1.进程综述

系统中的每个进程必有一个父进程。

进程间的关系存放在进程描述符中。每个task\_struct都包含一个指向其父进程tact\_struct、叫做parent的指针.还包含一个称为children的子进程链表.

对于当前进程，可以通过下面的代码获取其符进程的进程描述符：

struct task\_struct \*my\_parent = current->parent;

2.启动进程

2.1fork()

在fork()之后，父子进程都会执行fork之后的指令

子进程复制父进程的资源

pid=fork();

if(pid==0)

{子进程}

else

{父进程}

COW写时复制

为了降低开销，后来确实发生了写入时，才产生复制动作，每个进程拿到属于自己的一份

2.2vfork()

必须用exit(0)退出

父子进程共享地址空间，子进程修改变量将影响父进程

2.3exec族

v:vector//参数以数组形式

l:list//参数排列在后面

p:path//PATH环境变量,表示使用文件名，并从PATH环境下进行搜索。

e:envir//自己创建环境变量

char\* const argv[] = {"ls", "-l", NULL}; // 1

char\* const envp1[] = {"PATH=.", "XX=YY", NULL}; // 2

char\* const envp2[] = {"PATH=/bin:/sbin", "XX=YY", NULL}; // 3

char\* const envp3[] = {"AA=BB", "XX=YY", NULL}; // 4

（1）int execl(const char \*path,const char \*arg…..);（库函数）

// execl("/bin/ls", "ls", "-l", NULL);

（2）int execv(const char path,char const argv[]);（系统调用函数）

// execv("/bin/ls", argv);

（3）int execle(const char path,const char \*arg…..,char const envp[]);（库函数）

// execle("ls", "ls", "-l", NULL, envp);

（4）int execve(const char\* path,char\* const argv[],char\* const envp[]);（库函数）

// execve("ls", argv, envp2);

（5）int execlp(const char file,const char arg,…);（库函数）

// execlp("ls", "ls", "-l", NULL);

（6） int execvp(const char file,char const argv[]);（库函数）

// execvp("ls", argv);

2.4system

执行shell

system("ls -al");

3.获取线程

getpid() 真实的组织识别码

getgid() 父进程的组织识别码

4.进程退出

4.1正常退出

return

把控制权交给调用函数

exit()

有参数，执行完把控制权交给系统，参数为0代表正常终止，其他代表异常

所有缓冲区数据自动写回并关闭未关闭的文件

\_exit()

立刻结束，关闭未关闭的文件

传递SIGCHID给父进程，父进程可以由wait取得子进程结束状态

4.2异常退出

abort函数

进程收到某个信号，该信号使程序终止

5.进程等待

5.1僵尸进程

在每个进程退出的时候，内核释放该进程所有的资源，但是仍然保留了一些信息。这些保留的信息直到进程通过调用wait/waitpid时才会释放。如果产生大量的僵尸进程，将导致系统没有可用的进程号而导致系统不能创建进程。

如果子进程先结束而父进程后结束，即子进程结束后，父进程还在继续运行但是并未调用wait/waitpid那子进程就会成为僵尸进程；

如果子进程后结束，即父进程先结束了，但没有调用wait/waitpid来等待子进程的结束，此时子进程还在运行，父进程已经结束。那么并不会产生僵尸进程。因为每个进程结束时，系统都会扫描当前系统中运行的所有进程，看看有没有哪个进程时刚刚结束的这个进程的子进程，如果有，就有init来接管它，成为它的父进程；

同样的在产生僵尸进程的那种情况下，即子进程结束了但父进程还在继续运行（并未调用wait/waitpid）这段期间，假如父进程异常终止了，那么该子进程就会自动被init接管。那么它就不再是僵尸进程了。

查看：

top->zombie

ps -ef -> <defunct>

5.2pid\_t wait(int \*status)

使父进程阻塞

直到一个子进程结束或者该进程接收到了一个指定的信号为止。

如果该父进程没有子进程或者它的子进程已经结束，则wait()函数就会立即返回。

参数status用来保存被收集进程退出时的一些状态。

pid = wait(NULL);

返回被收集的子进程的进程ID

如果调用进程没有子进程，失败，返回-1

WIFEXITED(status) ，指出子进程是否为正常退出的，如果是，它会返回一个非零值，status是指针所指向的整数

WEXITSTATUS(status) ，当WIFEXITED返回非零值时，提取子进程的返回值。

如果子进程调用exit(5)退出，WEXITSTATUS(status) 就会返回5；

如果进程不是正常退出的，也就是说， WIFEXITED返回0，这个值就毫无意义。

5.3pid\_t waitpid(pid\_t pid,int \*status,int options)

一个非阻塞版本的 wait()功能，也能支持作业控制。

pid:

  pid>0，等待相应子进程

  pid=-1时，等待任何一个子进程退出，同wait

  pid=0时，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不会对它做任何理睬。

pid<-1时，等待一个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。

options:

常用0

WNOHANG，即使没有子进程退出，它也会立即返回

6.进程调度

6.1调度目标

CPU三种模式:

IO密集型

计算密集型

平衡型

极小化平均响应时间，极大化系统吞吐率

对于实时系统，达到截止时间前完成所应该完成的任务和提供性能的可预测性

6.2调度算法

FCFS:先来先处理

时间片轮询:周期性切换

STCF:短任务优先

分类:抢占式

非抢占式

问题:

长任务长时间不处理:办法：优先级调度

如何预估短任务:启发式估算（根据程序大小

执行一次后记录时间

优先级调度

给长任务分配更高的优先级的STCF

问题：短任务长时间不处理

办法：动态调整优先级

Linux进程调度策略

SCHED\_OTHER

FIFO

RR

实时系统

EDS动态算法

最早截止任务优先(贪心)

类似抢占式STCF

RMS静态算法

6.3优先级反转

低优先级任务持有高优先级所需的共享资源，因此高优先级受阻状态。此时介于两者优先级之间且不需要共享资源的任务将优先执行。

解决办法:

设置优先级上限

优先级继承

7.特殊进程

7.1idle进程（PID=0）

idle进程由系统自动创建，运行在内核态

系统创建的第一个进程，是唯一一个没有通过fork或者kernel\_thread产生的进程。完成加载系统后，演变为进程调度、交换。

7.2init进程（PID=1）

init进程由idle通过kernel\_thread创建，在内核空间完成初始化后，加载init程序，是系统中所有其他用户进程的祖先进程

首先Linux内核启动，然后在用户空间中启动init进程，再启动其他系统进程。在系统启动完成后，init将变成为守护进程监视系统其他进程。

7.3kthreadd（PID=2）

kthreadd进程由idle通过kernel\_thread创建，并始终运行在内核空间，负责所有内核进程的调度和管理。

循环执行一个kthread的函数，该函数的作用就是运行kthread\_create\_list全局链表中维护的kthread, 当我们调用kernel\_thread创建的内核线程会被加入到此链表中，因此所有的内核线程都是直接或者间接的以kthreadd为父进程 。