POSIXスレッド（２）

システムプログラミング

2009年10月26日

建部修見

スレッドセーフな関数

- マルチスレッドセーフ、MTセーフ、reentrant（リエントラント、再入可能）ともいう
- 同時に複数のスレッドで呼出しても良い関数
  - 呼出側で何もしなくてもよい
- ただし、呼出側で管理しているメモリ領域は守られない
  - 複数のスレッドがmemcpyで同じ領域にコピー
  - 別のスレッドがmemcpyのコピー元を変更
- 呼出側で見えない（opaque）オブジェクト（FILE構造体など）は守られる
  - 同じFILEに対してfprintfで同時に呼出せる

POSIXスレッドセーフ関数

- ほとんどの関数はスレッドセーフであるが、例外がある
- 本質的にスレッドセーフにできない関数
  - 関数内部のstaticバッファの値を更新してそのポインタを返す
- asctime_r, ctime_r, getgrgid_r, getgrnam_r, getlogin_r, getpwnam_r, getpwuid_r, gmtime_r, localtime_r, rand_r, readdir_r, strtok_r, ttyname_r

スレッドセーフな関数の補足

- メモリ確保
  - malloc, calloc, reallocはスレッドセーフである
  - すべてのスレッドで共有されるヒープ領域にメモリを確保する
  - 他のスレッドで確保した領域を別のスレッドで利用しても問題ない
- 標準入出力
  - 標準入出力関数はスレッドセーフである
  - 同じFILE構造体への呼出はシリアライズされる
    - まざらない
  - 複数の呼出はflockfileなどでアトミックにできる

標準入出力関数のロック

void **flockfile**(FILE *file);
void **ftrylockfile**(FILE *file);
void **funlockfile**(FILE *file);
  - Mutexと同じでlockしたスレッドだけがunlockできる

- ロックされていることを仮定している関数

int **getc_unlocked**(FILE *file);
int **getchar_unlocked**(void);
int **putc_unlocked**(int c, FILE *file);
int **putchar_unlocked**(int c);
  - 最内ループなどロックのオーバヘッドが問題になるところで、外側でロックして利用することを想定

プロセス資源

- スレッドはプロセス資源を共有する
  - Process address space
  - File descriptor table
  - Timers
  - User ID
- スレッドは異なるmachine state (general register, program counter, stack pointer), signal mask, errnoなどのthread-specific dataを持つ

ファイル読込の例

- 複数のスレッドで同一ファイルをreadで読み込むと同一内容を複数読むことがある
- lseek+write, lseek+readで位置を指定して読み込むのも、lseekとの間でファイルポインタの位置が競合
- Positioned I/O操作を利用する
- ssize_t **pread**(int fd, void *buf, size_t nbytes, size_t offset);
- ssize_t **pwrite**(int fd, void *buf, size_t nbytes, size_t offset);
  - 注：POSIXでは規定されていない

プロセス生成

- fork()は呼び出したスレッドだけを含む別プロセスを生成する
- exit(), _exit(), exec()はすべてのスレッドを破壊する
- fork()における注意点
  - メモリがコピーされるため、他スレッドでlockされているmutexはlockされたままとなる。かつ、unlockできない

シグナル

- 各スレッドは固有のシグナルマスクを持つ
- int pthread_sigmask(int how, const sigset_t *set, sigset_t *oset);
- how
  - SIG_BLOCK – スレッドのシグナルマスクに追加
  - SIG_UNBLOCK – スレッドのシグナルマスクから削除
  - SIG_SETMASK – スレッドのシグナルマスクを置き換え
- set
  - sigemptyset(), sigfillset(), sigaddset(), sigdelset()で生成
- マルチスレッドプログラムではプロセスに対するシグナルマスクはない
  - sigprocmask()は使ってはいけない

シグナル生成（1）

- 例外(traps)
  - SIGSEGV, SIGBUS, SIGFPE, SIGILL
  - 同期的に発生
- 割り込み(interrupts)
  - SIGINT, SIGHUP, SIGQUIT, SIGIO
  - kill(), sigsend()
  - 非同期的に発生

シグナル生成（2）

- int pthread_kill(pthread_t thread, int signal);
- 特定のスレッドにシグナルを送る
- シグナルがマスクされていた場合，アンマスクされるまでシグナル配送は遅らされる
- 同一プロセスのスレッドだけ配信される
- signalが0のとき，シグナルは配送されない
  - スレッドが生きているかどうか知るため

シグナル配送

- スレッドはシグナルのアクションを共有する
  - signal, sigactionなどで設定
- アクションがSIG_DFL, SIG_IGNのとき，全スレッドでアクション（exit, core dump, stop, continue, or ignore）が実行される
  - Stopの場合，全スレッドがstop。Exitの場合プロセスが終了。
- 例外が発生した場合，そのスレッドに配送される
- 非同期的に発生したシグナルは，マスクしていないどれかのスレッドに配送される
  - 独立な割り込みを複数スレッドで同時処理可能
- 全スレッドがマスクしていた場合，どれかのスレッドがアンマスクするまで配送が遅延される

シグナルハンドラ

- シグナル処理の方法
  - シグナルハンドラかシグナルを待つか
- シグナルハンドラはスレッドで共有
- シグナルハンドラは，マルチスレッドプログラムでは，通常同期的に発生するシグナルに用いられる
- 非同期的に発生するシグナルの処理は，専用スレッドで待つようにした方がよい
  - 非同期シグナルハンドラ実行中のプロセス状態の維持は大変煩雑（後述）

Async-safe関数

- シグナルハンドラで呼んでも良い関数
- 割り込み可能，再入可能
- Fork-safeでもある
- _exit, access, aio_{error,return,suspend}, alarm, cf{get,set}[io]speed, chdir, chmod, clock_gettime, close, creat, dup2, dup, execle, execve, fcntl, fdatasync, fork, fstat, fsync, get{e,}[gu]id, getgroups, get{p,}pid, kill, link, lseek, mkdir, mkfifo, open, pathconf, pause, pipe, read, rename, rmdir, sem_post, set{p,}gid, setsid, setuid, sig{action,addset,delset,emptyset,fillset,ismember,pending,procmask,suspend}, sleep, stat, sysconf, tc{drain,flow,flush,getattr,getpgrp,sendbreak,setattr,setpgrp}, time, timer_{getoverrun,gettime,settime}, times, umask, uname, unlink, utime, wait, waitpid, write

Async-safe関数（２）

- Async-unsafe関数でシグナルが割り込まれない事が保証されている場合，シグナルハンドラでasync-unsafe関数をよんでもよい
  - ある特定のところでシグナルマスクをはずす

非局所goto (nonlocal goto)

- setjmp(), sigsetjmp(), longjmp(), siglongjmp()のスコープはスレッドに限られる
  - setjmp(), sigsetjmp()でjmp_bufを初期化したスレッドが同じjmp_bufでlongjmp(), siglongjmp()を呼ぶ必要がある
- （逐次プログラムと同様に）非同期シグナルハンドラでのlongjmp()の利用は注意が必要
  - (mallocなど) Async-safeではない関数の途中で割り込まれ，longjmp()すると内部状態の一貫性が保たれなくなる。（malloc, freeは利用できなくなる）
  - Async-unsafe関数を呼ぶ場合はシグナルをブロックする必要がある

シグナルを待つ

- 非同期シグナルをより簡単に，安全に処理するためには，すべてのスレッドでシグナルをブロックし，専用スレッドで待つこと
- “ハンドラ”スレッドはasync-safe関数に制限されない
  - シグナルハンドラではないため
- 通常のスレッド関数で他のスレッドと協調動作可能

シグナルを待つ（２）

- #include <signal.h>
- int sigwait(const sigset_t *set, int *signalp);
- Setに含まれるペンディングシグナルを待つ
- 待っている間はsetに含まれるシグナルはアンブロックされる
- シグナルを受け取ると，そのシグナルをクリアし，シグナルマスクを元に戻し，*signalpにシグナル番号を返す
- sigwaitを複数スレッドで呼び出した場合，どれかのスレッドがsigwaitから戻る

シグナルを待つ（3）

- sigwait()を（複数の）専用スレッドで呼び出す
- 他のスレッドではシグナルマスクする

非同期シグナル処理の例

```
pthread_mutex_t m = PTHREAD_MUTEX_INITIALIZER;
int hup = 0;
sigset_t hupset;
/* signal hander thread. We're not restricted to async-safe
      functions since we're in a thread context */
void *
handle_hup(void *arg)
{
   int sig, err;

   err = sigwait(&hupset, &sig);
   if (err || sig != SIGHUP);
      abort();
   pthread_mutex_lock(&m);
   hup = 1;
   pthread_mutex_unlock(&m);
   return (NULL);
}
```

```
int main()
{
   pthread_t t;

   sigemptyset(&hupset);  /* initialize set to empty
      */
   sigaddset(&hupset, SIGHUP);    /* add SIGHUP */
   /* block signals in initial thread. New threads will
      inherit this signal mask */
   pthread_sigmask(SIG_BLOCK, &hupset, NULL);
   pthread_create(&t, NULL, handle_hup, NULL);
   for (;;) {
      … do stuff
      pthread_mutex_lock(&m);
      if (hup) {
         … cleanup
         break;    /* got SIGHUP. We're done. */
      }
      pthread_mutex_unlock(&m);
   }
   return (0);
}
```