Linux操作系统

进程管理

主讲：杨东平
中国矿大计算机学院

Linux 进程状态和状态转换

- Linux 进程在生存周期中呈现出各种状态及状态转换，反映了进程的获取系统资源的情况
- Linux 系统的进程状态模型

状态名称	说明
创建状态	进程正在被 Linux 内核创建
就绪	进程还没开始执行，但相关数据已被创建，只要内核调度它就可立即执行
内核状态	进程在内核状态下运行，被调度上 CPU 执行
用户状态	进程在用户状态下运行，等待被调度上 CPU 执行
睡眠	进程正在睡眠，等待系统资源或相关信号唤醒
唤醒	正在睡眠的进程收到 Linux 内核唤醒的信号
被抢先	具有更高优先级的进程强制获得进程的 CPU 时钟周期
僵死状态	进程通过系统调用结束，进程不再存在，但在进程表项中仍有记录，该记录可由父进程收集

子进程被 Linux 内核调度 CPU 执行的过程

- 进程在生命周期里并不一定要经历所有的状态
  - 1) 创建态：父进程调用 fork 创建的子进程
  - 2) 就绪态：
    - 内核就绪：如果内存空间足够，且Linux 内核为子进程配置了数据结构
    - swap 分区就绪：内存空间不足时子进程在 swap 分区就绪
  - 3) 内核状态：就绪态的进程被 Linux 内核调度且分配了 CPU 时钟周期，子进程处于内核状态并开始运行
  - 4) 用户态：
    - 被分配 CPU 时钟周期结束时，Linux 内核再次调度子进程，将子进程调出 CPU，子进程进入用户状态
    - 当子进程处于运行态时，有更高优先级的进程将抢占子进程的时钟周期，子进程又会回到用户态

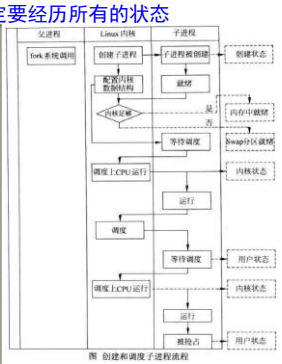

图 创建和调度子进程流程

子进程进入睡眠状态

- 子进程在运行时，如果请求的资源得不到满足将进入睡眠状态，睡眠状态的子进程被从内存调换到 swap 分区
- 被请求的资源可能是一个文件，也可能是打印机等硬件设备。如果该资源被释放，子进程被调入内存，继续以系统状态执行

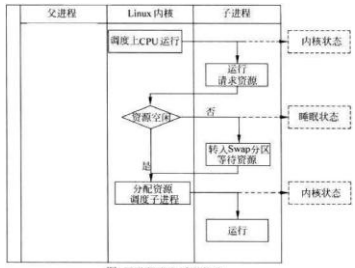

图 子进程进入睡眠状态

子进程结束

- 子进程可以通过 exit 系统调用结束，这时子进程将进入到僵死状态，生命周期结束
- 子进程在内核中的数据结构又被称为上下文，它包括3个部分
  - 用户级上下文：是子进程用户空间的内容
  - 寄存器上下文：是子进程运行时装入 CPU 寄存器的内容
  - 系统级上下文：是子进程在 Linux 内核中的数据结构
- 子进程切换时使用上下文进行现场保护与还原，整个过程称为上下文切换，保存上下文的数据空间称为 u 区。内核在以下情况下会进行上下文切换操作：1)子进程进入睡眠状态时；2)子进程时钟周期结束，被转为用户态时；3)子进程再次被调度上CPU 运行，转为系统状态时；4)子进程僵死时

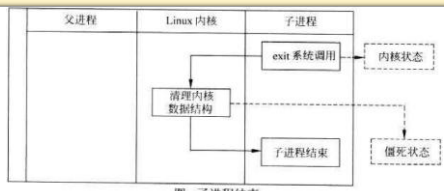

图 子进程结束

进程控制

- fork 系统调用是创建子进程的唯一方法
  - 1) Linux 内核在进程表中为子进程分配一个表项，然后分配 PID。子进程表项的内容来自父进程，fork 会将父进程的进程表项复制为副本，并分配给子进程
  - 2) Linux 内核使父进程的文件表和索引表的节点自增1，创建用户及上下文
  - 3) 将父进程上下文复制到子进程上下文空间中
  - 4) fork 调用结束后，子进程的 PID 将返回给父进程，而子进程获得的值为 0
- exit 系统调用将结束进程
  - exit 调用后，Linux 内核将删除进程的上下文，但保留进程表项，进程处于僵死状态。待合适的时候，再删除进程表项中的内容，释放进程 PID
- wait 系统调用用于父进程与子进程的同步
  - 父进程调用 wait 后，父进程的执行被阻断，直到子进程进入僵死状态。这时，子进程的退出参数可通过 wait 函数返回给父进程。wait 系统调用常用来判断子进程是否已结束
- exec 系统调用用于运行一个可执行文件
  - 它与 fork 的区别在于：exec 会结束原有进程，使用更新上下文的内容，并从头开始执行一个新的进程，两个进程间无父子关系

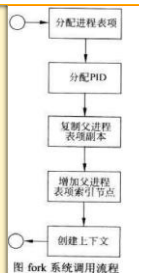

图 fork 系统调用流程

进程调度

- Linux 内核可同时执行多个进程，并为每个进程分配 CPU 时钟周期
- Linux 内核会为每个进程设定优先级，高优先级的进程能够抢占较低优先级进程的 CPU 时钟周期
- Linux 进程调度包括两个概念：
  - 调度时机：指进程何时被调度上 CPU 执行，如：
    - 转变为睡眠态的进程将获得较高的优先级，一旦所需资源得到满足，就可以立即被调度上 CPU 执行
    - 被抢占时钟周期的进程也将获得一个较高的优先级，抢占其 CPU 时钟周期的进程一旦转为用户态，被抢占的进程立即转为内核态
  - 调度算法：它所关心的内容是如何为进程分配优先级
- 通常不需要人为的设置进程的优先级，Linux 调度机制可保证所有进程都能够获得足够的运行时间

查看进程状态信息的命令：ps

- 语法：ps [options]
- 主要选项：
  - a：显示所有进程
  - -a：显示所有终端机下执行的进程，包括其他用户的进程
  - -A：显示所有进程
  - -e：等于"-A"
  - f：显示程序间的关系
  - r：显示当前终端的进程
  - -g<群组名称>：列出属于该群组的进程的状况，也可使用群组名来指定
  - g：显示现行终端机下的所有进程，包括群组领导者的进程
  - -j或j：采用工作控制的格式显示进程状况
  - -l或l：采用详细的格式来显示进程状况
  - -p<程序识别码>：指定进程识别码，并列出该进程的状况
  - -u<用户识别码>：列出属于该用户的进程的状况，也可使用用户名称来指定
  - u：以用户为主的格式来显示进程状况
  - x：通常与 a 这个参数一起使用，可列出较完整信息

ps 命令标识进程的状态码

- 进程的 5 种主要状态码
  - D 无法中断的休眠状态(通常 IO 的进程)
  - R 正在运行中(在运行队列上)
  - S 处于休眠状态
  - T 停止或被追踪
  - Z 僵尸进程
- 其它状态码有：
  - W 进入内存交换 (从内核2.6开始无效)
  - X 死掉的进程(很少见)
  - < 优先级高的进程
  - N 优先级较低的进程
  - L 有些页被锁进内存
  - s 进程的领导者(在它之下有子进程)
  - l 多线程，克隆线程(使用 CLONE_THREAD)
  - + 位于后台的进程组

例：

```
[root@localhost ~]# ps aux
USER       PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
root         1  4.0  0.1  19232  1500 ?        Ss   06:43   0:00 /sbin/init
root         2  0.0  0.0      0     0 ?        S    06:43   0:00 [kthreadd]
root         3  0.0  0.0      0     0 ?        S    06:43   0:00 [migration/0]
root         4  0.0  0.0      0     0 ?        S    06:43   0:00 [ksoftirqd/0]
root         5  0.0  0.0      0     0 ?        S    06:43   0:00 [migration/0]
```

- USER：进程属于那个使用者账号
- PID ：进程的进程 ID号
- %CPU：进程使用掉的 CPU 资源百分比
- %MEM：进程所占用的物理内存百分比
- VSZ ：进程使用掉的虚拟内存量 (Kbytes)
- RSS ：进程占用的固定的内存量 (Kbytes)
- TTY ：进程是在那个终端机上面运作，若与终端机无关，则显示？
  另外，tty1~tty6 是本机上面的登入者程序，若为 pts/0 等，则表示由网络连接进主机的程序
- STAT：程序目前的状态
- START：进程被触发启动的时间
- TIME ：进程实际使用 CPU 运作的时间
- COMMAND：程序的实际指令

进程的基本操作

- fork 系统调用
- exec 系统调用
- exit 系统调用
- wait 系统调用
- sleep 函数调用

fork 系统调用

- 原型： pid_t fork(void)
- 头文件： unistd.h
- 功能：
  - 创建一个与原来进程几乎完全相同的进程，即两个进程可以做完全相同的事，但如果初始参数或者传入的变量不同，两个进程也可以做不同的事
  - 一个进程调用 fork 函数后，系统先给新的进程分配资源，例如存储数据和代码的空间。然后把原来的进程的所有值都复制到新的进程中，只有少数值与原来的进程的值不同。相当于克隆了一个自己
- 返回值：fork 被调用一次却能够返回两次且可能有三种不同的返回值：
  - 1) 在父进程中，fork 返回新创建子进程的进程 ID(通常为父进程 PID+1)
  - 2) 在子进程中，fork 返回 0
  - 3) 如果出现错误，fork 返回一个负值
- fork 出错可能有两种原因：
  - 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
- 两个进程有不同的 PID，但执行顺序由进程调度策略决定

fork 示例(视频：39 进程管理：ex_fork.c)：

```
#include <unistd.h>/*系统调用，如 fork、pipe 以及各种 I/O 原语(read、write、close等)*/
#include <stdio.h>
int main()
{
   pid_t fpid;  // fpid 用于 fork 函数的返回值
   int count=0;
   fpid=fork(); // 父进程调用则创建子进程并返回子进程PID，子进程调度则返回0
   if (fpid < 0)  // 如果出错
      printf("error in fork!");
   else if (fpid == 0) {  // 如果是子进程
      printf("I am the child process, my process id is %d\n", getpid()); // getpid()取pid
      count++;                                                  // getppid()可以取父进程的 pid
   }
   else {  // 否则是父进程
      printf("I am the parent process, my process id is %d\n", getpid());
      count++;
   }
   printf("count = %d\n",count);
   return 0;
}
```

```
[root@localhost ~]# gcc -o ex_fork ex_fork.c
[root@localhost ~]# ./ex_fork
I am parent process,my process id is 1398
count=1
[root@localhost ~]# I am child process,my process id is 1399
count=1
```

注意：两个进程的变量(count、fpid)是相互独立的，存储在不同的地址中，不是共用的

exec 系统调用

- 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[])
- 头文件：unistd.h
- 功能：以新进程替代原有进程，但 PID 保持不变
- 返回值：出错则返回 -1，失败原因记录在 error 中

后缀	操作能力
l	希望接收以逗号分隔的参数列表，列表以 NULL 指针作为结束标志
v	希望接收一个以 NULL 结尾的字符串数组的指针
p	是一个以 NULL 结尾的字符串数组指针，函数可以利用 PATH 变量查找子程序文件
e	函数传递指定参数 envp，允许改变子进程的环境，无后缀 e 时，子进程使用当前程序的环境

exec 系统调用(续)

- exec 的最基本调用为：execve (pathname，argv，envp)
- 6 个函数的区别
  - 1) 查找方式不同
    - 前 4 个函数的查找方式都是完整的文件目录路径，而后 2 个函数(以 p 结尾的两个函数)可以只给出文件名，系统会自动从环境变量“$PATH”所指出的路径中进行查找
  - 2) 参数传递方式不同：exec 函数族的参数传递有两种方式
    - 函数名的第 5 位字母为“l”(list)的表示逐个列举的方式
    - 函数名的第 5 位字母为“v”(vertor)的表示将所有参数整体构造成指针数组传递，然后将该数组的首地址当做参数传给它，数组中的最后一个指针要求是NULL
  - 3) 环境变量不同：
    - 以“e”(environment)结尾的两个函数 execle、execve 可以在 envp[] 中指定当前进程所使用的环境变量替换掉该进程继承的环境变量
    - 其它函数把调用进程的环境传递给新进程

示例：exec调用(视频：40 进程管理：ex_exec)

- 源代码1:beexec.c

```
#include <stdio.h>
#include <unistd.h>
void main(int argc, char* argv[])
{
   int i;
   puts("execution params:");
   for(i=0; i<argc; i++)
      printf("param %d is: %s\n", i, argv[i]);
}
```

```
[root@localhost ~]# gcc -o doexec doexec.c
[root@localhost ~]# ./doexec test1
this info may not be exported
execution params:
param 0 is: ./doexec
param 1 is: test1
[root@localhost ~]# ./doexec test1 test2
this info may not be exported
execution params:
param 0 is: ./doexec
param 1 is: test1
param 2 is: test2
```

- 源代码2:doexec.c

```
#include <stdio.h>
#include <unistd.h>
void main(int argc, char* argv[])
{
   puts("this info may not be exported");  // 原因：此语句的输出可能在缓冲区中，当
                                           // 调用 beexec程序时，缓冲区可能被后者清
                                           // 空，因而不能被显示出来
   execv("beexec", argv);  //注意：一旦exec执行成功，其后的代码便不再执行
   puts("normally,this info cannot be exported");
}
```


exit 系统调用

- 原型1： void _exit(int status)
  - 头文件：unistd.h
- 原型2： void exit(int status)
  - 头文件：stdlib.h
- 功能：终止发出调用的进程，status 是返回给父进程的状态值，父进程可通过 wait 系统调用获得
- exit() 和 _exit() 的区别：
  - _exit() 的作用最简单：直接使进程停止运行，清除其使用的内存空间，并销毁其在内核中的各种数据结构
  - exit() 在终止进程之前要检查文件的打开情况，把文件缓冲区中的内容写回文件，即“清理 I/O 缓冲”
  - 两者最终都要将控制权交给内核
  - 因此，要想保证数据的完整性，就一定要使用 exit()

在 Linux 中如何让一个进程退出？

- 正常退出
  - 1) main() 函数执行完成或在 main() 函数中执行 return
  - 2) 调用 exit() 函数
  - 3) 调用 _exit() 函数
- 异常退出
  - 1) 调用 abort() 函数
  - 2) 进程收到某个信号，而该信号使程序终止

exit 和 return 的区别

- 1) exit 是函数，有参数，其执行完后把控制权交给系统
- 2) return 是函数执行完后把控制权交给调用函数

- 注意
  - exit() 终止进程时，将使终止的进程进入僵死状态，释放它占有的资源，撤除进程上下文，但仍保留 proc 结构
  - 子进程还未终止，但父进程已终止时，将交由 init 进程处理

wait 系统调用

- 原型1：pid_t wait(int *status)
- 原型2：pid_t waitpid(pid_t pid,int *status,int options)
- 头文件(两个)： sys/types.h 和 sys/wait.h
- 返回值：成功则返回成功结束运行的子进程的进程号 PID，否则返回 -1
- 参数：
  - status：返回子进程退出时的状态
  - pid：
    - pid>0时：等待进程号为 pid 的子进程结束
    - pid=0时：等待组 ID 等于调用进程组 ID 的子进程结束
    - pid=-1时：等待任一子进程结束，等价于调用 wait()
    - pid<-1时：等待组 ID 等于 PID 的绝对值的任一子进程结束
  - options：
    - WNOHANG：若 pid 指定的子进程没有结束，则 waitpid() 不阻塞而立即返回，此时的返回值为 0
    - WUNTRACED：为了实现某种操作，由 pid 指定的任一进程已被暂停，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

wait 系统调用(续)

- wait() 与 waitpid() 的区别
  - wait() 函数等待所有子进程的僵死状态
  - waitpid() 函数等待 PID 与参与 pid 相关的子进程的僵死状态
- 检查子进程的返回状态码 status

宏名	说明
WIFEXITED(status)	进程中通过调用 _exit() 或 exit() 正常退出，该宏值为非 0
WIFSIGNALED(status)	子进程因得到的信号没有被捕捉导致退出，该宏值为非 0
WIFSTOPPED(status)	子进程没有终止但停止了，并可重新执行时，该宏值为为 0 这种情况仅出现在 waitpid() 调用中使用了 WUNTRACED 选项
WEXITSTATUS(status)	如果 WIFEXITED(status) 返回非 0，该宏返回由子进程调用 _exit(status) 或 exit(status) 时设置的调用参数 status 值
WTERMSIG(status)	如果 WIFSIGNALED(status) 返回非 0，该宏返回导致子进程退出的信号的值
WSTOPSIG(statis)	如果 WIFSTOPPED(status) 返回非 0，该宏返回导致子进程停止的信号的值

sleep 函数调用

- 原型：unsigned int sleep(unsigned int seconds)
- 头文件：unistd.h
- 功能：调用 sleep() 会使进程主动进入睡眠状态，直到指定的秒数已到
- 返回值：睡眠时间已到则返回 0，如果睡眠进程被信号提前唤醒，则返回值为原始秒数减去已睡眠秒数的差

- 线程休眠函数：void usleep(unsigned long usec)
  - 单位：微秒

- 延时函数：void delay(unsigned int msec)
  - 可以延时 msec*4 毫秒，即 delay(250) 将延时一秒

进程控制综合示例1

```
// comprehensive.c
#include <stdio.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <unistd.h>
#include <wait.h>
#include <errno.h>
#include <stdlib.h>
// 功能：进程等待wait()方法的应用

void waitprocess();

void main(int argc, char * argv[])
{
   waitprocess();
}
```


进程控制综合示例1(视频：41 进程管理：comprehensive.c)(续)

```
void waitprocess() {
   int count = 0;
   pid_t pid = fork();
   int status = -1;
   if(pid<0)  {
      printf("fork error for %d\n", errno );
   }else if(pid>0)  {
      printf("this is parent ,pid = %d\n", getpid() );
      wait(&status);//父进程阻塞自己，直到有子进程结束。
   }else  {          //当发现有子进程结束时，就会回收它的资源
      printf("this is child , pid = %d , ppid = %d\n",getpid(), getppid() );
      int i;
      for (i = 0; i < 10; i++) {
         count++;
         sleep(1);
         printf("count = %d\n", count) ;
      }
      exit(5);
   }
   printf("child exit status is %d\n", WEXITSTATUS(status));//status按位存储状态信息
   printf("end of program from pid = %d\n", getpid() );
}
```

```
[root@localhost ~]# ./comprehensive
this is parent, pid=1374
this is child, pid=1374
count=1
count=2
count=3
count=4
count=5
count=6
count=7
count=8
count=9
count=10
child exit status is 5
end of program for pid=1374
```

进程的特殊操作

- 获得进程 ID
- setuid 和 setgid 系统调用
- setpgrp 和 setpgid 系统调用
- chdir 系统调用
- chroot 系统调用
- nice 系统调用

获得进程 ID

- 进程 ID 包括：
  - PID / PPID：进程号/父进程号，即登录时的用户名
  - UID / GID：用户 ID / 组 ID
  - EUID / EGID：有效用户 ID / 有效的组 ID，决定对文件的访问权限，SUID/SGID 的设置(如 setuid() 函数或者 chmod 命令)会影响这个值
    - EUID 还包括是否有使用 kill 系统调用发送软中断信息到 Linux 内核结束进程的权限
    - EGID：它与 GID 并不一定相同(因为进程执行时所属用户组可能改变)
  - PGID：进程组 ID

- GID 与 PGID 的区别
  - 一般地，执行进程的用户组 ID 就是该进程的 GID，如果该执行文件设置了 SGID 位，则文件所属群组 ID 就是该进程的 GID
  - 一个进程在 shell 下执行，shell 程序就将该进程的 PID 作为该进程组 PGID
  - 从该进程派生的子进程都拥有父进程所属进程组 PGID，除非父进程将子进程的 GPID 设置成与该子进程的 PID 一样

进程 ID 相关函数

- 原型：
```
pid_t getpid(void);          // 返回进程 ID
pid_t getppid(void);         // 返回父进程 ID
uid_t getuid(void);          // 返回实际用户 ID
uid_t geteuid(void);         // 返回有效用户 ID
gid_t getgid(void);          // 返回实际组 ID
gid_t getegid(void);         // 返回有效组 ID
pid_t getpgid(pid_t pid);    // 返回进程组 ID
pid_t getpgrp(void);         // 同上，POSIX.1 版
pid_t getpgid(pid_t pid);    // 同上， BSD 版
```
- 头文件：unistd.h
  sys/types.h

setuid 和 setgid 系统调用

- 原型：int setuid(uid_t uid);
  - 功能：该函数会检验用户的真实身份。如果参数 uid 为根用户 UID，为保障系统的安全性，Linux 内核将以进程表和 u 区中用户真实的 ID 来设置
  - 注意：使用该函数要十分小心
    - 当进程的 EUID 为根用户，而参数 uid 为普通用户 UID，则进程的 UID 不能再被设置为根用户
    - 若进程创建初期需要根用户权限，完成相应任务后不再需要根用户权限，则可设置可执行文件的 SUID 信息，并将所有者设置为根用户。这样，进程创建时的 UID 为根用户，不需要根用户时可用 setuid(getuid()) 恢复进程的 UID 和 EUID
- 原型：int setgid(gid_t gid);
  - 功能：该函数不会检验用户的真实身份而用参数 gid 直接设置
- 返回值：成功返回0，否则返回 -1 并设置 errno 值

setpgrp 和 setpgid 系统调用

- 原型：int setpgrp(void);
  - 功能：直接将进程的 PGID 设置为与 PID 相同的数值
- 原型：int setpgid(pid_t pid, pid_t pgid);
  - 功能：将进程号为 pid 的进程 PGID 设置为 pgid
    - pid=0 时，则修改调用者进程的 PGID
    - pgid=0 时，则将所有 PID 等于 pid 的进程的 PGID 修改为 pgid
    - 如果 PGID 原本为根用户所有，则只有在指定进程与调用进程的 EUID 相同时，或者指定进程为调用进程的子进程时才有效
- 返回值：成功返回 0，否则返回 -1 并设置 errno 值

chdir 系统调用

- 原型：int chdir(const char* path);
- 功能：在文件控制中，chdir() 函数将进程的当前工作目录改为由参数指定的目录
- 参数：path 为指定的目录路径，发出该调用的进程必须具备该目录的执行权限
- 返回值：成功返回 0，否则返回 -1 并设置 errno 值

chroot 系统调用

- 原型： int chroot(const char* path);
- 功能：该函数操作又称为根交换操作，作用通常是在一个 Linux 系统上虚拟另一个 Linux 系统，根交换后，所有的命令操作都被重新定向
- 参数：path 为新的根目录路径，执行后，进程将以该目录作为根目录，并且使进程不能访问该目录以外的内容
- 说明：
  - 该操作不能改变当前工作目录，如果当前工作目录在指定目录以外，则无法访问其中内容
  - 根交换操作只能由根用户发出
- 返回值：成功返回 0，否则返回 -1 并设置 errno 值

nice 系统调用

- 原型： int nice(int inc);
- 功能：改变进程的优先级
- 参数：inc 为调用 nice() 函数的进程优先级数值的增量，该值越小则优先级值越小，进程被调度运行的机会越大
- 说明：只有根用户能为 inc 参数设置负值，使进程优先级提高，普通用户只能设置正值来降低进程的优先级
- 返回值：成功返回 0，否则返回 -1 并设置 errno 值

进程综合示例2(视频：42 进程管理：comprehensive2.c)

```
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
#include <stdlib.h>
void main() {
   pid_t pid;
   pid=fork();
   if(pid<0) {
      printf("fork failed");
      exit(EXIT_FAILURE);
   } else if(pid==0)   {
      printf("I am child, uid=%d, euid=%d, gid=%d, egid=%d\n",
            getuid(), geteuid(), getgid(), getegid());
   } else   {
      printf("I am parent, uid=%d, euid=%d, gid=%d, egid=%d\n",
            getuid(), geteuid(), getgid(), getegid());
   }
}
```

```
[root@localhost ~]# gcc -o comprehensive2 comprehensive2.c
[root@localhost ~]# ./comprehensive2
I am parent, uid=0, euid=0, gid=0, egid=0
[root@localhost ~]# I am child, uid=0, euid=0, gid=0, egid=0
```