情報科学プロジェクト実験

第 15 回目 スレッドの同期

コンピュータシステム研究室

成蹊大学理工学部

2017 年 1 月 19 日

本日の内容

- 同期変数
- 条件変数

スレッドの同期

複数のスレッドがメモリ等の計算資源を操作しているときに競合が発生しないようにしたい。

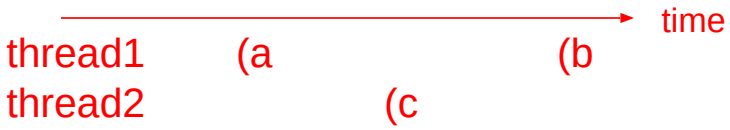
- **同期プリミティブ**
  - **排除ロック** (mutual exclusion lock)
  - **条件変数** (condition variable)
- join() **メソッドも同期の** 1 **つと考えてよい**

同期が必要な例

```
// Bank Account
int checking = 0, savings = 100;
void savings_to_checking(int amount)
{
   savings -= amount;  // ......(a
   checking += amount; // ......(b
}

int total_balance()
{
   return checking + savings; // .....(c
}
```

以下のタイミングでは問題が生じる



問題の原因と解決方法

原因:
- 2 つのスレッドが同時に同じデータを操作した
- データレースの発生: savings, checking

解決方法:
- 1 スレッドだけが共有データを操作する
- 1 スレッドしか実行できないコード部分を作る
  - 相互排除
  - クリティカルセクションの保護

同期変数によるクリティカルセクションの保護

```
// Bank Account2
int checking = 0, savings = 100;
mutex mx;
void savings_to_checking(int amount)
{
    mx.lock();
    savings -= amount;
    checking += amount;
    mx.unlock();
}

int total_balance()
{
    mx.lock();
    int balance = checking + savings;
    mx.unlock();
    return balance;
}
```

ロックを管理するオブジェクト

- アンロックのし忘れがよく起こる
- ロックを管理するオブジェクトの導入 (C++っぽさ)
  - lg のコンストラクタが mx.lock() を呼び
  - lg がスコープから外れると mx.unlock() を呼ぶ

```
// Bank Account3
int checking = 0, savings = 100;
mutex mx;
void savings_to_checking(int amount)
{
    lock_guard<mutex> lg(mx);
    savings -= amount;
    checking += amount;
}

int total_balance()
{
    lock_guard<mutex> lg(mx);
    int balance = checking + savings;
    return balance;
}
```

```
#include <iostream>
#include <thread>
#include <mutex>
using namespace std;
const int inittotal = 500000;
int checking = 0, savings = inittotal;
mutex mx;

void savings_to_checking(int amount) {
    lock_guard<mutex> lg(mx);
    savings -= amount;
    checking += amount;
}

int total_balance() {
    lock_guard<mutex> lg(mx);
    int balance = checking + savings;
    return balance;
}

void func() {
    for (int i = 0; i<inittotal; i++)
        savings_to_checking(1);
}

int main() {
    thread t(func);
    for (int i = 0; i<inittotal; i++)
        if (total_balance() != inittotal)
            cerr << "error\n";
    t.join();
    return 0;
}
```

出力のための同期

複数のスレッドが同時に呼び出しても出力結果を混ぜない

```
mutex mx;
void write_strings(vector<string>& sv)
{
    lock_guard<mutex> l(mx);
    for (string &s : sv )
        cout << s << " ";
}
```

条件変数

- **条件変数の 2 つの基本操作**:
  - wait: **取得したロックを一時的に解放し、通知されるのを待つ操作**
  - notify: **通知する操作**
- **条件変数で** wait **しているスレッドに対して通知がなされると**:
  - **条件変数で待っているスレッドが処理を再開**
    - **通知の種類により** 1or **全スレッドが再開**
    - **指定されたロックを獲得する**
  - 1 **つも待っていない場合には通知は無効**
    (通知は状態を持たない)

典型的な条件待ちのコード

```
mutex mx;
condition_variable cv;
...
func(){
    unique_lock<mutex> ul( mx );
    while (条件を満たしていない) cv.wait(ul);  // ロックを解放して待機

    // 再開時にはロックも再度獲得している

    // 条件が満たされたので処理を行う
    some_critical_work();
}
```

通知の種類

```
condition_variable cv;
// どれか 1 つのスレッドに通知
cv.notify_one()
```

または

```
// 待っている全てのスレッドに通知
cv.notify_all()
```

注意：

- **これらは通知だけでロックを解放するわけではない**
- cv.wait() **で待っている側は再度ロックを獲得しようとする**

```
// thread の id 順に出力させる
#include <iostream>
#include <thread>
#include <vector>
#include <mutex>
#include <condition_variable>
using namespace std;
int turn = 0;
void func(int id, mutex& mx, condition_variable& cv) {
    unique_lock<mutex> lock(mx);
    while (id != turn) cv.wait(lock);
    cout << id << "\n";
    turn ++;
    cv.notify_all(); // bug if it is cv.notify_one();
}

int main() {
    mutex mx;
    condition_variable cv;
    const int num = 10;
    vector<thread> a;
    for (int i = 0; i < num; i++)
        a.push_back(thread(func, i, ref(mx), ref(cv)));
    for (thread &t : a) t.join();
    return 0;
}
```

```
// main スレッドが他のスレッドの終了を待つ
int worker = 0;
void func(int id, mutex& mx, condition_variable& cv) {
    sleep(3); // とりあえず寝る（実際は何らかの計算をする）
    unique_lock<mutex> lock(mx);
    cout << id <<"/"<<worker<<"\n";
    -- worker;
    if (worker == 0) cv.notify_one(); // main スレッドに通知
}

int main() {
    mutex mx;
    condition_variable cv;
    const int num = 10;
    worker = num;
    for (int i = 0; i < num; i++)
        thread(func, i, ref(mx), ref(cv)).detach(); // detach: 切り離し

    unique_lock<mutex> lock(mx);
    while (worker > 0) cv.wait(lock); // 全 thread の終了を待つ
        cout << "done\n";
    return 0;
}
```

課題 15:スレッドの同期

素数とは 1 とその数自身以外に約数を持たない 2 以上の整数である。指定した数以下の自然数にいくつ素数があるかを数えるプログラムを作成し、それをスレッドにより並列化せよ。

- スレッド数と CPU 数を揃えること。
- スレッドプログラムではロックを使って一つの変数 count に数を数えること。さらにロックを使う期間を短く、回数を少なくすること。
- main スレッドは join を使わずに条件変数を使って自分の作成した子スレッドの終了を待つようにすること。

```
// 指定値までの素数の個数を数える
#include <iostream>
#include <cstdlib>
using namespace std;

bool is_prime(int x) {
   if (x == 2) return true;
   if (x % 2 == 0) return false;
   for (int i = 3; i*i <= x; i+=2 )
      if (x % i == 0) return false;
   return true;
}

int main(int argc, char *argv[]) {
   if (argc < 2) {
      cerr << "usage: "<<argv[0]<<" max\n";
      return 1;
   }
   int max = atoi(argv[1]);
   if (max <= 0) return 1;
   int count = 0;
   for (int i = 2; i <=max; i++) // prime number is greater than 1
       if (is_prime(i)) ++ count;
   cout << "# of prime (<="<<max<<") is "<<count<<"\n";
   return 0;
}
```