一、进程的概述

可执行程序运行起来后（没有结束之前)，它就成为了一个进程。程序是存放在存储介质上的一个可执行文件，而进程是程序执行的过程。进程的状态是变化的，其包括进程的创建、调度和消亡。程序是静态的，进程是动态的。

1、程序和进程的区别

程序 静态的 占磁盘空间（存放在存储介质上的一个可执行文件）

进程 动态的 （调度、执行、消亡），占内存空间。（进程是程序执行到结束间的这个过 程）

2、单道和多道程序设计

单道程序设计 所有进程一个一个排队执行。若A阻塞，B只能等待，即使CPU处于空闲状 态。这种模型在系统资源利用上及其不合理，大部分已被淘汰了。

多道程序设计 在计算机内存中同时存放几道相互独立的程序，它们在管理程序控制之 下，相互穿插的运行。当下常见CPU为纳秒级，由于人眼的反应速度是毫秒级，所以看似同时在运行。

而多道程序设计必须有硬件基础作为保证。时钟中断（强制让进程让出cpu资源）即为多道程序设计模型的理论基础。

3、并行和并发的区别

并行（微观）和并发（宏观）都是多个任务同时执行（多道程序）。

并行(parallel)：指在同一时刻，有多条指令在多个处理器上同时执行（微观上同时执行）（多核）

并发(concurrency)：指在同一时刻只能有一条指令执行，但多个进程指令被快速的轮换执 行，使得在宏观上具有多个进程同时执行的效果，但在微观上并不是同时执行的，只是把时 间分成若干段，使多个进程快速交替的执行（单核 ）

4、进程控制块（PCB）

进程运行时，内核为每个进程分配一个PCB（进程控制块），维护进程相关的信 息，Linux内核的进程控制块是task\_struct结构体。 PCB存在于进程的内核空间里面。系统会为每一个进程分配一个进程ID，其类型为pid\_t(非负整数) ，进程的状态，有就绪、运行、挂起、停止等状态。进程状态切换时需要保存和恢复的一些CPU寄存器。进程是系统分配资源的基本单位。

5、进程的状态

进程的三大状态：就绪态、执行态、等待态

就绪态：执行条件全部满足，等待CPU的执行调度

执行(运行)态：正在被CPU调度执行

等待态：不具备CPU调度执行的执行条件，等待条件满足。

状态机：

ps命令查看进程信息:

选项 含义

-a 显示终端上的所有进程，包括其他用户的进 程

-u 显示进程的详细状态

-x 显示没有控制终端的进程

-w 显示加宽，以便显示更多的信息

-r 只显示正在运行的进程

查看进程状态：ps -aux

stat中的参数意义如下:

以树状显示进程：pstree

二、进程号PID

每个进程都由一个进程号来标识，其类型为 pid\_t（整型），进程号的范围：0～32767。 进程号总是唯一的，但进程号可以重用。当一个进程终止后，其进程号就可以再次使用 。

进程号（PID）： 标识进程的一个非负整型数

父进程号（PPID）:父进程号

进程组号（PGID）： 进程组是一个或多个进程的集合。

1、获取进程号的函数

头文件：

#include<sys/type.h>

#include<unistd.h>

函数：

pid\_t getpid(void);

功能： 获取本进程号（PID）

参数： 无

返回值： 本进程号

2、获取父进程的ID

#include<sys/type.h>

#include<unistd.h>

pid\_t getppid(void);

功能： 获取调用此函数的进程的父进程号（PPID）

参数： 无

返回值： 调用此函数的进程的父进程号（PPID）

3、获取进程组的ID

#include<sys/type.h>

#include<unistd.h>

pid\_t getpgid(pid\_t pid);

功能： 获取进程组号（PGID）

参数： pid：进程号

返回值： 参数为 0 时返回当前进程组号，否则返回参数指定的进程的进程组号

查看父子进程号：ps -ef

查看所有进程号：ps -ajx

getchar();防止进程结束。

三、 fork创建进程

1、fork函数

系统允许一个进程创建新进程，新进程即为子进程，子进程还可以创建新的子进程，形成进 程树结构模型。

#include<sys/types.h>

#include<unistd.h>

pid\_t fork(void);

功能： 用于从一个已存在的进程中创建一个新进程，新进程称为子进程，原进程称为父进程。

参数： 无

返回值： 成功：子进程中返回 0，父进程中返回子进程 ID。pid\_t，为整型。 失败：返回-1。

失败的两个主要原因：

1）当前的进程数已经达到了系统规定的上限，这时 errno 的值被设置为 EAGAIN。

2）系统内存不足，这时 errno 的值被设置为 ENOMEM

2、fork出来的子进程和父进程之间的关系

使用fork函数得到的子进程是父进程的一个复制品，它从父进程处继承了整个进程的地址空间。 地址空间: 包括进程上下文、进程堆栈、打开的文件描述符、信号控制设定、进程优 先级、进程组号等。 子进程所独有的只有它的进程号，计时器等。因此，使用fork函数的代价是很大的 。

父子进程 从fork后开始继续执行。父子进程是同时运行，空间独立，子进程复制父进程的所有空间，谁先运行不确定。

#include <stdio.h>

#include <unistd.h>

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

{

//创建子进程

pid\_t pid=fork();

if(pid<0)

{

perror("创建失败\n");

}

else if(pid==0)

{

printf("%d为子进程\n",getpid());

}

else if(pid>0)

{

printf("%d为父进程\n",getpid());

}

getchar();

return 0;

}

3、子进程 复制 父进程的资源（各自独立）

4、父子进程同时运行

5、父进程 给子进程 足够的准备时间时

四、特殊的进程

孤儿进程、僵尸进程、守护进程。

1、孤儿进程（无危害）

父进程先结束、子进程就是孤儿进程，会被1号进程接管（1号进程负责给子进程回收资 源）

终止子进程：

2、僵尸进程（有害）

子进程结束，父进程没有回收子进程资源（PCB）,子进程就是僵尸进程。

当父进程结束后，僵尸进程的资源被回收。

3、守护进程

守护进程 是脱离终端的 孤儿进程。在后台运行。为特殊服务存在的。（一般用于服务器）

五、父进程回收子进程的资源

在每个进程退出的时候，内核释放该进程所有的资源、包括打开的文件、占用的内存等。但 是仍然为其保留一定的信息，这些信息主要主要指进程控制块PCB的信息（包括进程号、退 出状态、运行时间等）

父进程可以通过调用wait或waitpid得到它的退出状态同时彻底清除掉这个进程。 wait()和 waitpid()函数的功能一样，区别在于，wait()函数会阻塞，waitpid()可以设置不阻塞。注意：一次wait或waitpid调用只能清理一个子进程，清理多个子进程应使用循环。 wait、waitpid基本上都是在父进程调用。

1、wait函数

#include<sys/types.h>

#include<sys/wait.h>

pid\_t wait(int \*status);

功能： 等待任意一个子进程结束，如果任意一个子进程结束了，此函数会回收该子进程的资源。

参数： status : 进程退出时的状态信息。

返回值： 成功：已经结束子进程的进程号 失败： -1

注意：

调用wait的进程会阻塞(挂起)、直到该函数返回才被唤醒。

若调用进程没有子进程，该函数立即返回 子进程已经结束，该函数同样会立即返回，并且会回收那个早已结束进程的资源。

如果参数status 的值不是NULL，wait()就会把子进程退出时的状态取出并存入其中。这是一个整数值（ int），指出了子进程是正常退出还是被非正常结束的。

状态值： int中包含了多个字段，直接使用这个值是没有意义的，WIFEXITED(status) 如果子进程是正常终止的，取出的字段值非零。 WEXITSTATUS(status) 返回子进程的退出状态，退出状态保存在status变 量的8~16位，在用此宏前应先用宏WIFEXITED判断子进程是否正常退出，正常退出才可以使用此宏。

2、waitpid函数

wiatpid常用于等待多个子进程结束。

#include<sys/type.h>

#include<sys/wait.h>

pid\_t waitpid(pid\_t pid, int \*status, int options);

功能: 等待子进程终止，如果子进程终止了，此函数会回收子进程的资源。

参数：

pid : 参数 pid 的值有以下几种类型：

pid > 0 等待进程 ID 等于 pid 的子进程。

pid = 0 等待同一个进程组中的任何子进程，如果子进程已经加入了别的进程组， waitpid 不会等待它。

pid = -1 等待任一子进程，此时 waitpid 和 wait 作用一样。

pid < -1 等待指定进程组中的任何子进程，这个进程组的 ID 等于 pid 的绝对值。

status : 进程退出时的状态信息。和 wait() 用法一样。

options : options 提供了一些额外的选项来控制 waitpid()。

0：同 wait()，阻塞父进程，等待子进程退出。

WNOHANG：没有任何已经结束的子进程，则立即返回。

WUNTRACED：如果子进程暂停了则此函数马上返回，并且不予以理会子进程的结束状态。（由于涉及到一些跟踪调试方面的知识，极少用到）

返回值： waitpid() 的返回值比 wait() 稍微复杂一些，一共有 3 种情况：

当正常返回的时候，waitpid() 返回收集到的已经回收子进程的进程号；

如果设置了选项 WNOHANG，而调用中 waitpid() 还有子进程在运行,且没有子进程退出，返回0； 父进程的所有子进程都已经退出了 返回-1； 返回>0表示等到一个子进 程退出

如果调用中出错，则返回-1，这时 errno 会被设置成相应的值以指示错误所在， 如：当 pid 所对应的子进程不存在，或此进程存在，但不是调用进程的子进程，waitpid() 就会出错返回，这时 errno 被设置为 ECHILD

waitpid等价于wait的案例：

六、创建多个子进程

1、创建2个子进程出现的问题

2、防止子进程 创建孙进程

#include <stdio.h>

#include <unistd.h>

#include <sys/wait.h>

#define N 3

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

{

//创建3个子进程

int i=0;

for(;i<N;i++)

{

pid\_t pid=fork();

if(pid==0)//防止子进程创建孙进程

{

break;

}

}

//判断具体的子进程

if(i==0)//子进程1

{

//完成任务A

int i=3;

for (; i > 0; i--)

{

printf("子进程%d工作剩余时间%d\n",getpid(),i);

sleep(1);

}

\_exit(-1);

}

else if(i==1)//子进程2

{

//完成任务B

int i=5;

for (; i > 0; i--)

{

printf("子进程%d工作剩余时间%d\n",getpid(),i);

sleep(1);

}

\_exit(-1);

}

else if(i==2)//子进程3

{

//完成任务C

int i=7;

for (; i > 0; i--)

{

printf("子进程%d工作剩余时间%d\n",getpid(),i);

sleep(1);

}

\_exit(-1);

}

else if(i==N)//父进程

{

//回收子进程资源

while(1)

{

pid\_t pid=waitpid(-1,NULL,WNOHANG);//不阻塞

if(pid>0)

{

printf("子进程%d退出\n",pid);

}

else if(pid==0)//还有子进程在运行

{

continue;

}

else if(pid==-1)

{

printf("所有子进程已结束\n");

break;

}

}

}

return 0;

}

七、 进程相关

1、终端

用户通过终端登录系统后得到一个Shell进程，这个终端成为Shell进程的控制终端（Controlling Terminal），进程中，控制终端是保存在PCB中的信息，而 fork会复制PCB中的信息，因此由Shell进程启动的其它进程的控制终端也是这个终端。

2、进程组

一个或多个进程的集合，也称之为作业。当父进程创建子进程的时候，默认子进程与父进程属于同一进程组。当bash创建进程时，该进程自己创建与自己ID相同的进程组，不与父进程同属于一个进程组。

进程组ID为第一个进程ID(组长进程)： 进程ID和进程组ID相同的进程就是 组长进程。

可以使用kill -SIGKILL -进程组ID(负的)(正的为组长进程)来将整个进程组内的进程全部杀死 只要进程组中有一个进程存在，进程组就存在，与组长进程是否终止无关。 进程组生存期：进程组创建到最后一个进程离开(终止或转移到另一个进程组)。

#include<unistd.h>

pid\_t getpgrp(void);

功能：获取当前进程的进程组ID

参数：无

返回值：总是返回调用者的进程组ID

pid\_t getpgid(pid\_t pid);

功能：获取指定进程的进程组ID

参数： pid：进程号，如果pid = 0，那么该函数作用和getpgrp一样

返回值： 成功：进程组ID 失败：-1

int setpgid(pid\_t pid, pid\_t pgid)

功能： 改变进程默认所属的进程组。通常可用来加入一个现有的进程组或创建一个新进程组。

参数： 将参1对应的进程，加入参2对应的进程组中

返回值： 成功：0 失败：-1

3、会话

会话是一个或多个进程组的集合。 一个会话可以有一个控制终端。 一个会话中的几个进程组可被分为一个前台进程组以及一个或多个后台进程组;如果一个会话有一个控制终端，则它有一个前台进程组，其它进程组为后台进程组;如果终端接口检测到断开连接，则将挂断信号发送至控制进程（会话首进程）。

如果进程ID==进程组ID==会话ID 那么该进程为会话首进程。

创建新会话的步骤：

调用进程不能是进程组组长，该进程变成新会话首进程(session header)

该调用进程是组长进程，则出错返回 。

该进程成为一个新进程组的组长进程

需有root权限(ubuntu不需要)

新会话丢弃原有的控制终端，该会话没有控制终端

建立新会话时，先调用fork, 父进程终止，子进程调用setsid

#includ<unistd.h>

pid\_t getsid(pid\_t pid);

功能：获取进程所属的会话ID

参数： pid：进程号，pid为0表示查看当前进程session ID

返回值： 成功：返回调用进程的会话ID 失败：-1

#include<unistd.h>

pid\_t setsid(void);

功能： 创建一个会话，并以自己的ID设置进程组ID，同时也是新会话的ID。调用了setsid函数的进程，既是新 的会长，也是新的组长。

参数：无

返回值： 成功：返回调用进程的会话ID 失败：-1

4、创建守护进程

创建子进程，父进程退出(必须) 所有工作在子进程中进行形式上脱离了控制终端

在子进程中创建新会话(必须) setsid()函数 使子进程完全独立出来，脱离控制。

改变当前目录为根目录(不是必须) chdir()函数 防止占用可卸载的文件系统 也可以换成其它路径

重设文件权限掩码(不是必须) umask()函数 防止继承的文件创建屏蔽字拒绝某些权限，增 加守护进程灵活性。

关闭文件描述符(不是必须) 继承的打开文件不会用到，浪费系统资源，无法卸载

开始执行守护进程核心工作(必须) 守护进程退出处理程序模型

八、 vfork创建子进程

1、vfork函数说明

vfork函数：创建一个新进程

pid\_t vfork(void)；

功能： vfork函数和fork函数一样都是在已有的进程中创建一个新的进程，但它们创建的子进程是有区别的。

返回值: 创建子进程成功，则在子进程中返回0,父进程中返回子进程ID。出错则返回-1。

2、vfork函数和fork函数的区别

区别1：vfork创建的子进程 会保证子进程先运行，只有当子进程退出（调用 exec）的时候，父进程才运行。

区别2：vfork创建的子进程 和父进程 共用一个空间。

九、exec函数族

exec函数族功能：在进程中 启动另一个进程。

#include<unistd.h>

int execl(const char \*path, const char \*arg, .../\* (char \*) NULL \*/);

int execlp(const char \*file,cconst char \*arg, ... /\* (char \*) NULL \*/);

int execle(const char \*path, const char \*arg, .../\*, (char \*) NULL, char \* const envp[] \*/);

int execv(const char \*path, char \*const argv[]);

int execvp(const char \*file, char \*const argv[]);

int execvpe(const char \*file, char \*const argv[], char \*const envp[]);

int execve(const char \*filename, char \*const argv[], char \*const envp[]);

函数中有l(list)表明使用列表方式传参，函数中有v(vector)表明使用指针数组传参。 函数中有p（path）表明 到系统环境中 找可执行性文件 函数中有e(evn) 表明exec可以使用环境变量值

案例1：在代码中使用execl执行ls命令

查看ls命令的路径：

一个进程调用exec后不会返回，exec函数族取代调用进程的数据段、代码段和堆栈段。除了进程ID，进程还保留了下列特征不变： 父进程号 进 程组号 控制终端 根目录 当前工作目录 进程信号屏蔽集 未处理信号 ...

案例2：在代码中使用execlp执行ls命令

案例3：在代码中使用execvp执行ls命令

案例4：vfork和exec配合使用,会为子进程开辟新的空间

————————————————

版权声明：本文为博主原创文章，遵循 CC 4.0 BY-SA 版权协议，转载请附上原文出处链接和本声明。

原文链接：https://blog.csdn.net/m0\_75045191/article/details/133842707