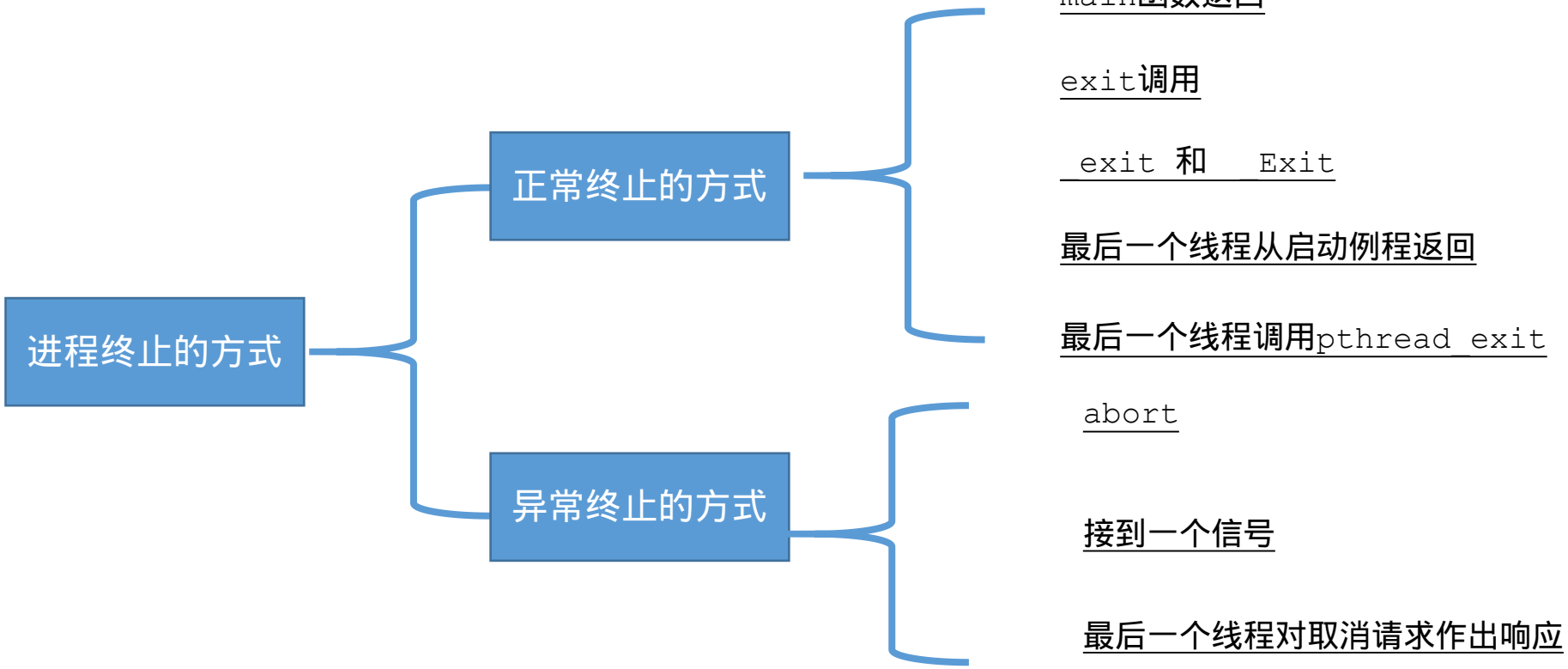
进程启动

```
int main(int argc, char *argv[], char *environ[])
```

当内核执行c的时候，会调用exec函数，在调用main前会执行一个特殊的历程，该历程从内核取得命令行参数和环境变量值，然后转调main函数

进程终止


进程终止的方式
正常终止的方式
异常终止的方式
main函数返回
exit调用
_exit 和 _Exit
最后一个线程从启动例程返回
最后一个线程调用pthread_exit
abort
接到一个信号
最后一个线程对取消请求作出响应

进程终止_退出函数

```
void exit(int status)
```

```
void _Exit(int status)
```

```
void _exit(int status)
```

1． exit会在进入内核之前，执行flose关闭该进程中所有的标准IO.

2． status： 终止状态

如果main函数返回值为void， 则终止状态为未定义的

```
int atexit(void (*func)(void))  //成功，返回0,出错，返回非0
```

注册终止处理程序

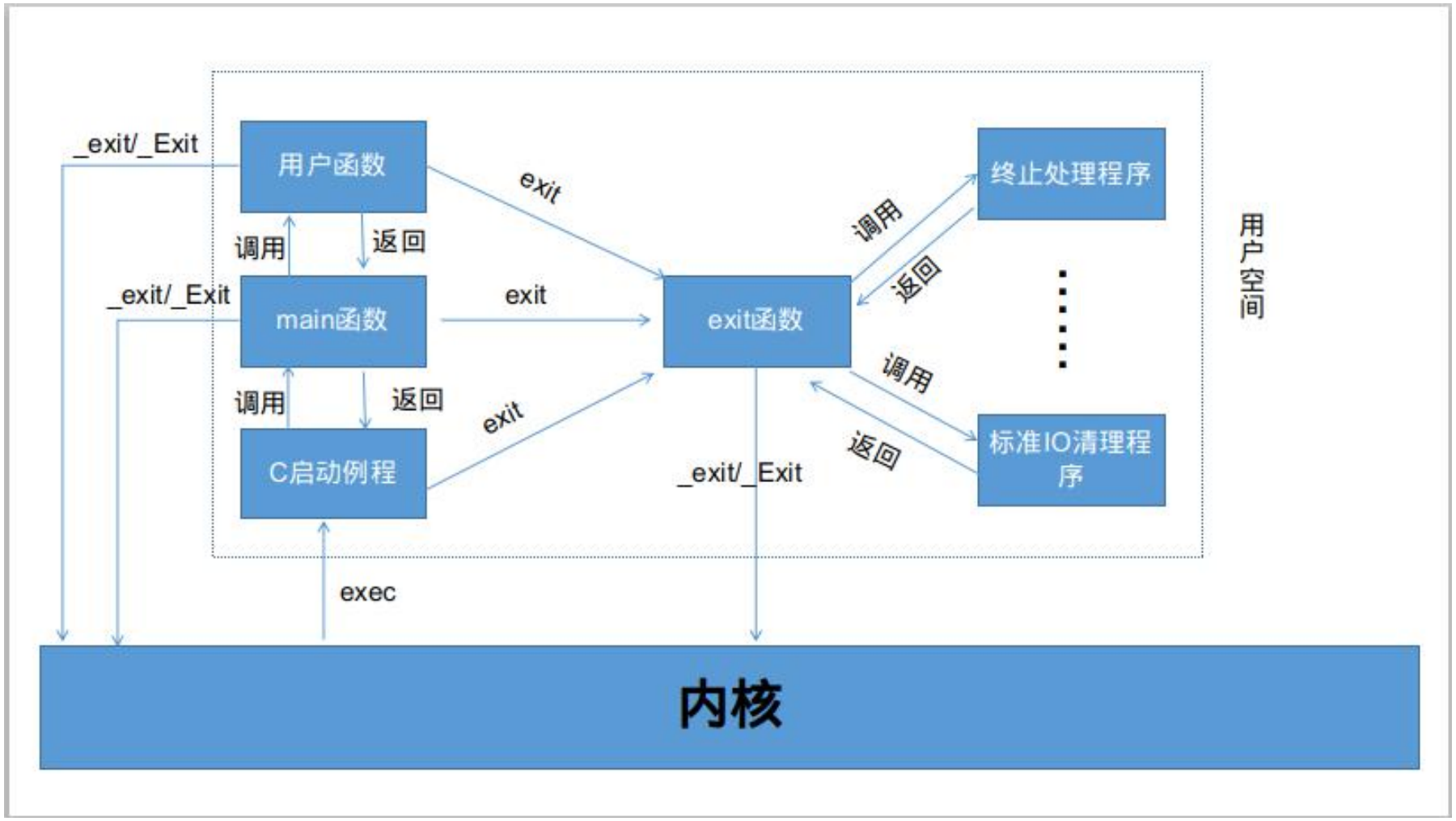
注册的程序执行顺序与登记顺序相反

注册多次也会多次执行

调用exec函数簇的任一函数，将会清空已经注册的终止处理程序

终止处理程序会在调用了exit函数后被执行，在fclose之前

C 程序启动和终止


_exit/_Exit
用户函数
exit
调用
返回
终止处理程序
调用
返回
用户空间
_exit/_Exit
main函数
exit
exit函数
调用
返回
调用
返回
exit
C启动例程
_exit/_Exit
返回
标准IO清理程序
exec
内核

命令行参数

1．　执行exec的进程可以把命令行参数传递给新程序

```c
#include<apue.h>


int main(int argc, char **argv)
{
    int i = 0;
    while(argv[i] != NULL)
        printf("argv[%d]: %s\n", i++, argv[i]);
    return 0;
```

环境变量

```c
#include<apue.h>

extern char **environ;

int main(int argc, char **argv)
{
    int i = 0;
    while(environ[i] != NULL)
        printf("environ[%d]: %s\n", i++, environ[i]);
    return 0;
```

环境变量

```
char *getenv(const char *name)
```

返回与name关联的value的指针，如果不存在,返回NULL

```
int putenv(char *str);
```

成功，返回0， 失败，返回非 0

```
int setenv(const char *name, const char *value, int rewrite);
```

```
int unsetenv(const char *name)
```

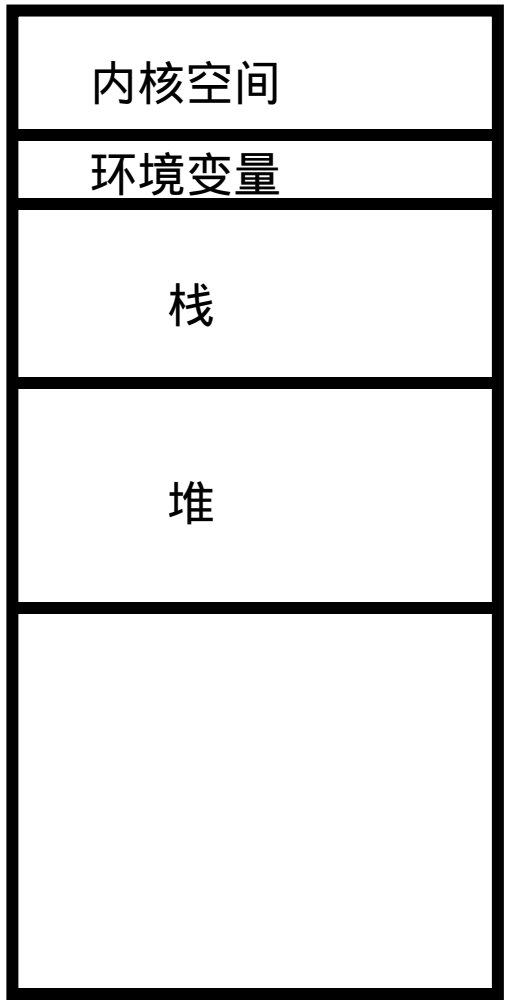
成功，返回0,失败，返回-1

putenv在插入新的环境变量的时候，如果已经存在就先删除，再插入，不存在，就直接插入， str的格式为”name=value”;

setenv在插入新的环境变量的时候，如果设置了rewrite为非零，和putenv相同，如果为 0 ，那么已经存在，将不会插入，也不会报错

unsetenv删除name的定义，即使不存在定义也不算出错

环境变量

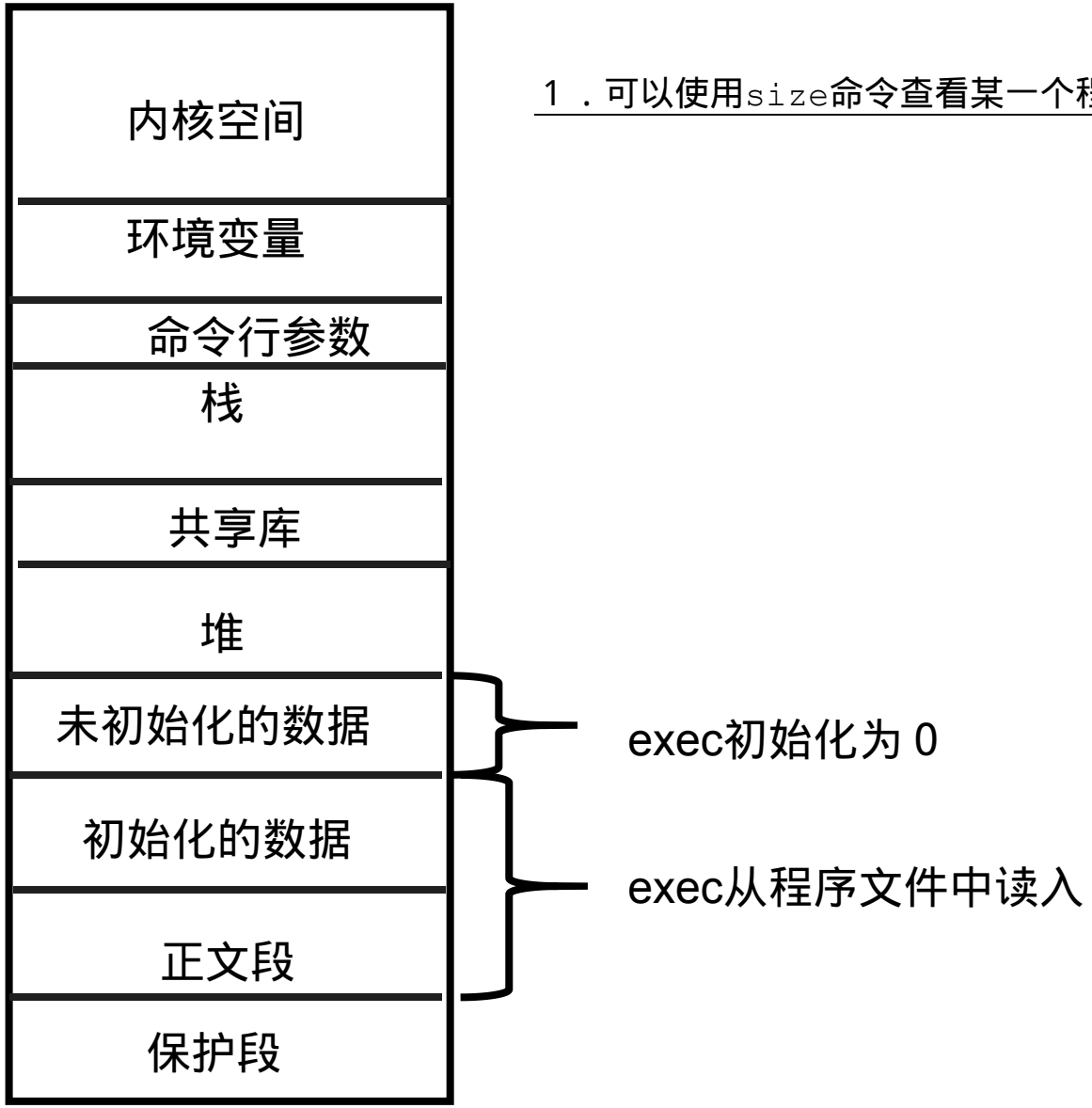


1． 环境变量的大小刚刚好够存储初始的环境变量指针表和对应的环境变量值

2． 修改一个现有的name的value，如果新的value比原始的value占用的空间小，那么直接放在原始的value空间就可以，如果大与原始的value空间，就会在堆中开辟一个新的空间，存储value,并更新环境变量指针表中的指针

3． 添加一个新的name， 要对环境变量指针表也要扩容，调用malloc开辟一个新的堆空间，先把原始的指针数据复制过来，再添加新的，这样操作之后，环境变量指针表中有一部分的指针指向堆中的nameValue， 有一部分指向原始的环境变量内存段。在后续的新增中，只需要调用realloc继续对环境变量指针表进行扩容就可以了。

C程序的0-4G地址空间分布



1．可以使用size命令查看某一个程序的.text \.data\.bss段的一个大小情况

共享库

1．　共享库使得可执行文件中不再需要包含公用的函数库，而只需要在所有进程都可以引用的存储区中保存这种库例程的一个副本

```
1
2 %:%.c ../lib/libapue.so
3     gcc -o $@ $< -L../lib -lapue -I../inc
4
5 ../lib/libapue.so: ../src/apue.o
6     gcc -shared -o ../lib/libapue.so ../src/apue.o
7
8 ../src/apue.o:../src/apue.c
9     gcc -fPIC -o ../src/apue.o -c ../src/apue.c -I../inc
```

静态存储空间分配

`void *malloc(size_t size)` //分配指定字节数的存储区，此存储区中的初始值不确定

`void *calloc(size_t nobj, size_t size)`//分配指定字节数的存储区，且被初始化为 0

`void *realloc(void *ptr, size_t size)` //改变原来的存储区的大小，如果是减少，那么会被截断，如果是增大，则会复制原始数据到新的空间，新增的部分初始值不确定。当`ptr==NULL`时，和`malloc`功能相同

成功，返回非空指针，失败，返回`NULL`

`void free(void *ptr)`

函数setjmp和longjmp

goto只会实现一个函数体内的跳转，setjmp和longjmp可以实现跨函数的跳转

```
#include<setjmp.h>

int setjmp(jmp_buf env)   //直接调用，返回0,从longjmp返回，则为非0
        env参数里面保存的是当前的栈帧的所有信息，用来给longjmp使用

void longjmp(jmp_buf, int val)
        将栈帧恢复到jmp_buf保存的栈帧状态，并交付给setjmp一个返回值val
```

再调用lomgjmp后，数据是否回滚到调用setjmp之前的状态，是不确定的，一般来说，存储在存储器内的数据会保持在longjmp时的状态，存储在寄存器里面的数据会回滚到调用setjmp时的状态。

进程资源限制信息

```
#include<sys/resource.h>

int getrlimit(int resource, struct rlimit *rlptr)

int setrlimit(int resource, const struct rlimit *rlptr)
```

成功，返回0,出错，返回非0

resource代表资源编号

struct rlimit 是资源的软限制和硬限制的结构体

```
struct rlimit {
    rlim_t rlim_cur;  /* Soft limit */
    rlim_t rlim_max;  /* Hard limit (ceiling for rlim_cur) */
};
```

更改资源限制的规则：

1．任何一个进程可以在满足软限制值小于等于硬限制值的情况下，修改软限制值

2．任何一个进程可以保证硬限制值大于等于软限制的情况下，降低硬限制值

3．只有超级用户可以增加硬限制值

4．资源限制修改影响到调用进程并被子进程继承