Linux とスレッド

Debian/Debian JP Project　後藤 正徳 <gotom@debian.or.jp>

本セッションの概要

マルチプロセッサシステムを有効に生かし、プロセス中で複数のタスクの制御を可能とするプログラミング手法として、スレッドが注目されている。近年 Linux においても様々なスレッド技術が利用可能となりつつある。本セッションでは、スレッドとは何かといった一般的・基礎的な事柄から概説する。続いて、カーネル・ライブラリ・各種言語においてどのようなスレッド機能が利用できるのかについて説明し、実際にプログラムする際の留意点や Linux に特有な話題まで、幅広く解説を行う。

1　スレッドとは

1 つのプログラムやプロセス中で、同時にいくつもの並行処理を行いたい場面のときに、とりわけ有効となるプログラミング手法、それがスレッドである。

Un*x では通常、プログラムはプロセス毎にわかれて、別々のメモリ空間を持ちながら独立して動作する。それぞれの制御の流れは 1 つだけである。そのため、制御を複数にわけて動作させたいときは`fork()`システムコールを実行することで子プロセスを生成し、各プロセスが相互にシグナルやセマフォ、共有メモリといった IPC (Inter Process Communication: プロセス間通信) を用いて通信させながら動作させる必要があった。

しかし、例えばサーバープログラムが複数のクライアントからの処理をこなさなければならない場合や、同時並行で様々な数値計算を行う場合、スレッドを用いることで比較的簡単にそれらが実現可能となる。以下にスレッドの利点と欠点をまとめる。

利点	欠点
○ 平行処理を比較的簡単に実現できる	× プログラムが煩雑になる
○ プロセス毎に管理されている資源を	× 変数の値を予想外に変更される危険性
そのプロセスに属すスレッド全てで共有可能	× スレッドセーフなライブラリが必要
○ コンテキストスイッチの負荷軽減	× 幾つかのプログラミング手法に制限が生じる
○ マルチプロセッサを有効に活用	× デバッグが困難

スレッドを使いこなすには、通常のプログラミングとは若干異なった概念が要求される。ここでは詳細には踏み込まないが、例えば生成・終了・合流の仕方や、同期・条件変数を用いたスレッド間の制御などの知識が必要であろう。また、スレッドを実際のプログラムで活用する場合は、クライアント/サーバー、生産者/消費者、リード/ライト、パイプライン、バリア、といった動作形態に関しても知っておくと、なお良いであろう。

2　Linux カーネルとスレッド

2.1　カーネルとスレッド

Linux カーネルとスレッドの関係を述べる前に、スレッドの実体がカーネル中でどのように表現されるかについて触れておきたい。これは、カーネル中におけるスレッド実体の実装方法によって、同じプログラムでもその速度や性能などが変わってくるためである。スレッド実体の実装方式は、大まかにわけて以下の 3 つに分類できるだろう。

- ユーザーレベル: アプリケーションやライブラリが、全てのスレッドを制御・スケジューリングする方式。
- カーネルレベル: カーネルが各スレッドを制御・スケジューリングする方式。
- 2 レベル: ユーザーレベルとカーネルレベルを組み合わせたもの。

	利点	欠点
ユーザレベル	○ スレッドの切り替えが高速 ○ ライブラリの移植性が良い	× あるスレッドがブロックされると、スレッド全体がブロックされる × マルチプロセッサに対応していない
カーネルレベル	○ あるスレッドがブロックされても、他がブロックされるわけではない ○ マルチプロセッサに対応	× スレッドの切り替えが重い × スケーリングが悪い × ライブラリの移植性が悪くなる
2 レベル	○ スレッドによってユーザ/カーネルレベルを組み合わせられる ○ あるスレッドがブロックしても、他がブロックされるわけではない ○ マルチプロセッサに対応	× スレッドの直接的な制御が難しくなる × 実装が複雑 × ライブラリの移植性が悪くなる

2.2 Linux カーネルのスレッド対応

現在の Linux カーネルは、スレッドに対して様々なサポートがされており、カーネルレベル実装の利用が可能である。元々 Linux の初期バージョンでは、スレッドに対して何ら考慮がされていなかった。しかしカーネル 2.0 からは、カーネルレベル実装を実現する `__clone()` システムコールが新たにサポートされた。また 2.2 からは、スレッド用のシグナルが新たに提供されるようになった。

最新バージョンである 2.3 系でも、例えば SYSV 共有メモリのスレッド対応や、大量スレッド生成への対応など、カーネル内部のスレッド対応化作業が続いている。また `/proc/sys/kernel/threads-max` にかかれた数値を変えることで、最大生成可能プロセス (スレッド) 数をコントロールできるようになるなど、便利な機能も増えつつある。

2.3 clone() システムコール

`__clone(2)` システムコールとは `fork(2)` システムコールと同様に子プロセスを生成するための関数である。`fork()` と異なる点は、子プロセスのメモリ空間、ファイルディスクリプタ、シグナルハンドラテーブルといったコンテキストのいづれかを親プロセスと共有可能にするか指定できることにある。この `clone()` システムコールを使ったスレッドライブラリを利用すれば、カーネルレベル実体を持つスレッドを生成することが可能となる。

3 Pthread

Pthread とは正式名称を POSIX 1003.1c-1995 と呼び、スレッドを操作するための API を規定した規格である。この Pthread を元にしたスレッドライブラリは、広く使われており、多くの Un*x で利用可能である。特徴としては、POSIX や ISO/IEC, XPG 等で標準化されていること、汎用性を持つこと、Pthread をサポートするシステム間での移植が容易なこと、などである。

Linux でも、様々な種類の Pthread ライブラリが利用可能である。その中でも、最も広く使われている代表的な Pthread ライブラリが LinuxThreads であろう。元々 Xavier Leroy 氏によって開発されていたが、現在は glibc の一部として多くのディストリビューション中に含まれる形で配布されている。また、かなり前から、それまでのユーザーレベルから `__clone()` システムコールを用いたカーネルレベル実装へ移行している。ただし、幾つかのあまり使われない関数がまだ実装されていないなど、改良すべき点がいくつかある。現在も活発に開発が続けられているため、出来れば最新版を利用する方が良いだろう。

4 スレッドと各言語の関係

各言語毎に、スレッドを利用する方法は異なっている。例えば C/C++ では Pthread ライブラリ中に含まれている関数を利用することが一般的だろう。言語によっては、元々の言語仕様中にスレッド機能が既に組み込まれているものもあり、Java や並列処理言語のいくつかがそれにあたる。最近では Perl 5.6 にスレッド機能が導入されるなど、スレッドが利用できる言語の数は、今後ますます増えていくと考えられる。

ただし、ライブラリや言語によって、使い方や概念などが違う場合があり、この点には注意されたい。また Linux で使う場合も、言語仕様やそれぞれの実装によって、スレッド実体がどのように表現されているか異なる点にも気をつけたい。例えば Java にはグリーンスレッド版とネイティブスレッド版があり、それぞれユーザレベルとカーネルレベルという別々の実装方式が利用可能である。

5 スレッドに関する有用な情報

規格	
POSIX Pthread	`POSIX 1003.1c-1995` など
Unix98/SUSv2	`http://www.opengroup.org/`
ISO/IEC	`ISO/IEC 9945-1:1996`
ライブラリ	
スレッドライブラリリスト	`http://www.gnu.org/software/pth/related.html`
LinuxThreads	`http://pauillac.inria.fr/~xleroy/linuxthreads/index.html`
glibc	`http://www.gnu.org/software/libc/` (配布アーカイブ中の文章などが参考になる)
PTL	`http://www.media.osaka-cu.ac.jp/~k-abe/PTL/index-ja.html`
FSU Pthreads	`http://www.cs.fsu.edu/~mueller/pthreads/`
GNU Pth	`http://www.gnu.org/software/pth/`
その他	
Thread Information	`http://www.media.osaka-cu.ac.jp/~k-abe/PTL/thread-info-ja.html` (日本語)
comp.programming.threads	`http://www.serpentine.com/~bos/threads-faq/`
Linux Threads FAQ	`http://linas.org/linux/threads-faq.html` (若干古い)

Pthread プログラミング例

```
----------- コンパイルと実行結果 -----------
$ gcc -o test test.c -lpthread -D_REENTRANT
$ ./test
Type some character:
abcde
-- Thread A: abcde
fghij
-- Thread B: fghij
exit A!
exit B!
-----------------------------------------
```

---------- プログラム: test.c -------------

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

/* スレッド関数 A */
void *thread_functA (char *arg)
{
  printf ("-- Thread A: %s", arg);
  sleep(1);
  return NULL;
}

/* スレッド関数 B */
void *thread_functB (char *arg)
{
  printf ("-- Thread B: %s", arg);
  sleep(2);
  return NULL;
}

/* メイン関数 */
int main(void)
{
  int s;
  char buf[256];
  pthread_t thread_idA, thread_idB;

  printf("Type some character:\n");
  fgets( buf, 256, stdin );
  /* スレッド A を生成 */
  s = pthread_create( &thread_idA, NULL, (void*)thread_functA, buf );
  if( s != 0 ) { printf("create error!\n"); exit(-1); }
  fgets( buf, 256, stdin );
  /* スレッド B を生成 */
  s = pthread_create( &thread_idB, NULL, (void*)thread_functB, buf );
  if( s != 0 ) { printf("create error!\n"); exit(-1); }

  /* スレッド A と合流 */
  s = pthread_join( thread_idA, NULL );
  if( s != 0 ) { printf("join error!\n"); exit(-1);
  } else {       printf("exit A!\n");  }

  /* スレッド B と合流 */
  s = pthread_join( thread_idB, NULL );
  if( s != 0 ) { printf("join error!\n"); exit(-1);
  } else {       printf("exit B!\n");  }
  return 0;
}
-------------------------------------------
```