POSIXスレッド（３）

システムプログラミング

2009年11月2日

建部修見

同期の戦略

- 単一大域ロック
- スレッドセーフ関数
- 構造的コードロッキング
- 構造的データロッキング
- ロックとモジュラリティ
- デッドロック

単一大域ロック (single global lock)

- 単一のアプリケーションワイドのmutex
- スレッドが実行するときに獲得，ブロックする前にリリース
  - どのタイミングでも一つのスレッドが共有データをアクセスする
- 単一プロセッサではよく利用された

- 単純であるが，複数プロセッサのメリットなし
- 全てのモジュールとライブラリで＊単一＊のロックを利用しなければならない！

スレッドセーフ関数

- モジュール，ライブラリをスレッドセーフにする
- 内部で適切に同期し，呼び出し側で同期の必要がない
- 呼び出しが簡単となり，モジュラリティが高くなる
  - モジュラプログラミングのよい例

構造的コードロッキング

- 関数、コードレベルで行うロック
  - 共有データが特定の関数群でアクセスされることを想定
  - 通常，一つのモジュールに閉じている
  - 関数レベル、コードレベルで並列性が制限される
- 単一のmutexを用い，競合セクション(critical section)となる関数、コードでロックする
  - 競合セクションとは，モジュールの共有データをアクセスするコードの部分

構造的コードロッキングの例（1）

```
/* global mutex for structured code locking */
pthread_mutex_t mq_lock = PTHREAD_MUTEX_INITIALIZER;

void mq_put(mq_t mq, msg_t m)
{
   struct mq_elt *new;

   new = malloc(sizeof(*new));
   new->msg = m;
   pthread_mutex_lock(&mq_lock);
   add_tail(mq, new);    /* critical section */
   pthread_cond_signal(&mq->notempty);
   pthread_mutex_unlock(&mq_lock);
}
```

構造的コードロッキングの例（２）

```
msg_t mq_get(mq_t mg)
{
   struct mq_elt *old;
   msg_t m;

   pthread_mutex_lock(&mq_lock);
   while (mq->head == NULL) /* critical section starts */
      pthread_cond_wait(&mq->notempty, &mq_lock);
   old = delete_head(mq);
   m = old->msg;
   pthread_mutex_unlock(&mq_lock);
   free(old);
   return (m);
}
```

モニタ (Monitors)

- 並行(concurrent)プログラミング言語におけるモニタの概念[Hoare 74]は構造化コードロッキングで模擬できる
- モニタは，モジュールを構成する外部関数群で構成
- 共有データ（共有状態）は，モジュールの外部関数群でのみ排他的にアクセスできる
  - 情報隠蔽のよい例
- モニタでは，コンパイラがモニタの中の関数に入ると自動的に排他ロックをかける

内部関数

- モニタロックを確保した状態で呼ばれる関数
  - 構造化コードロッキングの例のadd_tail(), delete_head()は内部関数
- モニタ型の外部関数は，ロックを確保したまま他の外部関数を呼べない
  - デッドロック
- 外部関数の内部関数版を利用

不変式 (invariants)

- モニタロックが誰にも保持されていないとき，共有状態で必ず成立する式
  - ロック獲得直後，ロックリリース直前で成立
- 例：演習6のll_check(), list_check(), dataq_check(), hash_check()
- デバッグのためによく利用される

```
#include <assert.h>
assert(list_check(list));
```

  - cc –DNDEBUGでassertionチェックはdisableされる

複数の条件と条件変数（１）

- 正しくない例

```
pthread_mutex_lock(&monitor_lock);
while (!cond1)
    pthread_cond_wait(&cond1_cv, &monitor_lock);
while (!cond2)
    pthread_cond_wait(&cond2_cv, &monitor_lock);
/* cond2 may be true, but cond1 may no longer be true */
…
pthread_mutex_unlock(&monitor_lock);
```

複数の条件と条件変数（２）

- 正しい例

```
pthread_mutex_lock(&monitor_lock);
while (!cond1 || !cond2) {
   if (!cond1)
      pthread_cond_wait(&cond1_cv, &monitor_lock);
   if (!cond2)
      pthread_cond_wait(&cond2_cv, &monitor_lock);
}
…
pthread_mutex_unlock(&monitor_lock);
```

構造化データロッキング

- データロッキングはコードではなく，データ（オブジェクト）に対するロック
  - 共有データが特定の関数群でアクセスされることを想定
  - 共有データは独立なオブジェクト（データ）に分けられることを想定
  - 例：演習６のlinked list, queue, ハッシュ表
- オブジェクトモニタと似た概念
- 独立なlistの操作などは，モニタの構造を失わず並列に実行可能
- オブジェクト間でデータを共有する場合は難しい

ロックとモジュラリティ

- モニタのネスト
- 長いオペレーション
- モニタへの再入

モニタのネスト

- モジュールAが（Aのモニタロックを保持したまま）モジュールBを呼び，モジュールBでブロックしてしまうと，Aのモニタロックを長時間確保したままとなる
- モジュールBのブロック条件が，モジュールAで満たされる場合，デッドロックとなる
- オブジェクトモニタも，可能性は低くなるが，同様の問題がある
- 解決策：モニタロックを確保したままブロックする可能性のある関数を呼ばない
  - strcmpなどは大丈夫だが，getcなどはだめ
  - モニタロックを確保する前，リリース後に移動する

長いオペレーション

- モニタロックを保持したまま，実行時間の長い操作を行うことは，他スレッドの実行の妨げとなる
- I/O操作や入力待ちなどではロックをリリースする方がよい
- 例えば，busyフラグを立てて，unlockする。終了したら，busyフラグを落とす
- ハッシュ表の例

モニタへの再入

- モジュールAが(モニタロック確保したまま)モジュールAを呼ぶと，デッドロックが生じる
- 解決策:モニタロックを保持したまま，可能性のある関数の呼び出しを避ける
  - ロックをリリースする
- 可能性のある例:リストモジュールが，メモリアロケータモジュールを利用し，メモリアロケータモジュールがリストモジュールを利用するなど

デッドロック

- 永遠にブロックしてしまうこと
- Self-deadlockとrecursive deadlock
- ロックの順番によるデッドロック
  - スレッド1：モジュールA→モジュールB
  - スレッド2：モジュールB→モジュールA
- 解決策：モニターロックを保持したまま可能性のある関数の呼び出しを避ける
- 解決策：保持が避けられない場合，ロックする順番を決める。Lock hierarchy
- リソース不足によるデッドロック
  - 幾つかのリソースを確保。複数のスレッドが部分的に確保し，誰も解放しない
  - 解決策：全てのリソースを確保する。失敗した場合，全て解放し，はじめからやり直し。

ロックのガイドライン

- 構造的ロッキングのスタイルを使う
  - 関数あるいは各データにmutexを利用
  - 不変式の定義と利用
  - ロックを保持したまま，再入の可能性のあるモジュール外の関数呼び出しを避ける
  - 性能に影響するため，I/O操作など長時間に渡るロックの保持は避ける