进程控制

进程相关知识

- 进程从可执行目标代码开始其生命周期。这些目标代码具有内核能够解析的可执行格式（Linux下最常用的格式是ELF），且可以由机器执行。可执行格式代码包含有元数据，多个代码和数据段。“段”是加载到线性内存块的线性目标代码块。所有字节内的字节将一视同仁，赋予相同的权限，一般也用于同样的目的。
- 最重要和通用的段莫过于代码段、数据段和bss段。代码段包含可执行代码和只读数据，如常量，经常标记为只读和可执行。数据段包含已初始化的数据，如定义了值的C变量，通常标记为可读写的。bss段包含未初始化的全局数据，因为C标准规定C变量的默认值全为0，因此没有必要在磁盘上的目标代码中保存这些0。相反，目标代码可以简单的列举bss段中未初始化的变量，内核将映射0页面（全0的内存页）到那个加载进内存的段，为了优化性能人们设计了bss段。其它ELF中可执行的通用段都是绝对地址段（包含不可再定位符号）和未定义段（容器）。
- 进程是一种虚拟的抽象。Linux内核支持抢占式多任务和虚拟内存，它给进程提供了虚拟处理器和内存虚拟视图。从进程的视角看，系统完全由该进程控制。也就是说，尽管给定的进程和其它的进程共同调度，但是看起来好像他在独立控制整个系统。系统将无缝透明的重新进行进程调度，将系统的处理器和所有进程共享，而进程不会感到区别。类似的，每一个进程获得一个独立的线性地址空间，就像它独立控制整个系统内存。通过虚拟内存和分页调度，内核允许多个进程共存在系统上，每个进程都有自己的地址空间。内核通过现代处理器的硬件支持来管理这种虚拟化，它使得操作系统能够并发管理多个独立的进程。
- 进程的内存映象

6．代码段：可执行代码，只读。如图 1所示

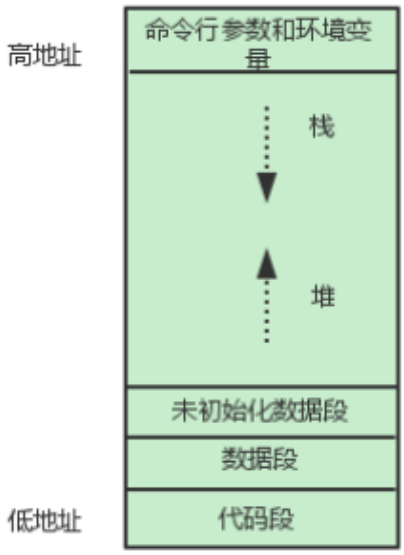

图 1构造图

可执行程序和内存映像的区别在于：

可执行程序位于磁盘中而内存映像位于内存中；

可执行程序没有堆栈，因为程序被加载到内存中才会分配堆栈；

可执行程序虽然也有未初始化数据段但它并不被存储于硬盘中的可执行文件；

可执行程序是静态的、不变的，而内存映像随着程序的执行是动态变化的；

- 进程PID
  - 每一个进程都由一个唯一的标识符表示，即进程ID，简称pid。系统保证在某个时刻每个pid都是唯一的。
  - 空闲进程（idle process）--当没有其它进程在运行时，内核所运行的进程--它的pid是0。在启动后，内核运行的第一个进程称为init进程，它的pid是1。一般来说,Linux中init进程就是init程序。
  - 创建新进程的那个进程被称为父进程，而新进程被称为子进程。每个进程都是由其它进程创建的（除了init进程），因此，每个子进程都有一个父进程。这种关系保存在每个进程的父进程ID（ppid）中。每个进程都被一个用户和组所拥有。这种从属关系是用来实现访问控制的。对于内核来说，用户和组仅仅是一些整数值。
  - pid_t 编程时，进程ID是由pid_t这种数据类型来表示的，它定义在<sys/types.h>中。

fork()

- 一般情况下，函数最多有一个返回值，但fork函数非常特殊，它有两个返回值，即调用一次返回两次。成功调用fork()会创建一个新的进程，它几乎与调用fork()的进程一模一样。这两个进程都会继续运行，调用者从fork()返回后，就好像没有特别的事情发生过。父子进程在调用fork函数的地方分开，在子进程中，fork调用返回0,在父进程中`fork`返回子进程的`pid`。以返回值来区分父子进程。

  进程创建失败返回-1。同时设置`errno`的值。

- 父子进程区别：

  1. 显然子进程的`pid`是新分配的，它是与父进程不同的。
  2. 子进程的`ppid`会设置为父进程的`pid`。
  3. 子进程的资源统计信息会清零。
  4. 任何挂起的信号会清楚，也不会被子进程继承。
  5. 任何文件锁都不会被子进程所继承。

当父进程先于子进程结束时，子进程会被`init`进程领养

example in Linux

```
// forktest.c
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>

int main()
{
    pid_t pid = fork();

    if (pid >0)
    {
        printf("pid of parent = %d\n", pid);
    }
    else if (pid == 0)
    {
        printf("I'm the son process and I will sleep 1s.\n");
        printf("pid of son = %d\n", pid);
        sleep(1);
        printf("son: I wake !\n");

        return 0;
    }
    else
    {
        perror("fork error");
    }

    printf("I'm parent and I had done my job and I need sleep 2s.\n");
    sleep(2);
    printf("parent: I wake! \n");

    return 0;
}
```

```
dyl@359f21020dd7:~/linux_lesson/process$ ./forktest
pid of parent = 12578
I'm parent and I had done my job and I need sleep 2s.
I'm the son process and I will sleep 1s.
pid of son = 0
son: I wake !
parent: I wake!
```

exec()

- 在UNIX中，载入内存并执行程序映像的操作与创建一个新进程的操作是分离的。**Unix有一些列系统调用可以将二进制文件的程序映像载入内存，替换原先进程的地址空间，并开始运行它。**这个过程称为运行一个新的程序，而相应的系统调用称为exec系统调用。
- exec函数族

```c
#include <unistd.h>
extern char **environ;
// exec函数族的函数执行成功后不会返回，调用失败时，会设置errno并返回-1，然后从原程序的调用点接着往下执行。
int execl(const char *path, const char *arg, ...);
int execlp(const char *file, const char *arg, ...);
int execle(const char *path, const char *arg,..., char * const envp[]);
int execv(const char *path, char *const argv[]);
int execvp(const char *file, char *const argv[]);
int execvpe(const char *file, char *const argv[],char *const envp[]);
/*
path：可执行文件的路径名字
arg：可执行程序所带的参数，第一个参数为可执行文件名字，没有带路径且arg必须以NULL结束
file：如果参数file中包含/，则就将其视为路径名，否则就按 PATH环境变量，在它所指定的各目录中搜寻可执行文件。
*/

/*
exec族函数参数极难记忆和分辨，函数名中的字符会给我们一些帮助：
l ： 使用参数列表
p：使用文件名，并从PATH环境进行寻找可执行文件
v：应先构造一个指向各参数的指针数组，然后将该数组的地址作为这些函数的参数。
e：多了envp[]数组，使用新的环境变量代替调用进程的环境变量
*/
```

以`execl`为例

```c
// execltest.c
#include <unistd.h>
#include <stdio.h>

int main()
{
    printf("before execl\n");

    if(execl("./echoarg","echoarg","abc",NULL) == -1)
    {
```

```
        printf("execl failed!\n");
    }

    printf("after execl\n");

    return 0;
}
```

```
// echoagr.c
#include <stdio.h>

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

```
dyl@359f21020dd7:~/linux_lesson/process$ ./execltest
before execl
argv[0]: echoarg
argv[1]: abc
```

以`execlp`为例

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
//函数原型：int execlp(const char *file, const char *arg, ...);
int main(void)
{
    printf("before execlp****\n");

    if(execlp("ps","ps","aux",NULL) == -1)
    {
        printf("execlp failed!\n");
    }

    printf("after execlp*****\n");

    return 0;
}
```

```
dyl@359f21020dd7:~/linux_lesson/process$ ./execlptest
before execlp****
F S   UID     PID    PPID  C PRI  NI ADDR SZ WCHAN  TTY          TIME CMD
4 S  1000   12497   12496  0  80   0 -  1090 do_wai pts/2    00:00:00 bash
4 R  1000   12642   12497  0  80   0 -  1454 -      pts/2    00:00:00 ps
dyl@359f21020dd7:~/linux_lesson/process$
```

environ 变量

environ变量存的是当前系统的环境变量，与在shell中执行`env`得到的结果是一样的

```
#include <stdio.h>
extern char ** environ;

int main()
{
    printf("environment:\n");

    for (int i = 0; environ[i] != NULL; i ++)
    {
        printf("%s\n", environ[i]);
    }

    return 0;
}
```

```
dyl@359f21020dd7:~/linux_lesson/process$ ./environ
environment:
SHELL=/bin/bash
TMUX=/tmp/tmux-1000/default,12496,0
JAVA_HOME=/usr/lib/jvm/java-8-openjdk-amd64
PWD=/home/dyl/linux_lesson/process
LOGNAME=dyl
MOTD_SHOWN=pam
HOME=/home/dyl
......
```

终止进程

- POSIX和C89都定义了终止当前进程的标准函数：`stdlib.h`里的`void exit(int status)`
- status参数用来标示进程退出的状态。其它进程（父进程）可以检测这个值。`EXIT_SUCCESS`和`EXIT_FAILURE`两个宏分别表示成功和失败，而且是可移植的。

等待进程 wait()

- 当一个进程终止时，内核会传递SIGCHLD信号给它的父进程。默认情况下，此信号会被忽略。父进程因此不会采取任何行动。然而，进程可经`signal()`或`sigaction()`系统调用选择处理此信号
- 若一个子进程先于父进程结束，则子进程会处于僵死状态，只保留最小的骨架：一些包含有用信息的内核数据结构。等待父进程来打听它的状态，等父进程获取了它的状态，子进程才会真正的退出

```
Linux内核提供了若干接口用于取得关于已终止子进程的信息。这些接口中最简单的就是POSIX所定义的
wait():
#include <sys/types.h>
#include <sys/wait.h>

pid_t wait(int *status);
```

调用wait()会返回一个已终止子进程的pid，发生错误时会返回-1.如果没有子进程终止，调用者会被阻塞，直到一个子进程终止。如果子进程终止了，它就会立刻返回。因此，调用wait()会响应一个子进程的死亡消息。
但如果收到了SIGCHILD信号，会立刻返回而不会阻塞在wait调用

如果指针status不是NULL，那么它包含了子进程的一些额外信息。
/*
int WIFEXITED(status) //如果子进程正常结束则为非0 值.

int WEXITSTATUS(status) //取得子进程exit()返回的结束代码，一般会先用WIFEXITED 来判断是否正常结束才能使用此宏.

int WIFSIGNALED(status) //如果子进程是因为信号而结束则此宏值为真

int WTERMSIG(status) //取得子进程因信号而中止的信号代码，一般会先用WIFSIGNALED来判断后才使用此宏.

int WIFSTOPPED(status) //如果子进程处于暂停执行情况则此宏值为真. 一般只有使用WUNTRACED时才会有此情况.

int WSTOPSIG(status) //取得引发子进程暂停的信号代码，一般会先用WIFSTOPPED 来判断后才使用此宏.
*/

waittest.c

```c
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>

int main()
{
    pid_t pid;
    pid = fork();

    if (pid == 0)
    {
        // in son process
        printf("I'm the son process.\n");
        sleep(1);
        return 100;
    }

    int status;
    pid = wait(&status);

    if (pid == -1)
    {
        perror("wait");
        printf("pid = %d\n", pid);
    }

    if (WIFEXITED(status))
    {
        printf("Normal termination with exit status = %d\n",
WEXITSTATUS(status));
```

```
    }

    return 0;
}
```

```
dyl@359f21020dd7:~/linux_lesson/process$ ./waittest
I'm the son process.
Normal termination with exit status = 100
```

waitpid(), waitid()

```
#include <sys/types.h>
#include <sys/wait.h>
pid_t waitpid(pid_t pid,int *status,int options);
int waitid(idtype_t idtype, id_t id, siginfo_t *infop, int options);
```

- `waitpid()`
  - 如果在调用`waitpid()`函数时，当指定等待的子进程已经停止运行或结束了，则`waitpid()`会立即返回；但是如果子进程还没有停止运行或结束，则调用`waitpid()`函数的父进程则会被阻塞，暂停运行。
  - 参数
    1. `pid_t pid`

参数值	说明
pid<-1	等待进程组号为pid绝对值的任何子进程。
pid=-1	等待任何子进程，此时的waitpid()函数就退化成了普通的wait()函数。
pid=0	等待进程组号与目前进程相同的任何子进程，也就是说任何和调用waitpid()函数的进程在同一个进程组的进程。
pid>0	等待进程号为pid的子进程。

    2. `int *status`

       这个参数将保存子进程的状态信息，有了这个信息父进程就可以了解子进程为什么会推出，是正常推出还是出了什么错误。如果status不是空指针，则状态信息将被写入指向的位置。当然，如果不关心子进程为什么推出的话，也可以传入空指针。

       Linux提供了一些宏，帮助解析这个status：

宏	说明
WIFEXITED(status)	如果子进程正常结束，它就返回真；否则返回假。
WEXITSTATUS(status)	如果WIFEXITED(status)为真，则可以用该宏取得子进程exit()返回的结束代码。
WIFSIGNALED(status)	如果子进程因为一个未捕获的信号而终止，它就返回真；否则返回假。
WTERMSIG(status)	如果WIFSIGNALED(status)为真，则可以用该宏获得导致子进程终止的信号代码。
WIFSTOPPED(status)	如果当前子进程被暂停了，则返回真；否则返回假。
WSTOPSIG(status)	如果WIFSTOPPED(status)为真，则可以使用该宏获得导致子进程暂停的信号代码。

    3. `int options`

       参数options提供了一些另外的选项来控制`waitpid()`函数的行为。如果不想使用这些选项，则可以把这个参数设为0。

参数	说明
WNOHANG	如果pid指定的子进程没有结束，则waitpid()函数立即返回0，而不是阻塞在这个函数上等待；如果结束了，则返回该子进程的进程号。
WUNTRACED	如果子进程进入暂停状态，则马上返回。

4. 如果像这样调用`waitpid`函数：`waitpid(-1, status, 0)`，这此时`waitpid()`函数就完全退化成了wait()函数。

- **waitid()**

```
int waitid(idtype_t idtype,id_t id,siginfo_t*infop,int options);
```

  - `idtype`和`id`指定等待的子进程或多个子进程
  - `idtype=P_PID`:等待指定进程号的子进程
  - `idtype=P_PGID`:等待指定进程组号的所有子进程
  - `idtype==P_ALL`:等待所有子进程

僵死进程

- 一个进程已经终止了，但是它的父进程还在等待获得它的状态，那么这个进程就叫做僵死进程。僵死进程还会消耗一些系统资源，尽管这些资源很少--仅仅能够描述进程曾经的状态。这些保留的资源主要是为了在父进程查询子进程的状态时提供相应的信息。一旦父进程得到了想要的信息，内核就会清楚这些信息，僵死的进程就不存在了 。

```
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <unistd.h>

int main()
{
    pid_t pid;

    pid = fork();

    if (pid < 0)
    {
        perror("fork error: ");
        exit(1);
    }
    if (pid == 0)
    {
        printf("I'm the child process.\n");
        printf("pid: %d\tppid:%d\tgpid:%d\n",getpid(),getppid(),getpgrp());
        printf("I will sleep 5 seconds.\n");

        sleep(5);

        printf("pid: %d\tppid:%d\tgpid:%d\n",getpid(),getppid(),getpgrp());
        printf("child process is exited\n");
    }
    else
    {
        printf("I am father process.\n");
        sleep(1);
        printf("father process is exited.\n");
        printf("pid: %d\tppid:%d\tgpid:%d\n",getpid(),getppid(),getpgrp());
    }

    return 0;
```

```
}
```

```
dyl@359f21020dd7:~/linux_lesson/process$ ./dyingfork
I am father process.
I'm the child process.
pid: 12901      ppid:12900      gpid:12900
I will sleep 5 seconds.
father process is exited.
pid: 12900      ppid:12497      gpid:12900
dyl@359f21020dd7:~/linux_lesson/process$ pid: 12901     ppid:1  gpid:12900
child process is exited
```

孤儿进程

- 如果父进程在子进程结束之前结束或者说如果父进程还没有机会等待僵死的子进程，它就结束了，出现这种情况的子进程被称为孤儿进程
- 孤儿进程在Linux中，会被`init`进程领养

```

```

线程的创建与运行

相关知识

- 一个进程包含一个或多个执行线程（通常只叫线程），线程是进程中的活动单位。线程是一种抽象，它负责执行代码和维护进程的运行状态。
- 线程包括栈（如同在非线程系统上的进程栈，主要用于存储局部变量）、处理器状态、目标代码的当前位置（通常是处理器的指令指针）。进程剩下的部分由所有线程共享。
- 线 程是进程中的运行单元，所有的进程都至少有一个线程。每一个线程都独自占有一个虚拟处理器：独自的寄存器组，指令计数器和处理器状态。
- **同一个进程内的所有线程共享同一地址空间（也就是同样的动态内存，映射文件，目标代码等等），打开的文件 队列和其它内核资源。**
- Linux内核对线程的观点独特而有趣。本质上，内核没有线程这个概念。广义上来说，两个无关进程和一个进程内的两个线程没有区别。内核把线程简化为共享资源的进程，也就是说，内核把一个进程中的两个线程，简化为共享一系列内核资源（地址空间，打开的文件列表等）的两个不同进程。

线程的创建

想要使用`pthread`库在编译时，需要加上链接库`-lpthread`

```
#include <pthread.h>

int pthread_create(pthread_t *thread, const pthread_attr_t *attr,
                          void *(*start_routine) (void *), void *arg);

 Compile and link with -pthread.
```

thread是一个指针，当线程创建成功时，返回线程ID。
attr：参数用来指定线程属性，一般用NULL。
start_routine 是一个函数指针，指向线程创建后要调用的函数。这个被线程调用的函数也称为线程函数。
arg：该参数指向传递给线程函数的参数。

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int * thread(void * args)
{
    pthread_t newthid;
    newthid = pthread_self();
    printf("this is a new thread , thread id is %lu\n", newthid);

    return 0;
}

void * arg_thread(void * t)
{
    int i;
    long  tid;
    tid = (long )t;

    pthread_t newthid = pthread_self();
    printf("this is a new thread2 , thread2 id is %lu\n", newthid);

    sleep(1);

    printf("arg is %ld\n", tid);
    pthread_exit(NULL);
}

int main()
{
    pthread_t thid;
    pthread_t thid2;

    printf("main thread , ID is %lu\n",pthread_self()); //打印主线程的ID
    if(pthread_create(&thid,NULL,(void *)thread,NULL) != 0)
    {
        printf("thread create failed\n");
        exit(1);
    }

    long i = 100;
```

```
    if(pthread_create(&thid2,NULL,arg_thread,(void *) i) != 0)
    {
        printf("thread create failed\n");
        exit(1);
    }

    void *ret_val, *ret_val2;
    if (pthread_join(thid, &ret_val) != 0)
        printf("thread not 0 return.\n");
    else
        printf("thread id %lu return code %ld\n", thid, (long)ret_val);

    if (pthread_join(thid2, &ret_val2) != 0)
        printf("thread not 0 return.\n");
    else
        printf("thread id %lu return code %ld\n", thid2, (long)ret_val2);

    return 0;
}
```

```
dyl@359f21020dd7:~/linux_lesson/thread$ ./createthread
main thread , ID is 140676300842816
this is a new thread2 , thread2 id is 140676292445952
this is a new thread , thread id is 140676300838656
thread id 140676300838656 return code 0
arg is 100
thread id 140676292445952 return code 0
```

线程退出

- Linux下有两种方式可以使线程终止。第一种是通过return从线程函数返回，第二种是通过调用函数`pthread_exit()`使线程退出。
- **线程终止最重要的问题是资源释放的问题，特别是一些临界资源。**临界资源在一段时间内只能被一个线程所持有，当线程要使用临界资源时需提出请求，如果该资源未被使用则申请成功，否则等待。临界资源使用完毕后要释放以便其它线程可以使用。
- Linux提供了这一对函数用于自动释放资源：`pthread_cleanup_push()`,`pthread_cleanup_pop()`,这对函数必须成对出现。

```
#include <pthread.h>

void pthread_exit(void *retval);
void pthread_cleanup_push(void (*routine)(void *), void *arg);
void pthread_cleanup_pop(int execute);
```

- 一般情况下，进程中各个线程的运行时相互独立的，线程的终止并不会相互通知，也不会影响其它线程，终止的线程所占用的资源不会随着线程的终止而归还系统，而是仍为线程所在的进程持有。正如进程之间可以通过`wait()`调用来等待其它进程结束一样，线程也有类似的函数：`pthread_join()`函数

```c
#include <pthread.h>

int pthread_join(pthread_t thread, void **retval);
int pthread_detach(pthread_t thread);
```

线程数据共享

- 在多线程环境下，进程内的所有线程共享进程的数据空间，**因此全局变量为所有线程共有。**
- 在程序设计中有时需要保存线程自己的全局变量，这种特殊的变量仅在某个线程内部有效。如常见的变量`errno`，它返回标准的出错代码。但它又不能作为一个全局变量，否则在一个线程里输出的很可能是另一个线程的出错信息。这个问题可以通过创建线程的私有数据（TSD）来解决。在线程内部，**线程私有数据可以被各个函数访问，但它对其它线程是屏蔽的。**
- `pthread_getpecific`和`pthread_setspecific`提供了在同一个线程中不同函数间共享数据即线程存储的一种方法。

```c
/*
#include <pthread.h>

int pthread_key_create(pthread_key_t *key, void (*destructor)(void*));
void *pthread_getspecific(pthread_key_t key);
int pthread_setspecific(pthread_key_t key, const void *value);
int pthread_key_delete(pthread_key_t key);
*/
1.调用pthread_key_create()来创建一个类型为pthread_key_t类型的变量
    该函数有两个参数，第一个参数就是声明的pthread_key_t变量，第二个参数是一个清理函数，用来在线程释放该线程存储的时候被调用。该函数指针可以设成NULL，这样系统将调用默认的清理函数。
2.调用pthread_setspcific()
    当线程中需要存储特殊值的时候调用该函数，该函数有两个参数，第一个为前面声明的pthread_key_t变量，第二个为void*变量，用来存储任何类型的值。
3.如果需要取出所存储的值，调用pthread_getspecific()
    该函数的参数为前面提到的pthread_key_t变量，该函数返回void*类型的值。

#include<stdio.h>
#include<pthread.h>
#include<string.h>
pthread_key_t p_key;

void func1()
{
        int *tmp = (int*)pthread_getspecific(p_key);//同一线程内的各个函数间共享数据。
        printf("%d is runing in %s\n",*tmp,__func__);

}
void *thread_func(void *args)
{

        pthread_setspecific(p_key,args);

        int *tmp = (int*)pthread_getspecific(p_key);//获得线程的私有空间
        printf("%d is runing in %s\n",*tmp,__func__);

        *tmp = (*tmp)*100;//修改私有变量的值
```

```
        func1();

        return (void*)0;
}
int main()
{
        pthread_t pa, pb;
        int a=1;
        int b=2;
        pthread_key_create(&p_key,NULL);
        pthread_create(&pa, NULL,thread_func,&a);
        pthread_create(&pb, NULL,thread_func,&b);
        pthread_join(pa, NULL);
        pthread_join(pb, NULL);
        return 0;
}
```

线程同步

- 线程的主要优势在于，能够通过全局变量来共享信息。而进程不能
- **互斥量**

  互斥量通过锁机制来实现线程间的同步。在同一时刻它通常只允许一个线程执行一个关键部分的代码。

  ```
  pthread_mutex_init();                  //初始化一个互斥锁
  pthread_mutex_destroy();              //注销一个互斥锁
  pthread_mutex_lock();                 //加锁，如果不成功，阻塞等待
  pthread_mutex_unlock();                  //解锁
  pthread_mutex_trylock();                 //测试加锁，如果不成功立即返回
  ```

  - 初始化mutex

    ```
    pthread_mutex_t *mutex = PTHREAD_MUTEX_INITIALIZER; //静态初始化

    int pthread_mutex_init(pthread_mutex_t *restrict mutex, //推荐
    const pthread_mutexattr_t *restrict attr);

    函数中的参数attr表示互斥量的属性，如果为NULL则使用默认属性。
    ```

  - 给互斥量加锁

    ```
    int pthread_mutex_lock(pthread_mutex_t *mutex);
    int pthread_mutex_trylock(pthread_mutex_t *mutex);

    // 用pthread_mutex_lock函数加锁时，如果mutex已经被锁住，当前尝试加锁的线程就会阻塞，直到互斥量被加锁线程释放。当pthread_mutex_lock函数返回时，说明加锁成功。
    // 调用pthread_mutex_trylock的函数稍有不同，如果mutex已经被锁住，则调用线程立即返回，而不是阻塞等待，并得到错误码EBUSY。
    ```

  - 解锁

```
int pthread_mutex_unlock(pthread_mutex_t *mutex);
```

解锁时，要满足两个条件：一是互斥量必须处于加锁状态，二是调用本函数的线程必须是给互斥量加锁的线程。

- 清除

```
int pthread_mutex_destroy(pthread_mutex_t *mutex);
```

清除一个互斥量意味着释放它所占用的资源。清除互斥量时要求互斥量处于开放状态。在linux中，互斥量并不占用内存。

- 示例

```
#include <stdio.h>
#include <unistd.h>
#include <pthread.h>
#include <string.h>

int globalnum = 0;
pthread_mutex_t mutex;

void add(void * args)
{
    pthread_mutex_lock(&mutex);
    globalnum ++;
    pthread_mutex_unlock(&mutex);
}

int get()
{
    int temp;
    pthread_mutex_lock(&mutex);
    temp = globalnum;
    pthread_mutex_unlock(&mutex);
    return temp;
}

void readnum(void * args)
{
    printf("Now the globalnum is %d\n", get());
}

int main()
{
    pthread_t threads[1000];
    pthread_mutex_init(&mutex, NULL);
    printf("main begins\n");

    for (int i = 0; i < 1000; i ++)
    {
        if (i == 500 || i == 700)
        {
            pthread_create(&threads[i], NULL, (void *)readnum, NULL);
            continue;
        }
```

```
        pthread_create(&threads[i], NULL, (void *)add, NULL);
    }

    for (int i = 0; i < 1000; i++)
    {
        pthread_join(threads[i], NULL);
    }

    readnum(NULL);
    pthread_mutex_destroy(&mutex);
    printf("main exit.");

    return 0;
}
```

```
说明：中间有两个线程去读globalnum的值，因为是多线程，所以值不一定是499，与699，但globalnum的值最后一定会是998，因为有998个线程对它++，如果不加锁，那么结果很有可能会小于998

dyl@359f21020dd7:~/linux_lesson/thread$ ./mutextest
main begins
Now the globalnum is 499
Now the globalnum is 676
Now the globalnum is 998
main exit.
```

- **条件变量**
  - 条件变量是利用线程间共享的全局变量进行同步的一种机制。**条件变量宏观上类似if语句，符合条件就能执行某段程序，否则只能等待条件成立。**
  - 使用条件变量主要包括两个动作：一个等待使用资源的线程等待“条件变量被设置为真”；另一个线程在使用完资源后“设置条件为真”，这样就可以保证线程间的同步了。这样就存在一个关键问题，**就是要保证条件变量能被正确的修改，条件变量要受到特殊的保护，实际使用中互斥量扮演着这样一个保护者的角色。**
  - 操作条件变量的相关函数

```
pthread_cond_init();//初始化条件变量
pthread_cond_wait();              //基于条件变量阻塞，无条件等待
pthread_cond_timedwait();  //阻塞直到指定事件发生，记时等待
pthread_cond_signal();      //解除特定线程的阻塞，存在多个等待线程时按入队顺序激活其中一个
pthread_cond_broadcast();      //解除所有线程的阻塞
pthread_cond_destroy();         //清除条件变量
```

  - 初始化

```
pthread_cond_t cond = PTHREAD_COND_INITIALIZER; //静态初始化

int pthread_cond_init(pthread_cond_t *restrict cond,const
pthread_condattr_t *restrict attr);
// attr参数是条件变量的属性，由于其并没有实现，值通常是NULL。
```

  - 等待条件成立

```
// pthread_cond_wait，pthread_cond_timedwait用于等待条件成立。

int pthread_cond_timedwait(pthread_cond_t *restrict cond,
pthread_mutex_t *
restrict mutex,const struct timespec *restrict abstime);

int pthread_cond_wait(pthread_cond_t *restrict cond,pthread_mutex_t *
restrict mutex);
```

`pthread_cond_wait`函数释放由mutex指向的互斥量，同时使当前线程关于`cond`指向的条件变量阻塞，直到条件被信号唤醒。

通常条件表达式在互斥锁的保护下求值，如果条件表达式为假，那么线程基于条件变量阻塞。

- 激活

  线程被条件变量阻塞后，可通过函数`pthread_cond_broadcast`和`pthread_cond_signal`激活

```
int pthread_cond_broadcast(pthread_cond_t *cond); // 通知所有
int pthread_cond_signal(pthread_cond_t *cond); // 通知阻塞队列靠前的那个线程
```

- 销毁

```
int pthread_cond_destroy(pthread_cond_t *cond);
```

  当一个条件变量不再使用时，需要将其清除。

  注意只有在没有线程等待该条件变量的时候才能清除这个条件变量。否则返回EBUSY。

- 示例

```
#include<stdio.h>
#include<unistd.h>
#include<pthread.h>
#include<string.h>

pthread_mutex_t mutex;
pthread_cond_t cond;
/*  使用pthread_cleanup_push((void *)pthread_mutex_unlock,&mutex);
    和thread_cleanup_pop(0); 这里 0 的话，是不会执行push绑定的函数的
    push进去的函数可能执行的点：
        1，显示的调用pthread_exit();
        2，在cancel点线程被cancel。
        3，pthread_cleanup_pop()的参数不为0时。

    这样保证了mutex的释放，相当于 try...catch 语句了
*/
void thread1()
{
    pthread_cleanup_push((void *)pthread_mutex_unlock,&mutex);
    while(1)
    {
        printf("thread1 is running\n");
        pthread_mutex_lock(&mutex);
        pthread_cond_wait(&cond,&mutex);
```

```
        printf("thread1 applied the condition\n");
        pthread_mutex_unlock(&mutex);
        sleep(2);
    }
    pthread_cleanup_pop(0);
}

void thread2()
{
    pthread_cleanup_push((void *)pthread_mutex_unlock,&mutex);
    while(1)
    {
        printf("thread2 is running\n");
        pthread_mutex_lock(&mutex);
        pthread_cond_wait(&cond,&mutex);
        printf("thread2 applied the condition\n");
        pthread_mutex_unlock(&mutex);
        sleep(1);
    }
    pthread_cleanup_pop(0);
}

int main()
{
    pthread_t thid1,thid2;
    pthread_mutex_init(&mutex,NULL);
    pthread_cond_init(&cond,NULL);

    printf("main thread begins\n");

    pthread_create(&thid1,NULL,(void *)thread1,NULL);
    pthread_create(&thid2,NULL,(void *)thread2,NULL);

    do{
        pthread_cond_signal(&cond);
    }while(1);

    sleep(10);
    printf("main thread exit\n");
    pthread_exit(0);
}
```

后续的输出一定是 2次thread2，1次thread1，因为thread2睡2秒，thread1睡1秒

```
dyl@359f21020dd7:~/linux_lesson/thread$ ./condition
main thread begins
thread2 is running
thread1 is running
thread2 applied the condition
thread1 applied the condition
thread2 is running
thread2 applied the condition
thread1 is running
thread1 applied the condition
thread2 is running
thread2 applied the condition
thread2 is running
```

```
thread2 applied the condition
thread1 is running
thread1 applied the condition
thread2 is running
thread2 applied the condition
thread2 is running
thread2 applied the condition
```

信号

- 信号是一种单向异步通知机制，信号可能是从**内核发送到进程**，也可能是从**进程发送到进程**，或者**进程给自己**。信号一般用于通知进程发生某些事件。**信号是一种软件中断。信号是异步的**
- 信号有一个非常明确的生命周期。首先，产生信号，然后内核存储信号直到可以发送它。最后一旦有空闲，内核会适当的处理信号。
  - 忽略信号

    不采取任何操作。但是有两种信号不能被忽略：SIGKILL和SIGSTOP。
  - 捕获并处理信号

    内核会暂停该进程正在执行的代码，并跳转到先前注册过的函数。接下来进程会执行这个函数。一旦进程从该函数返回，它会跳回到捕获信号的地方继续执行。

    `SIGINT`和`SIGTERM`是两个常见的可捕获的信号。
  - 执行默认操作

    该操作取决于被发送的信号。默认操作通常是终止进程。

信号处理

- Linux系统中对信号的处理主要由`signal`和`sigaction`函数来完成。

```
#include <signal.h>
void (*signal(int sig, void (*func)(int)))(int);

信号处理函数必须是无返回值的，并接收一个int类型的信号参数
```

- 示例

```
#include <stdlib.h>
#include <stdio.h>
#include <signal.h>

void my_handler(int singno)
{
    printf("recv my signal %d\n", singno);
    printf("now I am exit.\n");
    exit(1);
}

int main()
{
```

```
    signal(SIGINT, my_handler); //安装信号处理函数

    while(1)
    {
        ;
    }

    return 0;
}
```

```
dyl@359f21020dd7:~/linux_lesson/signal$ ./my_signal
^Crecv my signal 2
now I am exit.
```

等待信号

- 出于调试和演示代码的目的，POSIX定义的pause()系统调用，它可以使进程睡眠，直到进程接收到处理或终止进程的信号

```
#include <unistd.h>

int pause(void);
```

- 当程序运行时，pause会使当前的进程挂起（进入睡眠状态），直到我们向该进程发送SIGINT中断信号，进程才会被唤醒，并处理信号，处理完信号后pause函数才返回，并继续运行该程序。**任何信号都可使pause唤醒**
- 示例

```
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <signal.h>

void my_handler(int singno)
{
    printf("recv my signal %d\n", singno);
}

int main()
{
    printf("main begining\n");

    if (signal(SIGINT, my_handler) == SIG_ERR)
    {
        fprintf(stderr, "can not handle SIGINT!\n");
        exit(EXIT_FAILURE);
    }//安装信号处理函数

    pause();

    printf("main ending\n");

    return 0;
```

```
}
```

```
Ctrl + C 发送 SIGINT

dyl@359f21020dd7:~/linux_lesson/signal$ ./my_signal
main begining
^Crecv my signal 2
main ending
```

发送信号

- 信号的发送主要由函数`kill、raise、sigqueue、alarm、setitmer`以及`abort`来完成

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

int kill(pid_t pid, int sig);

该函数的行为与第一个参数pid的取值有关，第二个参数sig表示信号编号。

1. 如果pid是正数，则发送信号sig给进程号为pid的进程；
2. 如果pid为0，sig被发送给调用进程的进程组中的每个进程。
3. 如果pid是-1，则把信号sig广播至除了init进程（1号）和自身以外的所有进程（有权限）。
4. 如果pid小于-1，则把信号sig发送给属于进程组-pid的所有进程；
5. 如果sig是0，则kill将执行正常的错误检查，但不发送信号，用来检查一个进程是否有权限向另一个进程发送信号。

函数执行成功返回0，当有错误发生时返回-1并设置errno。
```

- 示例 kill

```
#include<wait.h>
#include<stdio.h>
#include<unistd.h>
#include<signal.h>
#include<stdlib.h>

void main()
{
    id_t pid;
    int status;

    if((pid = fork()) < 0)//创建一个子进程
    {
        printf("fork() error!\n");
        exit(-1);
    }
    else if(pid == 0)//让子进程处于休眠状态
    {
        printf("this is the child process!!\n");
        sleep(1000);
    }
    else
    {
```

```
        sleep(1);
        int i = waitpid(pid, &status, WNOHANG);//跳用waitpid函数来返回子进程的状
态
        printf("%d\n",i);
        printf("the state is %d\n",status);
        kill(pid,SIGKILL);//跳用kill函数传递SIGKILL命令

        //kill(pid,SIGINT);//跳用kill函数传递SIGINT命令
        sleep(3);//等待kill执行完毕
        int j = waitpid(pid, &status, WNOHANG);//查看进程结束的返回状态!
        printf("%d\n",j);

        printf("the state is %d\n",status);
    }

}
```

```
this is the child process!!
0
the state is 1220059424
145668
the state is 9
```

- 给自己发送信号

raise()函数是一种简单的进程给自己发送信号的方法：

```
#include <signal.h>
int raise(int sig);
```

该调用：

```
raise(signo);
```

和下面的调用是等价的：

```
kill (getpid(),signo);
```