POSIXスレッド（１）

システムプログラミング

2009年10月19日

建部修見

組込機器における並行処理

- GUIにおける反応性向上
  - ダイナミックなWaitカーソル
  - 各イベントを別制御で実行
  - Auto-save機能
- サーバの反応性向上
  - 各リクエストを別制御で実行
- マルチコア、マルチプロセッサでの並列実行

スレッドとは？

- プロセス内の＊独立した＊プログラム実行
  - 同一プロセスID
  - 注：LinuxThreadsは実装がプロセスなため異なる
- 論理メモリ空間を共有
- ファイルディスクリプタなどプロセス資源を共有
- 一般にスレッド生成はプロセス生成より軽い

プロセスvsスレッド

	スレッド	プロセス
生成、実行オーバヘッド	小	大
メモリ	共有	別々
プロセス資源	共有	別々
データ共有	メモリのポインタ渡し(コピーは不必要)	パイプ、ソケット、ファイルなど
保護	他スレッドのメモリ、資源を破壊する可能性あり。スレッドの実行制御が必要	他プロセスのメモリ、資源は保護される

その他のスレッドの便利な利用例

- RPC（遠隔手続き呼出）の遅延隠蔽
  - 別スレッドでRPCを発行、非同期IOと同等の処理
- プログラム構造の単純化
  - 表計算における入力処理（スレッド１）と合計値などの値更新（スレッド２）
  - スレッド1は入力を処理、スレッド2は変更を待ち変更があれば更新

スレッドを利用すべきでない例

- 逐次的な処理自体の高速化が必要な場合
  - 逐次的な数値計算
  - 逐次的なIO処理（ファイルの連続読込、連続書込）

→マルチスレッド処理はオーバヘッドとなる

並列性の制御

- スレッド間でメモリ、プロセス資源を共有しているため、破壊してしまう可能性がある
- 例：共有カウンタのインクリメント

counter++ →

1. counterの値をレジスタにロード
2. レジスタの値をインクリメント
3. レジスタの値をcounterにストア

スレッド1

1. counterの値をレジスタにロード
2. レジスタの値をインクリメント
3. レジスタの値をcounterにストア

スレッド2

1. counterの値をレジスタにロード
2. レジスタの値をインクリメント
3. レジスタの値をcounterにストア

スライドショウのプログラム例

```
main(int argc, char *argv[]) {
   for (i = 1; i < argc; ++i) {
      if (i == 1)
         /* get the first picture */
         buf = get_next_pic(argv[i]);
      else
         /* wait for the previous process */
         pthread_join(tid, &buf);
      if (i < argc – 1)
         /* get the next picture */
         pthread_create(&tid, NULL,
               get_next_pic, argv[i + 1]);
      display_buf(buf); /* display the picture */
      free(buf);
      if (getchar() == EOF)
         break;
   }
}
```

表示している合間に次の画像を読込

スレッドの生成、終了、終了待ち

- #include <pthread.h>
- int **pthread_create**(pthread_t *threadp,
  const pthread_attr_t *attr,
  void *(*start_routine)(void *), void *arg);
- void **pthread_exit**(void *status);
- void **pthread_join**(pthread_t thread,
  void **statusp);

Detachedスレッド

- スレッドの終了状態が必要ないスレッド
  - pthread_joinの対象とできない

```
pthread_attr_t attr;

pthread_attr_init(&attr);
pthread_attr_setdetachstate(&attr,
        PTHREAD_CREATE_DETACHED);
pthread_create(&t, &attr, func, NULL);
```

```
int pthread_detach(pthread_t thread);
```

同期

- Mutexロック
- pthread_mutex_t lock;
- int **pthread_mutex_init**(
  pthread_mutex_t *lock,
  const pthread_mutexattr_t *attr);
- pthread_mutex_t lock =
  PTHREAD_MUTEX_INITIALIZER;
- int **pthread_mutex_destroy**(
  pthread_mutex_t *lock);

Mutexロック（２）

- int **pthread_mutex_lock**(pthread_mutex_t *mp);
- int **pthread_mutex_unlock**(
  pthread_mutex_t *mp);
- int **pthread_mutex_trylock**(
  pthread_mutex_t *mp);

- Mutexをロックしたスレッドがオーナとなる
- オーナ以外はmutex_unlockできない
- Trylockはlockできる場合はlockし、できない場合はEBUSYを返す

データ競合 (data race)

- データ競合の定義
  - 他のスレッドがアクセス可能な領域を、同時にあるスレッドが修正してしまう可能性のある場合、プログラムはデータ競合があるという

- Mutexロックなどを利用して同期が必要
- データ競合のないプログラムをデータ競合フリーという
  - 共有変数を変更する場合、他スレッドがアクセスできないことを保証

共有カウンタのインクリメントの例

```
pthread_mutex_t count_mutex = PTHREAD_MUTEX_INITIALIZER;
int count ;

void increment_count() {
    pthread_mutex_lock(&count_mutex); // ensure atomic increment
    count++;
    pthread_mutex_unlock(&count_mutex);
}

int get_count() {
    int c;
    pthread_mutex_lock(&count_mutex);  // guarantee memory is synchronized
    c = count;
    pthread_mutex_unlock(&count_mutex);
    return (c);
}
```

条件変数 (condition variables)

- 条件が満たされるまで待つことに利用
- 基本操作
  - (条件が満たされたときに)シグナルを送る
  - (条件が満たされていなければ)シグナルが送られるのを待つ
- 動作例
  - 一つ以上のスレッドが条件変数で待つ
  - 条件変数にシグナルが送られたら、どれかのスレッドが実行を開始
- 誰も待っていない条件変数にシグナルが送られても無視される(状態を持たない)

条件変数の初期化

- pthread_cond_t cond;
- int **pthread_cond_init**(pthread_cont_t *cond,
  const pthread_condattr_t *attr);
- pthread_cont_t cond =
  PTHREAD_COND_INITIALIZER;
- int **pthread_cond_destroy**(
  pthread_cond_t *cond);

シグナルの待機、送信

- int **pthread_cond_wait**(pthread_cond_t *cond,
  pthread_mutex_t *mutex);
- int **pthread_cond_signal**(pthread_cond_t *cond);
- int **pthread_cond_broadcast**(
  pthread_cond_t *cond);

- cond_waitはmutexをunlockしてcondにシグナルが送信されるまで待つ（アトミックな操作）
- シグナルが送信されたら、mutexをlockして実行を再開する

巡回バッファ

```
#define QSIZE 10                        /* キューの長さ */
typedef struct {
    pthread_mutex_t buf_lock;           /* 構造体のロック */
    int start;                          /* バッファの開始 */
    int num_full;                       /* データの数 */
    pthread_cont_t notfull;             /* notfullの条件変数 */
    pthread_cont_t notempty;            /* notemptyの条件変数 */
    void *data[QSIZE];                  /* 巡回バッファ */
} circ_buf_t;
```

Puts new data on the queue

```
void put_cb_data(circ_buf_t *cbp, void *data) {
    pthread_mutex_lock(&cbp->buf_lock);
    /* wait while the buffer is full */
    while (cbp->num_full == QSIZE)
        pthread_cond_wait(&cbp->notfull, &cbp->buf_lock);
    cbp->data[(cbp->start + cbp->num_full) % QSIZE] = data;
    cbp->num_full++;
    /* let a waiting reader know there's data */
    pthread_cond_signal(&cbp->notempty);
    pthread_mutex_unlock(&cbp->buf_lock);
}
```

巡回バッファを操作するときはmutexロックする

バッファが一杯の場合条件変数notfullで待つ

データを挿入，条件式の更新

条件変数notemptyにシグナルを送信

mutexロックをアンロック

条件成立をwhile文で待つ理由

cond_waitを抜けたからといって直後にMutexロックを獲得できるのはースレッドのみ。そのスレッドが条件式を更新する可能性があるため。

Gets the oldest data in circular buffer

```
void *get cb_data(circ_buf_t *cbp) {
    void *data;

    pthread_mutex_lock(&cbp->buf_lock);
    /* wait while there's nothing in the buffer */
    while (cbp->num_full == 0)
        pthread_cond_wait(&cbp->notempty, &cbp->buf_lock);
    data = cbp->data[cbp->start];
    cbp->start = (cbp->start + 1) % QSIZE;
    cbp->num_full--;
    /* let a waiting writer know there's room */
    pthread_cond_signal(&cbp->notfull);
    pthread_mutex_unlock(&cbp->buf_lock);
    return (data);
}
```

巡回バッファを操作するときはmutexロックする

バッファが空の場合条件変数notemptyで待つ

データを取出す，条件式の更新

条件変数notfullにシグナルを送信

mutexロックをアンロック

論理条件(式)と条件変数

論理条件	式	条件変数
FIFOキューが空	cbp->num_full == 0	cbp->notempty
FIFO キューが一杯	cbp->num_full == QSIZE	cbp->notfull

- それぞれの条件に対して別々の条件変数を使う必要はないが、その場合、pthread_cond_waitで不必要に起こされてしまう可能性がある
- 条件変数の処理は比較的軽いため、無理に共有する必要はない
- mutexをunlockする前にsignalを送るのは本質的ではない
- mutexをlockせずに条件となる式を更新するとlost wake-upバグとなる(条件を評価したあと、条件が変更されシグナルが送信されると、次のpthread_cond_waitは決して起こされない)

タイムアウト付きで待つ

- int **pthread_cond_timedwait**(
  pthread_cond_t *cond,
  pthread_mutex_t *mutex,
  const struct timespec *abstime);
- 条件変数にシグナルが送信されるか、タイムアウトとなるまでブロックする
- タイムアウトの場合はETIMEDOUTを返す