[C++] UNIX 上的 C++程序设计守则(1)

原文:http://d.hatena.ne.jp/yupo5656/20040712/p1

Unix 跟 Windows 等那些”对于开发者易于使用”的 OS 比起来，在信号和线程的利用方面有诸多的限制。但是即使不知道这些知识就做构架设计和实现的情况也随处可见。这个就是那些经常不能再现的 bug 的温床吧。

因此，我想分成几回来写一些准则来防止陷入到这些圈套里。

准则 1：不依赖于信号收发的设计

·给其他进程以及自己发送异步信号并改变处理流程的设计不要做

- 异步信号是值用 kill 系统调用来创建?发送的信号、例如 SIGUSR1,SIGUSR2,SIGINT,SIGTERM 等
- 简单的使用忽略信号(SIG_IGN)则没有问题

·不要把线程和信号一起使用

- 这将使程序动作的预测和调试变得很困难

说明：

同步信号是指，因为某些特定的操作*1 而引起向自身进程发送某些特定的信号，例如 SIGSEGV,SIGBUS,SIGPIPE,SIGSYS,SIGILL,SIGFPE。异步信号就是这些以外的信号。在什么时机发送异步信号并不能被预测出来。我们会在程序里追加收到某些信号时做一些特殊处理(信号处理函数)的函数。那么根据收到的信号就跳到信号处理函数的程序就叫做”在任意代码处都能发生跳转”的程序。这样的程序往往隐藏这下面的那些问题：

1. **容易引入 BUG**。”任意的代码”虽然也包含”执行 C/C++里面的一条语句的过程中”的意思，但这很容易跳出程序员的正常思维以及默认的假定条件。编写程序的时候往往需要考虑比 C++异常分支还要多得多的分支情况。
2. **使测试项目激增**。即使根据白盒测试达成 100%的分支覆盖,也不能网罗到因为接受信号而发生的跳转分支处理。也就是说做到 100%的网罗信号跳转分支的测试是不能全部实现

的。一般的，加上要考虑”在实行某个特定代码时因为接受到信号而发生的误操作”这样的 BUG 会经常发生*2 的这种情况，测试困难往往就是导致软件的品质低下的诱因。

根据经验，”当检查到子进程结束(接收到 SIGCHLD 信号)时，要做必要的处理”像这样的信号处理不管做什么都是有必要的情况会有，但是除此以外的信号处理，例如

- 把自己的状态用信号告诉其他进程
- 主线程在输入输出函数里发送信号给被阻塞的子线程，并解除阻塞

等，是应该事先好好好好考虑过后再去做实际的实现。前者的话，如果不强制在”普通的”进程间进行通信的话可能会很好，后者是特意要使用线程，也要应该按照即使阻塞了也不能发生问题那样再设计。

不管怎么样，如果必须要使用信号的话，也要先全部*3 理解这些陷阱以及，和多线程软件设计的场合一样或者说比它更严格的制约.注意事项都需要铭记在心里。

*1：例如，引用空指针

*2：参照 id:yupo5656:20040703 的 sigsafe 说明

*3：暂时先掌握”准则 2”:-)

UNIX 上 C++程序设计守则 (2)

原文地址：http://d.hatena.ne.jp/yupo5656/20040712/p2

准则 2: 要知道信号处理函数中可以做那些处理

· 在用 sigaction 函数登记的信号处理函数中可以做的处理是被严格限定的

· 仅仅允许做下面的三种处理

1. 局部变量的相关处理
2. “volatile sig_atomic_t”类型的全局变量的相关操作
3. 调用异步信号安全的相关函数

· 以外的其他处理不要做！

说明：

因为在收到信号时要做一些处理，那通常是准备一个信号处理函数并用 sigaction 函数把它和信号名进行关联的话就 OK 了。但是，在这个信号处理函数里可以做的处理是像上面那样被严格限定的。没有很好掌握这些知识就随便写一些代码的话就会引起下面那样的问题：

· **问题 1: 有程序死锁的危险**

  o 这是那些依赖于某一时刻，而且错误再现比较困难的 BUG 产生的真正原因

  o 死锁是一个比较典型的例子，除此之外还能引起函数返回值不正确，以及在某一函数内执行时突然收到 SEGV 信号等的误操作。

*◆译者注 1：**SEGV** 通常发生在进程试图访问无效内存区域时（可能是个 NULL 指针，或超出进程空间之外的内存地址）。当 bug 原因和 **SEGV** 影响在不同时间呈现时，它们特别难于捕获到。*

· **问题 2: 由于编译器无意识的优化操作，有导致程序紊乱的危险**

  o 这是跟编译器以及编译器优化级别有关系的 bug。它也是"编译器做了优化处理而不能正常动作"，"因为 inline 化了程序不能动作了"，"变换了 OS 了程序也不能动作"等这些解析困难 bug 产生的原因。

还是一边看具体的代码一边解说吧。在下面的代码里至少有三个问题，根据环境的不同很可能引起不正确的动作[*1]、按照次序来说明里面的错误。

```
1  int gSignaled;
2 void sig_handler(int signo) {
3     std::printf("signal %d received!\n", signo);
4     gSignaled = 1;
5 }
6 int main(void) {
7     struct sigaction sa;
8      // (省略)
9      sigaction(SIGINT, &sa, 0);
10    gSignaled = 0;
11    while(!gSignaled) {
12     //std::printf("waiting…\n");
```

```
13        struct timespec t = { 1, 0 }; nanosleep(&t, 0);
14    }
15 }
16
```

错误 1: 竞争条件

在上面的代码里有竞争条件。在 sigaction 函数被调用后、在 gSignaled 还未被赋值成 0 值之前，如果接受到 SIGINT 信号了那会变得怎么样呢? 在信号处理函数中被覆写成 1 后的 gSignaled 会在信号处理函数返回后被初始化成 0、在后面的 while 循环里可能会变成死循环。

错误 2: 全局变量 gSignaled 声明的类型不正确

在信号处理函数里使用的全局变数 gSignaled 的类型没有声明成 volatile sig_atomic_t 。这样的话、在执行 while 循环里的代码的时候接收到了了 SIGINT 信号时、有可能引起 while 的死循环。那为什么能引起这样的情况呢：

· 信号处理函数里，把内存上 gSignaled 的值变更成 1 ， 它的汇编代码如下：

```
movl    $1, gSignaled
```

· 但是，就像下面的代码描述的那样，main 函数是把 gSignaled 的值存放到了寄存器里。在 while 循环之前，仅仅是做了一次拷贝变量 gSignaled 内存上的值到寄存器里、而在 while 循环里只是参照这个寄存器里的值。

```
    movl    gSignaled, %ebx
.L8:
        testl   %ebx, %ebx
        jne     .L8
```

在不执行优化的情况下编译后编译器有可能不会生成上面那样的伪代码。但 Gcc 当使用-O2 选项做优化编译时，生成的实际那样的汇编代码产生的危害并不仅仅是像上面说的威胁那样简单。这方面的问题，是设备驱动的开发者所要知道的常识，但现实情况是对于应用程序的设计者.开发者几乎都不知道这些知识。

为了解决上面的问题，全局变量 gSignaled 的类型要像下面那样声明。

volatile sig_atomic_t gSignaled;

volatile 则是提示编译器不要像上面那样做优化处理，变成每次循环都要参照该变量内存里的值那样进行编译。所以在信号处理函数里把该变量的值修改后也能真实反映到 main 函数的 while 循环里。

sig_atomic_t 是根据 CPU 类型使用 typedef 来适当定义的整数值，例如 x86 平台是 int 类型。就是指"用一条机器指令来更新内存里的最大数据[*2]"。在信号处理函数里要被引用的变量必须要定义成 sig_atomic_t 类型。那么不是 sig_atomic_t 类型的变量(比如 x86 平台上的 64 位整数)、就得使用两条机器指令来完成更新动作。如果在执行一条机器指令的时候突然收到一个信号而程序执行被中断，而且在信号处理函数中一引用这个变量的话，就只能看到这个变量的部分的值。另外，由于字节对齐的问题不能由一条机器指令来完成的情况也会存在。把该变量的类型变成 sig_atomic_t 的话，这个变量被更新时就只需要一条机器指令就可以完成了。所以在信号处理函数里即使使用了该变量也不会出现任何问题。

2006/1/16 补充: 有一点东西忘记写了。关于 sig_atomic_t 详细的东西，请参考 C99 规范的§7.14.1.1/5 小节。在信号处理函数里对 volatile sig_atomic_t 以外的变量进行修改，其结果都是"unspecified"的(参照译者注 2)。另外，sig_atomic_t 类型的变量的取值范围是在 SIG_ATOMIC_MIN/MAX 之间 (参见§7.18.3/2)。有无符号是跟具体的实现有关。考虑到移植性取值在 0 ~ 127 之间是比较合适的。C99 也支持这个取值范围。C++规范(14882:2003)里也有同样的描述、确切的位置是§1.9/9 这里。在 SUSv3 的相关描述请参考 sigaction 这里[*3]。此外、虽然在 GCC 的参考手册里也说了把指针类型更新成原子操作，但在标准 C/C++却没有记载[*4]。

◆*译者注 2：*

When the processing of the abstract machine is interrupted by receipt of a signal, the value of objects with type other than volatile ***sig_atomic_t*** *are* ***unspecified****, and the value of any object not of volatile* ***sig_atomic_t*** *that is modified by the handler becomes undefined.*

------ ISO/IEC FDIS 14882:1998(E) 的 1.9 小节

错误 3: 在信号处理函数里调用了不可重入的函数

上述的样例代码中调用了 printf 函数，但是这个函数是一个不可重入函数，所以在信号处理函数里调用的话可能会引起问题。具体的是，在信号处理函数里调用 printf 函数的瞬间，引起程序死锁的可能性还是有的。但是，这个问题跟具体的时机有关系，所以再现起来很困难，也就成了一个很难解决的 bug 了。

下面讲一下 bug 发生的过程。首先、讲解一下 printf 函数的内部实现。

・printf 函数内部调用 malloc 函数

・malloc 函数会在内部维护一个静态区域来保存 mutex 锁、是为了在多线程调用 malloc 函数的时候起到互斥的作用

・总之、malloc 函数里有"mutex 锁定，分配内存，mutex 解锁"这样"连续的不能被中断"的处理

main 関数:

call printf // while 循环中的 printf 函数

call malloc

call pthread_mutex_lock(锁定 malloc 函数内的静态 mutex)

// 在 malloc 处理时..

☆收到 SIGINT 信号！

call sig_handler

call printf // 信号处理函数中的 printf 函数

call malloc

call pthread_mutex_lock(锁定 malloc 函数内的静态 mutex)

// 相同的 mutex 一被再度锁定，就死锁啦!!

知道上面的流程的话、像这样的由于信号中断引起的死锁就能被理解了吧。为了修正这个 bug，在信号处理函数里就必须调用可重入函数。可重入函数的一览表在 UNIX 规范 (SUSv3)有详细记载[*5]。你一定会惊讶于这个表里的函数少吧。

另外，一定不要忘记以下的几点：

・虽然在 SUSv3 里有异步信号安全(async-signal-safe)函数的一览，但根据不同的操作系统，某些函数是没有被实现的。所以一定要参考操作系统的手册

・第三者做成的函数，如果没有特别说明的场合，首先要假定这个函数是不可重入函数，不能随便在信　号处理函数中使用。

・调用不可重入函数的那些函数就会变成不可重入函数了

最后，为了明确起见，想说明一下什么是" 异步信号安全(async-signal-safe)"函数。异步信号安全函数是指"在该函数内部即使因为信号而正在被中断，在其他的地方该函数再被调用了也没有任何问题"。如果函数中存在更新静态区域里的数据的情况(例如，malloc)，一般情况下都是不全的异

步信号函数。但是，即使使用静态数据，如果在这里这个数据时候把信号屏蔽了的话，它就会变成异步信号安全函数了。

◆译者注 3：不可重入函数就不是异步信号安全函数

*1：sigaction 函数被调用前，一接收到 SIGINT 信号就终止程序，暂且除外吧

*2：“最大”是不完全正确的。例如，Alpha 平台上 32/64bit 的变量用一条命令也能被更新，但是好像把 8/16bit 的数据更新编程了多条命令了。http://lists.sourceforge.jp/mailman/archives/anthy-dev/2005-September/002336.html 请参考这个 URL 地址。

*3：If the signal occurs other than as the result of calling abort(), kill(), or raise(), the behavior is undefined if the signal handler calls any function in the standard library other than one of the functions listed in the table above or refers to any object with static storage duration other than by assigning a value to a static storage duration variable of type volatile sig_atomic_t. Furthermore, if such a call fails, the value of errno is unspecified.

*4：在这个手册里“ In practice, you can assume that int and other integer types no longer than int are atomic. ”这部分是不正确的。请参照 Alpha 的例子

*5：The following table defines a set of functions that shall be either reentrant or non-interruptible by signals and shall be async-signal-safe. 后面有异步信号安全函数一览