排他処理

排他処理とは複数のプロセス(スレッド)が共有するメモリへの読み書きを行う際に、整合性を保てるようにするための処理です。

昨今の CPU はシングルコアでのクロック数向上による高速化の道を諦めて、マルチコアによる並列処理で高速化していく方向になっています。このためプロのプログラマとして働いていく上で排他処理は避けて通れない道になっています。

文章の説明だけではイメージしにくいと思いますので、Github にサンプルプログラムをアップしています。Database を pull して下記のパスの ThreadTest プロジェクトをビルドして実行してみて下さい。

C:¥GitHub¥Database¥ソフトウェア授業資料¥排他処理¥ThreadTest¥ThreadTest.sln

このプログラムは下記のようなコードが記述されています。

```
#include <future>
#include <windows.h>
int count = 0;
void Thread1()
{
	//countを10万回インクリメント
	for (int i = 0; i < 100000; i++) {
		count++;
	}
}
void Thread2()
{
	//countを10万回インクリメント
	for (int i = 0; i < 100000; i++) {
		count++;
	}
}
int main()
{
	std::thread th1 = std::thread([] {Thread1(); }); //Thread1を起動。
	std::thread th2 = std::thread([] {Thread2(); }); //Thread2を起動。
	th1.join();                    //Thread1が終了するまで待機。
	th2.join();                    //Thread2が終了するまで待機。
	//結果を表示。
	char message[256];
	sprintf_s( message, "count = %d¥n", count);
	MessageBox(NULL, message, "結果", MB_OK);   //20万になる？
	return 0;
}
```

std::thread あたりのコードが意味不明に感じるかと思いますが、今はスルーしておいてください。このコードで分かってほしいのは Thread1 関数で count を 10 万回インクリメントされ、Thread2 関数でも count を 10 万回インクリメントしています。そして Thread1 と Thread2 は並列に動作するということです。

さて、最後の行で出力される count の値はいくつになるでしょうか？恐らく人によって異

なる結果になっているはずです。また実行するたびに結果も変わるはずです。なぜこのようなことが起きるのか詳しく説明していきます。

まず、少し話が横にずれるのですが C 言語、C++、Java、C#など、どの言語もそのままではコンピュータは実行できません。そのためプログラミング言語をコンピュータが理解できる言葉(これをアセンブリ言語といいます)に翻訳する必要があります。この翻訳を行ってくれるソフトウェアをコンパイラといいます。皆さんがいつも VisualStudio で行っていることです。今回の説明はサンプルプログラムを逆アセンブルしたものを使って説明を行います。

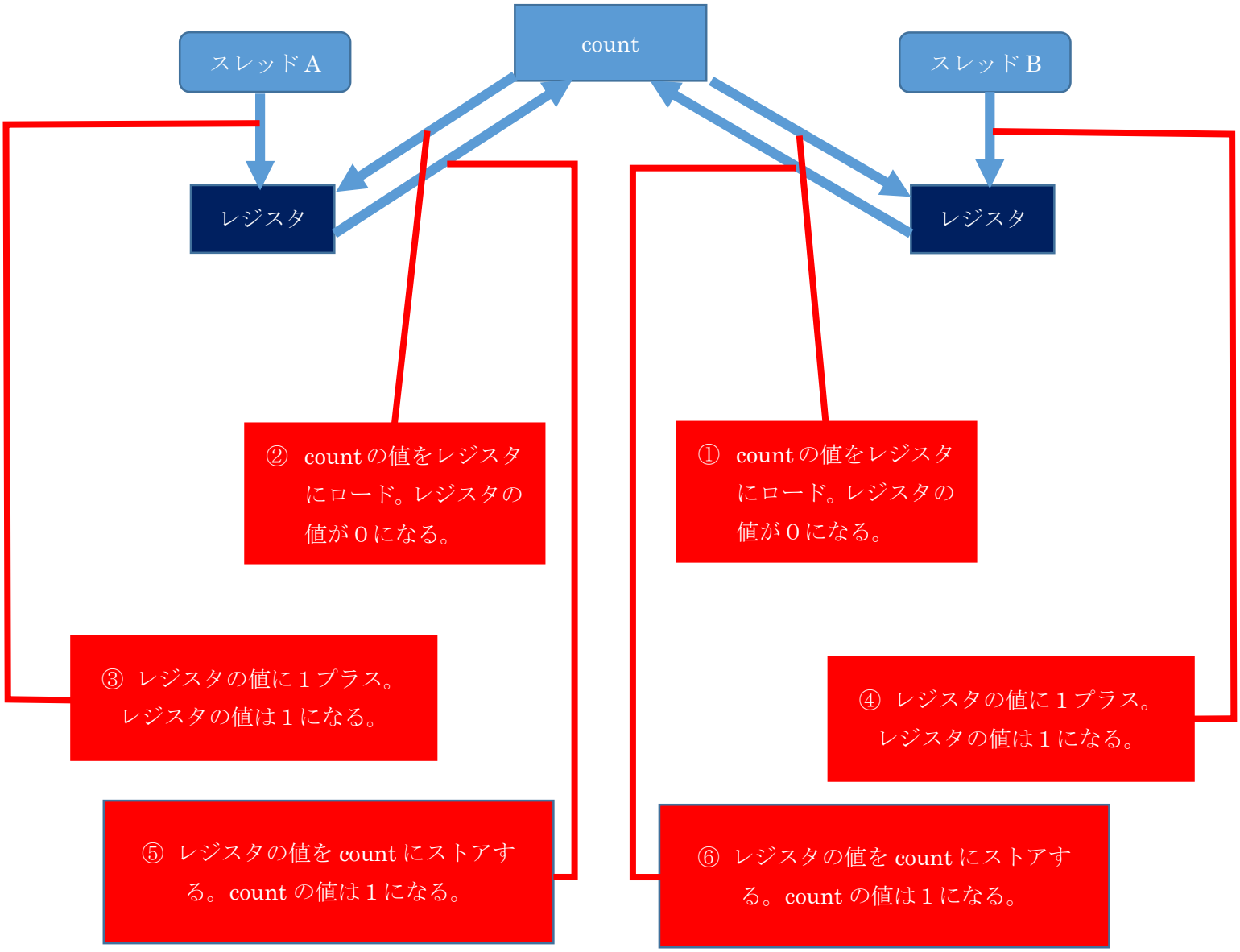
下記のコードは Thread1 関数を逆アセンブルしたものです。

```
void Thread1()
{
push        ebp
mov         ebp,esp
sub         esp,0CCh
push        ebx
push        esi
push        edi
lea         edi,[ebp-0CCh]
mov         ecx,33h
mov         eax,0CCCCCCCCh
rep stos    dword ptr es:[edi]
mov         dword ptr [ebp-8],0
jmp         Thread1+30h (0F33BC0h)
mov         eax,dword ptr [ebp-8]
add         eax,1
mov         dword ptr [ebp-8],eax
cmp         dword ptr [ebp-8],186A0h
jge         Thread1+48h (0F33BD8h)
mov         eax,dword ptr [count (0F3E4C8h)]
add         eax,1
mov         dword ptr [count (0F3E4C8h)],eax
jmp         Thread1+27h (0F33BB7h)
}
```

見慣れないコードになっているかと思います。実はこれが実際に CPU に命令を送っているコードになります。では問題となっている count のインクリメントの部分だけ抜き出してみます。

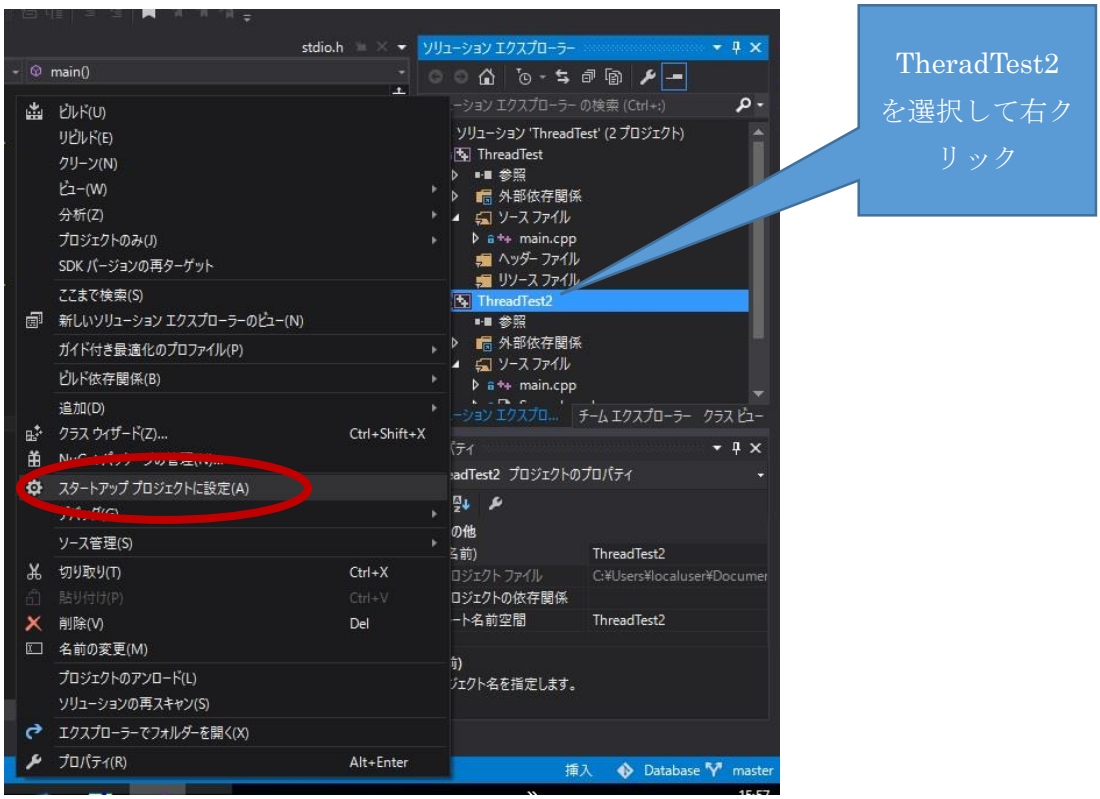
```
mov         eax,dword ptr [count (0F3E4C8h)]  //countの値をeaxレジスタにロード。
add         eax,1                             //eaxレジスタに1プラス。
mov         dword ptr [count (0F3E4C8h)],eax  //eaxレジスタの値をcountにストア。
```

このように、実は count++のコードだけで 3 命令実行されています。この count++が二つのスレッドで並列に実行されるとどうなるのか考えてみます。赤色の吹き出しが処理の流れです。count の初期値は 0 とします。



このようにcountのインクリメントは2回行われたのに、countの値は1になってしまっています。この処理の順番はマルチスレッドプログラミングではどうなるかは不明です。スレッドAがcountの値をストアしてからスレッドBが動き出すこともあります。これはCPUの状況次第になります。このような不整合を防ぐために変数を読み込み/書き込みを行っているときには他のスレッドがアクセスできないようにロックを書けることを排他処理といいます。ではThreadTestソリューションのThreadTest2プロジェクトを実行してください。セマフォで排他処理を行うように変更しています。

VisualStudio のソリューションにプロジェクトが二つあるのを見るのは初めてでしょうか？実はソリューションには複数のプロジェクトを入れることができます。ThreadTest2 を実行したい場合は下記の図のようにスタートアッププロジェクトを ThreadTest2 に設定してください。



実行できたでしょうか？今度は count の値は何回やっても 20 万になるはずです。ではコードを見て説明をします。

```
void Thread1()
{
   //countを10万回インクリメント
   for (int i = 0; i < 100000; i++) {
      //セマフォ資源を取得(1減算)。
      //セマフォを獲得できなかったら待ち行列に追加されてスレッド眠る。
      sem.GetSemaphore();
      count++;
      //待ち行列入っているスレッドがいたら起こします。
      //いなければセマフォ資源を返却(1加算)。
      sem.ReleaseSemaphore(); //セマフォ資源を返却(1加算)。
   }
}
```

count のインクリメントの前後にセマフォの資源の取得と返却が追加されています。セマフォ資源を獲得できない場合にスレッドは待ち行列に入りスレッドは眠ります。これによ

り count++の処理はスレッドがいくつ立ち上がっても、必ず一つのスレッドしか実行できなくなります。これが排他処理です。