进程标示及获取

1．进程id：同一个时刻是唯一的，可以用来区分不同的进程，如果一个进程结束后，id 可以被其他进程使用，一般采用延迟复用原则

2．进程id为 0 的进程是交换进程， 是内核的一部分，不执行任何的磁盘上的程序

3． 进程id为 1 的是init进程 ，在自举过程结束后由内核调度，用来初始化系统的状态，是一个以超级用户权限执行的用户进程，也是所有孤儿进程的父进程。

```
pid_t getpid(void);//调用进程的进程id

pid_t getppid(void); //调用进程的父进程id

uid_t getuid(void); //调用进程的实际用户id

uid_t geteuid(void);  //调用进程的有效用户id

gid_t getgid(void); //调用进程的实际组id

gid_t getegid(void); //调用进程的有效组id
```

新进程的创建

pid_t fork(void); //子进程返回0，父进程返回子进程id，出错返回-1

1．调用一次，返回两次

2．写时复制原则

3．fork之后标准IO缓冲区的问题

4．fork之后文件共享的问题：fork会复制文件描述表，然后子进程从父进程继承下来的文件描述符指向同一个文件列表

5．fork失败的原因：资源不足

```c
int main()
{
    FILE *fp = fopen("./tmp.txt", "w");
    if(fp == NULL)
        err_exit("fopen");
    fprintf(fp, "hahahah");

    pid_t pid = fork();

    if(pid == 0)
    {
        sleep(3);
    }
    else
    {
        sleep(2);
    }
    return 0;
}
```

```c
int main()
{
    int fd = open("tmp.txt", O_RDWR);
    if(fd == -1)
        err_exit("open");

    pid_t pid = fork();

    if(pid == 0)
    {
        write(fd, "&&&", 3);
    }
    else
    {
        sleep(2);
        write(fd, "---", 3);
    }
    return 0;
}
```

父子进程共享的内容

实际用户id，实际组id，有效用户id，有效组id，附属组id，进程组id，会话id

控制终端

设置用户id标志和设置组id标志

当前工作目录

根目录

文件模式和创建屏蔽字

信号屏蔽和安排

文件描述符标志

环境变量表

共享存储段

存储映像

资源限制

父子进程区别

父进程id和进程id不同

fork的返回值不同

未决信号集被设置为空

不继承父进程设置的文件锁

未处理闹钟被清除

子进程的tms_utime, tms_stime, tms_cutime, tms_ustime的值设置为0

vfork函数

pid_t vfork(void); //子进程返回0,父进程返回子进程id,失败返回-1

vfork适用于子进程立刻进行exec的情况，因为vfork不会进行父进程地址空间的完全复制， 在子进程调用fork之前，其在父进程的地址空间运行

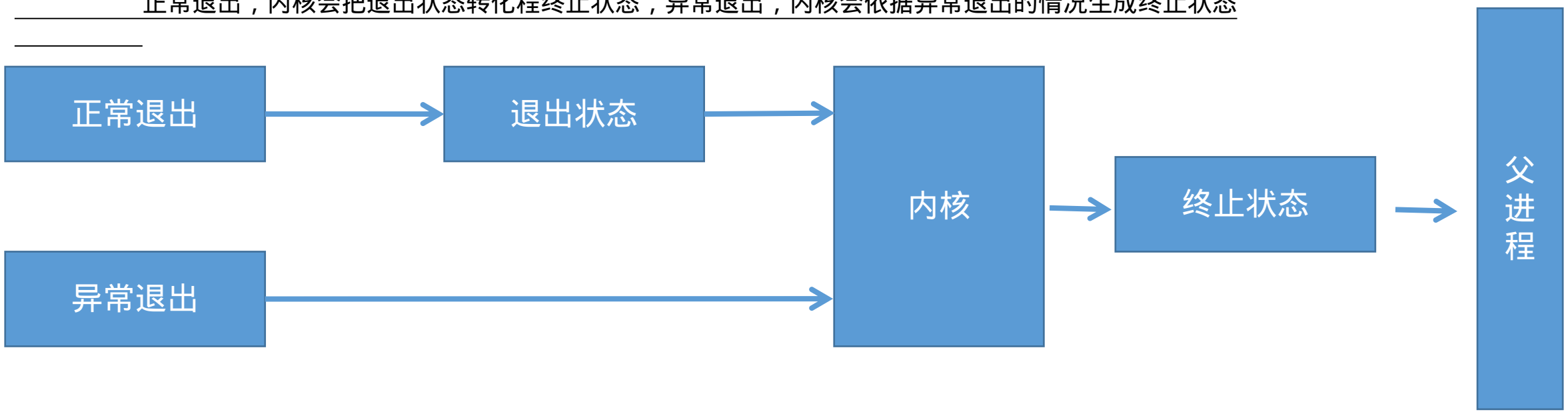
vfork保证子进程先运行，直到子进程调用exec或者exit后父进程才会被调度运行

进程结束深入再谈

正常退出：1.return语句，相当于调用exit
2.exit语句，会先调用终止处理程序，然后刷新标准io缓冲区，一般现代的实现，exit不再负责fclose
3._exit和_Exit，不调用终止处理程序，不刷新标准io缓冲区，直接陷入内核
4.进程的最后一个线程执行return，但是该返回值不作为进程的返回值，进程以状态0返回
5.进程的最后一个线程执行pthread_exit，同上，返回值不作为进程的返回值，进程以状态0返回

异常退出：1.abort调用
2.信号
3.最后一个线程接受pthread_cancel的请求

进程终止后，如何通知父进程其如何终止的？
正常退出，内核会把退出状态转化程终止状态，异常退出，内核会依据异常退出的情况生成终止状态



孤儿进程与僵尸进程

孤儿进程：父进程先于子进程死亡， 则子进程变为孤儿进程。当一个进程终止的时候，内核会检查所有的活动进程，查看他们是不是属于该即将终止进程的子进程，如果是的话，就把他们的父进程修改为 1 ，即init进程。

僵尸进程：子进程死亡，父进程没有对内核为子进程残留的资源[终止状态等]进行回收[调用wait簇函数]，则变为僵尸进程。僵尸状态会跟随父进程结束一块被更高层的进程回收

进程回收_wait

当一个进程结束的时候，会向其父进程发送SIGCHLD信号， 父进程可以忽略，也可以设置信号处理函数进行捕获，也可以调用wait函数进行接收

```
pid_t wait(int *status);

pid_t waitpid(pid_t pid, int *status, int options)

        //成功，返回回收的子进程id,失败，返回-1或者0
```

wait：如果有子进程结束后，此时调用wait则立即返回， 如果没有子进程结束，调用wait将会阻塞等待，如果没有子进程，返回-1

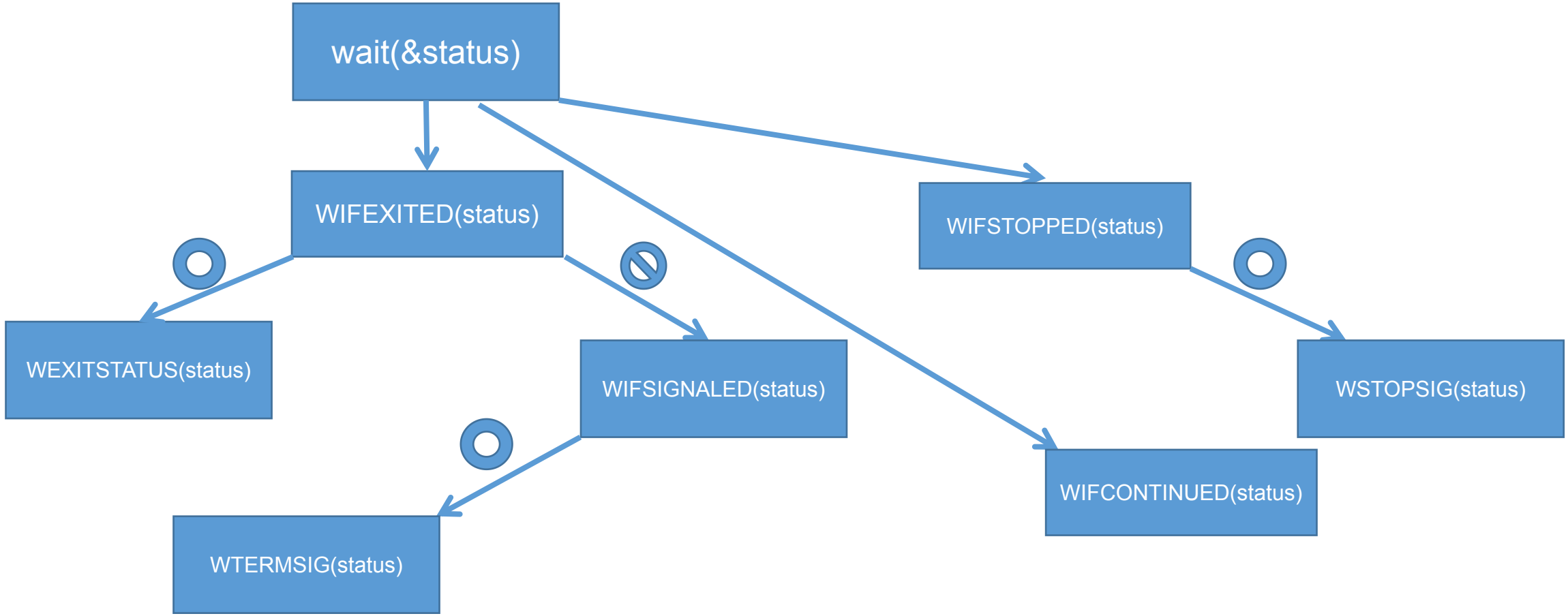
waitpid:

- pid==-1: 等待任意子进程
- pid>0: 等待进程ID为pid的子进程
- pid==0：等待组id等于调用进程组id的任一子进程
- pid<-1:等待组id等于pid绝对值的任一子进程

```
options  //不仅可以获取终止状态，还可以获取停止状态的信息
[
    WCONTINUED
    //实现支持作业控制的前提下，如果由pid指定的子进程在停止后已经继续，但是其状态尚未报告，则返回状态
    WNOHANG
    //非阻塞模式，如果没有终止的满足要求的子进程，则立刻返回0
    WUNTRACED
    //实现支持作业控制的前提下，如果由pid指定的子进程处于停止状态，但是其状态尚未报告，则返回状态
]
```

进程回收_wait

如何从终止状态status中获取到所需要的信息？



进程回收_waitid

使用到的时候再细究

```
int waitid(idtype_t idype, id_t id, siginfo_t *info, int options)
        //成功，返回0,出错，返回-1
```

进程回收_waitid

```
pid_t wait3(int *status, int options, struct rusage *rusage);

pid_t wait4(pid_t pid, int *status, int options, struct rusage *rusage);

        //成功，返回进程id，失败， 返回-1
```

允许返回终止进程的资源统计信息，例如cpu时间分布，缺页次数，接收到信号的次数等等

竞争条件

1.什么是竞争条件？

在多个进程执行环境下，最后运行的结果取决域进程运行的顺序。

```c
static void charatatime(const char *str)
{
    char *ptr;
    int c;
    setbuf(stdout, NULL);
    for(ptr=str; (c=*ptr++)!= 0;)
    {
        putc(c, stdout);
    }
}


int main()
{
    pid_t pid;
    if((pid = fork()) < 0)
        err_exit("fork");
    else
    {
        if(pid == 0)
        {
            charatatime("output for child\n");
        }
        else
        {
            charatatime("output for parent\n");
        }
    }
    exit(0);
    return 0;
}
```

```
zhaosong@zhaosong:~/WorkSpace/APUE/8$ ./8_5
output for parenotu
tput for child
```

exec函数簇

```c
extern char **environ;
int main()
{
    //1.
    int ret = execl("/bin/ls", "ls", "-l", "-a", NULL);
    if(-1 == ret)
        err_exit("execl");
    //2.
    char *argv[] = {"ls", "-l", "-a", NULL};
    int ret = execv("/bin/ls", argv);
    if(-1 == ret)
        err_exit("execv");
    //3
    int ret = execle("/bin/ls", "ls", "-l", "-a", NULL, environ);
    if(-1 == ret)
        err_exit("execle");
    //4
    char *argv[] = {"ls", "-l", "-a", NULL};
    int ret = execve("/bin/ls", argv, environ);
    if(-1 == ret)
        err_exit("execve");
    //5
    int ret = execlp("ls", "ls", "-l", "-a", NULL);
    if(-1 == ret)
        err_exit("execlp");
    //6
    char *argv[] = {"ls", "-l", "-a", NULL};
    int ret = execvp("ls", argv);
    if(ret == -1)
        err_exit("execvp");
    return 0;
}
```

l代表参数以可变参数的方式传入

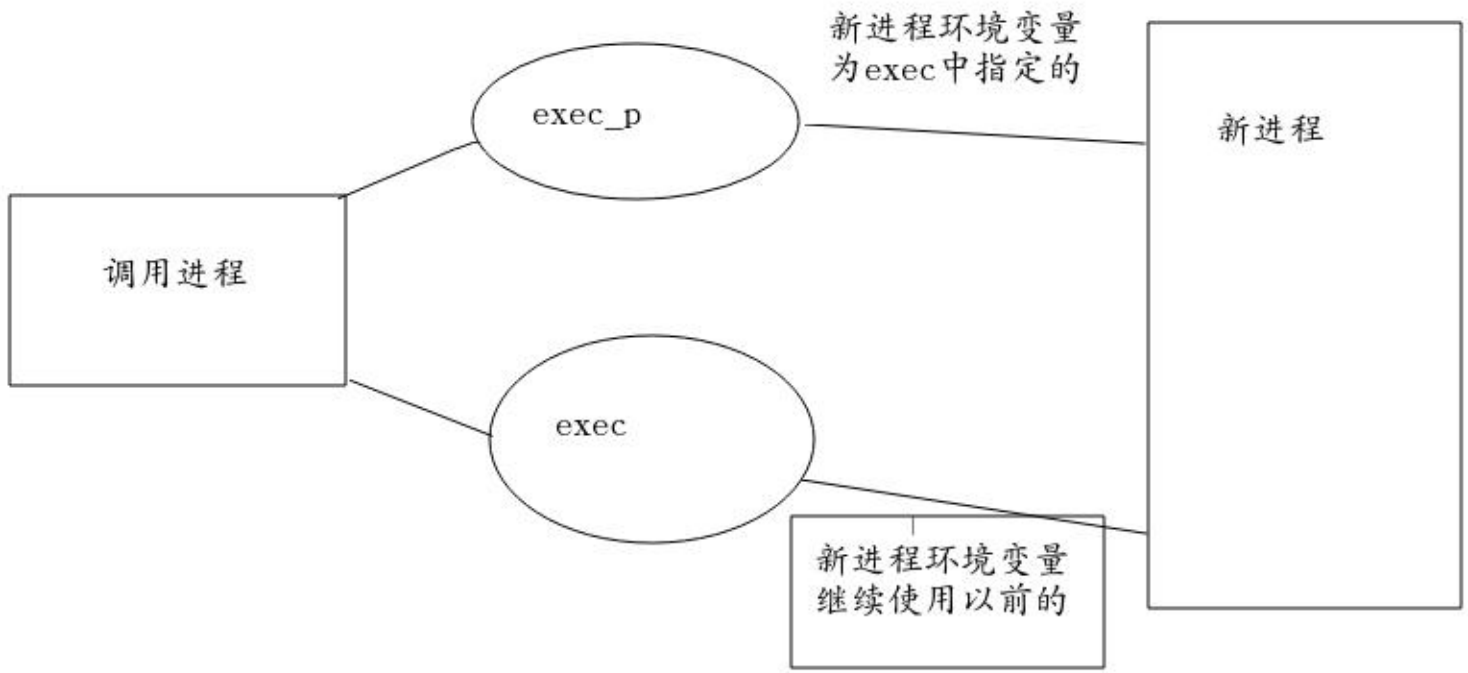
v代表参数以一个指针数组传入

p代表可执行文件直接指定名称也可以，这时候会从PATH中寻找

e代表可以同时传入环境变量

当exec找到了一个可执行文件，但是该文件不是由连接编译器产生的机器可执行文件，则认为该文件是一个shell脚本，试着调用/bin/sh, 并将filename作为shell的输入

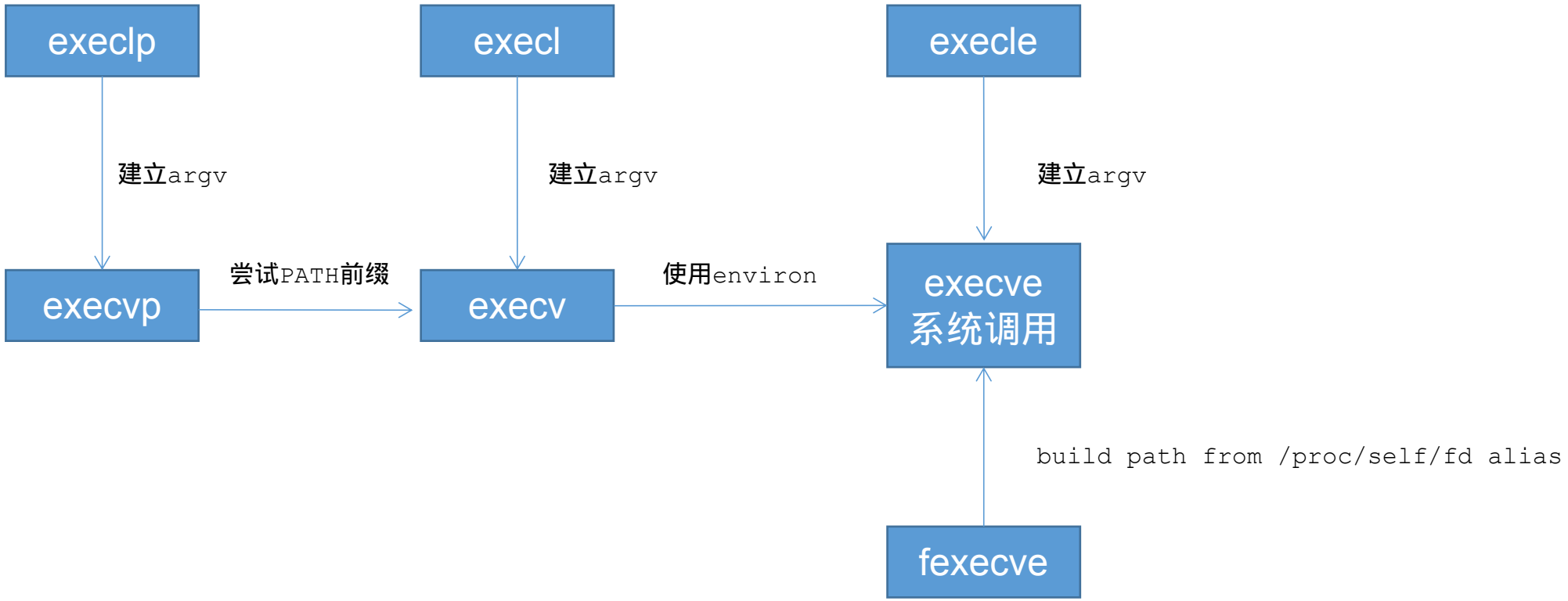
exec函数簇



文件描述符状态close-on-exec

如果设置了close-on-exev位，那么在exec后，将关闭该文件描述符，否则，不关闭。一般，除非特地使用fcntl，那么默认的都是不关闭。

在exec前后实际用户id和实际组id保持结束，而有效id是否改变则取决于所指向程序文件的设置id位是否设置。



更改用户id和更改组id

当用户A创建一个文件的时候，那么文件的实际ID就是该用户的ID，文件的有效ID一般等于实际ID，但是当调用exec执行了某些设置了设置用户ID为的文件后，A用户的有效ID会变的和exec调用的文件的实际ID一致。

要注意，真正权限检查时看的是有效用户ID

```
int setuid(uid_t uid);

int setgid(gid_t gid);

//更改用户的实际和有效id， 成功，返回0，出错，返回-1
```

注意：

如果进程是超级用户，则setuid将实际用户id,有效用户id和保存的设置用户id设置为uid

如果进程不是超级用户，但是uid等于实际用户ID或者保存的设置用户id，函数将有效用户id设置为uid

如果上不满足，出错，设置errno为EPERM，并返回-1

更改用户id和更改组id

```
int setreuid(uid_t ruid, uid_t euid);

int setregid(gid_t rgid, gid_t egid);

//设置实际id和有效id， 成功，返回0，出错，返回-1
```

注意：

非特权用户可以交换实际用户id和有效用户id

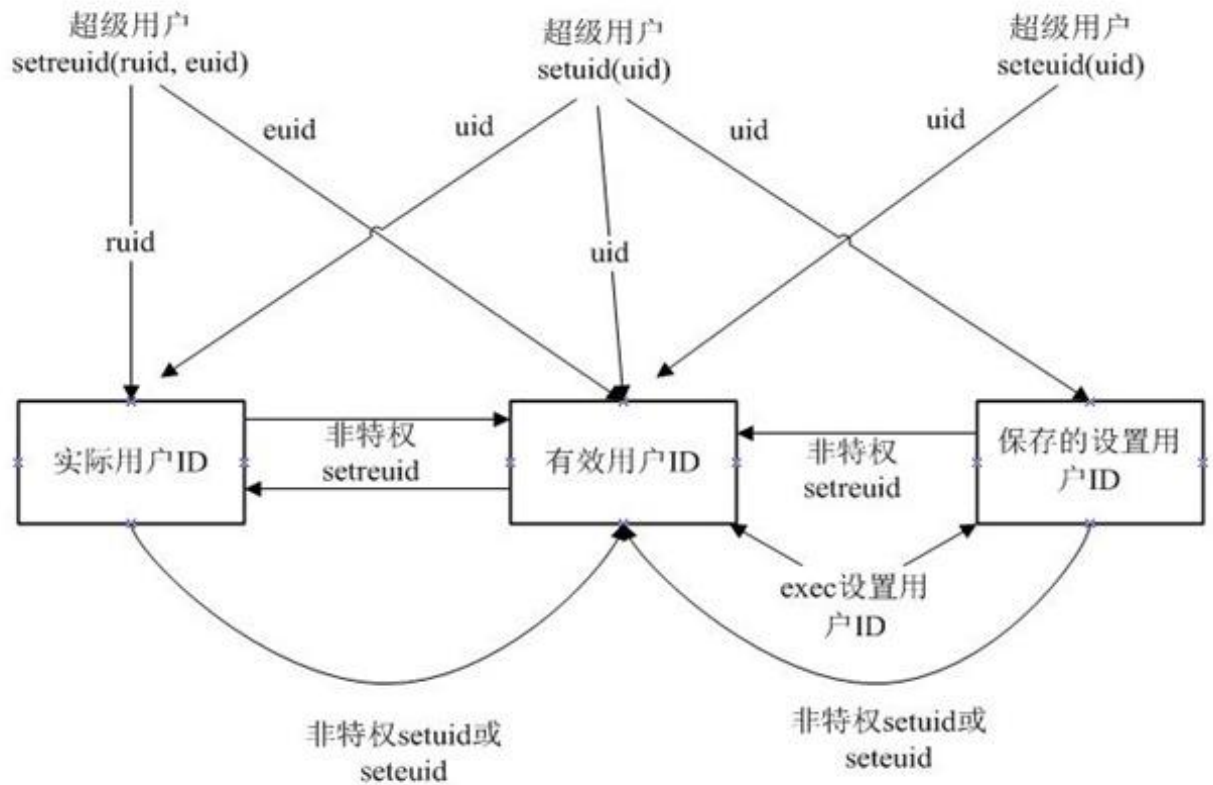
更改用户id和更改组id

```
int seteuid(uid_t uid);

int setegid(gid_t gid);

//设置有效id， 成功，返回0，出错，返回-1
```

注意：

非特权用户可以设置其有效id位实际id或者保存的设置id

更改用户id和更改组id



解释器文件

解释器文件是一个文本文件， 其开始的参数为

```
#! pathname [option-argument]
```

内核执行的不是解释器文件本身，而是第一行pathname指定的解释器

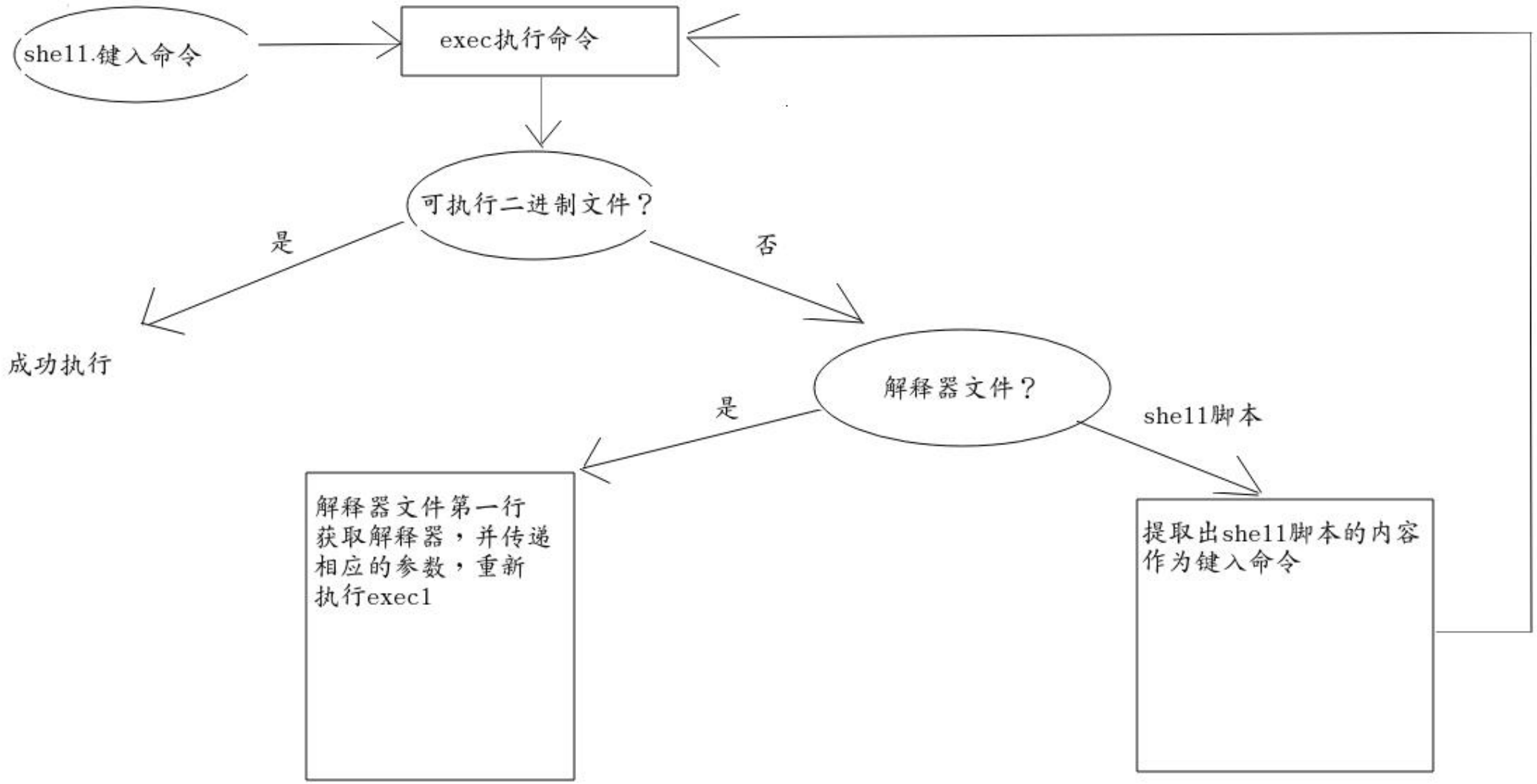
当一个进程调用exec去执行某个解释器文件的时候：

1.解释器文件需要有可执行权限

2.系统会先把解释器文件当成机器文件去执行，发现格式不符合，就认为是一个解释器文件，去第一行寻找pathname，进一步调用解释器

3.调用解释器后，将解释器文件的路径作为argv[2]， 将解释器文件第一行的解释器参数作为argv[1]， 将exec函数的参数列表的第二个开始作为argv[2+]的参数，argv[0]和一般的调用保持一致是解释器（可执行二进制文件）路径。

解释器文件


she11.键入命令
exec执行命令
可执行二进制文件？
是
否
成功执行
解释器文件？
是
she11脚本
解释器文件第一行
获取解释器，并传递
相应的参数，重新
执行exec1
提取出she11脚本的内容
作为键入命令

解释器文件

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

    if((pid = fork()) < 0)
    {
        err_exit("fork");
    }
    else
    {
        if(pid == 0)
        {
            if(execl("/home/zhaosong/WorkSpace/APUE/8/testinterp", "testinterp", "my1", "my2", (char*)0) < 0)
                err_exit("execl");
        }
    }
    if(waitpid(pid, NULL, 0) < 0)
        err_exit("waitpid");
    return 0;
}
```

```
zhaosong@zhaosong:~/WorkSpace/APUE/8$ ./8_8
argv[0]: /home/zhaosong/WorkSpace/APUE/8/echoargc
argv[1]: foo java python c_-
argv[2]: /home/zhaosong/WorkSpace/APUE/8/testinterp
argv[3]: my1
argv[4]: my2
zhaosong@zhaosong:~/WorkSpace/APUE/8$ cat testinterp
#! /home/zhaosong/WorkSpace/APUE/8/echoargc foo java python c_-
```

解释器文件

缺点：解释器文件由内核识别，会造成内核的额外开销

优点：

1. 解释器文件可以隐藏具体的语言细节，使得语言脚本被视为一个可执行程序
2. 用一个shell脚本替代解释器需要更多的开销
3. 可以使用其他的shell完成任务

函数system

```
int system(const char *cmdstring)
```

if cmdstring is NULL, return !0 when the system function is useful in this system

system 在实现中调用了fork, exec, waitpid，其返回值如下：

fork失败，或者waitpid返回除了EINTR之外的错误，system返回-1，并且设置errno

exec失败(该命令不能执行)，返回值如同shell执行了exit(127)

成功返回shell的终止状态

```c
int system(const char *cmdstring)
{
    pid_t pid;
    int status;

    iuf(cmdstring == NULL)
        return 1;
    if((pid = fork()) < 0)
        status = -1;
    else
    {
        if(pid == 0)
        {
            execl("/bin/sh", "sh", "-c", cmdstring, (char*)0);
            _exit(127);
        }
        else
        {
            while(waitpid(pid, &status, 0) < 0)
            {
                if(errno != EINTR)
                {
                    status = -1;
                    break;
                }
            }
        }
    }
    return status;
}
```

用户标识

```
char *getlogin(void)
```

成功，返回指向登录名字符串的指针，出错，返回NULL

进程调度

友好值：

友好值代表一个进程对其他进程的友好程度，友好值越大，代表越能容忍其他进程先抢占CPU，及本身的优先级越低

友好值取值范围[-NZERO, NZERO-1]，NZERO是系统默认的友好值 可以通过sysconf(_SC_NZERO)获取，LINUX为20

The range of the nice value is +19 (low priority) to -20 (high priority).

```
int nice(int incr); //将incr加在原始的友好值上面
```

成功，返回新的友好值，出错，返回-1

```
int getpriority(int which, id_t who);
```

成功，返回友好值，出错，返回-1

```
int setpriority(int which, id_t who, int value);
```

成功，返回0， 出错，返回-1

友好值在调用exec后保持，在fork之后是否保持取决于实现

进程时间

```
clock_t times(struct tms *buf);
```

成功，返回墙上时钟时间，出错，返回-1

```
struct tms {
    clock_t tms_utime;  /* user time */
    clock_t tms_stime;  /* system time */
    clock_t tms_cutime; /* user time of children */
    clock_t tms_cstime; /* system time of children */
};
```