**12-Linux-API编程**

1. **进程线程**

1.1 **进程**

1.1.1 **进程规范**

进程退出

fork创建子进程退出，由父进程wait() 或者 waitpid()，不然会形成僵尸进程

父子进程

父进程杀死子进程，

waitpid(pid, &status, 0);

等待子进程结束防止子进程形成僵尸进程

fork后tcp连接归父子进程共享

1.2 **线程**

1.2.1 **线程规范**

1.2.1.1 **线程销毁**

线程属性值中有一个分离状态，什么是线程的分离状态呢？在任何一个时间点上，线程是可结合的（joinable），或者是分离的（detached）。一个可结合的线程能够被其他线程收回其资源和杀死；在被其他线程回收之前，它的存储器资源（如栈）是不释放的。相反，一个分离的线程是不能被其他线程回收或杀死的，它的存储器资源在它终止时由系统自动释放。

1、pthread\_join退出

线程退出，需要释放资源，不然一个线程最大创建数目三万左右（小于32767）

（1）线程默认创建是joinable状态

当线程函数自己返回退出时或pthread\_exit时都不会释放线程所占用堆栈和线程描述符（总计8k多）。只有当你调用了pthread\_join之后这些资源才会被释放。

（2）更多情况下，我们也不能一直等待线程退出，此时加二种方式可以使其像windows自动销毁退出线程资源

2、pthread\_detach(pthread\_self())

在线程函数中加pthread\_detach(pthread\_self())，将状态改为unjoinable状态。退出线程自动销毁线程占用资源。

pthread\_detach(threadid)函数的功能是使线程ID为threadid的线程处于分离状态，一旦线程处于分离状态，该线程终止时底层资源立即被回收；

3、pthread\_attr\_setdetachstate

也可创建时设置线程属性为PTHREAD\_CREATE\_DETACHED，线程退出后释放线程资源

（1）int pthread\_attr\_setdetachstate(pthread\_attr\_t \*attr, int detachstate)；

• attr：指向一个线程属性的指针。

• detachstate：如果值为 PTHREAD\_CREATE\_DETACHED，则表示线程是分离状态，如果值

为 PTHREAD\_CREATE\_JOINABLE 则表示线程是结合状态

如果想要获取某个线程的分离状态，那么可以通过 pthread\_attr\_getdetachstate() 函数获取：

（2）int pthread\_attr\_getdetachstate(const pthread\_attr\_t \*attr, int detachstate);

|  |
| --- |
| C++ pthread\_t t; pthread\_attr\_t a; //线程属性 pthread\_attr\_init(&a); //初始化线程属性 pthread\_attr\_setdetachstate(&a, PTHREAD\_CREATE\_DETACHED); //设置线程属性 pthread\_create( &t, &a, GetAndSaveAuthviewSDRStub, (void\*)lp); //建立线程  pthread\_attr\_destroy(&a) |

4、线程在特定的点退出

某些情况下，希望通过pthread\_cancel结束某个线程，但是被结束的线程必须在某一个点来进行退出操作，就需要用到pthread\_testcancel。pthread\_testcancel在指定点退出。

|  |
| --- |
| C++ void \*inotify\_read\_event(void \*arg) {     pthread\_setcancelstate(PTHREAD\_CANCEL\_ENABLE,NULL);     pthread\_setcanceltype(PTHREAD\_CANCEL\_DEFERRED,NULL);     struct inotify\_event \*event;     while(1)     {         pthread\_testcancel();         printf("test");         pthread\_testcancel();         inotify\_all\_observer(event);     } } pthread\_t tid; pthread\_create(&tid, 0, (void \*)pthreadFunc, NULL); pthread\_cancel(tid); pthread\_join(tid,NULL); |

1.3  **线程锁**

1.3.1 **线程锁**

**锁的创建**

方式1

锁可以被动态或静态创建，可以用宏PTHREAD\_MUTEX\_INITIALIZER来静态的初始化锁：

|  |
| --- |
| C++ pthread\_mutex\_t mutex = PTHREAD\_MUTEX\_INITIALIZER; |

方式2，这种方式需要

锁可以用pthread\_mutex\_init函数动态的创建，函数原型如下：

|  |
| --- |
| C++ int pthread\_mutex\_init(pthread\_mutex\_t \*mutex, const pthread\_mutexattr\_t \* attr) |

**锁的释放**

调用pthread\_mutex\_destory之后，可以释放锁占用的资源，但这有一个前提上锁当前是没有被锁的状态。

**锁操作**

加锁 pthread\_mutex\_lock()

解锁 pthread\_mutex\_unlock()

测试加锁 pthread\_mutex\_trylock()

pthread\_mutex\_trylock()语义与pthread\_mutex\_lock()类似，不同的是在锁已经被占据时返回EBUSY而不是挂起等待

**锁的属性**

设置其属性；

pthread\_mutexattr\_init(pthread\_mutexattr\_t \*mattr); 互斥锁属性初始化

int pthread\_mutexattr\_setpshared(pthread\_mutexattr\_t \*mattr, int pshared) 设置

pthread\_mutexattr\_getshared(pthread\_mutexattr\_t \*mattr,int \*pshared) 获取

参数

PTHREAD\_PROCESS\_SHARE 锁可以在不同进程间的线程共享

PTHREAD\_PROCESS\_PRIVATE 进程内使用锁，默认

**设置互斥锁的类型**

pthread\_mutexattr\_settype(pthread\_mutexattr\_t \*attr , int type) 设置线程互斥锁属性中的锁类型

pthread\_mutexattr\_gettype(pthread\_mutexattr\_t \*attr , int \*type) 获取线程互斥锁属性中的锁类型

取值：

PTHREAD\_MUTEX\_TIMED\_NP 这是缺省值，也就是普通锁。当一个线程加锁以后，其余请求锁的线程将形成一个等待队列，并在解锁后按优先级获得锁。这种锁策略保证了资源分配的公平性。

PTHREAD\_MUTEX\_RECURSIVE\_NP 嵌套锁，允许同一个线程对同一个锁成功获得多次，并通过多次unlock解锁。如果是不同线程请求，则在加锁线程解锁时重新竞争。

PTHREAD\_MUTEX\_ERRORCHECK\_NP 检错锁，如果同一个线程请求同一个锁，则返回EDEADLK，否则与PTHREAD\_MUTEX\_TIMED\_NP类型动作相同。这样就保证当不允许多次加锁时不会出现最简单情况下的死锁。

PTHREAD\_MUTEX\_ADAPTIVE\_NP 适应锁，动作最简单的锁类型，仅等待解锁后重新竞争。

1.4 **读写锁**

初始化一个读写锁 pthread\_rwlock\_init

读锁定读写锁 pthread\_rwlock\_rdlock

非阻塞读锁定 pthread\_rwlock\_tryrdlock

写锁定读写锁 pthread\_rwlock\_wrlock

非阻塞写锁定 pthread\_rwlock\_trywrlock

解锁读写锁 pthread\_rwlock\_unlock

释放读写锁 pthread\_rwlock\_destroy

读写锁是用来解决读者写者问题的，读操作可以共享，写操作是排他的，读可以有多个在读，写只有唯一个在写，同时写的时候不允许读。

具有强读者同步和强写者同步两种形式:

强读者同步：当写者没有进行写操作，读者就可以访问；

强写者同步：当所有写者都写完之后，才能进行读操作，读者需要最新的信息，一些事实性较高的系统可能会用到该所，比如定票之类的。

**初始化一个读写锁pthread\_rwlock\_init**

#include <pthread.h>

int pthread\_rwlock\_init(pthread\_rwlock\_t \*rwlock, \ const pthread\_rwlockattr\_t \*attr);

返回值：函数成功返回0；任何其他返回值都表示错误

初始化rwlock指定的读写锁，它的属性被参数attr指定。如果attr的值为NULL，初始化读写锁时使用缺省的读写锁属性。其效果和attr指向一个缺省的读写锁属性对象是一样的。

成功初始化后，读写锁的状态为非锁定的。

对一个已经初始化过的读写锁调用pthread\_rwlock\_init函数会产生不可预测的后果。使用一个未经初始化的读写锁也会产生不可预测的后果。

如果需要缺省属性的读写锁，可以用宏PTHREAD\_RWLOCK\_INITIALIZER初始化静态的读写锁变量。但是，静态初始化读写锁不进行错误检查。如：

pthread\_rwlock\_t rwlock = PTHREASD\_RWLOCK\_INITIALIZER;

获取读写锁的读锁操作：分为阻塞式获取和非阻塞式获取,如果读写锁由一个写者持有，则读线程会阻塞直至写入者释放读写锁。

**读锁定读写锁pthread\_rwlock\_rdlock**

#include <pthread.h>

int pthread\_rwlock\_rdlock(pthread\_rwlock\_t \*rwlock);

返回值：函数成功返回0；任何其他返回值都表示错误

函数在rwlock读写锁上进行读锁定。

如果一个线程写锁定了读写锁，调用pthread\_rwlock\_rdlock函数的线程将无法读锁定读写锁，并将被阻塞，直到线程可以读锁定这个读写锁为止。

如果一个线程写锁定了读写锁后又调用pthread\_rwlock\_rdlock函数来读锁定同一个读写锁，结果将无法预测。

标准中没有指定当没有线程写锁定这个读写锁，但有线程试图写锁定而被阻塞在这个读写锁上时，当前线程是否能读锁定读写锁。大多数线程库往往优先考虑激活试图写锁定而被阻塞的线程，这样做是为了防止试图写锁定而被阻塞的线程长时间得不到调度。

一个线程可能对一个读写锁进行了多次读锁定(即成功调用了pthread\_rwlock\_rdlock()函数多次)。如果是这样的，线程必须调用pthread\_rwlock\_unlock()函数对读写锁进行同样次数的解锁。

当一个读写锁被读锁定了，而一个线程阻塞在这个读写锁上时，如果这时来了一个信号，那么当线程从信号处理程序中返回时，将继续阻塞在这个读写锁上。就好象线程没有被中断过。

**非阻塞读锁定pthread\_rwlock\_tryrdlock**

#include <pthread.h>

int pthread\_rwlock\_tryrdlock(pthread\_rwlock\_t \*rwlock);

返回值：函数成功返回0；任何其他返回值都表示错误

pthread\_rwlock\_tryrdlock()函数和pthread\_rwlock\_rdlock函数的功能相近，不同的是，当已有线程写锁定读写锁，或是有试图写锁定的线程被阻塞时，pthread\_rwlock\_tryrdlock函数失败返回。

获取读写锁的写锁操作：分为阻塞和非阻塞，如果对应的读写锁被其它写者持有，或者读写锁被读者持有，该线程都会阻塞等待。

**写锁定读写锁pthread\_rwlock\_wrlock**

#include <pthread.h>

int pthread\_rwlock\_wrlock(pthread\_rwlock\_t \*rwlock);

返回值：函数成功返回0；任何其他返回值都表示错误

写锁定读写锁rwlock。如果没有线程读或写锁定读写锁rwlock，当前线程将写锁定读写锁rwlock。否则线程将被阻塞，直到没有线程锁定这个读写锁为止。

如果调用这个函数之前，本线程已经读或写锁定了这个读写锁，那么pthread\_rwlock\_wrlock函数运行的结果是不可预测的。

读写锁被解开以后，激活阻塞在读写锁上的线程时，往往优先考虑试图写锁定而被阻塞的线程，这样做是为了防止试图写锁定而被阻塞的线程长时间得不到调度。

当一个读写锁被写锁定了，而一个线程阻塞在这个读写锁上时，如果这时来了一个信号，那么当线程从信号处理程序中返回时，将继续阻塞在这个读写锁上。就好像线程没有被中断过。

**非阻塞写锁定pthread\_rwlock\_trywrlock**

#include <pthread.h>

int pthread\_rwlock\_trywrlock(pthread\_rwlock\_t \*rwlock);

返回值：函数成功返回0；任何其他返回值都表示错误

pthread\_rwlock\_trywrlock函数的功能和pthread\_rwlock\_wrlock函数相近。不同的是如果有其他线程锁定了读写锁，pthread\_rwlock\_trywrlock函数会失败返回。

**解锁读写锁pthread\_rwlock\_unlock**

#include <pthread.h>

int pthread\_rwlock\_unlock(pthread\_rwlock\_t \*rwlock);

返回值：函数成功返回0；任何其他返回值都表示错误

解锁一个读写锁。

调用pthread\_rwlock\_unlock函数之前必须调用pthread\_rwlock\_rdlock函数或pthread\_rwlock\_wrlock函数锁定读写锁。否则结果是不可预测的。

在pthread\_rwlock\_unlock函数被用来解锁对读写锁的读锁定后，如果本线程对这个读写锁还有其他读锁定，那么这个读写锁对本线程将继续保持读锁定状态。如果pthread\_rwlock\_unlock函数解开了当前线程在这个读写锁上的最后一个读锁定，那么当前线程将不再拥有对这个读写锁的读锁定。如果pthread\_rwlock\_unlock函数解开了这个读写锁上的最后一个锁定，那么这个读写锁将处在非锁定状态。

如果pthread\_rwlock\_unlock函数被用来解锁对读写锁的写锁定，那么函数返回后，这个读写锁将处在非锁定状态。

如果用pthread\_rwlock\_unlock函数解锁一个读写锁时，有多个线程在等待对这个读写锁进行写锁定，系统将用调度策略决定激活哪个线程对读写锁进行写锁定。

如果用pthread\_rwlock\_unlock函数解锁一个读写锁时，有多个线程在等待对这个读写锁进行读锁定，系统将用调度策略决定按什么顺序激活各个线程对读写锁进行读锁定。

如果用pthread\_rwlock\_unlock函数解锁一个读写锁时，有多个线程在等待对这个读写锁进行写锁定和读锁定，一般先激活需要写锁定的线程对读写锁进行写锁定(标准没有指定在这种情况下应该先激活写线程还是先激活读线程)。

**释放读写锁pthread\_rwlock\_destroy**

#include <pthread.h>

int pthread\_rwlock\_destroy(pthread\_rwlock\_t \*rwlock);

返回值：函数成功返回0；任何其他返回值都表示错误

释放读写锁rwlock，并释放这个读写锁占用的资源。一般情况下，pthread\_rwlock\_destroy函数将rwlock指向的互斥锁对象设置为非法值。

释放了一个读写锁之后，在pthread\_rwlock\_init重新初始化它之前，再使用这个读写锁会有不可预料的结果。

如果有线程锁定了某个读写锁，那么用pthread\_rwlock\_destroy()函数释放这个读写锁会引起不可预知的结果。

试图释放一个未初始化的读写锁会引起不可预知的结果。

1.4.1 **只执行一次锁**

|  |
| --- |
| C++ pthread\_once\_t once\_control = PTHREAD\_ONCE\_INIT; void init\_function(void) {  printf("Initialization function executed.\n"); }  void\* thread\_function(void\* arg) {  pthread\_once(&once\_control, init\_function); // 第一次执行  printf("Thread 1 completed.\n");  // 重置控制变量，手动设置可以执行2次  once\_control = PTHREAD\_ONCE\_INIT;  pthread\_once(&once\_control, init\_function); // 第二次执行  printf("Thread 2 completed.\n");  return NULL; } |

2. **IPC通信**

2.1 **各种方式优缺点**

1.管道（Pipe）：

* 特点：管道是一种半双工通信方式，适用于具有父子关系的进程间通信。它只能在具有公共祖先的进程之间使用，通常用于单向通信。
* 优点：简单易用，无需复杂的设置和管理。
* 缺点：只能在具有父子关系的进程之间使用，只能实现单向通信。

2.命名管道（Named Pipe）：

* 特点：命名管道也是一种半双工通信方式，但它可以用于非相关进程间的通信。它在文件系统中创建一个特殊文件，进程可以通过读写该文件来进行通信。
* 优点：适用于非相关进程间的通信，可以在不同的进程中进行读写操作。
* 缺点：只能实现单向通信，对于不同进程的读写操作需要进行协调。

3.共享内存（Shared Memory）：

* 特点：共享内存是一种高效的进程间通信方式，它通过将一块内存区域映射到不同的进程地址空间，使得这些进程可以直接读写共享的内存数据。
* 优点：速度快，适用于大量数据的高性能共享。
* 缺点：需要对共享内存的访问进行同步和互斥，以避免竞争条件和数据一致性问题。

4.信号量（Semaphore）：

* 特点：信号量是一种用于进程同步和互斥的通信方式，它可以用于进程间的互斥操作和资源控制。
* 优点：提供了进程间的同步和互斥机制，可以避免竞争条件。
* 缺点：对于复杂的通信需求，使用信号量可能较为繁琐。

5.消息队列（Message Queue）：

* 特点：消息队列是一种消息传递机制，通过在进程间传递消息实现通信。进程可以将消息发送到队列中，其他进程可以从队列中接收消息。
* 优点：支持多对多通信，进程可以按照优先级顺序接收消息。
* 缺点：对于大量数据传输，消息队列的效率可能较低。

6.套接字（Socket）：

* 特点：套接字是一种网络通信机制，可以用于不同主机上的进程间通信。它通过网络协议进行数据交换。
* 优点：支持不同主机上的进程间通信，可以实现跨网络的通信。
* 缺点：与其他通信方式相比，使用套接字进行进程间通信需要更多的网络设置和管理。

2.2 **IPC命令**

|  |
| --- |
| C++ 查看共享内存 ipcs -m 删除共享内存 ipcrm -m <shmid> |

3. **信号**

3.1 **信号类型**

1. SIGTERM（信号值为15）：

* SIGTERM是一个软终止信号。它是一种向进程发送的通知，请求进程自行安全地关闭。
* SIGTERM允许进程进行优雅的关闭，进行资源的回收，保存状态，并且做一些清理工作后再退出。
* 进程可以捕获SIGTERM信号，对其进行处理，甚至忽略它（虽然忽略此信号并不是一个好的做法）。
* 通常使用kill命令发送SIGTERM信号，如果没有指定信号类型，那么默认就是发送SIGTERM。

1. SIGKILL（信号值为9）：

* SIGKILL是一个硬终止信号，用于立即终止进程。
* 相比SIGTERM，SIGKILL信号并不安全，因为它不给进程自我终止的机会。操作系统会立即停止进程，进程无法捕获或忽略SIGKILL信号。
* SIGKILL经常用在无法使用SIGTERM关闭的进程上，例如进程挂起或不响应终止请求时。
* 发送SIGKILL信号时，进程不会有机会进行任何关闭前的清理工作，这可能会导致数据丢失或不一致的状态。

1. 信号 2：SIGINT (Interrupt signal)

中断信号，通常由用户通过终端发送（如Ctrl+C），用来请求中断一个程序。

1. 信号 6：SIGABRT (Abort signal)

异常终止信号，通常是由abort函数调用产生，表示程序异常终止。

1. 信号 10：SIGUSR1 (User-defined signal 1)

用户定义信号1，其具体行为不由POSIX标准定义，程序员可以用它来告知应用程序发生了一个事件。

1. 信号 11：SIGSEGV (Segmentation violation)

段违例信号，通常是程序尝试访问无效的内存段时产生，如访问未分配的内存或无权访问的内存。

1. 信号 15：SIGTERM (Termination signal)

终止信号，用来请求程序干净地退出。它允许程序捕捉该信号并执行清理操作，例如删除临时文件。

4. **事件**

4.1 **IO复用**

4.1.1 **epoll事件**

epoll\_ctl函数用于控制epoll实例对事件的监听，可以添加、修改或删除对文件描述符的监听。

* EPOLL\_CTL\_ADD：用于向epoll实例中添加要监听的文件描述符。
* EPOLL\_CTL\_MOD：用于修改已经添加到epoll实例中的文件描述符的监听事件。
* EPOLL\_CTL\_DEL：用于删除epoll实例中已经监听的文件描述符。

epoll\_wait函数用于监听事件的返回值中所包含的事件类型标志。

* EPOLLHUP表示发生了挂起事件。当文件描述符上的连接被断开或关闭时，会触发EPOLLHUP事件。
* EPOLLERR表示发生了错误事件。当文件描述符上发生了错误条件时，会触发EPOLLERR事件。

设置非阻塞模式

|  |
| --- |
| C++  //set NONBLOCK  int flag = ::fcntl(\_connfd, F\_GETFL, 0);  ::fcntl(\_connfd, F\_SETFL, O\_NONBLOCK | flag); |

5. **时间**

5.1 **定时工具**

**Crontab定时任务**

Crontab 是 Unix 类系统（包括 Linux 和 macOS）中用来定期执行任务的一种工具。它允许用户安排定时任务，这些任务称为 Cron 作业。Crontab 是 “cron table”的缩写，意味着它是 cron 守护进程的配置文件，该守护进程用以检查定时任务。

Crontab 文件包含了一系列的条目，每个条目代表一个要执行的作业。每个条目都是由六个字段组成的，前五个字段是时间和日期的指定，第六个字段是要执行的命令。

时间和日期字段包括：

1. 分钟 （0 - 59）
2. 小时 （0 - 23）
3. 日 （1 - 31）
4. 月 （1 - 12）
5. 星期几 （0 - 7，其中0和7都代表星期日）

命令字段 是系统命令或者脚本的路径。

这里是一个 crontab 文件的示例：

|  |
| --- |
| C++ Example of job definition: .---------------- minute (0 - 59) | .------------- hour (0 - 23) | | .---------- day of month (1 - 31) | | | .------- month (1 - 12) OR jan,feb,mar,apr ... | | | | .---- day of week (0 - 7) (Sunday=0 or 7) | | | | | \* \* \* \* \* user-name command to be executed  30 04 \* \* \* root /usr/bin/find / -size +10000 -exec ls -lh {} \; > /var/log/file\_sizes.log |

上面的 crontab 作业将在每天凌晨 4:30 为 root 用户执行查找大于 10000 字节的文件，并将结果输出到 /var/log/file\_sizes.log 文件中。

要编辑当前用户的 crontab 配置，可以在终端执行：

|  |
| --- |
| Plaintext crontab -e |

要列出当前用户的 crontab 配置，可以执行：

|  |
| --- |
| Plaintext crontab -l |

要删除当前用户的 crontab 配置，可以执行：

|  |
| --- |
| Plaintext crontab -r |

6. **网络编程**

6.1 **curl 网络工具**

|  |
| --- |
| Perl -X, --request <METHOD>：指定HTTP请求方法，如GET、POST、PUT、DELETE等。 -H, --header '<HEADER>'：设置HTTP请求头，可以多次使用以设置多个请求头。 -d, --data <DATA>：发送POST请求时，设置请求的数据体。 -F, --form <KEY=VALUE>：发送Multipart/form-data请求时，设置请求的表单字段。 -i, --include：在输出信息中包含响应的头部信息。 -o, --output <FILE>：将响应内容保存到指定文件中。 -L, --location：跟随重定向。 -u, --user <USER:PASSWORD>：设置验证用户和密码。 -s, --silent：静默模式，不显示额外信息。 -A, --user-agent <STRING>：设置请求的用户代理。 -c, --cookie <COOKIE>：设置cookie。 -b, --cookie-jar <FILE>：将cookie保存到指定文件中。 -k, --insecure：允许不安全的HTTPS连接。 -v, --verbose：显示详细的调试信息。 |