版本号：v1.0 阶段：初稿

# 操作系统面试宝典

MRL Liu

（未经许可，不得传播）

2022年01月22日

本文以《现代操作系统（原书第3版）》》为参考，在具备基本逻辑的基础上只记录笔者认为对计算机网络领域面试相关的知识，深入研究可参考其他资料。

# 操作系统基础

操作系统的定义有很多种，但总体来说，我们可以认为它扮演着这样一个角色：

|  |
| --- |
| 操作系统是介于硬件资源和应用程序之间的一个系统软件，它可以管理硬件资源，同时也可以管理应用程序。 |

### 根据以上的叙述，我们也可以总结出操作系统的主要功能：

1. 硬件资源管理

计算机必要的硬件资源有CPU、内存、硬盘和I/O设备。这些硬件资源是有限的，所以必须进行合理地资源分配和资源回收。

1. 应用程序管理

操作系统为应用程序提供硬件资源的服务，同时也要管理应用程序，即控制进程的生命周期、资源分配等。

## 一、进程管理

### 1、进程（process）

### （1）程序和进程

**程序**在运行前是存储在硬盘里的静态文件，运行时才被**CPU**加载到内存。**进程本质上就是一个正在运行的程序**，是对运行时程序的封装，其包含了该程序运行的所有资源和操作。

我们通过以下比喻来理解CPU对进程的管理：

|  |
| --- |
| CPU的每个核是一个厨师，程序是保存在存储箱里的食谱，其用来制作菜谱前一直放在存储箱里，程序的各种输入数据就是做菜的各种原料，程序运行后可以实现的各种功能就是最终做好的菜品。  进程就是厨师阅读食谱、获取各种原料并做菜的一系列动作的总和，厨房就是该进程对应的地址空间，厨师烹饪每一道菜品都必须在对应的厨房按照对应的食谱进行，即使烹饪同一道菜，也必须在不同的厨房，所以计算机中的同一个程序运行了两遍，也是两个不同的进程。 |

### （2）CPU的并发

单核的CPU在任意时刻只能运行一个程序（一个厨师在任意时刻只能按照一个食谱做一道菜），但是现在的复杂操作系统基本都是多任务的，即CPU可以同时运行多个程序。我们继续上述的比喻来描述这种CPU同时运行多个程序的机制：

|  |
| --- |
| 假设现在需要一个单核CPU同时运行多个程序：A、B，即厨师要同时按照不同的食谱A、B做不同的菜品A、B。  厨师的做法是，在某段时间内，先去厨房A按照食谱A烹饪菜品A，然后做一小段时间后，记录下此时食谱A的进度（保存进程的当前状态）；然后跑去厨房B按照食谱B烹饪菜品B，同样做一小段时间后，记录下此时食谱B的进度，返回厨房A继续按照食谱A烹饪菜品A。  在上述过程中，一个厨师在任意时刻只能按照一个食谱进行烹饪，每工作一段时间就切换到另一个食谱。这样子的好处是，对每个菜品来讲，每次它被烹饪时都独占了厨师的精力，对顾客来讲，这些菜品看起来是同时在烹饪的。 |

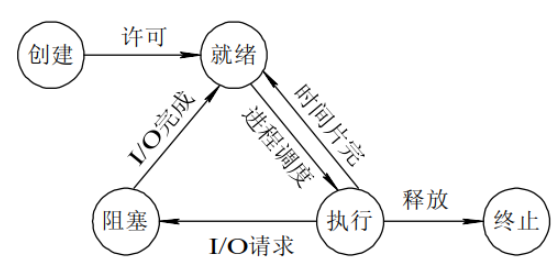
同样的原理，单核CPU在任意时刻只能运行一个程序，所以单核CPU在每个时间片（由程序计时器计数的一小段时间）内运行一个程序，时间片时结束就切换到另一个程序。由于每个时间片足够小，对每个程序来讲，每次运行都似乎独占了CPU的资源；对每个用户来讲，每个程序似乎都可以同时运行。我们将这种机制称为CPU的并发。注意以下CPU的并发和并行概念的区别：

|  |
| --- |
| CPU的**单个核心**在**每个时间片**内分别执⾏多个进程，称为**CPU的并发**；  CPU的**多个核心**在**同一个时间片**内同时执行多个进程，称为**CPU的并行**。 |

我们注意到，对于CPU的并发来讲，CPU从一个进程切换到另一个进程需要保存进程的状态信息。

### （3）进程的状态

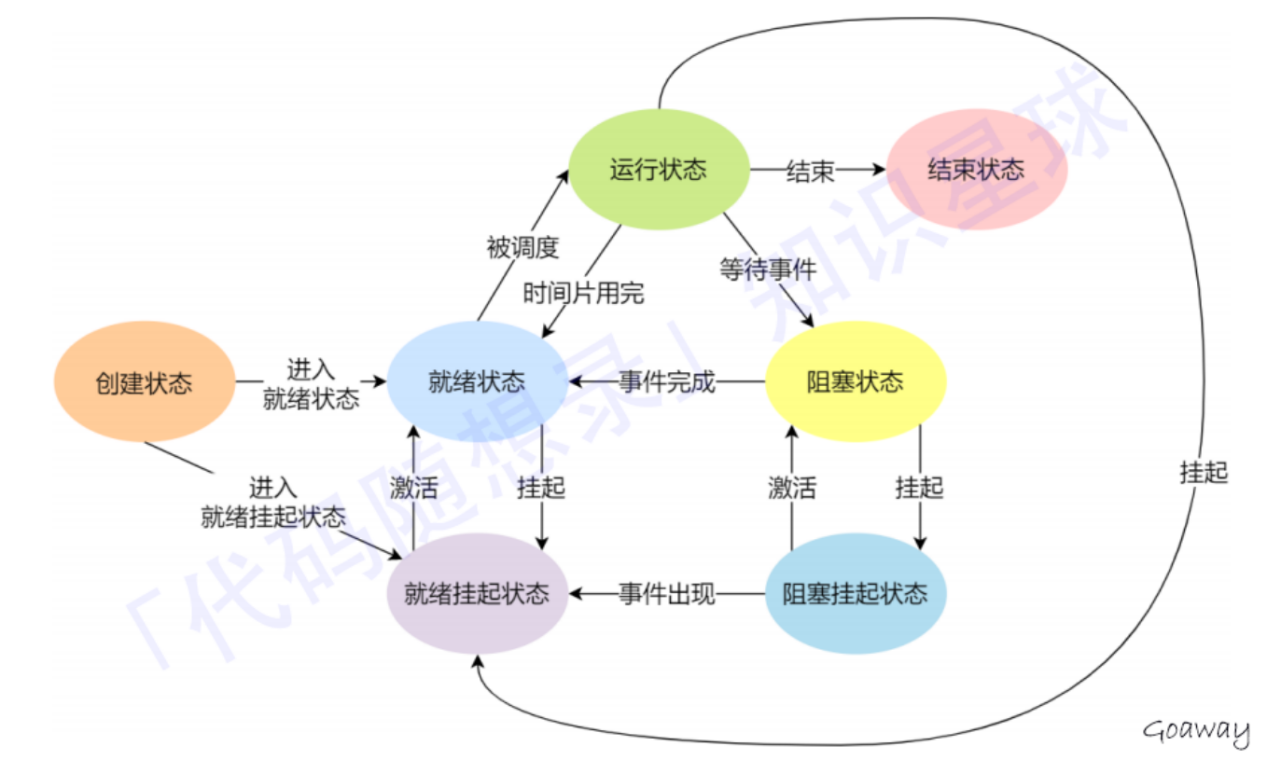
在前文描述中，我们已经知道，同一个进程存在多个状态，具体来讲，进程从创建到终止大致分为如下5种状态，即进程的五态模型：



|  |  |  |
| --- | --- | --- |
| **状态类型** | **状态特点** | **状态切换条件** |
| 创建态 | 创建进程，调度程序为其分配内存空间等资源 | 如果成功获取除处理机以外的其他资源则自动进入就绪态。 |
| 就绪态 | 因为其他进程正在运行而暂时停止。 | 等待分配处理机资源，得到后可立即运行 |
| 执行态  （运行态） | 该进程正在运行，处于独占用一个CPU核心的时间片 | 时间片用尽后自动进入就绪状态 |
| 阻塞态 | 该进程正在等待某一事件发生而暂时停止运行。 | 比如等待客户端连接或用户输入。 |
| 终止态 | 进程结束，资源被回收。 | 可能是任务结束或者遇到已知问题而主动结束也可能是遇到严重未知错误或被其他进程杀死而被动结束 |

为了满足系统和用户观察、分析和及时修改进程的需要，还引入了一个对进程的重要操作——挂起，增加挂起操作后，进程的就绪状态分为活动就绪态（激活）和静止就绪态（挂起），阻塞状态分为活动阻塞态（激活）和静止阻塞态（挂起）。

进程的七态模型：



注意，在所有状态的切换中，只有就绪态和运行态是双向转换的，其他都是单向转换。

### （4）进程的实现

为了实现上述描述的进程模型，操作系统维护着一张进程表（本质是一个结构数组或结构链表），每一个进程占用其中的一个进程表项，即进程控制块（PCB），每个PCB包含了一个进程的重要状态信息，例如程序计时器、堆栈指针、内存分配状况、所打开文件的状态、账号的调度信息等。

### （5）守护进程

启动操作系统时，通常会创建若干个进程，其中有些需要同用户交互并替用户完成某些工作，这时这些进程是前台进程，如果进程不需要与用户进行交互但是还有执行一些必要的功能，那么这些进程就是后台进程。

守护进程（daemon，也叫精灵进程）是一种特殊的后台进程，其通常完全独立于控制终端，并且周期性地执行某种任务或等待处理某些事件，例如收发电子邮件的网络进程，在大部分事件下都处于休眠状态，但是当接收到电子邮件时就被唤醒。操作系统中有很多守护进程。

### （6）进程间通信（Inter Process Communication）

操作系统的每个进程都有不同的用户地址空间,任何一个进程的全局变量在另一个进程中都不可见,所以进程之间必须通过内核交换数据,在内核中开辟一块缓冲区,进程1把数据从用户空间拷到内核缓冲区,进程2再从内核缓冲区把数据读走,内核提供的这种机制称为进程间通信(IPC,InterProcess Communication)。

注意我们现在讨论的进程间通信都是同一主机上的不同进程间的通信，不同主机间的进程通信是网络编程中的套接字（socket）。

：

无名管道

有名管道

信号

消息队列

共享内存

信号量

存储映射

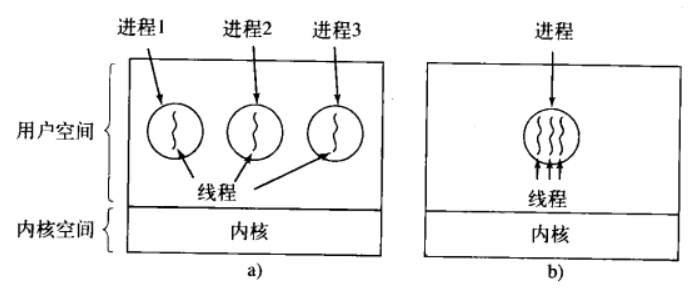
### （7）进程间调度（Inter Process Communication）

在计算机系统是多道程序设计系统时，多个进程或线程间需要同时竞争CPU。在操作系统中，负责完成这一工作的就是调度程序（scheduler），该类程序使用的算法就是调度算法（scheduling algorithm）

### 线程（thread）

#### （1）传统操作系统和现代操作系统的线程

线程概念也是随着操作系统不断完善的。传统操作系统中，每个进程都有一个对应的地址空间（address space）和一个控制线程。但是现代操作系统中，同一个进程中可以并行运行多个线程，多个线程共享同一个进程的地址空间和其他资源等，所以从现在意义看，线程也叫做轻量级进程。如下图：



传统操作系统a） 现代操作系统b)

多线程的独特优势？（为什么需要多线程设计？）

|  |
| --- |
| （1）多线程可以满足需要同时进行多种功能的应用需求。很多应用程序的功能实现需要同时有多种活动同时进行，而且有些活动可能需要阻塞，比如服务器等待客户端连接，服务器在等待客户端连接的同时，可能需要同时处理用户的输入，显然这种需求单进程/单线程是完成不了的。  （2）多线程是轻量级进程，比单纯的多进程设计更快。多线程是同一个地址空间内的共享同一个地址空间和所有可用数据的并行实体，线程比进程更加轻量级，更容易创建和撤销，在许多系统中创建线程比创建进程要快10~100倍。  （3）多线程可以大大提升需要大量计算或大量I/O处理的应用的执行速度。即使一个应用不需要同时执行多种功能，但是只要其计算量大或者读写操作多，都可以考虑多线程。 |

线程的优点：

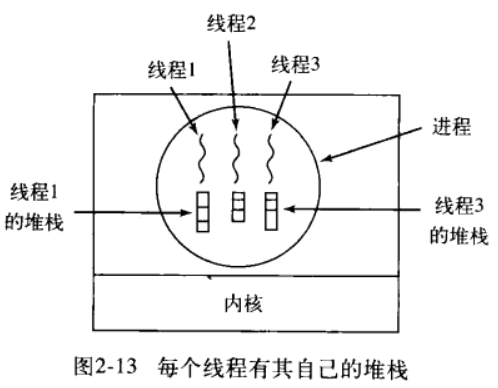
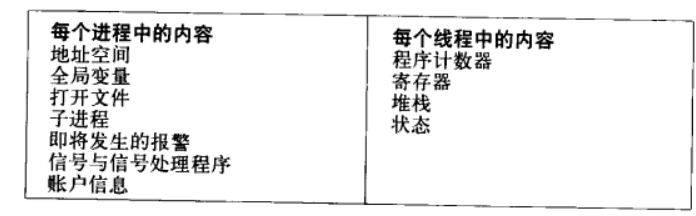
1. 提高程序并发性
2. 在并发性程序上比多进程开销更小、
3. 数据通信、共享数据比多进程方便

线程的缺点：

1. 依赖线程包的库函数，实现上可能不稳定；
2. 调试、编写困难、gdb不支持
3. 对信号支持不好

#### （2）进程和线程的区别

请注意进程与线程的区别：进程是操作系统最小的资源分配的单位，操作系统只为每一个进程分配资源和回收资源；线程是操作系统的最小的执行单位，同一个进程中的所有线程共享该进程的资源，下图是所有线程共享的内容和每个线程独有的内容的对比：

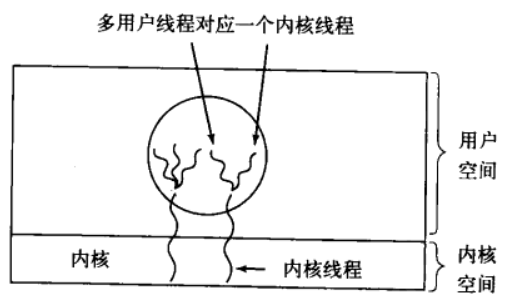
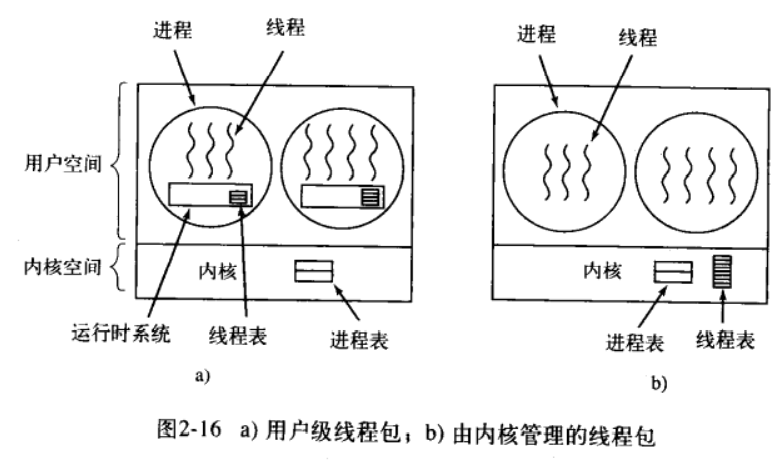


所有线程共享的资源：文件描述符表、每种信号的处理方式、当前工作目录、用户ID和组ID

线程非共享资源：线程ID、内核栈、用户空间栈、信号屏蔽字、errno变量

#### 多线程的实现

前文我们知道传统操作系统中每个进程中只有一个控制线程，其本身是不支持多线程的。有两种主要的方法来实现多线程机制：用户级线程和内核级线程。前者在操作系统的用户空间实现线程包，后者在操作系统的内核空间实现线程包。



|  |  |  |  |
| --- | --- | --- | --- |
| 线程类型 | 特点 | 优点 | 缺点 |
| 用户级线程 | 在用户空间实现线程包，在用户空间每个进程都有其独立的线程表 | 1. 移植性强，可用于不支持多线程的操作系统 2. 线程切换快，用户空间的线程切换不需要操作系统在内核态和用户态转换，比内核级线程快一个数量级以上。 3. 自由度高，允许开发者自定义每个进程中多线程的调度算法。 | 1. 某个用户线程阻塞时该进程下的其他线程都会阻塞 2. 某个用户线程运行后且不主动停止，其他线程就不能运行。 |
| 内核级线程 | 在内核空间实现线程包，在内核空间存在记录系统中所有线程的线程表 | 1. 某个内核线程阻塞时不会影响其他内核线程的执行。 2. 每个内核线程都可以直接获取CPU资源，拥有更多的CPU使用时间 | 1. 需要由内核来维护内核线程的信息，占用内核资源 2. 内核线程的创建、终止和切换都是在内核空间中进行的，开销比较大。 |
| 轻量级线程（LWP） | 轻量级进程(LWP)是建立在内核之上并由内核支持的用户线程，每一个轻量级进程都与一个特定的内核线程关联。 | 综合了内核级线程和用户级线程的优点 |  |

### 3、进程间通信（process）

## **存储系统**

## **文件系统**

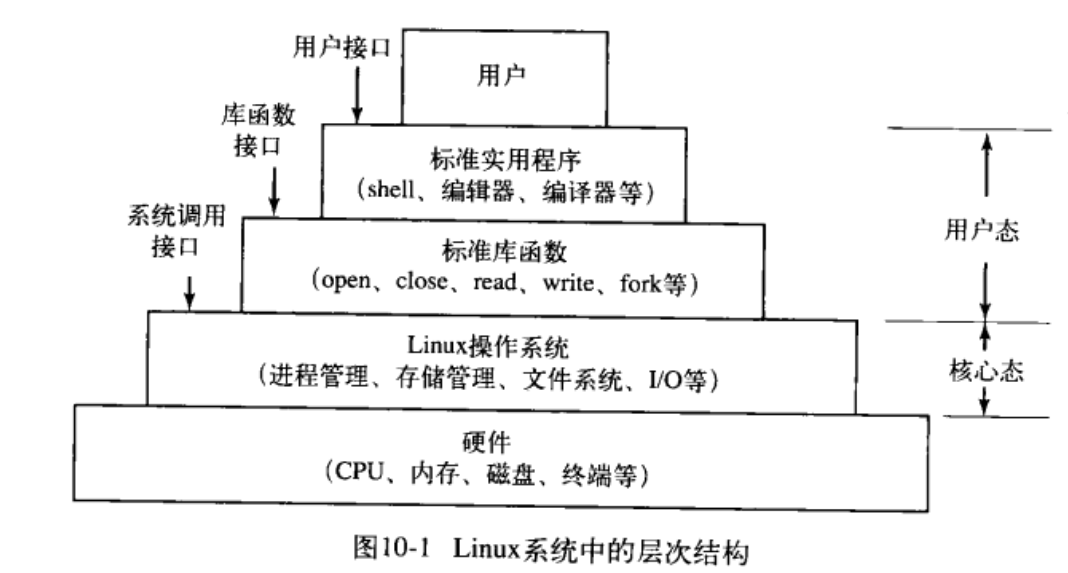
## **输入输出**

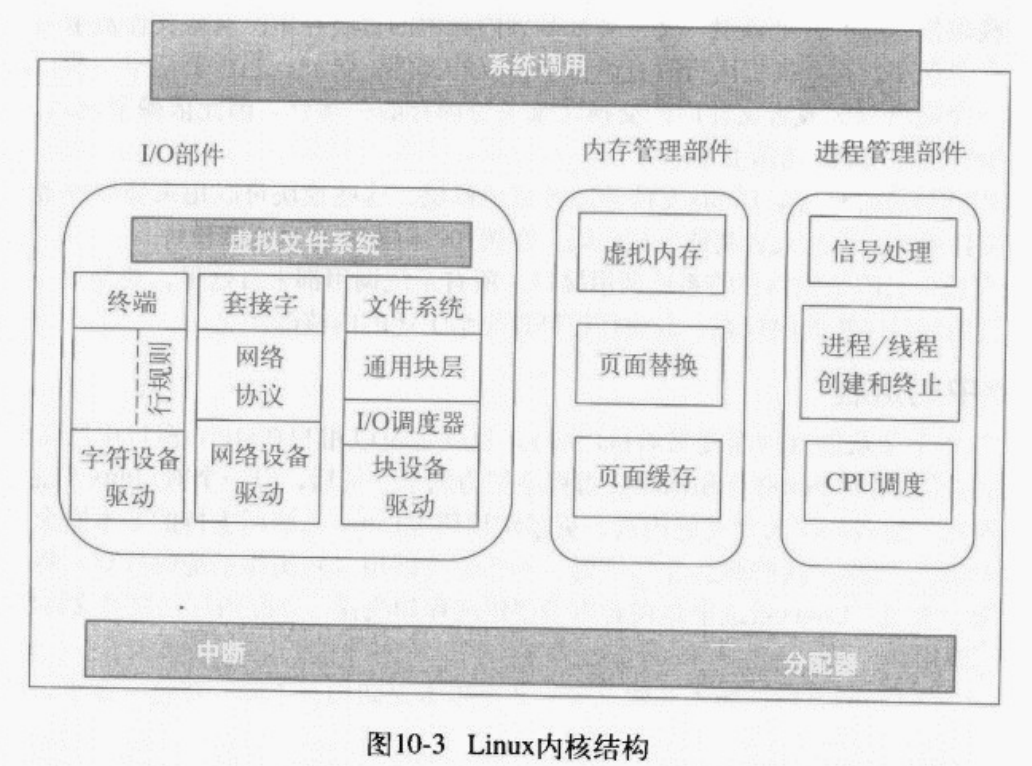
## **互斥同步**

# **Linux操作系统**

Linux的三种接口：

系统调用接口、库函数接口、用户接口



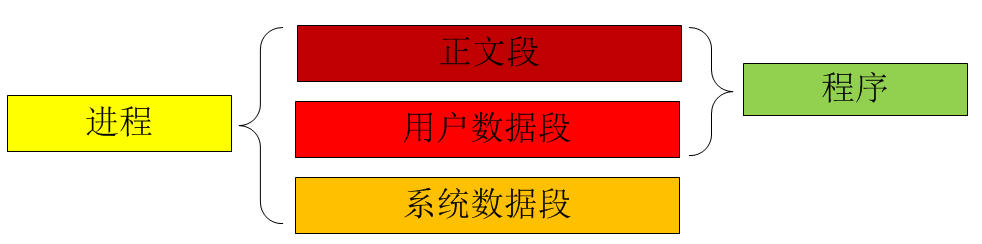


Linux的内核系统分为三个主要部分：

1. 虚拟文件系统：所有的I/O操作都通过设备驱动程序封装为对一个文件的读写操作
2. 存储管理系统：维护虚拟内存到物理内存的映射、负责维护最近访问页面的缓存以及实现一个好的页面置换算法，根据需要将数据和代码页读入内存中
3. 进程管理系统：使用进程调度器进行 进程的创建和终止

## 一、进程管理系统

### Linux系统的PCB



Linux系统中进程分为三部分：

1. 数据段：存放全局变量、常数、以及动态分配的数据空间
2. 正文段：存放程序代码
3. 堆栈段：存放函数返回地址、函数参数以及局部变量

其中数据段根据实现的位置不同分为用户数据段和系统数据段，用户数据段在用户空间实现，系统数据段在内核空间实现。操作系统中的进程表来存放每一个进程的系统数据段，即PCB（Process Control Block），也是操作系统描述进程的单位。Linux中的PCB就是task\_struct结构体，task\_struct也被称为Linux的进程描述符。

task\_struct中有许多变量，涉及进程管理、存储管理、文件系统等，这里我们先介绍如下几个重要变量：

（1）进程标识符（PID）。Linux内核通过唯一的进程标识符来标识每一个线程对应的任务进程，新创建的进程通常是前一个进程的PID加1，PID值有最大值（32767）。

（2）父进程标识符（PPID）。Linux内核通过复制原有的进程来创建新的进程，为了描述这种创建的父子关系，子进程中会保留有其父进程标识符。

（3）线程组标识符（TGID）。Linux内核支持多线程，通过TGID来标识每一个多线程组。一个多线程程序对应的进程只有一个主线程，当主线程创建其他线程时，就复制了主线程的TGID，内核可以通过TGID判断该线程属于哪个线程组，即属于哪个进程，getpid()返回的就是TGID。

（4）进程状态（state）。进程当前的状态，取值为TASK\_RUNNING、TASK\_RUNNING、TASK\_INTERRUBTIBLE、TASK\_UNINTERRUPTIBLE、TASK\_KILLABLE等。

### 2、进程的状态转换

Linux中的进程运行状态有以下几种主要状态：

A、运行状态（TASK\_RUNNING)：程序当前正在运行，或者在运行队列中等待调度。

B、可中断的阻塞状态（等待状态）（TASK\_INTERRUPTIBLE)：进程处于阻塞（睡眠）状态，正在等待某些事件的发生或者能够占用某些资源。处在这种状态下的进程可以被信号中断。接收到信号或被显示的唤醒呼叫（如调用wake\_up系列宏：wake\_up、wake\_up\_interruptible等）唤醒之后，进程将转变为运行（TASK\_RUNNING）状态。

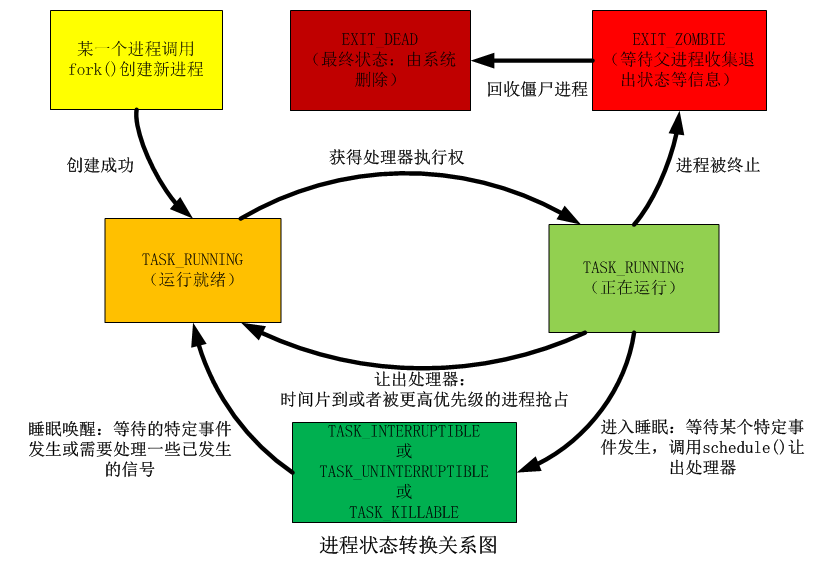
C、不可中断的阻塞状态（等待状态）(TASK\_UNINTERRUPTIBLE)：该状态类似于可中断的阻塞状态（TASK\_INTERRUPTIBLE)，但是它不会处理信号，把信号传递到这种状态下的进程不能改变它的状态。在一些特定的情况下（进程必须等待，知道某些不能被中断的事件发生），这种状态是很有用的。它不能被信号唤醒，只有它所等待的事件发生时，进程才能被显示唤醒呼叫唤醒。

D、暂停状态(TASK\_STOPPED)：进程的执行被暂停，当进程收到SIGSTOP、SIGTSTP、SIGTTIN、SIGTTOU等信号，就会进入暂停状态。

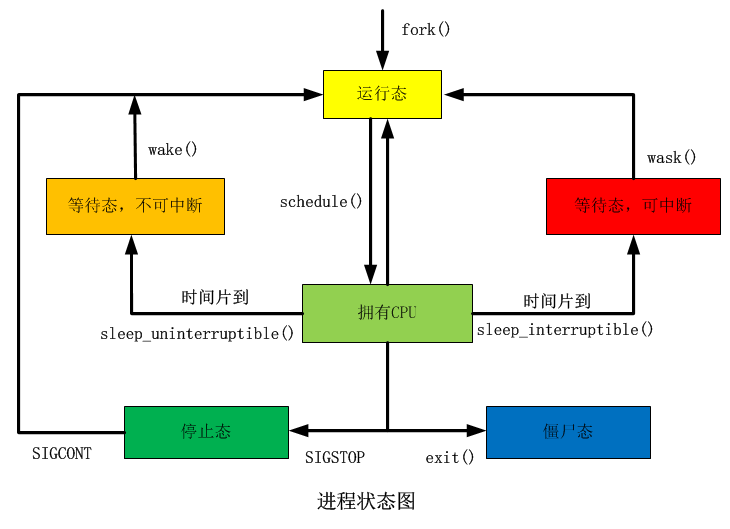
E、僵尸状态(TASK\_ZOMBLE)：子进程运行结束，父进程未退出，并且未使用wait函数族（如使用waitpid（）函数）等系统调用来回收子进程的退出状态。处在该状态下的子进程已经放弃了几乎所有的内存空间，没有任何可执行代码，也不能被调度，仅仅在进程列表中保留一个位置，记载该进程的退出状态等信息供其父进程收集。

F、消亡状态（死亡状态）(TASK\_DEAD)：这是最终状态，这是一个已终止的进程，但还在内核的进程表中占有一个task\_struct结构。父进程调用wait函数族回收之后，子进程由系统彻底删除，不可见。

它们之间的转换关系如下图所示：



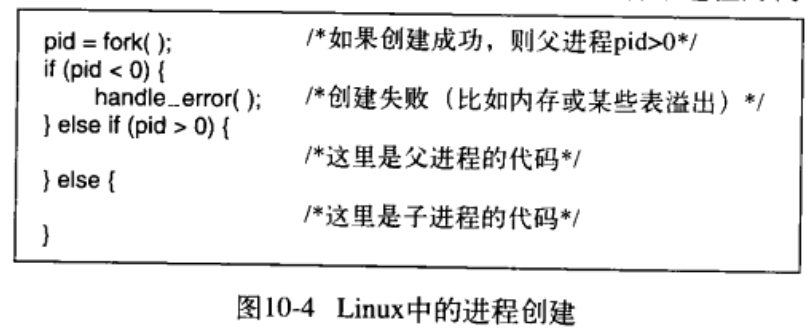
调用各个函数的状态如下：



### 3、进程的创建、执行和终止

#### （1）进程的创建

Linux系统中通过调用库函数接口fork()来复制当前进程创建一个新进程，调用fork函数的进程叫做父进程，新创建的进程叫做子进程。子进程与父进程的区别仅仅在于不同的PID、PPID和某些资源及统计量上。注意，Linux中的fork()函数使用的是写时复制页的技术，也就是内核在创建进程时，其资源并没有被复制过来，所有的资源以只读的方式和父进程共享数据，一旦子进程需要写入数据时才进行资源复制。这种写时复制技术可以使Linux拥有快速执行的能力，因此这个优化是非常重要的。



#### 进程的执行

exec函数族负责读取可执行文件并将其载入地址空间开始运行新创建的子进程。

#### （3）进程的终止

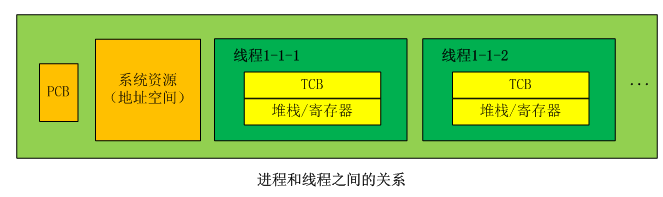
当父进程退出而它的子进程还在运行时，这些子进程就成为孤儿进程，孤儿进程将被Init进程（PID=1）收养，所以孤儿进程不会浪费系统资源。

父进程创建子进程后一般需要监控子进程的状态，如果子进程退出但是父进程没有读取其状态，则此时的子进程为僵尸进程。具体说，一个进程使用fork创建子进程，如果子进程退出但是父进程并没有调用wait或waitpid来获取子进程的状态信息，那么子进程的进程描述符task\_struct将会仍然存在，该子进程就是僵尸进程。僵尸进程会浪费系统资源。

僵尸进程产生原因：  
1、子进程结束后向父进程发出SIGCHLD信号，父进程默认忽略了它；  
2、父进程没有调用wait()或waitpid()函数来等待子进程的结束。  
避免僵尸进程方法：  
1、父进程调用wait()或者waitpid()等待子进程结束，这样处理父进程一般会阻塞在wait处而不能处理其他事情。  
2、捕捉SIGCHLD信号，并在信号处理函数里面调用wait函数，这样处理可避免1中描述的问题。

### 4、线程的用户态和内核态

从用户空间看，Linux中用户空间的线程使用一套通用的第三方pthread线程库，因此具有很好的移植性。一个进程可以拥有多个线程，其中每个线程共享该进程所拥有的的资源。要注意的是，由于线程共享了进程的资源和地址空间，因此，任何线程对系统资源的操作都会给其它线程带来影响。由此可知，多线程中的同步是非常重要的问题。在多线程系统中，进程与线程的关系如下图所示：



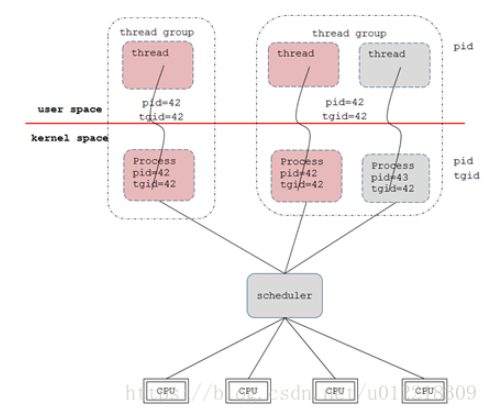
从内核空间看，Linux系统的每一个线程都是一个单独的任务，一个任务等同于一个内核级的进程，一个内核级的进程对应于一个线程组，对比如下：

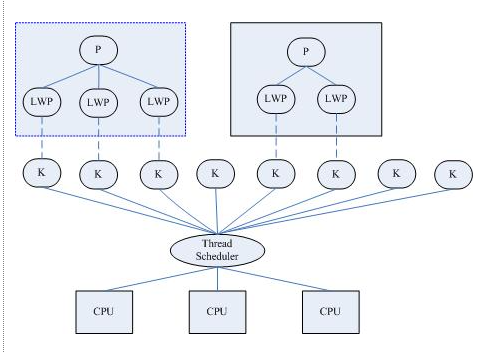
Linux创建进程时只有一种方式，即调用fork函数从一个父进程中复制而来。新创建的子进程在内核空间的进程表中创建一个新的task\_struct，其进程标识符PID和线程组标识符TGID都设为一个新值，然后在用户空间创建对应的控制线程。

Linux创建线程时只有一种方式，即调用pthread\_create函数从一个父线程中复制而来。新创建的子线程在内核空间的进程表中创建一个新的task\_struct，其进程标识符PID设为一个新值，线程组标识符TGID不变，然后在用户空间创建对应的用户级线程。

从上述内核层面的进程和线程的创建过程中，可以发现Linux系统中的进程与线程的概念和实现非常接近，只是调用的函数不一样，设置的task\_struct的参数不一样，实际上确实如此，fork和pthread\_create两个函数在内部实现中都调用了clone，只是传入的参数有区别，所以Linux系统中线程和进程已经非常接近，线程也被称作轻量级进程。

Linux系统中的每一个线程都对应一个内核空间的task\_struct，所以对于调度算法而言，每一个线程都参与CPU资源的竞争。





### 5、进程间通信

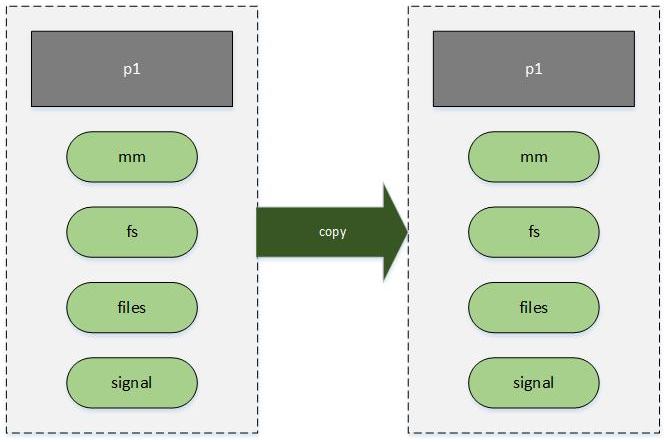
Linux系统中使用管道（pipe）进行同一主机间的进程间通信。管道将和另一个进程的通信看做是对一个文件的读写操作，popen()函数类似于fopen()函数，pipe()函数类似于open()函数，返回的是对象描述符。管道必须在亲属进程(同一父进程创建出的相关进程)之间进行数据传输的。

Linux系统也可以使用信号（signal）进行同一主机间的进程间通信。一个进程只可以给它的同一进程组的进程发送信号，进程A可以向进程B发送信号，进程B必须制定一个信号处理函数来获取所有收到的信号，当A的信号达到时，进程B立即切换到信号处理函数。

## 二、内存管理系统

## 三、文件管理系统

执行fork后，父进程的task\_struck对拷给子进程，父子进程最初资源完全一样，但是是两份不同的拷贝，因此任何改动都造成二者的分裂。

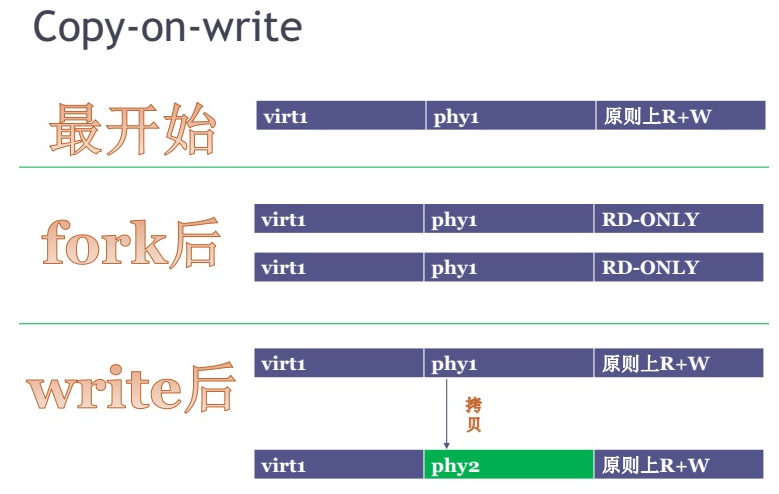


父子进程对内存资源(mm)的管理使用了COW(Copy-On-Write, 写时拷贝)技术：

（1）在fork之前，一片内存区对应一份物理地址和一份虚拟地址，内存区的权限为RW；

（2）在fork之后，父子进程看到的内存区虚拟地址相同，物理地址也相同，父子进程使用的其实是同一片物理内存，未发生内存拷贝，操作系统会将此内存区权限改为RO；

（3）父或子进程对内存区执行写操作将触发PageFault，操作系统此时会将内存区拷贝一份，父子进程看到的虚拟地址仍然一样，但是物理地址已经不同。各进程虚拟地址到物理地址的映射由MMU(Memory Management Unit, 内存管理单元)管理。



# **Windows操作系统**

# 迭代日志表

|  |  |  |
| --- | --- | --- |
| **迭代版本** | **迭代工作** | **迭代日期** |
| V1.0 | 建立初稿文档，完成第1章和第2章的整理。 | 2022-01.22-01.23 |
| V1.1 | 添加Linux操作系统的进程、线程的相关笔记。 | 2022-01.24-01.26 |
| V1.2 |  |  |