123

通过 Linux 理解操作系统（一）：概述

用了那么多年电脑，操作系统从 WinXP，Vista，再到 Win7，然后是现在用的 Ubuntu，这么长的时间里，一直没有搞明白这操作系统是个什么东西，为什么这么神奇，只要点一点，按一按，那些一块一块的硬件就可以完成我们的工作。直到学了操作系统这门课程，才开始有点朦朦胧胧的理解，最近又看了一些 linux 系统设计的资料，觉得有些领悟，所以写出来跟大家分享一下。

先声明，本人不是 linux 技术极客，所以本文不会讲一些很酷的 linux 使用技术，也不会讲一些很深入的 linux 内核分析，这些都有相应的书籍和资料可以学习，比如鸟哥的 linux 私房菜和 Linux 内核源码剖析，我只是希望通过 linux 这样一个开源的操作系统实例来帮助像我一样的菜鸟们理解操作系统这个东西，更多地是从系统本身的设计，数据结构，算法，代码来讲，如何实现了 I/O，进程管理，内存管理这些东西，有讲错的地方请指正或者补充。（还有请轻喷~)

1、操作系统究竟是个神马东西

操作系统 （英语：Operating System，简称 OS ）是管理计算机 硬件与软件资源的计算机程序 ，同时也是计算机系统的内核与基石。 操作系统需要处理如管理与配置内存 、决定系统资源供需的优先次序、控制输入与输出设备、操作网络与管理文件系统等基本事务。 操作系统也提供一个让用户与系统交互的操作界面。（摘自维基百科）

以上定义言简意赅，但是看起来还是云里雾里，好像知道了是这么个东西，其实心里疑问多多。其实说白了，抛开所有面纱，所有特殊地位，它就是一个计算机程序，它一样是通过源代码编译出来的，一样是由程序设计语言，数据结构，算法支撑起来的，它的目的就是管理计算机的硬件资源（硬盘，屏幕，键盘，鼠标，打印机，网卡等等）和软件资源（系统上运行的程序，像浏览器，播放器，文本编辑器，游戏等等），而这个目的的目的最终还是为用户服务，用户通过它，就只要点一点，按一按，高级一点的就输入一些命令，写写脚本什么的就能够完成那么多复杂的任务（你能想象自己去控制那么多硬件计算吗，想想电路实验的时候各种插线拔线吧~）。意识到这一点，我们才能抛开对它的恐惧，来好好的研究一下它。

2、Linux 的历史发展

linux 的历史已经要被讲烂了，几乎所有的资料里都会花很大的篇幅来讲 linux 的发展，但是为了后面的理解需要，这里还是要简单提一提，详细的可以参阅：鸟哥的 linux 私房菜这个站点：http://linux.vbird.org/linux_basic/0110whatislinux.php

提到 Linux，就不得不提到 Unix，因为 linux 就是由 unix 发展而来的。unix 最早诞生与 20 世纪四五十年代，是贝尔实验室的 Ken Thompson 在一台叫 PDP-7 的机器上写出来的，这个东西出来以后他的同事都被这个系统的能力感到惊讶，因为在此之前操作计算机是很痛苦的，（这里涉及到计算机的历史，就不展开了）。很多人开始加入到 Unix 的开发中，而由于这个系统一开始是用汇编写的，这意味着要把它用在其他机器上，就必须在新的机器上重新编写整个系统。怎么办呢？学过编译原理的同学就知道应该使用一种高级语言来编写这个程序，然后在不同的机器上通过编译器将其编译成能在该机器上运行的机器代码就行了。Thompson 于是自己设计了一种语言叫做 B，然后用这种语言重写了 Unix，但是由于这个语言设计的缺陷最终不是很成功，这时候大名鼎鼎的 Dennis Ritchie 出现了，根据 B，他又设计出了大家都相当熟悉的 C 语言，而且还为 C 写了一个很棒的编译器，然后他们两个就用了 C 把 Unix 给重写了一遍，而 C 语言也成为了一种影响深远的程序设计语言(这两位大哥真心牛叉~）。

Unix 出来之后，受到了很多关注，很多的大公司，科研机构都找贝尔实验室拿源码，当时贝尔实验室的老板 AT&T 公司由于对这个东西不是很重视 所以都给了（因为人家当时有的是钱~，推荐看看《浪潮之巅》就知道了，作者：吴军)，这些机构拿到了源码之后，又进行了很多的研究和开发，于是产生了很多不同版本的 Unix，其中一个就是伯克利大学，他们的版本就是很多人知道的 BSD 了 （Berkeley Software Distribution), 它的特殊之处在于它第一个将网络引入到操作系统中，使得网络协议（TCP/IP）的支持成为了 Unix 的一个标准，为之后因特网的发展中，基于 Unix 的服务器统治整个市场打下了基础。

由于很多厂商，机构都发布了自己的 Unix 版本，这就带来了一个兼容的问题，不同的 Unix 版本提供不同的功能，甚至同个功能又有不同的系统调用接口，这意味着开发者必须针对每一个版本编写自己的程序，这又是一件很痛苦的事情。为了解决这个问题，IEEE 标准委员会展开了一个项目叫做 POSIX（前三个字母表示 Potable Operating System)，主要根据当时最流行的两个 Unix 版本，一个是 BSD，一个是 AT&T 原生的 Unix (System V), 这个标准定义了所有 Unix 系统必须提供的一套库函数，开发者只要通过这些库函数就可以满足他们开发利用 Unix 系统的需要。当然，标准制定的过程肯定有很多冲突，这个过程也是挺有趣的，有兴趣的话可以自己再去了解一下。

讲了那么多 Unix，终于要讲到 Linux 了。因为当时所有的 Unix 系统都十分庞大和复杂，很难用于学校教学目的，有位大学教授就根据 Unix，又写了一个相似的简化了的系统，叫做 Minix，它的内核只有 1600 多行 C 代码和 800 多行的汇编，这个东西出来之后同样受到了很多人的欢迎，不仅仅是学习，而且还移植了许多 Unix 的程序过来。随着它的发展，有人就希望原作者能够给这个系统内核增加更多的功能，但是作者为了保证这个系统的规模足够小，能够被学生在短时间理解，所有没有同意添加功能。于是一位芬兰的大学生 Linus Torvalds 又根据 Minix 重写了一个系统叫做 Linux，支持了很多扩展功能如网络通信，其最为特殊的一点，也是它能取得现在这样的成功的一点是在与它的商业模式，它是一个开源的自由软件，意味着任何人都能够对它的代码进行研究，修改，由此创造了很大的活力，对这个有兴趣的可以查阅一些开源软件，开源协议方面的资料。

。。。本来说简单提一提的，想不到说了这么多，接下来还是正式进入 Linux 系统的所谓概述吧。

3、Linux 系统概述

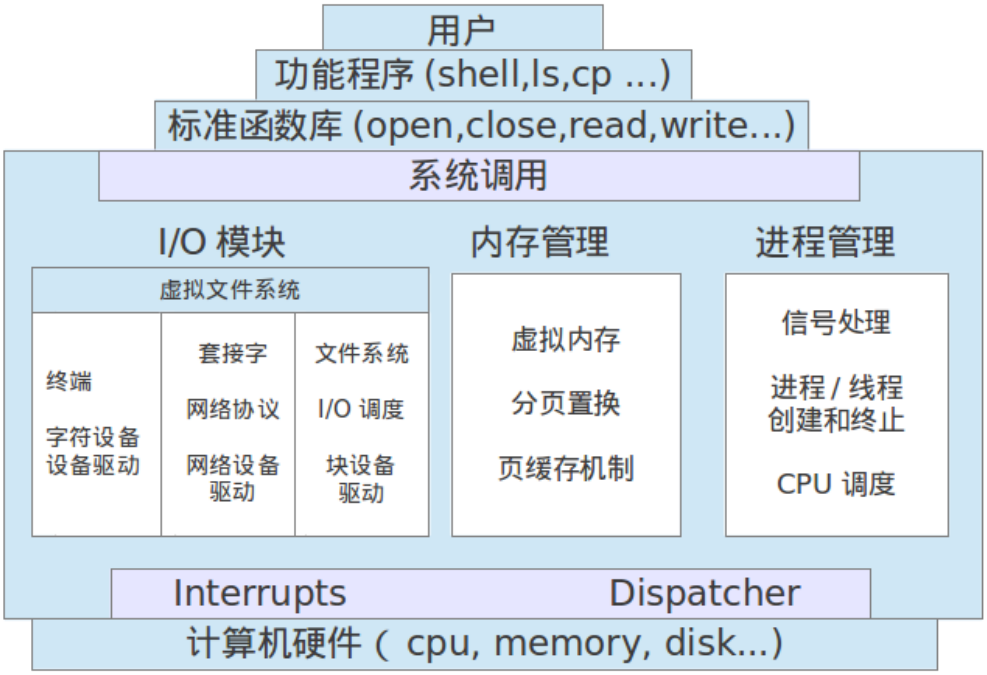
首先通过一个简单的例子来引出一些概念，思考一下，当我们在 shell 终端里输入：ls，这个时候系统发生了什么？

我们知道 ls 是一个系统命令，可以列出一个目录里的文件列表，那么系统命令是什么，它是如何实现的？

熟悉 linux 的都知道，其实它是一个编译好的程序，就放在/usr 文件目录里，这些程序叫做 Core Utility Program, 那么这些程序又是如何来完成其相应的功能的呢？这里就涉及到了一些概念，如库函数，系统调用，内核模式，用户模式等。了解 C 语言的肯定对 c 库函数不陌生，我们的程序里经常都会有的 scanf，printf，fopen，fwrite，malloc 这些就是，通过使用这些函数我们可以进行 I/O 操作和内存管理等，这些库函数也就是前面讲 linux 历史时提到的 POSIX 的成果，但是这些函数又是如何完成其功能的呢？

没错，是通过系统调用，系统调用是操作系统内核的一种方式，具体是怎么实现的我也不知道，但是我们只要知道，我们的库函数内部是通过系统调用让系统的内核去控制内存，进程，还有 I/O 设备的管理就行，但是库函数又不完全等价于系统调用，因为有些库函数并没有进行系统调用，比如数学函数 abs，sin，cos 或是字符函数 isnum，isdigt，tolower 这些，而前面提到的几个函数则有进行系统调用，它们之间的区别大概就在于有没有操作到内存，I/O，或是进程。而关于内核模式和用户模式，又有相关的进程的内核空间跟用户空间的概念，这些之后会再提到，我们现在只要知道，所有进行系统调用的操作都要切换到内核模式下完成，普通的操作则是在用户模式下运行即可。为什么要这样呢？简单地讲就是为了安全，这样做的话可以保证系统内核进行的所有操作都是系统本身定义的，而用户定义的操作都是在内核以外运行，这样即使用户的操作出了问题（恶意的或者无意的），那么系统的内核也不会受到很大的影响。

为了清晰地展示一下这些概念还有 Linux 内核的结构，还是上个图吧~



从上图我们可以看到那一大块的就是我们的操作系统，它位于计算机硬件的上层，支持了程序和硬件设备的交互，而整个操作系统内核又可以大致分成三个模块，分别为 I/O，内存管理和进程管理。

Linux 通过一个虚拟文件系统将所有的 I/O 设备都抽象成文件，即所有的 I/O 设备的输入输出都可通过文件的 read/write 操作完成，而不需要考虑它实际上是在操作硬盘，屏幕，键盘还是网卡，而内存模块负责虚拟地址空间到物理地址空间的映射，内存分页管理和缓存等，进程管理模块负责进程线程的创建和终止，进程调度和进程间通信。这三个模块之间相互依赖，支撑起了操作系统的所有功能，在后续的文章里会分别进行深入的介绍。（这样讲可能笼统了一点，再举个例子吧）比如当我们的程序要读取一个磁盘上的文件时，就需要访问到文件系统，而为了减少磁盘读取延迟，提高效率，一次读取就会读取一个 block 的数据然后保存在内存里，而在程序等待磁盘读取的过程中，为了提高 CPU 的利用率，系统又会调度其他进程在这个过程中运行，这就是它们之间一个简单的依赖关系，当然它们还有其他许多的依赖。

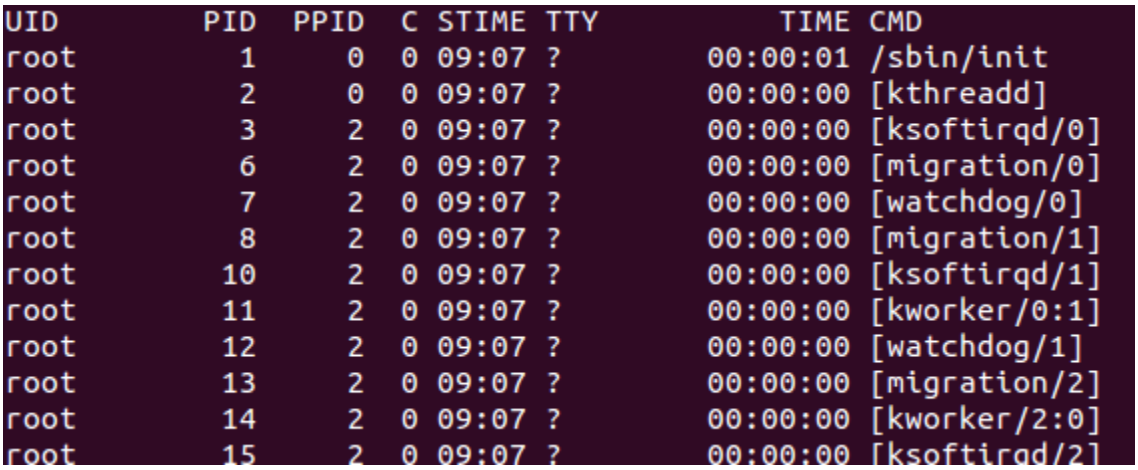
好了，作为第一篇博客写得实在有点长了，下次再继续吧~

通过 Linux 理解操作系统（二）：进程管理（上）

在前文完成了概述之后，本文就要开始进入戏肉了，之前我们将操作系统的内核结构分成了三个模块，现在就先从进程管理模块来开始深入探讨一下。

1、进程间的关系

我们知道一个 Linux 系统里同时运行着大量的进程，当你在 shell 终端里输入 ps -ef 命令时，你会看到像下面这样一长串的东西，有很多我就截了一部分。



```
UID        PID  PPID  C STIME TTY          TIME CMD
root         1     0  0 09:07 ?        00:00:01 /sbin/init
root         2     0  0 09:07 ?        00:00:00 [kthreadd]
root         3     2  0 09:07 ?        00:00:00 [ksoftirqd/0]
root         6     2  0 09:07 ?        00:00:00 [migration/0]
root         7     2  0 09:07 ?        00:00:00 [watchdog/0]
root         8     2  0 09:07 ?        00:00:00 [migration/1]
root        10     2  0 09:07 ?        00:00:00 [ksoftirqd/1]
root        11     2  0 09:07 ?        00:00:00 [kworker/0:1]
root        12     2  0 09:07 ?        00:00:00 [watchdog/1]
root        13     2  0 09:07 ?        00:00:00 [migration/2]
root        14     2  0 09:07 ?        00:00:00 [kworker/2:0]
root        15     2  0 09:07 ?        00:00:00 [ksoftirqd/2]
```

这些进程之间是什么关系呢？

首先，我们从最基本的父子关系说起，我们知道在一个程序里只要调用 fork 函数就可以创建一个新的进程，在这种情况下，调用 fork 函数的就是**父进程**，新创建的进程就是**子进程**，而子进程又可以创建子进程，这样层层往下延伸，就构成了一棵进程树。然后看上图，每个进程都对应了一个 PID 和一个 PPID，在我们的程序里可以通过 getpid 和 getppid 函数获取，PID 标识了进程本身，而 PPID 标识了父进程，所有进程的 PPID 一层层往上推最终都会汇集到 0。然后这里又有一个概念叫做**进程组**，需要注意的是不能把进程组和前面提到的进程树等同，进程组只包括了一个进程的父进程（还有父进程往上的祖先进程），兄弟进程，以及子进程（还有子进程往下的子孙进程），它只是整个进程树的部分而已（可以自己画个图就明白了）。为什么要区分这个概念呢，是因为在进程通信中一个进程只能发送信号给同一个进程组的进程，如果混淆了的话就会以为能够发信号给所有进程了。进程组以外又有一个概念叫**会话**，一个会话由多个进程组组成，一个会话对应一个控制终端，也就是 tty1-tty7。
关于父子进程又有另外两个概念，一个叫作**僵尸进程**，在父进程程序中，通常会调用 waitpid 函数等待子进程终止，而如果在父进程调用 waitpid 之前，子进程就先终止了，那么子进程就会成为僵尸进程（因为还是要等到父进程调用 waitpid 它才能真正终止，要死不死的所以叫僵尸），而相应的如果父进程在子进程终止之前自己先终止了，那么子进程就变成了**孤儿进程**，这样该进程以下的进程组就和整个系统的进程树隔离开了，这时候有一种机制叫做"收养"，就是把这个孤儿进程的父进程设为 init 进程（就是 PID 为 1 的那个）。

2、进程间通信

进程间通信是一个很常见的问题，有很多种方式，网上资料也是大把，所以我也不讲怎么实现，只是提一下方式还有稍微解释一下。

首先，关于进程通信不能只是狭隘地理解为交换数据，虽然基本上是这样，但是其实发送**信号**通知另外一个进程要干什么事，这也是一种通信。一个进程可以在程序里使用函数 kill(pid,sig)发送信号给另一个进程（kill 不只是杀死，也能发信号的），然后在接收信号的进程程序里可以定义处理相应的信号函数，这实际上是通过软中断实现的，指一个进程在运行的过程中，收到信号后，停止当前运行的指令，并保存进程状态，然后跳转到信号处理过程，处理完成后又回到原先停止的位置继续执行。
进程通信的另一种方式是**管道**，它通过阻塞的方式实现了同步，管道分为**匿名管道**和**命名管道**两种，匿名管道存在与内存中，只能用于父子进程间，熟悉 shell 的应该对这样的命令不陌生：
sort <f | head，它实际上创建了一个进程 sort 从文件 f 中都入数据进行排序，而 sort 进程又创建了一个子进程 head，并构建了一个匿名管道，把 sort 的结果传到 head，然后 head 输出前 n 行数据，当管道满了的时候，sort 会停止输出等到 head 将管道的数据取出后再继续输出。

命名管道则存在与文件系统中，可以用于任意进程之间，因为这只是相当于一个进程创建一个文件并写如数据，然后另一个进程只要知道那个文件的路径，然后打开它读取就行了。

进程间通信还有其他的方式如使用 socket 监听本机的端口，或者使用 IPC 对象如共享内存，信号量（信号量不同于信号），消息队列等，这些就等到之后谈到 I/O，和内存管理的时候再详细介绍。

3、Linux 用于进程管理的系统调用

在了解 Linux 系统内核是怎么实现进程管理之前，我们先来了解我们的程序如何通过系统调用进行进程管理。

（1）前面已经提到了一个函数 fork 用于创建一个子进程，它实际上是创建了一份父进程的拷贝，他们的内存空间里包含了完全相同的内容，包括当前打开的资源，数据，当然也包含了程序运行到的位置，也就是说 fork 后子进程也是从 fork 函数的位置开始往下执行的，而不是从头开始。而为了判别当前正在运行的是哪个进程，fork 函数返回了一个 pid，在父进程里标识了子进程的 id，在子进程里其值为 0，在我们的程序里就根据这个值来分开父进程的代码和子进程的代码。

（2）通常情况下，在 fork 之后，子进程需要执行一个和父进程不同的程序，而父进程则阻塞等待子进程终止，这也是 shell 程序的执行方式，shell 本身是一个进程，当我们输入一个命令时，shell 本身停止，该命令对应的程序作为子进程执行（有时直接返回结果，有时进入程序执行界面），当程序返回后，shell 又继续运行等待下一个指令。这个过程中涉及到两个函数，一个是前面提到的 waitpid，还有一个是 execve 函数，它们的函数原型分别为：

waitpid(pid, &staloc, opts); execve(name,argv,envp);

waitpid 的第一个参数用于指定等待终止的子进程 id，也可以设为-1，表示任意一个子进程， 第二个参数用于返回子进程的终止状态，比如我们经常调用的 exit(0),exit(1),所传的参数就在这里返回，表示是否正常终止，第三个参数用于指定父进程是否阻塞等待子进程终止。

看到 execve 的参数列表，很容易想到我们平时写程序时的 main 函数，

int main(int argc, char **argv); (这里其实也可以有第三个参数 envp）

这里其实就是对应的，execve 的第一个参数表示可执行程序的名字，第二个是一个参数数组指针，第三个是环境变量数组的指针，在调用 execve 时，它实际上就是使用这些参数又调用了我们程序的 main 函数。例如：在 shell 中输入 cp file1 file2 时，shell 进程 fork 出一个子进程后，子进程再调用 execve，然后把参数传给 cp 程序的 main 函数，这时 cp 的 main 里得到的 argc 就是 3， argv[0] 是 cp， argv[1] 是 file1，argv[2]是 file2。另外在库函数中有一组函数叫做 exec 族函数，包括 execl，execv， execle 这些，它们内部最终调用的都是 execve 函数，只是参数和处理方式不同而已（库函数有很多是这样的关系，比如 fread 内部调用的又是 read）。

（3）还有一个经常使用的系统调用就是信号处理，我们最经常用来终止一个程序的操作就是直接按 Crtl+C 了，其实这就是向一个进程发出了终止的信号，但这不是唯一的方式，还可以在程序里使用前面提到的 kill 函数来给进程发信号，那么一个进程收到信号又是怎么处理的呢，可以调用 **sigaction** 来声明该进程会接收什么信号，并如何进行处理，它有三个参数，第一个是要捕捉的信号 ID，第二个是一个结构指针，指向的结构体里保存了自定义的信号处理函数的函数指针和相关信息，第三个也是一个结构指针。我们在自己写的函数里定义对接到信号的处理过程然后将它传给 sigaction 即可。

（4） 有些时候我们又希望进程能够在没事情干的时候或者把要 cpu 让给其他进程时自己暂停下来，这时候可以使用 sleep 函数和 pause 函数，注意两者的区别，sleep 可以传进去一个时间参数表示暂停多长时间后回复，而 pause 则没有参数，它要等到接收到另外一个信号时才会恢复。

有了这些系统调用之后，虽然还是受限于系统，但是我们已经能通过我们写的程序来实现我们想要的进程管理方式了，更详细的系统调用介绍还请参阅相关手册。OK~今天就到这，下次我们再往下深入看看系统又是怎么样实现进程管理的吧。

通过 Linux 理解操作系统（三）：进程管理（下）

在前文我们大致了解了程序中如何使用系统调用实现我们想要的进程管理方式，在本文中我们将要看看 linux 系统内核又是如何实现进程的管理的。正如在概述中讲的，操作系统本身也只是一个计算机程序，只要是程序，就会有数据结构和算法，就同样会利用到内存空间甚至磁盘空间，在往下看之前，读者不妨先根据自己的知识思考一下可以用什么样的方式来实现，说不定就搞出了一个新的系统哦~

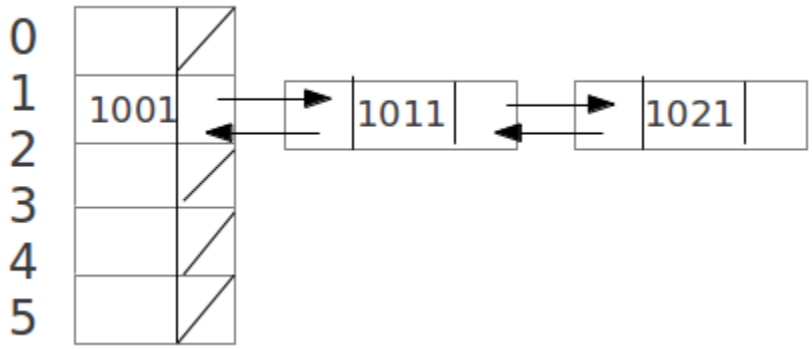
1、进程的表示方式

linux 将一个个进程抽象为一个个任务，并定义了一个结构体 task_struct 用于表示一个任务，对于每一个进程，在其生命周期里都会有一个相应 task_struct 类型的进程描述符存在于内存中，保存了内核用于管理进程所需要的重要信息，一个 task_struct 包含了以下这些域：

1. 调度参数：进程优先级，已使用的 CPU 时间，已休眠的时间，用于系统决定调度哪个进程执行；
2. 内存镜像：指向进程 text，data，stack 内存段或 page table 的指针；
3. 信号：指定哪些信号将被处理，哪些信号将被忽略等；
4. 寄存器：进程切换至内核模式时，用于保存当前正在运行的寄存器信息；
5. 系统调用状态：保存当前进行系统调用的信息，包括参数，结果等；
6. 文件描述符表：保存进程打开的文件的 i-node 数据；
7. 统计信息：记录了进程使用的 cpu 时间，栈空间大小，分页帧数等；
8. 内核栈空间：本进程专属的内核栈空间地址；
9. 其他：当前进程状态，正在等待的事件，进程 ID，父进程 ID，用户 ID 等信息。

通过为每一个进程保存以上这些信息，系统内核才得以合理地进行进程的管理。比如当进行进程调度时，内核需要得到每个进程的优先级，来决定要分多少时间片；在进程接收到一个信号时，内核需要查看进程指定了什么方式进行处理，这些所有的信息都需要通过在进程描述符中查找。

由于系统中同时运行了多个进程，内存中也就保存了多个进程描述符，为了方便地管理和支持快速查找，内核维护了一个哈希表，使用 PID 作键值，采用开散列的方式解决冲突，同一个槽的元素使用双向链表连接，如下图所示：(只是示意图)



通过上面这种存储方式，当内核需要查找一个进程描述符时，只需要将进程的 ID 映射到哈希表的一个槽中，并在该槽的链表上进行顺序查找即可。接下来我们再看看一个进程在 linux 系统中是如何创建的，有了以上信息，这个过程就很容易理解了。

当一个 fork 系统调用执行时，调用 fork 的进程将切换至内核模式并创建一个 task_struct 类型的进程描述符（还有其他一些结构，这里就不提了）,新创建的进程描述符中的大多数内容将根据父进程的进程描述符进行设置，然后系统分配一个新的 PID，并根据这个 PID，映射到哈希表里对应的槽，若该槽已被占，则新建一个元素添加到链表里，该元素保存了新的进程描述符的内存地址。接下来，系统再为子进程分配内存空间，并将父进程的内存空间中的内容复制过来，这个过程完成之后，子进程便可以开始运行了。

2、进程调度

在了解了进程的表示方式之后，我们再看看 linux 如何实现进程调度。

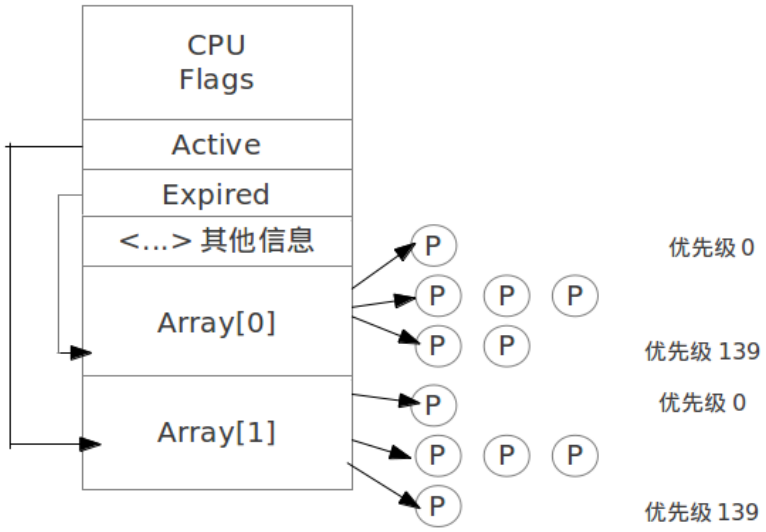
（1）关于线程：linux 系统的调度是基于线程的，一般将进程视为资源容器，而将线程视为一个执行单元（也就是一段连续，独立的执行过程），事实上前面对应的 task_struct 是对应了一个线程，一个单线程的进程表示为一个 task 结构，而一个多线程的进程有多个 task 结构，每一个线程有一个。（这里可能有点乱，因为 linux 里进程和线程的概念有些模糊，不像其他系统还区分了进程，轻量级进程和线程，不过我们编程的时候不需要考虑这么多，所以没什么影响）

（2）优先级：linux 将线程分成三类：实时 FIFO 线程、实时轮转线程、普通分时线程

其中实时跟普通分时的区别仅在于优先级不同，实时线程为 0-99，普通分时线程为 100-139（优先级总共 140 个，0-139），优先级数值越低表示优先级越高，而 FIFO 跟轮转的区别在于 FIFO 是非抢占的，即到达的线程任务必须完全完成之后，下一个线程任务才能进行，轮转则是每个线程分配了一个时间片，在该时间片内线程可以运行，当时间片耗完则切换下一个线程运行不管当前任务是否完成。

除了优先级之外,与进程调度有关的还有另外一个值叫 NICE 值，它的范围是-20~+19，nice 值的意思是表示其他进程的友好程度，友好是指把 cpu 时间让出来，所有一个进程的 NICE 值越大，它本身使用的 cpu 时间会越少，其默认的值为 0，可以使用 nice 系统调用进行修改。优先级和 NICE 值共同决定了一个进程在 cpu 的运行时间。

(3)进程调度的数据结构：linux 的调度器为每一个 cpu 维护了一个数据结构成为 runqueue，示意图如下:



如上图所示，一个 runqueue 中有两个域 active 和 expired，它们是一个指针分别指向了一个长度为 140 的数组，而这每个数组里有存储了 140 个链表的头指针，每个链表实际上对应了一个优先级，链表里存了属于该优先级的进程。调度器进行调度的过程基本上是：先从

active 里优先级最高的链表中取出一个任务执行，当该进程的时间片耗光后，则把它移动 expired 里对应的优先级的链表中，然后再取出下一个进程执行，这样一级级往下，就保证了优先级高的进程先被执行，同时进程优先级越高，所分配的时间片也越长。而当 active 里的所有进程都已经执行过后，只要将 active 和 expired 的这两个指针的指向交换，则原先 expired 里的现在进程现在又变成了 active，之后再重复上面的步骤即可，这种方式就保证了低优先级的进程能够得到 cpu 时间。

OK，关于 linux 系统的进程管理的部分就到这里了，其实实现这一部分还是很复杂的，我的水平有限也没有办法讲得很清楚，而且因为对实际应用的影响也不是很大，所以也没什么兴趣进行再深入的研究了，我们只要知道它实现的机制和思路，证明它真的只是一个程序，而且设计的方式多种多样，没有绝对标准就达到目的了，下一篇将进入到 linux 的内存管理模块，敬请期待。

通过 Linux 理解操作系统（四）：内存管理（上）

关于内存，最直观的理解可以将其想象成一个个格子，每个格子由一个地址标记出来并且存了一个字节的数据，对于 32 位的机器，可以有 2^32 个地址，也就是理论上可以存 4GB 的数据(实际的机器不一定是 4G 的物理内存）。的确，对于程序员而言这样的理解已经足以满足我们编写程序的要求了，而内存实际的物理模型也是这个样子的。但是，对于系统而言，这样简单的模型是不够的，因为正常情况下系统中都会运行着多个程序，如果这些程序都可以直接对任意一个内存地址进行操作，那么一个程序就很有可能直接的修改了另外一个程序保存在内存中的数据，这种情况下会发生什么，不好说，但肯定会悲剧。所以操作系统必须实现一些机制，来保证各个进程可以和谐友爱地使用这有限的内存，同时又要保证内存的使用效率，这些就是我本文要讲的主要内容了。

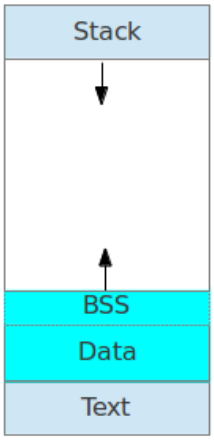
1、基本概念

为了解决前面提到的问题，操作系统对物理内存做了抽象，得到一个重要的概念叫做**地址空间**，指可以用来访问内存的地址集合，也就是 0X00000000 到 0xFFFFFFFF，大小是 4 个 G 。每个进程都有自己的地址空间，且在每个进程自己看来这 4G 就相当于物理内存，它可以使用任意地址去访问他们，而不需要担心影响到其他进程。因为这里的地址并不是实际的物理地址，而是虚拟地址，它需要经过系统转化成物理地址后再去访问内存，且系统保证了不同的进程中的同一个虚拟地址会映射到不同的物理地址,也就不会操作到同一块内存（除非那一块内存是共享的）。又因为通常一个程序也不会使用到 4G 的内存，所以 4G 的物理内存可以同时存放多个程序的数据而不会重叠，即使 4 个 G 都已经放满了，也可以通过将一部分暂时没用的数据保存到磁盘的方式来腾出空间放其它的数据，具体如何操作，我们之后再讲，这里只要知道，我们的程序是通过虚拟地址来访问内存的，而系统保证了每个进程通过地址空间访问到的都会是自己的数据就可以了。

有了地址空间的概念后，在讨论程序如何使用内存的时候，我们就可以将物理内存的概念抛到一边了，接下来我们就看看 Linux 里的进程是如何使用地址空间的：

在 Linux 中，虽然每个进程有 4G 地址空间，但是其中只有 3G 是属于它自己的，也就是所谓的**用户空间**，剩下的 1G 则是所有进程共享的，也就是**内核空间**，这 1G 的内核空间里保存了重要的内核数据比如用于分页查询的页表，还有之前提到的进程描述符等，这些内容在系统运行过程中将一直保存在内存当中，且对于运行在**用户模式**下的进程是不可见的，只有当进程切换到**内核模式**后，才能够对内核空间的资源进行访问（以及进行系统调用的权限），又因为内核空间是所有进程共享的，所以利用内核空间进行进程间通信就是一件理所当然的事情了，所有 IPC 对象如消息队列，共享内存和信号量都存在于内核空间中。

而用户空间又根据逻辑功能分成了 3 个段：Text，Data 和 Stack，如下图所示。



其中，Text 段的内容是只读的且整个段的大小不会改变，它保存了程序的执行指令，来源于可执行文件，我们知道程序经过编译之后会得到一个可执行文件，这个可执行文件里就保存了程序执行的机器指令，在运行时，就将这些指令拷贝到 Text 段里然后 CPU 从这里读取指令并执行。

data 段顾名思义是保存了程序中的数据，包括各种类型的变量，数组，字符串等，它包括两个部分，一个是有初始化的数据区，保存了程序中有初始值的数据，一个是无初始化的数据区（通常叫做 BSS），保存了程序中没有初始值的数据，且 BSS 区的数据在程序加载时会自动初始化为 0。注意这里的数据不包括函数内的局部变量，因为那是在 stack 段中的。举个例子，熟悉 C/C++的人知道如果我们程序中的全局变量没有设置初始值的话，会自动初始化为 0，而局部变量没有设置初始值的话，则他们的值是不确定的，其原因就在这里，当全局量不设初始值时，会保存在 BSS 区里，这里自动为 0，若有初始值，则在有初始化的区，而局部变量在 Stack 段则是没有初始化。跟 Text 段不同，data 段里的数据可以被修改，而且 data 段的大小也可能在程序运行过程中改变，比如说当调用 malloc 时，data 段的地址会往上扩展，而这些动态分配的内存就称为**堆。**

stack 段位于用户空间的最顶部，可以向下增长，它被用来存放进行函数调用的栈。当程序执行时，main 函数的栈最先创建，伴随着传进来的环境变量和执行参数，并压入系统栈中（指 stack 段）,当在 main 函数中调用另一个函数 A 时，系统会先在 main 的栈中压入函数 A 的参数和返回地址，并为 A 创建一个新的栈并压入系统栈中，而当 A 返回时，则 A 的栈被弹出，这样就使得当前执行的函数总是在系统栈的顶部（这里的顶部在上图中是在下方，因为 stack 段是往下增长的），这就是函数调用的一个粗略过程。

2、地址空间的应用

前面已经提到了地址空间的概念，进程只管使用地址空间里的地址去读写数据，而不管实际的数据是放在什么地方，接下来我们就看看系统利用这点可以干些什么。

（1）共享 text 段：我们已经知道了 text 段是存放程序运行的机器指令的，那么当多个进程运行同一个程序的时候，它们的 text 段肯定也是一样的，在这个时候，为了节省物理内存，系统是不会把每个进程的 text 段内容都放到物理内存的，而是只保存了一份，然后让各个进程的地址空间的 text 都映射到这一区域，这样做对于每个进程的运行不会有任何影响，同时又节省了宝贵的物理内存。实际上系统还保证了同一份指令在内存中只会存在一份，一个实际例子就是动态连接库的使用。

（2）内存映射文件：因为进程使用地址空间的地址读写数据时不用管实际的数据在哪，那就意味着这些数据甚至可以不在内存中，内存映射文件就是利用了这一点，通过保留进程地址空间的一个区域，并将这块区域映射到磁盘上的一个文件，进程就可以像操作内存一样来访问这个文件（即像访问数组一样可以使用指针，偏移量等），而不用使用到文件的 IO 操作，当然这其中肯定需要操作系统提供相应的机制来去实

现逻辑地址到实际文件存放位置的转换，但这就不是我们所关心的了。使用内存映射文件还有一个好处就是，多个进程可以同时映射到同一个文件，又因为此时的这一份文件在进程看来就是内存，也就是说可以将其视为一块共享内存，这意味着每个进程对这块区域的修改对于其他进程都是实时可见的，当然这里的效率会比将数据实际放在内存时要低，但是却带来了另一个好处就是磁盘空间相对于内存来讲是无限的，因此可以实现大数据量的数据共享。

好了，到这里我们对内存就有了一个比较细致的理解，其中地址空间是一个值得细细体味的概念，下一篇文章我们再看看系统是通过怎样的机制来使得我们的程序可以如此方便地访问内存的。

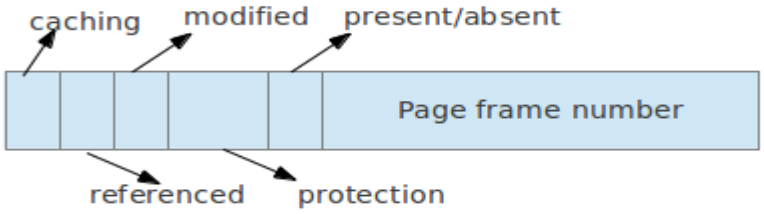
通过 Linux 理解操作系统（五）：内存管理（下）

前文主要讲了我们的程序是通过虚拟地址进行内存访问的，那么操作系统是如何实现了虚拟地址到实际物理地址的转换，又是如何对有限的物理内存进行管理，才能让多个进程共同在有限的内存里跑起来的呢？总的来说，系统要做的工作包括：监控物理内存的使用情况、在程序需要更多内存时进行内存分配、把不同进程的地址空间映射到物理内存的不同区域、动态地把程序运行需要的资源移进内存或把暂时不需要的资源移出内存以腾出空间，接下来将对 Linux 是通过怎样的机制完成这些工作做一个简要的介绍。

1、分页和页表

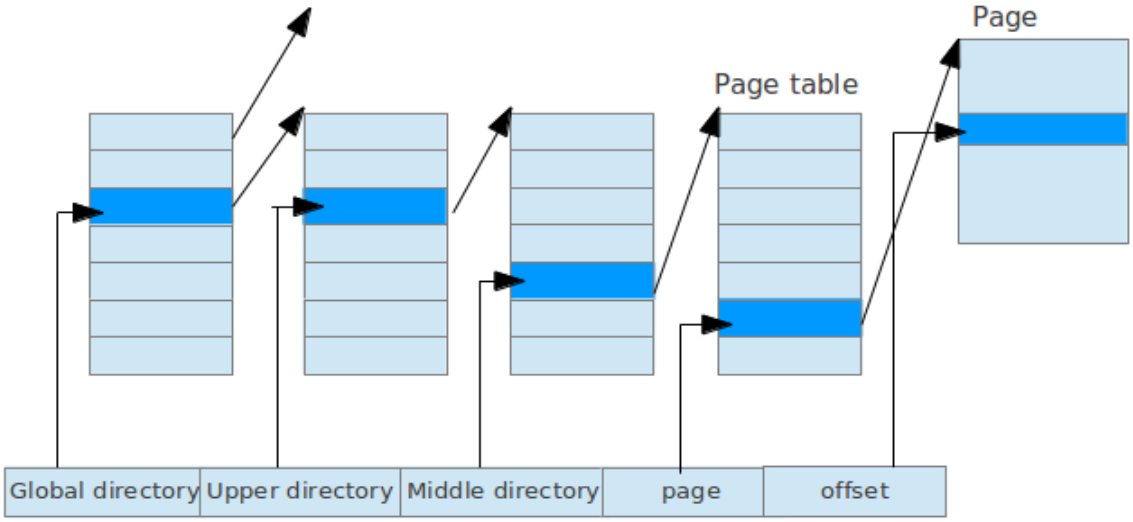
首先，分页的概念相信很多人都不陌生，我这里想说的是"分"的思想，学习计算机两年多，我最大的感受是计算机就是在利用有限的资源干无限的事，而这很多时候都是基于"分而治之"的思想实现的。问题规模太大，太复杂怎么办？就是要分解，分就意味着更简单，更灵活，更容易处理，我这里并不只是指算法设计，而是指解决很多实际的复杂问题，就像现在很火的大数据处理，一台机器根本无法完成这么大量的存储和计算工作，就是需要通过"分"，把数据分到多台机器上存，把计算任务也分到多台机器上完成，才有了问题解决的可能。好吧，扯远了~其实我只是想说作为程序员一定要理解分治的思想。回到正题，我们已经知道每个进程都有 4G 的地址空间，但是程序在运行的时候并不需要程序中所有的内容，需要的只是当前正在执行的一部分和相关的数据就可以，因此可以将整个地址空间分成一个个连续的大小固定的块（比如 4KB），只要那些需要的块在内存中即可，这一个个的块就是**页**（**page**），好处就是当程序运行或者停止时，系统不用把整个进程的内容移进移出，而只要移动一个个页就可以，这样既提高了效率又节省了空间，才使得多个进程能够同时存在于内存中。注意，这里的页是指虚拟地址空间里连续的一段，而物理内存中这样连续的段则称为 **page frame**，它们的大小相同，进程的页放进内存中时就放到一个个的 page frame 中。

有了 page 的概念后，我们再来看系统是如何实现逻辑地址到物理地址的转换的。Linux 为每个进程维护了一个 **page table**，这个表里的每一条记录表示一个 page，保存了以下信息：



每条记录中通过一些位标识了这个页是否存在于实际内存中，允许什么方式的访问，是否被修改过，是否正在被使用，是否进行缓存等信息，其中最重要的就是 page frame number，系统就是由此得到的该页在实际内存中的位置。在 32 位的机器上，一个逻辑地址有 32 位，它可以分成两部分，一部分用于表示页号，用于在 page table 中查找该 page 的记录，从而得到 page frame number，也就是这个 page 在物理内存的起始位置，还有一部分是页内偏移量，这两者相加就得到了我们实际要访问的物理地址。通过这种方式系统可以实现逻辑地址到物理地址的转换，但是又带来一个问题，由于在程序运行期间需要对 page table 进行查找，也就是说 page table 也要存在于内存中，假设一个页有 4KB，对于 32 位的机器，page table 中刚好有 2^20（1M）条的记录，似乎还可以接受，但是对于 64 位的机器，则需要

2^52 条记录，如果光是存个 page table 就用掉这么多内存，那程序也不用跑了，为了解决这个问题，Linux 使用了**多级索引**技术：

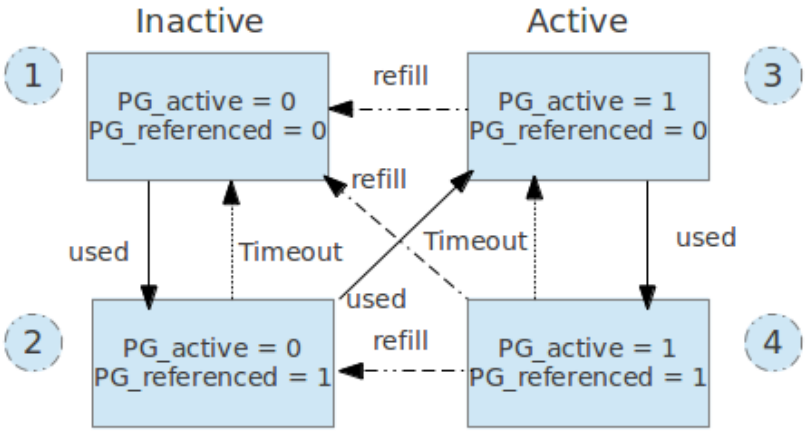


Linux 不是直接把整个 page table 放到内存中，而是根据页的大小将整个 page table 又分成一个个小的 page table，然后通过索引的方式来进行访问。具体方式如上图，虚拟地址被分成了 5 个部分，globalupperlmiddle directory, page 和 offset，首先根据 global directory 在 global 目录里进行查找，global 目录里每一条记录保存了一个指向下一级目录的指针，在取得一个指针后，根据这个指针定位到一个下一级的 upper 目录，然后根据 upper directory 又可以在 upper 目录里得到一个指针来得到一个 middle 目录，而 middle 目录里得到的指针才是指向真正的 page table，此时再根据 page 域取得一条 page table 中的记录。那为什么这样能省内存？假设一个 page 目录里有 n 条记录，那么根据 gloabl 目录可以索引到 n 个 upper 目录，而每个 upper 目录又可以索引到 n 个 middle 目录，以此类推，原来一个大的 page table 分成了 n^3 个小的 page table，通过三级目录一级级往下索引就可以得到我们最终需要的那个 page table，而需要放进内存的就只有在查找过程中需要的三个目录和最终的 1 个 page table 即可，这样当然就省了很多内存咯。

2、页的回收

在系统运行期间，随着越来越多的程序的启动和运行，越来越多的物理内存会被占用，而系统必须保证有空闲的内存来维持正常的运行，Linux 使用了一种叫 PFRA（page frame reclaiming algorithm) 的算法将一些暂时没用的 page frame 释放来腾出空间，的目标就是从内存中选出要释放的 page frame。首先 linux 将所有的 page frame 分为四类：unreclaimable, swappable, syncable, discardable。unreclaimable 表示该页的内容不能被取出，即新的 page 不能放到这里；swappable 表示该页的内容可以被取出内存但需要写到磁盘，而 syncable 也表示可以取出内存但是只有当这个页的内容被标记为已修改过的才需要写会磁盘，discardable 则表示该页的内容可以直接被覆盖而不需要保存到磁盘。这四个类代表了需要完成页的置换操作的难易程度，在发生页的置换时，系统会优先选择容易完成页进行置换。

linux 在启动的过程中会开启一个后台进程，这个进程在系统运行过程中每隔一段时间就会对内存的使用情况进行检查，如果它判断当前的内存已经快要不过用了，就会开始运行 PFRA 算法。



linux 将内存中所有 page frame 组织成两个链表，active list 和 inactive list，这两个链表也叫做 LRU list，active list 里的页是最近被访问过的，而 inactive 里的则是最近没被问的，所以系统可以从 inactive list 里选择 page frame 释放。如上图所示，每个 page 有 2 个标识位，编码了 4 个状态，这四个状态之间可以进行转换，从而导致了一个 page 会在 active list 和 inactive list 之间移动。观察这个图可以发现，当 PG_active 为 0 时，页在 inactive list 上，当 PG_active 为 1 时，页在 active list 上。对于一个在 inactive list 的页，如果一开始处于状态 1，么该页被访问过一次后，它的 referenced 标识位会变为 1，变成状态 2，而再被访问一次后，它的 active 就变成 1，referenced 变为 0（状态 3），这样才从 inactive 变成了 active，而如果在状态 2 的时候，在经过一段特定的时间后还没有再被访问一次，则它又会自动回到状态 1，也就是说一个页要从 inactive 变成 active，中间需要经过一个中间状态。为什么需要这个中间状态呢，主要是考虑到下面这种情况，有些程序可能会周期性的访问一个内存页，比如说 1 小时访问一次，那么在这 1 小时内，它是不需要再被访问到的，如果没有这个中间状态的话，它会直接变成 active，所以会一直保存在内存中，而有了这个中间状态，如果这个也在一定时间内没有再被访问一次，它就仍然是 inactive 的，也就可以被取出内存。当然，有些时候系统可能会急需更多的内存，所以即使是存在 active list 里的页，有时候也需要被取出内存，图中的 refill 箭头就表示了直接从 active 到 inactive 的状态转换。linux 系统通过维护这些状态变换，就可以选择出合适的页来将其取出内存，从而尽量减少发生 page fault 的机会。