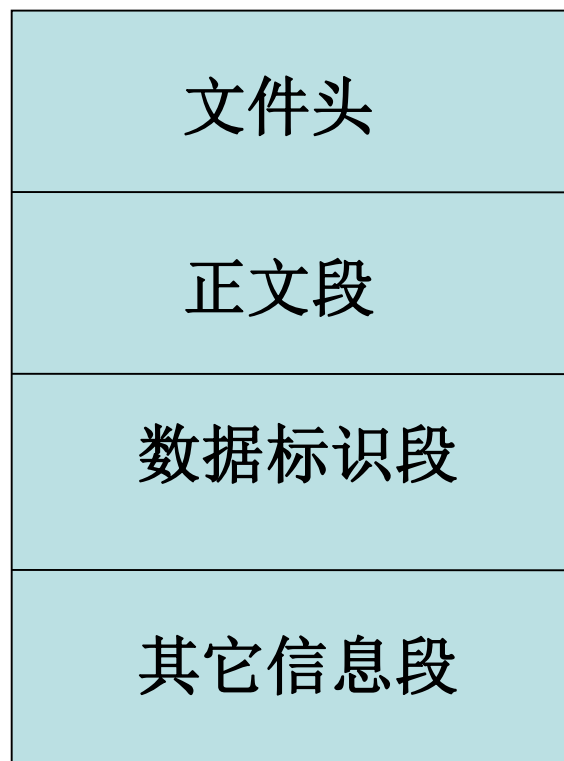
程序以及与动态执行该程序有关的各种信息的集合(类似于历史档案)。它包括存储器映象、通用寄存器映象，地址映射空间、打开文件状态等。

#### 进程——

对映像的执行。对映像的执行也就是一个程序在虚拟机上动态执行的过程。

## 1. 可执行文件的构成：

进程是可执行文件的一次执行实例，高级语言程序经过编译或汇编语言程序经过汇编后所产生的、缺省名为**a.out**的可执行文件的结构包括图示四个部分：



## 文件头——

- 文件的幻数（**magic number**）
- 编译器的版本号
- 机器类型
- 正文段、数据标识段、其它信息段的大小
- 程序入口点

## 正文段——

程序的功能代码

## 数据标识段——

标识未初始化的数据要占用的空间大小

## 其它信息段——

主要用于存放符号表

## 2. 程序的执行

一个进程在执行系统调用**exec**时，把可执行文件装入本进程的三个区域中：

**正文区**：对应可执行文件的正文段

**数据区**：对应可执行文件的数据标识段

**堆栈区**：新建立的进程工作区

堆栈主要用于传递参数，保护现场，存放返回地址以及为局部动态变量提供存储区。

进程在核心态下运行时的工作区为核心栈，在用户态下运行时的工作区为用户栈。核心栈和用户栈不能交叉使用。



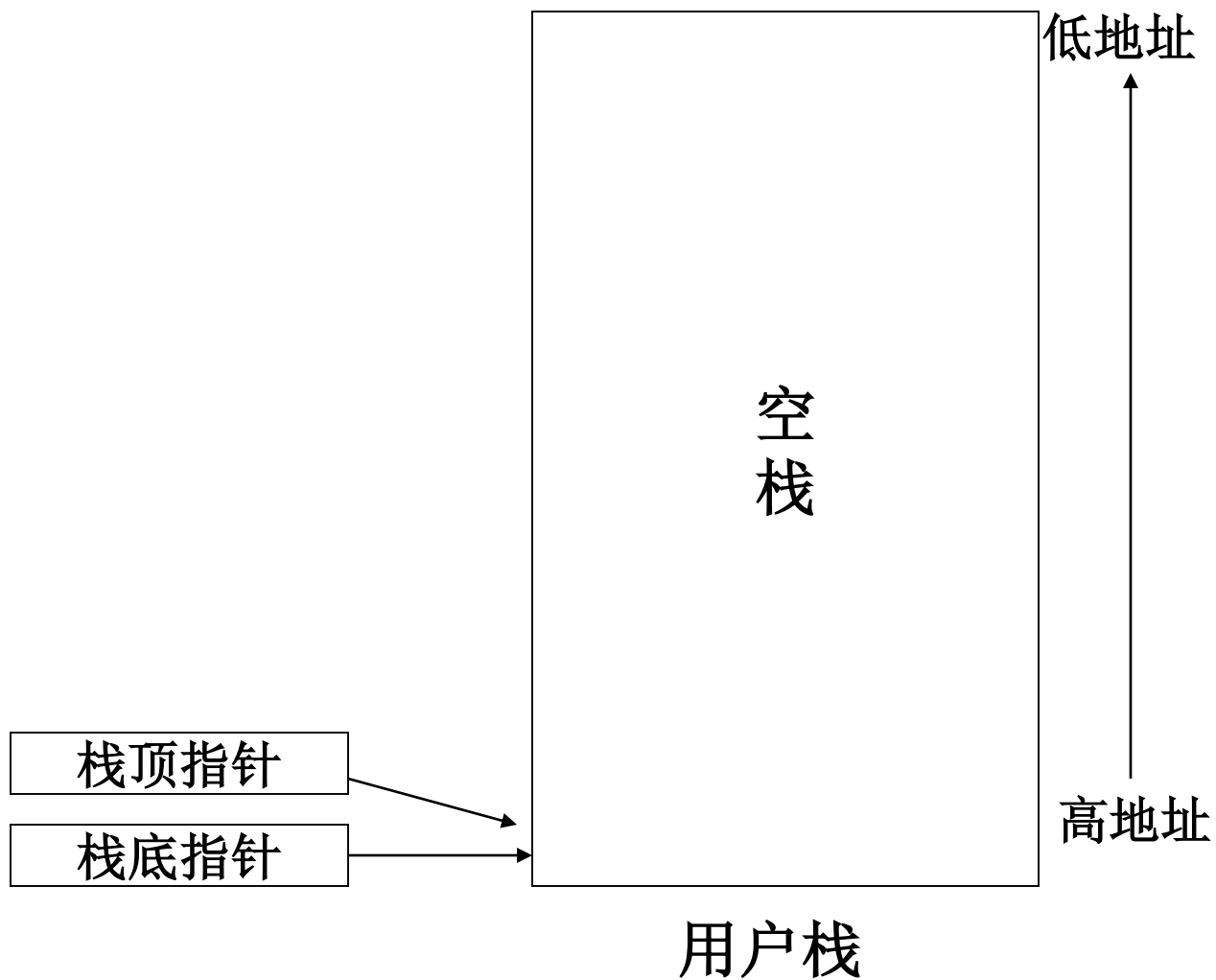
堆栈使用举例：如下程序在主程序中调用函数，并进行参数传递：

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

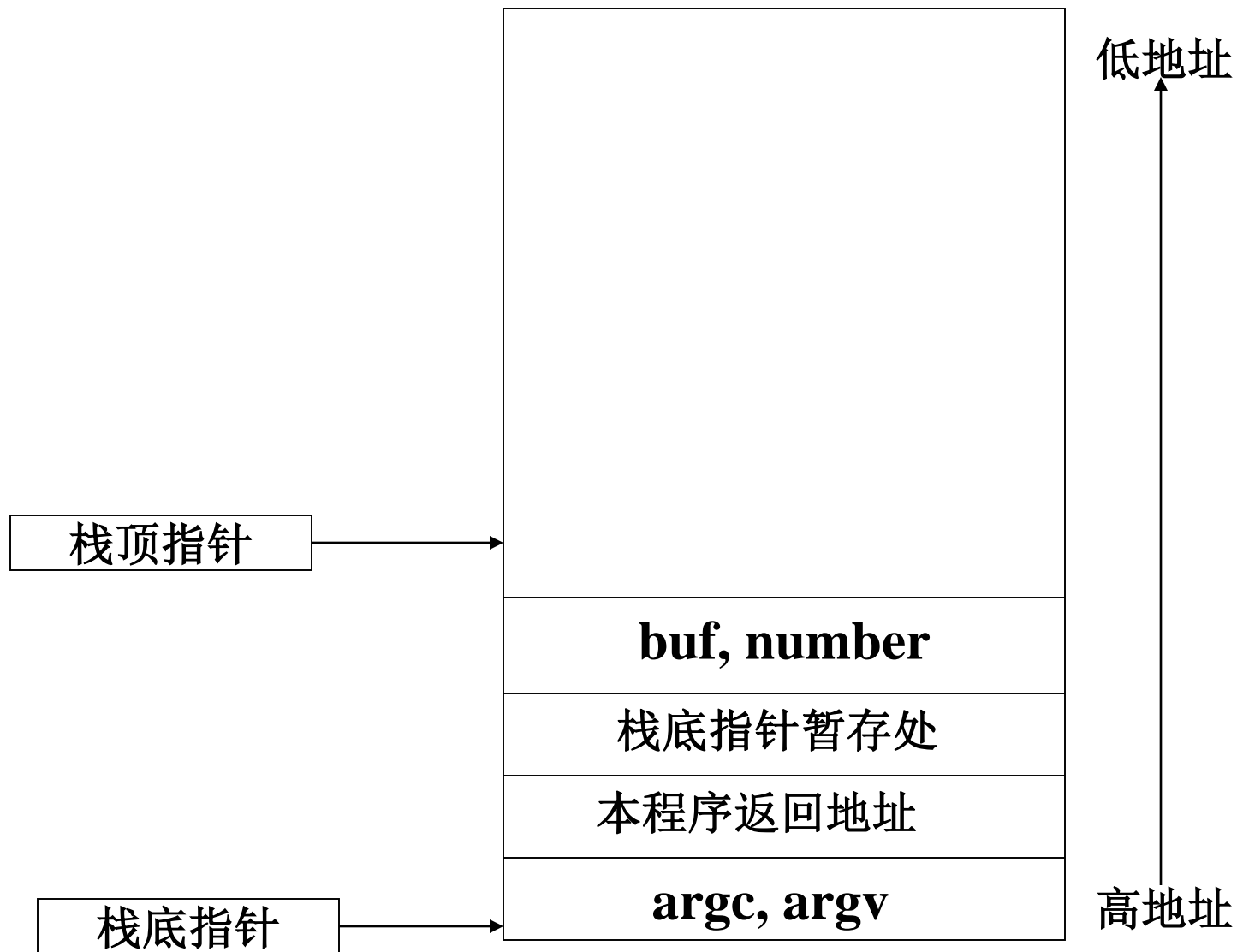
```
{  
    char buf[1024];  
    int number;  
    ...  
    readfile (buf, number);  
    ...  
}
```

```
readfile (char buffer[ ], int line)
```

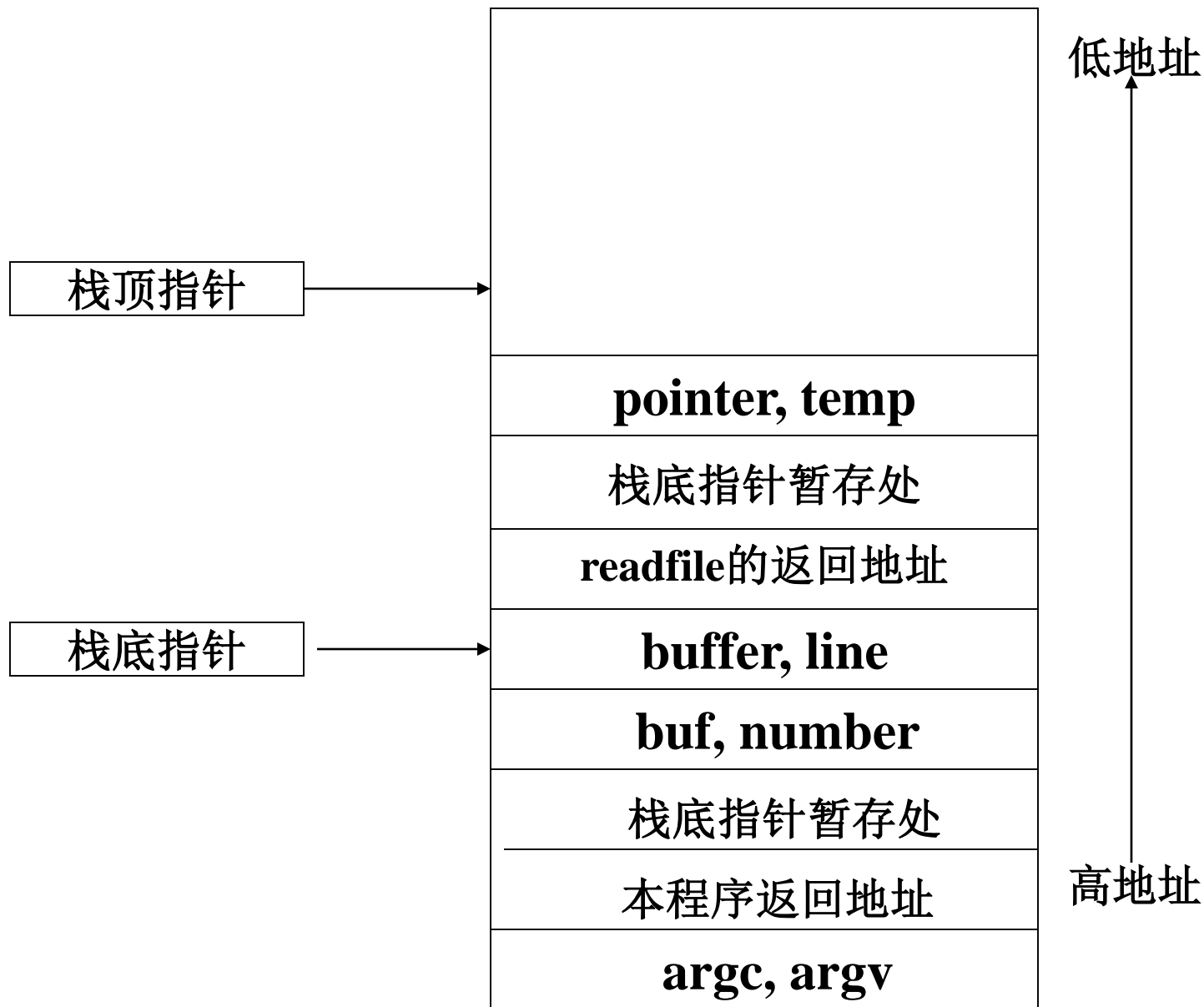
```
{  
    char *pointer;  
    int temp;  
    ...  
}
```



进入主程序时的堆栈状况



调用main()时



调用readfile时

### 3. 进程的标识

进程由其进程标识号**PID**来识别。

#### 0#进程

是由机器上电时“手工”创建的，调用**fork**创建了1#进程后，成为对换进程（**swap**）。

#### 1#进程

**init**进程，由它来创建系统初始化过程中所需的其它所有的进程。

#### 父进程

调用**fork**系统调用的进程

#### 子进程

由系统调用**fork**产生的进程

除0#进程外，其它所有进程都是另一个进程调用**fork**后产生的。

## 4. 进程状态及状态转换

### ①运行状态

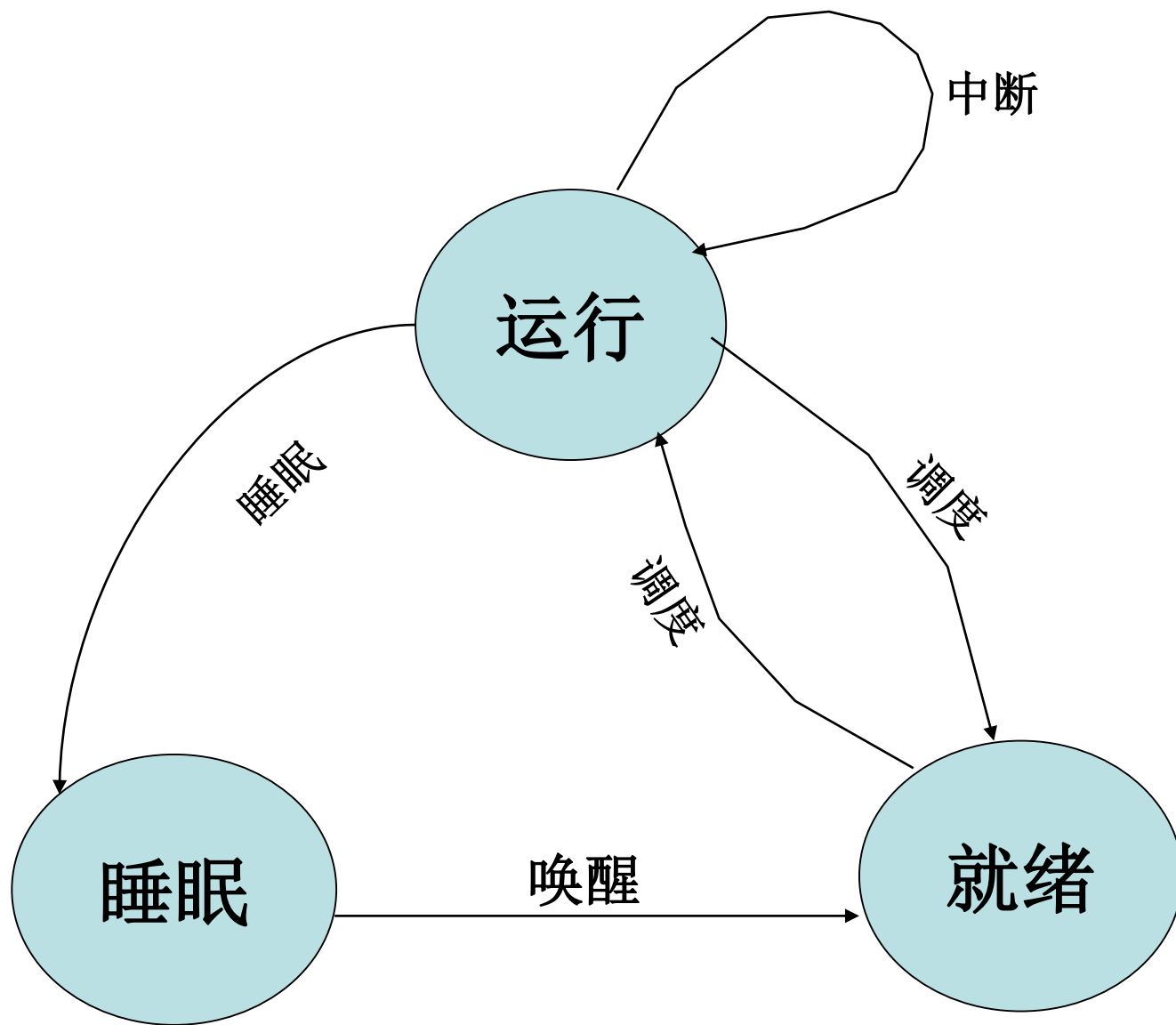
此时进程正在占用处理机，进程的全部映像驻在内存中。

### ②就绪状态

此时进程基本具备了运行条件，正在等待使用处理机。

### ③睡眠状态

进程不具备运行条件，需等待某种事件的发生，无法继续执行下去。



## 5. 在UNIX环境下，进程有如下特征：

- ① 每个进程在核心进程表（**proc**数组）都占有一项，在其中保留相应的状态信息。
- ② 每个进程都有一个“每进程数据区（**per process data area---ppda**）”保留相应进程更多的信息和核心栈；
- ③ 处理机的全部工作就是在某个时候执行某个进程
- ④ 一个进程可生成或消灭另一进程
- ⑤ 一个进程中可申请并占有资源
- ⑥ 一个进程只沿着一个特定的指令序列运行，不会跳转到另一个进程的指令序列中去，也不能访问别的进程的数据和堆栈。（抗病毒传播的重要原因之一）