进程

课程目标

1. 理解进程及其生命周期的含义。
2. 理解linux进程管理的基本原理。
3. 掌握并理解创建进程的相关函数。
4. 掌握并理解创建守护进程的一般步骤。

课程实践

1. 创建进程。
2. 实现一个c文件编译并运行的工具。
3. 创建一个守护进程。
4. 实现一个进程相互保护的简单例子。

Linux进程

理解进程

一、进程是如何被创建起来的？

当你双击某个ELF文件，或者在命令行中输入可执行指令时，系统会做如下动作：

1、检查该文件是否有可执行权限。

2、格式是否是ELF格式。

3、系统分配物理内存，为新进程创建一个虚拟内存空间。

4、系统解析ELF文件格式，将合适的结构（函数代码、局部变量、全局变量等）放入该进程的虚拟地址空间中，完成ELF文件的内存载入。

5、为该进程分配CPU，执行进程代码（main）。

理解进程

进程和ELF文件格式类似，都有以下的部分：

1、代码，存放编译的汇编代码；也称正文段。

2、数据，存放全局变量，常量；也称数据段。

进程中还有一个栈区，用于存放临时变量、函数调用参数和函数调用返回地址等。

还有一些不同的部分是ELF格式解析经过变换的，如ELF中存储的调用的函数名称在进程中将会转换成动态库中函数的地址。

理解进程

进程和程序的区别

1. 程序是静态的，它是一些保存在磁盘上的静态数据，没有任何执行的概念，其可执行权限也仅仅是数据中的某个字段的值而已。
2. 进程是一个动态的概念，它是运行在内存中的一个实体、是程序在内存中加载和执行的过程，包括创建、调度和消亡。

- 进程存在与内存中，和ELF文件一样，有一定的格式。
- 进程有独立的内存空间。
- 进程是程序的一次执行的过程。

思考

思考1：你下载了一个破坏力极强的木马病毒，你复制它，剪切它，但是没有运行它，它会伤害你的电脑吗？

思考2：你在终端运行一个进程，然后删掉对应的ELF程序，进程会终止吗？

理解进程

二、进程是如何运行的？

进程创建成功以后，进程代码就可以得到执行，完成作者设计的功能，同时进程还有如下特点：

1、进程的代码一部分是开发者编写的，一部分是调用系统的代码，（如open、read、write等系统调用，其具体实现在系统里面）。

2、运行一个进程的时候，其他进程仍能被运行，一个进程崩溃的时候，其他进程也还正常运行。

3、一个程序被运行很多次，如运行多个终端，每个终端都独立工作。

理解进程

1. 进程在运行过程中，大部分情况下，除了不断在用户编写的各个函数间切换之外，也不断的在用户编写的代码空间和系统实现的代码空间中来回切换。
2. 每个进程相互独立，都运行在自己的虚拟内存空间中，与其他进程相互隔离（系统为每个进程维护了一个内存映射表，表明物理内存和虚拟内存的对应关系）。
3. 进程间是以调度的方式并行运行的。

理解进程

1. 进程不仅包含了用户写的程序代码，申请的资源，打开的文件，而且还包含了系统为其分配的各种安全标识符，环境变量等。
2. 所有的代码运行环境，加上代码实体构成了进程。
3. 可以使用pmap指令查看某个进程的内存分布情况。

思考：同一个程序，运行多次，产生多个进程，如何区分它们？

进程PID

1. 主要的进程标识
   1. 进程号(Process Identity Number，PID)
   2. 父进程号(Parent Process ID，PPID)
2. PID唯一地标识一个进程

```
ziyang@ubuntu:~$ ps -elf
F S UID          PID    PPID  C PRI  NI ADDR SZ WCHAN  STIME TTY          TIME CMD
4 S root           1       0  0  80   0 -  6142 poll_s 08:34 ?        00:00:01 /sbin/init
1 S root           2       0  0  80   0 -     0 kthrea 08:34 ?        00:00:00 [kthreadd]
1 S root           3       2  0  80   0 -     0 smpboo 08:34 ?        00:00:00 [ksoftirqd/0]
```

```
0 S ziyang      2928    2861  0  80   0 -  6829 wait   08:35 pts/0    00:00:00 /bin/bash
0 S ziyang      2985       1  0  80   0 - 142056 poll_s 08:35 ?        00:00:00 /usr/bin/python
0 Z ziyang      3003    2583  0  80   0 -     0 exit   08:35 ?        00:00:00 [check_gl_textur
0 S ziyang      3007    2534  0  80   0 - 107170 poll_s 08:36 ?        00:00:00 update-notifier
4 S root        3020       1  0  80   0 - 23855 poll_s 08:36 ?        00:00:00 /usr/bin/python
0 S ziyang      3024       1 13  90  10 - 169923 poll_s 08:36 ?        00:00:10 /usr/bin/python
0 S ziyang      3031       1  0  80   0 - 65492 poll_s 08:36 ?        00:00:00 /usr/lib/dconf/d
4 S root        3055       1  8  85   5 - 48248 poll_s 08:36 ?        00:00:05 /usr/bin/python
0 S ziyang      3070    2534  0  80   0 - 73670 poll_s 08:37 ?        00:00:00 /usr/lib/deja-du
0 R ziyang      3077    2928  0  80   0 -  5592 -      08:37 pts/0    00:00:00 ps -elf
```

getpid

```
#include <sys/types.h>
#include <unistd.h>

pid_t getpid(void);
pid_t getppid(void);
```

功能：得到当前进程（父进程）的pid。

返回值：

成功：当前进程（父进程）的pid

没有失败

查看进程的指令

常用命令

ps –elf (ps -ajx)	查看当前所有的进程信息
kill -9 pid	杀掉某个pid进程
top	查看进程的资源使用情况
pmap –d pid	查看进程的内存映射

实验1

1、动手创建一个进程，打印自己及父进程的pid，然后使用getchar或者sleep阻塞那里。
2、用kill命令杀掉第一步运行的进程。

进程状态

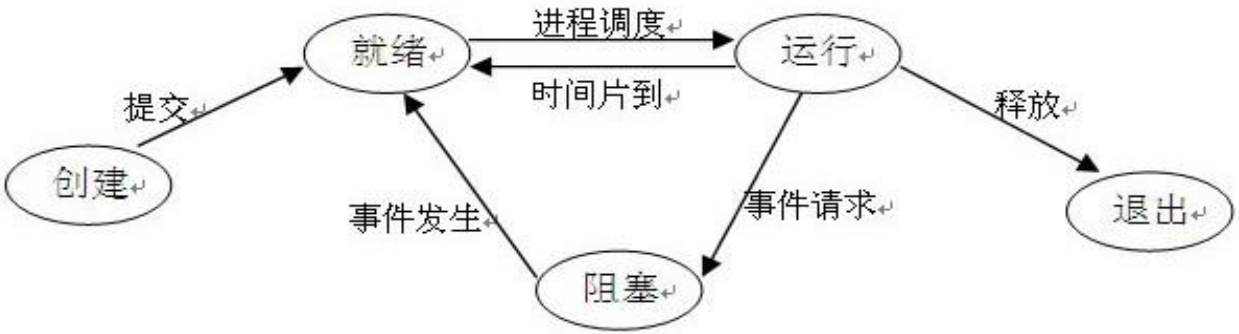
运行态：此时进程正在运行。

就绪态：此时进程正在等待CPU调度，可运行。

阻塞态：此时进程在等待一个事件的发生或某种系统资源，从而阻塞，进入睡眠模式（可中断，响应信号；不可中断，忽略无关信号）。

死亡态：这是一个已终止的进程，但还在进程向量数组中占有一个task_struct结构。

进程三态状态装换图



进程启动

手工启动

由用户输入命令直接启动进程

前台运行和后台运行

调度启动

系统根据用户事先的设定自行启动进程

计划任务在指定时刻执行相关进程

进程创建：fork()

所需头文件	#include <sys/types.h> // 提供类型pid_t的定义 #include <unistd.h>
函数原型	pid_t fork(void);
函数返回值	0：子进程
	子进程PID(大于0的整数)：父进程
	-1：出错

fork

1. fork函数会创建一个新进程的虚拟内存空间，并将自己的虚拟内存中的数据复制到新的虚拟内存空间中。
2. fork在当前进程中返回子进程的pid，在子进程中返回0。

fork

由于是完全的复制，所以堆栈，变量都会复制，同时EIP即指向下一条要指向的指令的变量也会被复制，fork之后，两个进程执行的下一条语句都是fork后面的指令。

fork

```c
int main()
{
    pid_t pid;

    pid = fork();
    if (pid == -1){
        return  -1;
    }
    else if (pid = = 0){/*返回值为0代表子进程*/
        printf("In child process!");//这段代码在子进程中运行
    }
    else {/*返回值大于0代表父进程*/
        printf("In parent process!");//这段代码在原来的进程中运行
    }
     return  0;
}
```

fork函数

由于之前的fork完整地拷贝了父进程的整个地址空间，因此执行速度是比较慢的。为了提高效率，Unix系统设计者创建了现代unix版的fork。现代unix版的fork也创建新进程，但不产生父进程的副本。它通过允许父子进程可访问相同物理内存从而伪装了对进程地址空间的真实拷贝，当子进程需要改变内存中数据时才拷贝父进程。这就是著名的“写操作时拷贝”(copy-on-write)技术。

fork函数

1. fork函数是创建新进程经常要用到的函数。
2. 一般情况下，父进程被杀掉，子进程也会被杀掉。
3. 和windows下的进程管理机制不同，子进程结束之后，需要其父进程回收子进程的资源。

实验2

- 1、用fork创建一个子进程，子进程打印”This is a child”，父进程打印”This is a father ”。
- 2、将实验2.1完成打印后阻塞（sleep）起来，用pmap查看父子进程的内存分布是否一致。

wait函数

1. 父进程要为子进程回收资源，需要一个函数监测子进程是否结束，wait函数就诞生了。
2. 调用该函数使进程阻塞，直到任一个子进程结束或者是该进程接收到了一个信号为止。
3. 如果该进程没有子进程或者其子进程已经结束，wait函数会立即返回。
4. 如果该进程有子进程，会在一个子进程退出是回收内存。

wait

所需头文件	#include <sys/types.h> #include <sys/wait.h>
函数原型	pid_t wait(int *status)
函数参数	status是一个整型指针，指向的对象用来保存子进程退出时的状态。 • status若为空，表示忽略子进程退出时的状态 • status若不为空，表示保存子进程退出时的状态
函数返回值	成功：子进程的进程号 失败：-1

waitpid

所需头文件	#include <sys/types.h> #include <sys/wait.h>	
函数原型	pid_t waitpid(pid_t pid, int *status, int options)	
函数参数	pid	pid>0：只等待进程ID等于pid的子进程，不管已经有其他子进程运行结束退出了，只要指定的子进程还没有结束，waitpid就会一直等下去。
		pid=-1：等待任何一个子进程退出，此时和wait作用一样。
		pid=0：等待其组ID等于调用进程的组ID的任一子进程。
		pid<-1：等待其组ID等于pid的绝对值的任一子进程。
	status	同wait
	options	WNOHANG：若由pid指定的子进程并不立即可用，则waitpid不阻塞，此时返回值为0
		WUNTRACED：若某实现支持作业控制，则由pid指定的任一子进程状态已暂停，且其状态自暂停以来还未报告过，则返回其状态。
		0：同wait，阻塞父进程，等待子进程退出。
函数返回值	正常：结束的子进程的进程号，使用选项WNOHANG且没有子进程结束时：0	
	调用出错：-1	

注意

1. wait是用waitpid实现的：

   static pid_t wait(int * wait_stat) { return waitpid(-1,wait_stat,0); }

1. 设置waitpid的第三个参数option为WNOHANG，可监测子进程的状态，且不阻塞（适合适时观测子进程情况，作出不同的反应）。
2. 如果父进程先结束，init进程会接收子进程并在子进程结束时为其回收资源。
3. 如果子进程结束时父进程还在，那么父进程应该调用wait函数为其回收资源，否则子进程会变成僵死进程，直到父进程死亡。

实验3

1、fork一个子进程，让子进程先退出，父进程不调用wait，观察是子进程僵死，用

ps -elf |grep defunct

指令查看产生的僵死进程。

```
ziyang@ubuntu:~/Documents$ ps -elf |grep defunct
1 Z ziyang     2562   2533  0  80   0 -     0 exit   02:24 ?        00:00:00 [lightdm-session] <defunct>
1 Z ziyang     3275   3263  0  80   0 -     0 exit   02:35 ?        00:00:00 [/usr/bin/termin] <defunct>
0 Z ziyang     3354   2582  0  80   0 -     0 exit   02:36 ?        00:00:00 [check_gl_textur] <defunct>
0 S ziyang     3427   3276  0  80   0 -  3405 pipe_w 02:44 pts/0    00:00:00 grep --color=auto defunct
```

2、fork一个子进程，让子进程sleep(100)，父进程每秒调用waitpid监测子进程的状态，如果子进程还活着，就打印“这个月生活费1000元。”，否则打印“你毕业了，自己花钱自己挣！”，期间手动杀掉子进程，观察打印，体会waitpid的用法。

exec函数族

1. fork只能创建和自己一模一样的进程，使用范围受限，要创建第三方的进程，要使用exec函数族。
2. exec函数族提供了一种在进程中启动另一个程序执行的方法。它可以根据指定的文件名或目录名找到可执行文件，并用它来取代原调用进程的数据段、代码段和堆栈段。在执行完之后，原调用进程的内容除了进程号外，其他全部都被替换了。

exec函数族

所需头文件	#include <unistd.h>
函数原型	int execl(const char *path, const char *arg, ...);
	int execv(const char *path, char *const argv[]);
	int execle(const char *path, const char *arg, ..., char *const envp[]);
	int execve(const char *path, char *const argv[], char *const envp[]);
	int execlp(const char *file, const char *arg, ...);
	int execvp(const char *file, char *const argv[]);
函数返回值	-1：出错

exec函数族成员区别

- 可执行文件查找方式
  - 表中的前四个函数的查找方式都是指定完整的文件目录路径，而最后两个函数(以p结尾的函数)可以只给出文件名，系统会自动从环境变量“$PATH”所包含的路径中进行查找。
- 参数表传递方式
  - 两种方式：逐个列举或是将所有参数通过指针数组传递
  - 以函数名的第五位字母来区分，字母为“l”(list)的表示逐个列举的方式；字母为“v”(vertor)的表示将所有参数构造成指针数组传递，其语法为char *const argv[]
- 环境变量的设置
  - exec函数族可以默认使用系统的环境变量，也可以传入指定的环境变量。这里，以“e”(Enviromen)结尾的两个函数execle、execve就可以在envp[]中传递当前进程所使用的环境变量

exec函数族注意

1、在使用exec函数族时，如果是list方式传递参数，第一个参数传递给argv[0]，第二个参数传递给argv[1], 依次类推，最后参数要以NULL结尾。

2、在使用exec函数族时，一定要加上错误判断语句

常见的错误原因有：

- 找不到文件或路径，此时errno被设置为ENOENT
- 数组argv和envp忘记用NULL结束，此时errno被设置为EFAULT
- 没有对应可执行文件的运行权限，此时errno被设置为EACCESS

实验4

1、使用exec函数运行指令”ls -lt”。
2、使用exec函数运行空中接龙游戏
sol --freecell，并用ps查看exec调用前后的进程名变化。
3、编写一个编译c文件，并运行的工具：

- 检查gcc要生成的目标文件是否已存在，存在就删除。
- fork一个子进程，在子进程中调用gcc编译C文件。
- 父进程监视子进程退出后，是否生成了目标文件。
- 如果生成了就运行可执行文件取代自己。

进程结束

- 进程依赖的终端被关闭。
- 用户注销、退出登录。
- 进程自身运行出问题，退出。

如何创建一个后台运行的程序，不依赖与终端和用户？

守护进程

1. 守护进程（精灵进程），也就是通常所说的Daemon进程，是Linux中的后台服务进程。它是一个生存期较长的进程，通常独立于控制终端并且周期性的执行某种任务或等待处理某些发生的事件
2. 守护进程常常在系统启动时开始运行，在系统关闭时终止
3. Linux系统有很多守护进程，大多数服务都是用守护进程实现的

守护进程

1. 在Linux中，每一个系统与用户进行交流的界面称为终端。从该终端开始运行的进程都会依附于这个终端，这个终端称为这些进程的控制终端。当控制终端被关闭时，相应的进程都会被自动关闭。

2. 守护进程能够突破这种限制，它从开始运行，直到整个系统关闭才会退出。如果想让某个进程不会因为用户或终端的变化而受到影响，就必须把这个进程变成一个守护进程。

守护进程的编写步骤

1. 创建子进程，父进程退出
2. 在子进程中创建新会话
3. 改变当前目录为根目录
4. 重设文件权限掩码
5. 关闭文件描述符

第一步、创建子进程，父进程退出

父进程退出以后，子进程就在形式上做到了与控制终端的脱离

由于父进程已经先于子进程退出，子进程变成孤儿进程

```
pid = fork();

if (pid > 0) /*父进程退出*/
{
        exit(0);
}
```

第二步、在子进程中创建新会话

进程组

进程组是一个或多个进程的集合。进程组由进程组ID来唯一标识。

每个进程组都有一个组长进程，进程组ID就是组长进程的进程号。

会话期

会话组是一个或多个进程组的集合

第二步、在子进程中创建新会话

setsid函数作用

setsid函数用于创建一个新的会话，并使得当前进程成为新会话组的组长

setsid函数能够使进程完全独立出来，从 而脱离所有其他进程的控制。

```
#include <unistd.h>
pid_t setsid(void);
```

成功：返回一个新的session id
失败：返回-1（除非已经是进程组的组长，一般不失败）。

第三步、改变当前目录为根目录

通常的做法是让“/”或”/tmp”作为守护进程的当前工作目录。

在进程运行过程中，当前目录所在的文件系统是不能卸载的。

chdir函数可以改变进程当前工作目录

```
#include <unistd.h>
int chdir(const char *path);
```

第四步、重设文件权限掩码

1. 文件权限掩码是指文件权限中被屏蔽掉的对应位。把文件权限掩码设置为0，可以增加该守护进程的灵活性。

2. 设置文件权限掩码的函数是umask

3. 通常的使用方法为umask(0)

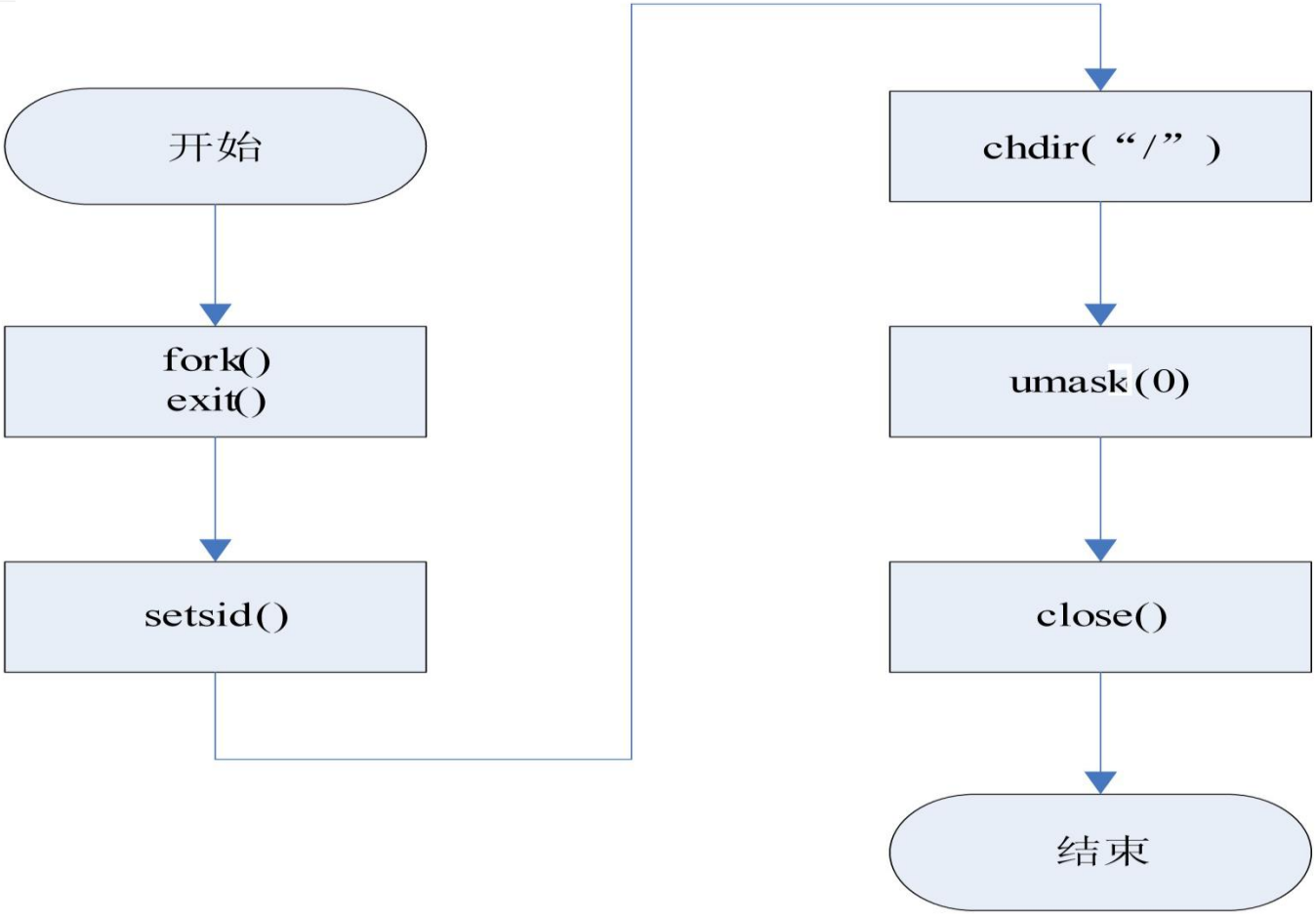
第五步、关闭文件描述符

- 新建的子进程会从父进程那里继承所有已经打开的文件。

- 在创建完新的会话后，守护进程已经脱离任何控制终端，应当关闭用不到的文件

```
fdtablesize = getdtablesize();
for (fd = 0; fd < fdtablesize; fd++)
    close(fd);
```

守护进程创建步骤



```c
void change_to_daemon()
{
    pid_t id;
    int i, num;

    id = fork();          // 步骤1  父进程退出
    if(id > 0)
        exit(0);

    setsid();             // 步骤2  创建新会话

    chdir("/tmp");        // 步骤3  改变运行目录

    umask(0);             // 步骤4  重设文件权限掩码

    num = getdtablesize();     // 步骤5 关闭文件描述符
    for(i = 0; i < num; i++)
    {
        close(i);
    }
}
```

实验5

1、创建一个守护进程。

守护进程

- #include <unistd.h>
- int daemon(int nochdir, int noclose);
- 说明：将本进程变成一个守护进程
- nochdir：
  - 若为1，进程的当前目录不变、
  - 若为0，进程的当前目录改为根目录”/”
- noclose：
  - 若为1，进程打开的文件描述符不变、
  - 若为0，进程将标准输入、标准输出、标准出错重定向到/dev/null
- 返回值：提问

实验6（扩展）

- 1、创建一个守护进程，并保护自己不被退出。思路如下：
- 运行如果发现没有文件/tmp/1则生成，同时不断删除/tmp/2，
- 如果发现没有文件/tmp/2则生成，同时不断删除/tmp/1；
- 如果发现自己生成的文件没有被删除，则将自己再运行一遍。