



developerWorks 中国 技术主题 Linux 文档库

Linux 多线程应用中如何编写安全的信号处理函数

关于代码的可重入性，设计开发人员一般只考虑到线程安全，异步信号处理函数的安全却往往被忽略。本文首先介绍如何编写安全的异步信号处理函数；然后举例说明在多线程应用中如何构建模型让异步信号在指定的线程中以同步的方式处理。

周婷，软件工程师，在上海交通大学获得自动控制专业学士和硕士学位。主要从事 Linux OS 下的开发工作，工作领域包括视频解码，IPV6，网络安全，Web 开发。

刘坚，Linux 爱好者，软件工程师，在上海交通大学获得学士和硕士学位。主要从事 Linux OS 下的开发工作，熟悉网络，Linux 内核。

唐桂峰，资深软件工程师，在南京大学获得计算机专业学士和硕士学位。研究方向包括人工智能，网络。

2009 年 6 月 18 日

Linux 多线程应用中编写安全的信号处理函数

在开发多线程应用时，开发人员一般都会考虑线程安全，会使用 `pthread_mutex` 去保护全局变量。如果应用中使用了信号，而且信号的产生不是因为程序运行出错，而是程序逻辑需要，譬如 `SIGUSR1`、`SIGRTMIN` 等，信号在被处理后应用程序还将正常运行。在编写这类信号处理函数时，应用层面的开发人员却往往忽略了信号处理函数执行的上下文背景，没有考虑编写安全的信号处理函数的一些规则。本文首先介绍编写信号处理函数时需要考虑的一些规则；然后举例说明在多线程应用中如何构建模型让因为程序逻辑需要而产生的异步信号在指定的线程中以同步的方式处理。



在 IBM Bluemix 云平台上
开发并部署您的下一个应用。

开始您的试用

线程和信号

Linux 多线程应用中，每个线程可以通过调用 `pthread_sigmask()` 设置本线程的信号掩码。一般情况下，被阻塞的信号将不能中断此线程的执行，除非此信号的产生是因为程序运行出错如 `SIGSEGV`；另外不能被忽略处理的信号 `SIGKILL` 和 `SIGSTOP` 也无法被阻塞。

当一个线程调用 `pthread_create()` 创建新的线程时，此线程的信号掩码会被新创建的线程继承。

POSIX.1 标准定义了一系列线程函数的接口，即 POSIX threads(Pthreads)。Linux C 库提供了两种关于线程的实现：LinuxThreads 和 NPTL(Native POSIX Threads Library)。LinuxThreads 已经过时，一些函数的实现不遵循 POSIX.1 规范。NPTL 依赖 Linux 2.6 内核，更加遵循 POSIX.1 规范，但也不是完全遵循。

基于 NPTL 的线程库，多线程应用中的每个线程有自己独特的线程 ID，并共享同一个进程 ID。应用程序可以通过调用 `kill(getpid(), signo)` 将信号发送到进程，如果进程中当前正在执行的线程没有阻碍此信号，则会被中断，信号处理函数会在此线程的上下文背景中执行。应用程序也可以通过调用 `pthread_kill(pthread_t thread, int sig)` 将信号发送给指定的线程，则信号处理函数会在此指定线程的上下文背景中执行。

基于 LinuxThreads 的线程库，多线程应用中的每个线程拥有自己独特的进程 ID，`getpid()` 在不同的线程中调用会返回不同的值，所以无法通过调用 `kill(getpid(), signo)` 将信号发送到整个进程。

下文介绍的在指定的线程中以同步的方式处理异步信号是基于使用了 NPTL 的 Linux C 库。请参考“[Linux 线程模型的比较：LinuxThreads 和 NPTL](#)”和“[pthreads\(7\) - Linux man page](#)”进一步了解 Linux 的线程模

型，以及不同版本的 Linux C 库对 NPTL 的支持。

编写安全的异步信号处理函数

信号的产生可以是：

用户从控制终端终止程序运行，如 Ctrk + C 产生 SIGINT；

程序运行出错时由硬件产生信号，如访问非法地址产生 SIGSEGV；

程序运行逻辑需要，如调用 kill、raise 产生信号。

因为信号是异步事件，即信号处理函数执行的上下文背景是不确定的，譬如一个线程在调用某个库函数时可能会被信号中断，库函数提前出错返回，转而去执行信号处理函数。对于上述第三种信号的产生，信号在产生、处理后，应用程序不会终止，还是会继续正常运行，在编写此类信号处理函数时尤其需要小心，以免破坏应用程序的正常运行。关于编写安全的信号处理函数主要有以下一些规则：

信号处理函数尽量只执行简单的操作，譬如只是设置一个外部变量，其它复杂的操作留在信号处理函数之外执行；

errno 是线程安全，即每个线程有自己的 errno，但不是异步信号安全。如果信号处理函数比较复杂，且调用了可能会改变 errno 值的库函数，必须考虑在信号处理函数开始时保存、结束的时候恢复被中断线程的 errno 值；

信号处理函数只能调用可以重入的 C 库函数；譬如不能调用 malloc ()，free () 以及标准 I/O 库函数等；

信号处理函数如果需要访问全局变量，在定义此全局变量时须将其声明为 volatile，以避免编译器不恰当的优化。

从整个 Linux 应用的角度出发，因为应用中使用了异步信号，程序中一些库函数在调用时可能被异步信号中断，此时必须根据 errno 的值考虑这些库函数调用被信号中断后的出错恢复处理，譬如 socket 编程中的读操作：

```
rilen = recv(sock_fd, buf, len, MSG_WAITALL);
if ((rilen == -1) && (errno == EINTR)){
    // this kind of error is recoverable, we can set the offset change
    // 'rilen' as 0 and continue to recv
}
```

在指定的线程中以同步的方式处理异步信号

如上文所述，不仅编写安全的异步信号处理函数本身有很多的规则束缚；应用中其它地方在调用可被信号中断的库函数时还需考虑被中断后的出错恢复处理。这让程序的编写变得复杂，幸运的是，POSIX.1 规范定义了 sigwait()、sigwaitinfo() 和 pthread_sigmask() 等接口，可以实现：

以同步的方式处理异步信号；

在指定的线程中处理信号。

这种在指定的线程中以同步方式处理信号的模型可以避免因为处理异步信号而给程序运行带来的不确定性和潜在危险。

sigwait

sigwait() 提供了一种等待信号的到来，以串行的方式从信号队列中取出信号进行处理的机制。

sigwait () 只等待函数参数中指定的信号集，即如果新产生的信号不在指定的信号集内，则 sigwait () 继续等待。对于一个稳定可靠的程序，我们一般会有一些疑问：

多个相同的信号可不可以不在信号队列中排队？

如果信号队列中有多个信号在等待，在信号处理时有没有优先级规则？

实时信号和非实时信号在处理时有没有什么区别？

笔者写了一小段测试程序来测试 `sigwait` 在信号处理时的一些规则。

清单 1. `sigwait_test.c`

```
#include <signal.h>
#include <errno.h>
#include <pthread.h>
#include <unistd.h>
#include <sys/types.h>

void sig_handler(int signum)
{
    printf("Receive signal. %d\n", signum);
}

void* sigmgr_thread()
{
    sigset_t waitset, oset;
    int sig;
    int rc;
    pthread_t ppid = pthread_self();

    pthread_detach(ppid);

    sigemptyset(&waitset);
    sigaddset(&waitset, SIGRTMIN);
    sigaddset(&waitset, SIGRTMIN+2);
    sigaddset(&waitset, SIGRTMAX);
    sigaddset(&waitset, SIGUSR1);
    sigaddset(&waitset, SIGUSR2);

    while (1) {
        rc = sigwait(&waitset, &sig);
        if (rc != -1) {
            sig_handler(sig);
        } else {
            printf("sigwaitinfo() returned err: %d; %s\n", errno, strerror(errno));
        }
    }
}

int main()
{
    sigset_t bset, oset;
    int i;
    pid_t pid = getpid();
    pthread_t ppid;

    sigemptyset(&bset);
    sigaddset(&bset, SIGRTMIN);
    sigaddset(&bset, SIGRTMIN+2);
    sigaddset(&bset, SIGRTMAX);
    sigaddset(&bset, SIGUSR1);
    sigaddset(&bset, SIGUSR2);

    if (pthread_sigmask(SIG_BLOCK, &bset, &oset) != 0)
        printf("!! Set pthread mask failed\n");

    kill(pid, SIGRTMAX);
    kill(pid, SIGRTMAX);
    kill(pid, SIGRTMIN+2);
    kill(pid, SIGRTMIN);
    kill(pid, SIGRTMIN+2);
    kill(pid, SIGRTMIN);
    kill(pid, SIGUSR2);
    kill(pid, SIGUSR2);
    kill(pid, SIGUSR1);
    kill(pid, SIGUSR1);

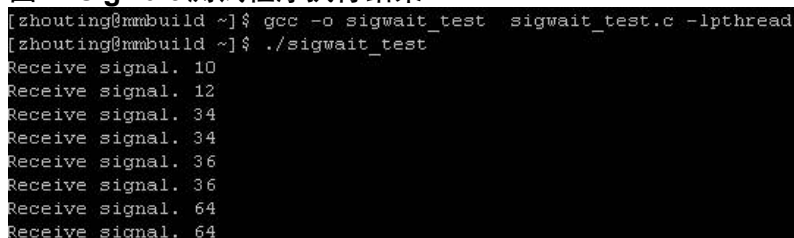
    // Create the dedicated thread sigmgr_thread() which will handle signals synchronously
    pthread_create(&ppid, NULL, sigmgr_thread, NULL);

    sleep(10);

    exit (0);
}
```

程序编译运行在 RHEL4 的结果如下：

图 1. `sigwait` 测试程序执行结果



```
[zhouting@mmmbuild ~]$ gcc -o sigwait_test sigwait_test.c -lpthread
[zhouting@mmmbuild ~]$ ./sigwait_test
Receive signal. 10
Receive signal. 12
Receive signal. 34
Receive signal. 34
Receive signal. 36
Receive signal. 36
Receive signal. 64
Receive signal. 64
```

从以上测试程序发现以下规则：

对于非实时信号，相同信号不能在信号队列中排队；对于实时信号，相同信号可以在信号队列中排队。

如果信号队列中有多个实时以及非实时信号排队，实时信号并不会先于非实时信号被取出，信号数字小的会先被取出：如 SIGUSR1（10）会先于 SIGUSR2（12），SIGRTMIN（34）会先于 SIGRTMAX（64），非实时信号因为其信号数字小而先于实时信号被取出。

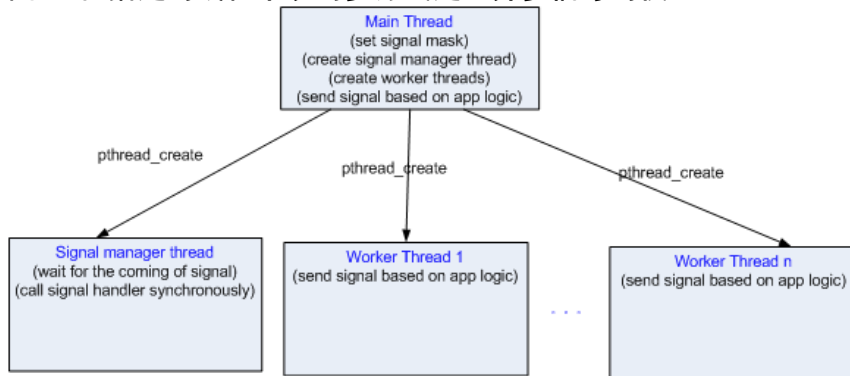
sigwaitinfo（） 以及 sigtimedwait（） 也提供了与 sigwait（） 函数相似的功能。

Linux 多线程应用中的信号处理模型

在基于 Linux 的多线程应用中，对于因为程序逻辑需要而产生的信号，可考虑调用 sigwait（） 使用同步模型进行处理。其程序流程如下：

1. 主线程设置信号掩码，阻碍希望同步处理的信号；主线程的信号掩码会被其创建的线程继承；
2. 主线程创建信号处理线程；信号处理线程将希望同步处理的信号集设为 sigwait（） 的第一个参数。
3. 主线程创建工作线程。

图 2. 在指定的线程中以同步方式处理异步信号的模型



代码示例

以下为一个完整的在指定的线程中以同步的方式处理异步信号的程序。

主线程设置信号掩码阻碍 SIGUSR1 和 SIGRTMIN 两个信号，然后创建信号处理线程

sigmgr_thread（）和五个工作线程 worker_thread（）。主线程每隔10秒调用 kill（） 对本进程发送 SIGUSR1 和 SIGTRMIN 信号。信号处理线程 sigmgr_thread（） 在接收到信号时会调用信号处理函数 sig_handler（）。

程序编译：gcc -o signal_sync signal_sync.c -lpthread

程序执行：./signal_sync

从程序执行输出结果可以看到主线程发出的所有信号都被指定的信号处理线程接收到，并以同步的方式处理。

清单 2. signal_sync.c

```
#include <signal.h>
#include <errno.h>
#include <pthread.h>
#include <unistd.h>
#include <sys/types.h>

void sig_handler(int signum)
{
    static int j = 0;
    static int k = 0;
    pthread_t sig_ppid = pthread_self();
    // used to show which thread the signal is handled in.

    if (signum == SIGUSR1) {
        printf("thread %d, receive SIGUSR1 No. %d\n", sig_ppid, j);
        j++;
        //SIGRTMIN should not be considered constants from userland,
        //there is compile error when use switch case
    } else if (signum == SIGRTMIN) {
        printf("thread %d, receive SIGRTMIN No. %d\n", sig_ppid, k);
        k++;
    }
}

void* worker_thread()
{
    pthread_t ppid = pthread_self();
    pthread_detach(ppid);
    while (1) {
        printf("I'm thread %d, I'm alive\n", ppid);
        sleep(10);
    }
}
```

```

    }
}

void* sigmgr_thread()
{
    sigset_t  waitset, oset;
    siginfo_t info;
    int       rc;
    pthread_t ppid = pthread_self();

    pthread_detach(ppid);

    sigemptyset(&waitset);
    sigaddset(&waitset, SIGRTMIN);
    sigaddset(&waitset, SIGUSR1);

    while (1) {
        rc = sigwaitinfo(&waitset, &info);
        if (rc != -1) {
            printf("sigwaitinfo() fetch the signal - %d\n", rc);
            sig_handler(info.si_signo);
        } else {
            printf("sigwaitinfo() returned err: %d; %s\n", errno, strerror(errno));
        }
    }
}

int main()
{
    sigset_t bset, oset;
    int      i;
    pid_t    pid = getpid();
    pthread_t ppid;

    // Block SIGRTMIN and SIGUSR1 which will be handled in
    //dedicated thread sigmgr_thread()
    // Newly created threads will inherit the pthread mask from its creator
    sigemptyset(&bset);
    sigaddset(&bset, SIGRTMIN);
    sigaddset(&bset, SIGUSR1);
    if (pthread_sigmask(SIG_BLOCK, &bset, &oset) != 0)
        printf("!! Set pthread mask failed\n");

    // Create the dedicated thread sigmgr_thread() which will handle
    // SIGUSR1 and SIGRTMIN synchronously
    pthread_create(&ppid, NULL, sigmgr_thread, NULL);

    // Create 5 worker threads, which will inherit the thread mask of
    // the creator main thread
    for (i = 0; i < 5; i++) {
        pthread_create(&ppid, NULL, worker_thread, NULL);
    }

    // send out 50 SIGUSR1 and SIGRTMIN signals
    for (i = 0; i < 50; i++) {
        kill(pid, SIGUSR1);
        printf("main thread, send SIGUSR1 No. %d\n", i);
        kill(pid, SIGRTMIN);
        printf("main thread, send SIGRTMIN No. %d\n", i);
        sleep(10);
    }
    exit (0);
}

```

注意事项

在基于 Linux 的多线程应用中，对于因为程序逻辑需要而产生的信号，可考虑使用同步模型进行处理；而对会导致程序运行终止的信号如 SIGSEGV 等，必须按照传统的异步方式使用 `signal()`、`sigaction()` 注册信号处理函数进行处理。这两种信号处理模型可根据所处理的信号的不同同时存在一个 Linux 应用中：

不要在线程的信号掩码中阻塞不能被忽略处理的两个信号 SIGSTOP 和 SIGKILL。

不要在线程的信号掩码中阻塞 SIGFPE、SIGILL、SIGSEGV、SIGBUS。

确保 `sigwait()` 等待的信号集已经被进程中所有的线程阻塞。

在主线程或其它工作线程产生信号时，必须调用 `kill()` 将信号发给整个进程，而不能使用 `pthread_kill()` 发送某个特定的工作线程，否则信号处理线程无法接收到此信号。

因为 `sigwait()` 使用了串行的方式处理信号的到来，为避免信号的处理存在滞后，或是非实时信号被丢失的情况，处理每个信号的代码应尽量简洁、快速，避免调用会产生阻塞的库函数。

小结

在开发 Linux 多线程应用中，如果因为程序逻辑需要引入信号，在信号处理后程序仍将继续正常运行。在这种背景下，如果以异步方式处理信号，在编写信号处理函数一定要考虑异步信号处理函数的安全；同

时，程序中一些库函数可能会被信号中断，错误返回，这时需要考虑对 EINTR 的处理。另一方面，也可考虑使用上文介绍的同步模型处理信号，简化信号处理函数的编写，避免因信号处理函数执行上下文的不确定性而带来的风险。

免责声明：

1. 本文所提出的方式方法仅代表作者个人观点。
2. 本文属于原创作品，资料来源不超出参考文献所列范畴，其中任何部分都不会侵犯任何第三方的知识产权。

参考资料

“Advanced Programming in the UNIX® Environment: Second Edition”，W. Richard Stevens, Stephen A. Rago，清楚、全面地介绍了Linux线程和信号方面的知识，但是对Linux线程的介绍是基于 LinuxThreads 而不是 NPTL，有些过时。

参看“[Linux 线程模型的比较：LinuxThreads 和 NPTL](#)”，了解 LinuxThreads 和 NPTL 线程模型的历史和区别。

参看“[Signal Handling](#)”，全面地介绍了 Linux C 库的信号机制。

参看“[The Native POSIX Thread Library for Linux](#)”，了解 Linux 的 NPTL 线程库。

参看“[Use reentrant functions for safer signal handling](#)”，了解编写安全的异步信号处理函数的一些规则。

在 [developerWorks Linux 专区](#) 寻找为 Linux 开发人员（包括 [Linux 新手入门](#)）准备的更多参考资料，查阅我们 [最受欢迎的文章和教程](#)。

在 developerWorks 上查阅所有 [Linux 技巧](#) 和 [Linux 教程](#)。



IBM Bluemix 资源中心

文章、教程、演示，帮助您构建、部署和管理云应用。



developerWorks 中文社区

立即加入来自 IBM 的专业 IT 社交网络。



Bluemixathon 挑战赛

为灾难恢复构建应用，赢取现金大奖。