准则 3：多线程程序里不准使用 fork

Posted on 2008-06-01 20:鉄則 3:　マルチスレッドのプログラムでの fork はやめよう

准则 3：多线程程序里不准使用 fork

マルチスレッドのプログラムで、「自スレッド以外のスレッドが存在している状態」で fork

何が起きるか

能引起什么问题呢?

実例から見てみましょう。次のコードを実行すると、子プロセスは実行開始直後の doit()　呼び出し時、高い確率でデッドロックします。

那看看实例吧. 一执行下面的代码, 在子进程的执行开始处调用 doit()时, 发生死锁的机率会很高.

```
1 void* doit(void*) {
2 
3     static pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
4 
5     pthread_mutex_lock(&mutex);
6 
7     struct timespec ts = {10, 0}; nanosleep(&ts, 0); // 10 秒寝る
8                                                      // 睡 10 秒
9 
10     pthread_mutex_unlock(&mutex);
11 
12     return 0;
13 
14 }
15 
16 
17 
18 int main(void) {
```

```
19
20 pthread_t t;
21
22 pthread_create(&t, 0, doit, 0); // サブスレッド作成・起動
23
24                          // 做成并启动子线程
25
26   if (fork() == 0) {
27
28       // 子プロセス。
29
30       // 子プロセスが生成される瞬間、親のサブスレッドは nanosleep 中の場合が多い。
31
32       //子进程
33
34       //在子进程被创建的瞬间,父的子进程在执行 nanosleep 的场合比较多
35
36       doit(0); return 0;
37
38   }
39
40 pthread_join(t, 0); // サブスレッド完了待ち
41
42                 // 等待子线程结束
43
44 }
45
```

以下にデッドロックの理由を説明いたします。

以下是说明死锁的理由.

一般に、fork を行うと

一般的,fork 做如下事情

1. 親プロセスの「データ領域」は子プロセスにそのままコピー
2. 子プロセスは、シングルスレッド状態で生成
1. 父进程的内存数据会原封不动的拷贝到子进程中
2. 子进程在单线程状态下被生成

されます。データ領域には、静的記憶域を持つ変数*2 が格納されていますが、それらは子プロセスにコピーされます。また、親プロセスにスレッドが複数存在していても、子プロセスにそれらは継承されません。fork に関する上記 2 つの特徴がデッドロックの原因となります。

在内存区域里,静态变量*2mutex 的内存会被拷贝到子进程里.而且,父进程里即使存在多个线程,但它们也不会被继承到子进程里. fork 的这两个特征就是造成死锁的原因.

译者注: 死锁原因的详细解释 ---

1. *线程里的 doit()先执行.*
2. *doit 执行的时候会给互斥体变量 mutex 加锁.*
3. *mutex 变量的内容会原样拷贝到 fork 出来的子进程中(在此之前,mutex 变量的内容已经被线程改写成锁定状态).*
4. *子进程再次调用 doit 的时候,在锁定互斥体 mutex 的时候会发现它已经被加锁,所以就一直等待,直到拥有该互斥体的进程释放它(实际上没有人拥有这个 mutex 锁).*
5. *线程的 doit 执行完成之前会把自己的 mutex 释放,但这是的 mutex 和子进程里的 mutex 已经是两份内存.所以即使释放了 mutex 锁也不会对子进程里的 mutex 造成什么影响.*

例えば次のようなシナリオを考えてみてください。上記のマルチスレッドプログラムでの不用意な fork によって子プロセスがデッドロックすることがわかると思います*3。

例如,请试着考虑下面那样的执行流程,就明白为什么在上面多线程程序里不经意地使用 fork 就造成死锁了*3.

1. fork 前の親プロセスでは、スレッド 1 と 2 が動いている
2. スレッド 1 が doit 関数を呼ぶ
3. doit 関数が自身の mutex をロックする

4. スレッド 1 が nanosleep を実行し、寝る
5. ここで処理がスレッド 2 に切り替わる
6. スレッド 2 が fork 関数を呼ぶ
7. 子プロセスが生成される。
8. この時、子プロセスの doit 関数用 mutex は「ロック状態」である。また、ロック状態を解除するスレッドは子プロセス中には存在しない!
9. 子プロセスが処理を開始する。
10. 子プロセスが doit 関数を呼ぶ
11. 子プロセスがロック済みの mutex を再ロックしてしまい、デッドロックする

1. 在 fork 前的父进程中,启动了线程 1 和 2
2. 线程 1 调用 doit 函数
3. doit 函数锁定自己的 mutex
4. 线程 1 执行 nanosleep 函数睡 10 秒
5. 在这儿程序处理切换到线程 2
6. 线程 2 调用 fork 函数
7. 生成子进程
8. 这时,子进程的 doit 函数用的 mutex 处于”锁定状态”,而且,解除锁定的线程在子进程里不存在
9. 子进程的处理开始
10. 子进程调用 doit 函数
11. 子进程再次锁定已经是被锁定状态的 mutex,然后就造成死锁

この doit 関数のように、マルチスレッド下での fork で問題を引き起こす関数を、「fork-unsafe な関数」と呼ぶことがあります。逆に、問題を起こさない関数を「fork-safe な関数」と呼ぶことがあります。一部の商用 UNIX*4 では、OS の提供する関数について、ドキュメントに fork-safety の記載がありますが、Linux(glibc)にはもちろん! 記載がありません。~~POSIX でも特に規定がありませんので、どの関数が fork-safe であるかは殆ど判別不能です。わからなければ unsafe と考えるほうが良いでしょう。~~ (2004/9/12 追記) Wolfram Gloger さんが非同期シグナルセーフな関数を呼ぶのは規格準拠と言っておられるので調べてみたら、pthread atfork のところに ″In the meantime*5, only a short list of async-signal-safe library routines

are promised to be available.″ とありました。そういうことのようです。

像这里的 doit 函数那样的,在多线程里因为 fork 而引起问题的函数,我们把它叫做”fork-unsafe 函数”.反之,不能引起问题的函数叫做”fork-safe 函数”.虽然在一些商用的 UNIX 里,源于 OS 提供的函数(系统调用),在文档里有 fork-safety 的记载,但是在 Linux(glibc)里当然!不会被记载.~~即使在 POSIX 里也没有特别的规定,所以那些函数是 fork-safe 的,几乎不能判别.不明白的话,作为 unsafe 考虑的话会比较好一点吧.~~(2004/9/12 追记)Wolfram Gloger 说过,调用异步信号安全函数是规格标准,所以试着调查了一下,在 pthread_atfork の这个地方里有” In the meantime*5, only a short list of async-signal-safe library routines are promised to be available.”这样的话.好像就是这样.

ちなみに、malloc 関数は自身に固有の mutex を持っているのが通例ですので、普通は fork-unsafe です。malloc 関数に依存する数多くの関数、例えば printf 関数なども fork-unsafe となります。

随便说一下,malloc 函数就是一个维持自身固有 mutex 的典型例子,通常情况下它是 fork-unsafe 的.依赖于 malloc 函数的函数有很多,例如 printf 函数等,也是变成 fork-unsafe 的.

いままで thread+fork は危険と書いてきましたが、一つだけ特例があります。「fork 直後にすぐ exec する場合は、特例として問題がない」のです。何故でしょう..? exec 系関数*6 が 呼ばれると、プロセスの「データ領域」は一旦綺麗な状態にリセットされます。したがって、マルチスレッド状態のプロセスであっても、fork 後にすぐ、危 険な関数を一切呼ばずに exec 関数を呼べば、子プロセスが誤動作することはないのです。ただし、「すぐ」と書いてあることに注意してください。**exec 前に printf(“I’m child process”); を一発呼ぶだけでもデッドロックの危険があります!**

直到目前为止,已经写上了 thread+fork 是危险的,但是有一个特例需要告诉大家.”fork 后马上调用 exec 的场合,是作为一个特列不会产生问题的”. 什么原因呢..? exec 函数*6 一被调用,进程的”内存数据”就被临时重置成非常漂亮的状态.因此,即使在多线程状态的进程里,fork 后不马上调用一切危险的函数,只是调用 exec 函数的话,子进程将不会产生任何的误动作.但是,请注意这里使用的”马上”这个词.**即使 exec 前仅仅只是调用一回 printf(“I’m child**

process”）, 也会有死锁的危险.

译者注:exec 函数里指明的命令一被执行, 改命令的内存映像就会覆盖父进程的内存空间. 所以, 父进程里的任何数据将不复存在.

災いをどう回避するか

如何规避灾难呢?

マルチスレッドのプログラムでの fork を安全に行うための、デッドロック問題回避の方法はあるでしょうか? いくつか考えてみます。

为了在多线程的程序中安全的使用 fork, 而规避死锁问题的方法有吗?试着考虑几个.

回避方法１: fork を行う場合は、それに先立って他スレッドを全て終了させる

规避方法 1:做 fork 的时候, 在它之前让其他的线程完全终止.

fork に先立って他スレッドを全て終了させておけば、問題はおきません。ただ、それが可能なケースばかりではないでしょう。また、何らかの要因で他スレッドの終了が行われないまま fork してしまった場合、解析困難な不具合して問題が表面化してしまいます。

在 fork 之前, 让其他的线程完全终止的话, 则不会引起问题. 但这仅仅是可能的情况. 还有, 因为一些原因而其他线程不能结束就执行了 fork 的时候, 就会是产生出一些解析困难的不具合的问题.

回避方法２: fork 直後に子プロセスが exec を呼ぶようにする

规避方法 2:fork 后在子进程中马上调用 exec 函数

(2004/9/11　書き忘れていたので追記)

(2004/9/11　追记一些忘了写的东西)

回 避方法 1 が取れない場合は、子プロセスは fork 直後に、どんな関数(printf などを含む)も呼ばずにすぐに execl など、exec ファミリーの関 数を呼ぶようにします。もし、"exec しない fork"を一切使わないプログラムであれば、現実的な回避方法でしょう。

不用使用规避方法 1 的时候,在 fork 后不调用任何函数(printf 等)就马上调用 execl 等,exec 系列的函数.如果在程序里不使用”没有 exec 就 fork”的话,这应该就是实际的规避方法吧.

译者注:笔者的意思可能是把原本子进程应该做的事情写成一个单独的程序,编译成可执行程序后由 exec 函数来调用.

回避方法 3： 「他スレッド」では fork-unsafe な処理を一切行わない

规避方法 3:”其他线程”中,不做 fork-unsafe 的处理

fork を呼ぶスレッドを除く全てのスレッドが、fork-unsafe な処理を一切行わない方法です。数値計算の速度向上目的でスレッドを使用している場合*7などは、なんとか可能かもしれませんが、一般のアプリケーションでは現実的ではありません。~~どの関数が fork-safe なのか把握することだけでも容易ではないからです。~~fork-safe な関数、要するに非同期シグナルセーフな関数ですが、それは数えるほどしかないからです。この方法では malloc/new, printf すら使えなくなってしまいます。

除了调用 fork 的线程,其他的所有线程不要做 fork-unsafe 的处理.为了提高数值计算的速度而使用线程的场合*7,这可能是 fork-safe 的处理,但是在一般的应用程序里则不是这样的.~~即使仅仅是把握了那些函数是 fork-safe 的,做起来还不是很容易的.~~fork-safe 函数,必须是异步信号安全函数,而他们都是能数的过来的.因此,malloc/new,printf 这些函数是不能使用的.

回避方法 4： pthread_atfork 関数を用いて、fork 前後に自分で用意したコールバック関数を呼んでもらう

规避方法 4:使用 pthread_atfork 函数,在即将 fork 之前调用事先准备的回调函数.

pthread_atfork 関数を用いて、fork 前後に自分で用意したコールバック関数を呼んでもらい、コールバック内で、プロセスのデータ領域を掃除する方法です。しかし、OS 提供の関数(例: malloc)については、コールバック関数から掃除する方法がありません。malloc の使用するデータ構造は外部からは見えないからです。よって、 pthread_atfork 関数はあまり実用的ではありません。

使用 pthread_atfork 函数,在即将 fork 之前调用事先准备的回调函数,在这个回调函数内,协商清除进程的内存数据.但是关于 OS 提供的函数(例:malloc),在回调函数里没有清除它的方法.因为 malloc 里使用的数据结构在外部是看不见的.因此,pthread_atfork 函数几乎是没有什么实用价值的.

回避方法５：　マルチスレッドのプログラムでは、fork を一切使用しない

规避方法 5:在多线程程序里,不使用 fork

fork を一切使用しない方法です。fork するのではなく、素直に pthread_create するようにします。これも、回避策 2 と同様に現実的な方法であり、推奨できます。

就是不使用 fork 的方法.即用 pthread_create 来代替 fork.这跟规避策 2 一样都是比较实际的方法,值得推荐.

*1：子プロセスを生成するシステムコール

*1：生成子进程的系统调用

*2：グローバル変数や関数内の static 変数

*2：全局变量和函数内的静态变量

*3：Linux を使用するのであれば、pthread_atfork 関数の man page を見るとよいです。この種のシナリオについて若干の解説があります

*3：如果使用 Linux 的话,查看 pthread_atfork 函数的 man 手册比较好.关于这些流程都有一些解释.

*4：Solaris や HP-UX など

*4：Solaris 和 HP-UX 等

*5：fork 後 exec するまでの間

*5：从 fork 后到 exec 执行的这段时间

*6：≒execve システムコール

*6：≒execve 系统调用

*7：四則演算しか行わないなら fork-safe

*7：仅仅做四则演算的话就是 fork-safe 的

原文地址:http://d.hatena.ne.jp/yupo5656/20040715/p1