POSIXスレッド（４）

システムプログラミング

2009年11月9日

建部修見

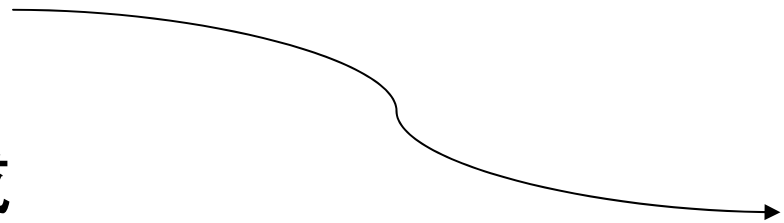
Thread-Specific Data (TSD)

- スレッド単位で別々のデータを持つ仕組み
- 内部の静的データのポインタを返すなどの関数を，スレッドセーフ化できる

- TSDは，ポインタ（void *）の集合であり，ポインタは各スレッドごとにある
  - ポインタをTSDの値という
- TSDはkeyにより管理される
- keyは全スレッドで共有され，そのスレッド専用のポインタを参照，設定するのに利用される

TSDの操作

key



TSD = array of void *

P1 (for thread 1)
P2 (for thread 2)
P3 (for thread 3)
. . .
P*n* (for thread *n*)

TSDの作成

```
pthread_key_t key;
int pthread_key_create(&key, destructor);
```

TSDへの代入

```
int pthread_setspecific(key, address);
```

TSDの参照

```
void *pthread_getspecific(key);
```

TSD keyのallocate（１）

- int pthread_key_create(pthread_key_t *keyp, void (*destructor)(void *));
- 新しいkeyが*keypに返される
- 各スレッドのポインタ（TSDの値）はNULLで初期化される
- システムがサポートするkeyの数の最大値はlimits.hのPTHREAD_KEYS_MAXで定義される
  - 少なくとも128以上
- 実際の最大値はsysconf(_SC_THREAD_KEYS_MAX)で分かる

TSD keyのallocate（２）

- destructorはオプションで指定できる
- スレッドが終了したとき，TSDの値がNULLでなければTSDの値を引数としてdestructorが呼ばれる
  - mallocしたTSDの値をfreeする
- exit(), _exit(), abort()で終了した場合，destructorは呼ばれない
- destructorを呼んだ後，TSDの値を再びチェックしNULLでなければ再びdestructorが呼ばれる
  - 少なくともPTHREAD_DESTRUCTOR_ITERATIONS回は反復される
    - すくなくとも4
  - 実際の値はsysconf(_SC_THREAD_DESTRUCTOR_ITERATIONS)でわかる

TSDの値の参照，更新

- int pthread_setspecific(pthread_key_t key, const void *value);
- 指定したkeyのTSDの値にvalueをsetする
- keyが不正な場合，allocateされていない場合はエラー番号を返す
- void *pthread_getspecific(pthread_key_t key);
- 指定したkeyのTSDの値を返す

TSDの例（１）

```
static pthread_once_t once =
    PTHREAD_ONCE_INIT;

/* key for per-thread data */
static pthread_key_t key;

/* per-thread data */
typedef struct {
   ....
} data_t;
```

```
static void
init(void)
{
   /*
    * allocate the key for a TSD.
    * free() is used to destroy the
    * allocated per-thread data.
    */
   if (pthread_key_create(&key,
    free) != 0) {
     fprintf(stderr, “cannot create
    key¥n”);
     exit(1);
   }
}
```

TSDの例（２）

```
static data *
getdata()
{
   data_t *datap;

   /* allocate a key for a TSD only once */
   pthread_once(&once, init);
   /* get the pointer to data for this thread */
   datap = pthread_getspecific(key);
   if (datap == NULL) {
      /* first time called in this thread, allocate data */
      datap = malloc(sizeof(*datap));
      pthread_setspecific(key, datap);
   }
   return (datap);
}
```

TSD keyの消去

- int pthread_key_delete(pthread_key_t key);
- 指定されたkeyを消去する
- 注意:どれかのスレッドのTSDの値がNULLでなくても，destructorは呼ばれない

スレッドの取消

- 長い計算処理を別スレッドで計算していたが，ユーザの指示により取り消されたため，その計算が不要となった
- 複数スレッドでのデータベース検索。どれかのスレッドで目的とするレコードを発見したため，他のスレッドは不要となった

- 解決法１：終了フラグの利用

終了フラグの利用（1）

```
static int terminate_flag = 0;
static pthread_mutex_lock lock =
    PTHREAD_MUTEX_INITIALIZER;

int
does_terminate()
{
   int t;
   pthread_mutex_lock(&lock);
   t = terminate_flag;
   pthread_mutex_unlock(&lock);
   return (t);
}
```

```
void
terminate()
{
   pthread_mutex_lock(&lock);
   terminate_flag = 1;
   pthread_mutex_unlock(&lock);
}

/* thread that can be terminated */
void
do_something()
{
   /* regularly check the flag */
   while (!does_terminate()) {
      …. do something
   }
}
```

終了フラグの利用(2)

- 定期的な終了フラグのチェックが必要
  - 誰か他の人の書いたライブラリには適用できない
- プログラムが外部イベントを待っているとき，定期的にチェックできない
  - チェックするために，タイムアウトを設定して待つなどが必要となる
- 簡単であるが，小規模プログラム向き

- 解決法2：非同期シグナルの利用

非同期シグナルの利用（1）

```
jmp_buf cancel;
static pthread_mutex_t lock =
    PTHREAD_MUTEX_INITIALIZER;
static pthread_once_t once =
    PTHREAD_ONCE_INIT;

/* signal handler to terminate thread
    */
static void
handler_usr1(int sig)
{
   /* nonlocal jump to a cleanup
     routine */
   longjmp(cancel, 1);
}
```

```
/* once function to initialize some
    variables */
static void
init()
{
   struct sigaction sa;

   sa.sa_handler = handler_usr1;
   sigemptyset(&sa.sa_mask);
   sa.sa_flags = 0;
    /* init SIGUSR1 handler */
   sigaction(SIGUSR1, &sa, NULL);
}
```

非同期シグナルの利用（2）

```
/*
 * thread function to do something.
 * SIGUSR1 assumed to be masked.
 */
void *
do_something(void *arg)
{
   sigset_t nset;

   /* initialize */
   pthread_once(&once, init);

   /* set SIGUSR1 to a signal set */
   sigemptyset(&nset);
   sigaddset(&nset, SIGUSR1);
```

```
if (setjmp(cancel) == 0) {
      pthread_sigmask(SIG_UNBLOCK,
     &nset, NULL);
      /* do something */
      fd = open(file, …);
      data = malloc(sizeof(*data));
      ….
      close(fd);
      do_morething(data);
   }
   else {
      /* cleanup from signal */
      pthread_mutex_lock(&lock);
      /* free all allocated memory */
      pthread_mutex_unlock(&lock);
   }
   return (NULL);
}
```

非同期シグナルの利用（3）

- do_morething()の中でlockをロック中にシグナルが到着すると，シグナルのクリーンアップコードでdeadlockする
- malloc, pthread_mutex_{,un}lockはasync-safeではないため，実行中にシグナルが到着するとそれ以降利用できなくなる
- 解決法：シグナルが到着してもよい場所だけSIGUSR1をアンブロックする
  - 常に意識してプログラムする必要があり，エラーが生じやすい
  - 他のプログラムが利用できない
- 様々な問題があるため，POSIXスレッドでは，スレッドを取消す機構がある

スレッドの取消

- int pthread_cancel(pthread_t thread);
- threadで指定したスレッドに実行終了の要求を出す
- キャンセルの要求を出すだけで，終了するまでは待たない
- キャンセルポイント
  - スレッドは直ちに終了するわけではなく，キャンセルポイントまでは通常通りに実行する
  - この種のキャンセルはdeferred cancellabilityとよばれる
- 注意：キャンセルポイントとなる関数が（定期的に）実行されている必要がある

必須POSIXキャンセルポイント関数

- aio_suspend, close, creat, fcntl(, F_SETLKW,), fsync, mq_{send,receive}, msync, nanosleep, open, pause, pthread_cond_{,timed}wait, pthread_join, pthread_testcancel, read, sem_wait, sigsuspend, sig{,timed}wait, sigwaitinfo, sleep, system, tcdrain, wait, waitpid, write
- （ブロックする可能性のある関数）

注意点（1）

- キャンセルポイントでスレッドの実行が取り消された場合，副作用がおこる可能性がある
  - read()の呼び出し中で取り消された場合，既にデータを読んでしまっている可能性がある
  - その場合，次のread()では，そのデータは読み出せない
- 一般的に，副作用はシグナル割込によりEINTRが返されたときと同様

注意点（2）

- キャンセルポイント関数では，いつもキャンセルのチェックをする必要はなく，ブロックするときにチェックをすればよいと定められている
  - ブロックしない場合の余計なオーバヘッドを削減できる
- pthread_mutex_lockはブロックする可能性があるが，キャンセルポイント関数ではない
  - mutexは短いクリティカルセクションにしか利用されないことを想定
- 実際には必須（選択）POSIXキャンセルポイント関数以外の関数もキャンセルポイントとなりうる
  - EINTRを返す関数はキャンセルポイントとなりやすい

キャンセルクリーンアップハンドラ（１）

- Deferred cancellationにより，非同期キャンセルの問題は解決したが，キャンセル後の中間的な状態のクリーンアップは依然必要
- void pthread_cleanup_push(void (*handler)(void *), void *arg);
- pthread_cleanup_push()は，指定されたhandlerとargをスレッドごとのLIFOクリーンアップハンドラスタックにプッシュする
- スレッドが（pthread_exit()を呼ぶか）キャンセルされると，スタックに積まれたハンドラを（LIFO）順に実行し，thread-specific data destructorsが実行され，PTHREAD_CANCELEDの終了状態で終了する
- exit(), _exit(), abort()で終了した場合はクリーンアップハンドラは呼ばれない

キャンセルクリーンアップハンドラ（２）

- void pthread_cleanup_pop(int execute);
- pthread_cleanup_pop()は，クリーンアップスタックの最も最近にプッシュされたハンドラをポップする
- executeが0ではない場合は，そのハンドラを実行する
- pthread_cleanup_pushとpopは同一スコープにある必要がある
  - ‘{‘と’}’に囲まれた範囲
  - マクロで実装されている場合があるため
  - pthread_cleanup_pushとpopの間から外にジャンプしたり，間に戻ったりしてはならない

クリーンアップハンドラの例

```
/*
 * cleanup handler
 * it just call free
 */
void
cleanup(void *bufp)
{
   free(bufp);
}
```

```
int
checksum(int fd)
{
   char *bufp;
   int nbutes, i, sum = 0;

   bufp = malloc(4096);
   /* push a cleanup handler to free bufp */
   pthread_cleanup_push(cleanup, bufp);
   while ((nbytes = read(fd, bufp, 4096)) > 0) {
      for (i = 0; i < nbytes; ++i)
         sum += (int) bufp[i];
   /* pop a cleanup handler and execute it */
   pthread_cleanup_pop(1);
   return (sum);
}
```

クリーンアップハンドラの注意点

- 非局所gotoを除きほとんど制限がない
- 一度スレッドがキャンセルされたら，再びキャンセルされることはない
  - 割り込まれることはない

クリーンアップハンドラと条件変数

- Pthread_cond_wait()あるいはpthread_cond_timedwait()の途中でキャンセルされた場合，mutexを獲得してからクリーンアップハンドラが呼ばれる

```
void
a()
{
   pthread_mutex_lock(&lock);
   pthread_cleanup_push(cleanup, …);
   while (!condition)
      pthread_cond_wait(&cond, &lock);      /* cancellation point */
   read(…);                                 /* cancellation point */
   pthread_cleanup_pop(0);
   pthread_mutex_unlock(&lock);
}
```

- cleanupは，lockが獲得されていることを仮定してよい

スレッドの取消の例

- スレッドを取り消す場合，以下のように呼ぶ

```
pthread_cancel(thread);
pthread_join(thread, &result);
```

- 取り消された場合，resultには PTHREAD_CANCELEDが返される

キャンセル中のスレッドのデタッチ

- pthread_joinはキャンセルポイント関数
- pthread_joinでキャンセルされた場合，待っていたスレッドの終了を待ってキャンセルするより，クリーンアップハンドラで待っていたスレッドをpthread_detachでデタッチするほうがよい

キャンセルのチェック

- void pthread_testcancel(void);
- キャンセルが要求されたかチェックする
- 計算バウンドなプログラムやキャンセルポイントがなかなかない場合に利用する
- 512反復ごとにキャンセルをチェックする例

```
for (i = 0; i < size; ++i) {
    … do something
    if ((i % 512) == 0)
        pthread_testcancel();
}
```

非同期スレッドキャンセル

- 非同期シグナルによるスレッドのキャンセルの例は様々な落とし穴があったが，有用なときがある
- int pthread_setcanceltype(int type, int *oldtypep);
- typeがPTHREAD_CANCEL_ASYNCHRONOUSのとき，非同期キャンセル型とする
- *oldtypepにはこれまでのキャンセルの型がはいる
- typeがPTHREAD_CANCEL_DEFERREDのときキャンセルポイントまでキャンセルは遅延される

非同期スレッドキャンセルの例

- pthread_testcancelを入れるとコードが読みにくくなる場合

```
pthread_setcanceltype(PTHREAD_CANCEL_ASYNCHRONOUS,
    &old);
for (i = 0; i < size; ++i)
   … do something
pthread_setcanceltype(old, &old);
```

- この例ではforループはキャンセルポイントとなる
- Async-cancel safe

スレッドキャンセルのdisable

- int pthread_setcancelstate(int state, int *oldstatep);
- stateがPTHREAD_CANCEL_DISABLEのとき，スレッドのキャンセルをできなくする
- stateがPTHREAD_CANCEL_ENABLEのとき，スレッドキャンセルをできるようにする
- disable中は，スレッドキャンセルのリクエストは延期される
- 複雑なキャンセルハンドラが必要な場合などに利用される

スレッドキャンセルについて

- スレッドキャンセルの仕組みはきれいにスレッドを終了させることができる
  - が，依然さまざまな注意点がある
  - クリーンアップハンドラを正しく設定するなどの必要がある
  - 通常のライブラリはキャンセルに関して正しく動作しない
  - ライブラリを利用するときはdisableする方がよい
- 一般的には，汎用ライブラリはdeferred cancellationという意味でのcancellation safeである方がよい