機械工学総合演習第二 計算機演習
並列計算

東京大学大学院情報理工学系研究科
知能機械情報学専攻 講師 新山龍馬
TA：中里、漆戸、和田、臼井、行澤
連絡先メールアドレス：niiyama@isi.imi.i.u-tokyo.ac.jp

May 23, June 2, July 1, and July 15, 2016

1　はじめに

コンピュータの OS は、プロセッサのコア数に関わらず、複数のアプリケーションを見かけ上並列に処理するマルチタスクを実現している。ユーザープログラムにおいても、そのような並行・並列処理が必要な場合がある。例えば、ロボット制御プログラムは、アクチュエータを制御すると同時にセンサ情報を処理する必要がある。そこで、複数の処理を同時に実行するために POSIX スレッドを使ったマルチスレッドプログラミングについて学ぶ。

また、最近の計算機ではマルチコア CPU の搭載が一般的になっており、シングルプロセッサを見かけ上並列に使うだけでなく、複数の処理を効率よく並列に処理する技術の重要性が増している。そこで、並列化計算を容易に実現する OpenMP についても学ぶ。

2　マルチスレッド

2.1　プロセスとスレッドの違い

プロセスとは、ひとつのアプリケーションの実行に対応する処理のまとまりである。普通、プロセスごとに別々のメモリ空間が割り当てられる。一方、スレッドとは、プロセスの中に含まれるより小さな処理の単位である。プロセスは複数のスレッドを持つことができる。図 1 にマルチスレッドとマルチプロセスの違いを示す。

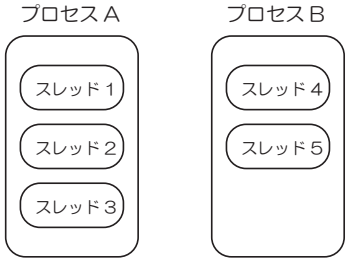

図 1: マルチスレッドとマルチプロセスの違い。プロセス A とプロセス B には別々のメモリ空間が割り当てられており、プロセス A の挙動がプロセス B に影響を与えることはほとんどないが、情報の共有に工夫が必要。スレッドはプロセス内で複数実行可能であり、同じプロセス内ではスレッドはメモリ空間を共有している。この図ではスレッド 1、2、3 がメモリ空間を共有し、スレッド 4 と 5 とは共有していない（スレッド 4 と 5 は共有している）。

プロセス間のデータのやり取りには、ソケット通信やファイルシステムへの書き込み・読み込みなどを利用する必要がある。一方、同じプロセスに含まれるスレッド間のデータのやり取りは、同じメモリ空間を共有しているのでもっと簡単になる。

つまり、複数のスレッドを使うことで、相互に連携した複数の処理を行うプログラムの作成が容易になる。具体例としては、複数のセンサの値を取り込みながら運動制御を行うロボットのプログラムや、ボタン入力を待ちながら画面描画を行うゲームのプログラムが挙げられる。

2.2　スレッドを用いたサンプルプログラム

課題 0

サンプルプログラムをダウンロードして、適当なディレクトリで解凍せよ。ファイルの解凍方法は以下の通り。

```
$ unzip multithread.zip
```

早速、マルチスレッドを用いたサンプルプログラムを見てみることにする。これは 2 つのスレッドがそれぞれ一秒おきに 0 から 9 まで表示するものである。

```
#include <pthread.h>

void *task1(void *arg)
{
  int i;
  pthread_t thread;
  thread=pthread_self(); /*スレッド ID の取得 */
  for(i=0;i<10;i++){
    sleep(1); /*1 秒待つ*/
    printf("Thread [%d]: %d\n",thread,i); /*スレッド ID とカウントを表示*/
  }
  return(NULL);
}

int main()
{
  pthread_t thread1, thread2; /*スレッドの定義*/

  pthread_create(&thread1,NULL,task1,(void *)NULL); /*スレッド 1 を作成*/
  pthread_create(&thread2,NULL,task1,(void *)NULL); /*スレッド 2 を作成*/

  pthread_join(thread1,NULL); /*スレッド 1 の終了を待つ*/
  printf("Thread %d finished.\n", thread1);
  pthread_join(thread2,NULL); /*スレッド 2 の終了を待つ*/
  printf("Thread %d finished.\n", thread2);
  return(0);
}
```

課題 1

サンプルプログラムに含まれる multithread.c をコンパイルせよ。できた実行ファイル multithread を実行し、二つのスレッドが 1 秒おきに数字を 0 から 9 までカウントし、スレッド ID と共に表示されることを確認せよ。コンパイルには「-lpthread」オプションをつける。具体的には以下のようにする。

```
$ gcc -lpthread -o multithread multithread.c
```

まず main 関数からみると、はじめの行で二つのスレッド（thread1 と thread2）を定義している。pthread_t はスレッドを定義するための構造体である。次に定義したスレッド（thread1 と thread2）を pthread_create 関数を用いて実際に作成している。この二行を行うことで **task1 という関数を thread1 と thread2 が別々に実行する**ことになる。スレッドを生成する pthread_create 関数の詳細を次に示す。

```
int pthread_create ( pthread_t *thread, const pthread_attr_t *attr,
                     void *(*start_routine)(void *), void *arg);
```

正常終了すると、pthread_create() は 0 を返す。それ以外はエラー番号を返す。

pthread_t *thread pthread_create により新たに作成されたスレッドを識別するためのポインタ。phread_t 型である。

const pthread_attr_t *attr スレッド属性オブジェクトという構造体へのポインタ。特に属性を指定する必要が無い時は、NULL を指定する事が出来る。NULL を指定した場合、デフォルトの属性が指定された事になる。

void *(*start_routine)(void *) 新しいスレッドが実行する関数へのポインタ。

void *arg 新しいスレッドが実行する関数へ渡される引数へのポインタ。

pthread_create の呼び出しにより、task1 が実行されるので、task1 の関数を見ることにする。

task1 内の三行目で pthread_self 関数が呼ばれているが、これにより呼び出したスレッドのスレッドハンドルを得ることができる。サンプルプログラムではスレッドハンドルを表示することで、異なる二つのスレッドが実行されていることを確かめている。次に for 文の中で、一秒待って（sleep(1)）現在のカウントとそれをカウントしているスレッドハンドルを表示させている（printf 文)。10 回カウントするとそのスレッドは終了する。

task1 から main 関数の説明に戻る。スレッドは同時に実行できることがメリットであるが、二つのスレッドを実行した場合に、それぞれのスレッドがいつ終わるかが分からない問題が生じることになる。例えば、thread1 の終了を待ってからある仕事を行い、thread2 が終了していることを確認してまた他の仕事を行う場合などである。これはスレッドの**同期**と呼ばれるものである。プログラムではこの同期を実現するために pthread_join 関数を用いている。pthead_join は、この関数を呼び出した側を pthread_join 関数で指定したスレッドが終了するまでサスペンドすることで、スレッドの同期を実現する。サンプルプログラムでは、thread1 が終了するのを確認して、thread1 が終了したことを表示し、さらに thread2 が終了することを確認して、thread2 が終了したことを表示している。pthread_join の詳しい説明を以下に示す。

```
int pthread_join(pthread_t thread, void **status)
```

pthread_t thread thread で指定したスレッドが終了するまで実行をサスペンドする。既に thread が終了していた場合は何も起こらない。

void **status thread の返り値を status が参照する変数に格納する。status に NULL を指定した場合、返り値はどこにも格納されない。

また、スレッドのジョインを行う必要が無い場合は、pthread_detach 関数を利用する。これによりスレッドの終了を確認する必要がなくなる。

課題 2

multithread.c を改良し、2 秒おきに数字を 0 から 9 までカウントする task2 という関数を作成し、3 つ目のスレッドとして起動するように修正せよ。

このサンプルプログラムでは、main 関数が thread1 と thread2 が終了するのを待ってから、main 関数自身を終了している。これとは別にスレッドは、明示的に終了させることも可能である。スレッドを明示的に終了させる関数は pthread_exit 関数である。この関数はスレッドが実行する関数内で呼ばれるのが一般的である。

```
void pthread_exit(void *status);
```

void *status スレッドの終了に伴う返り値をセットする。この値は例えば、pthread_join で受け取ることが可能であり、スレッドの終了状態を知ることに利用される。

2.3　スレッド間のデータ共有と同期

スレッドによるデータの共有は非常に簡単である。グローバル変数を用いることで、その変数を複数のスレッドで共有することが可能となる。しかしながら、同時に実行するスレッドが同じグローバル変数にアクセスする場合にスレッド同士が干渉する。

ここでは、ある人が、当座用の口座（cheking）と、貯蓄用の口座（saving）の 2 種類の口座を持っていて、これらの口座間でお金を移動するプログラムの作成例を考える。つまり、2 種類の口座の合計金額（balance）は常に一定である必要がある。

```
struct account {
   int checking;
   int savings;
};
void savings_to_checking(struct account *ap, int amount)
{
   ap->savings -= amount;
   ap->checking += amount;
}
int total_balance(struct account *ap)
{
   int balance;
   balance = ap->checking + ap->savings;
   return (balance);
}
```

この例のように、スレッド 1 が savings_to_checking() を呼び出している間に、スレッド 2 が total_balance() を呼び出す時に、問題が起こる可能性がある。

1. スレッド 1 が savings -= amount; を行う。
2. スレッド 2 が balance = checking + savings; を行う。
3. スレッド 1 が checking += amount; を行う。
4. スレッド 2 が return(balance); を行う。

この場合、明らかに total_balance() が返す値は口座にあるはずの合計金額よりも amount の分だけ少なくなる。この問題は、複数の制御スレッドが同時に同じデータを操作したために生じる。つまり、一度に一つのスレッドしか共有データを操作出来ないようにする必要がある。コードのある部分で一つのスレッドが他の全てのスレッドを排除するので、相互排除 (mutual exclusion) と呼ばれている。一度に 1 つのスレッドしか実行出来ないコード部分は、クリティカルセクション (critical section) と呼ばれる。相互排除は、mutex を用いて行うことができる。mutex には locking と unlocking という 2 つの主要な操作がある。

```
struct account {
   int checking;
   int savings;
};
pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER; /*追加*/
void savings_to_checking(struct account *ap, int amount)
{
   pthread_mutex_lock(&lock); /*追加*/
   ap->savings -= amount;
   ap->checking += amount;
   pthread_mutex_unlock(&lock); /*追加*/
}
int total_balance(struct account *ap)
{
   int balance;
   pthread_mutex_lock(&lock); /*追加*/
   balance = ap->checking + ap->savings;
   pthread_mutex_unlock(&lock); /*追加*/
   return (balance);
}
```

pthread_mutex_lock() 関数は、mutex をロックする。その mutex をロックしようとする他のスレッドは、その mutex がアンロックされるまで待たなければならない。pthead_mutex_unlock() の呼び出しにより mutex がアンロックされると、ロックが解放されるのを待っていたスレッドのどれかが mutex をロックして、pthread_mutex_lock() からリターンする。

課題 3

サンプルプログラムに含まれる mutex.c をコンパイルし、実行せよ。実行の最初と最後で balance と cash の和が表示されるが、初期値 10000 から変化している。プログラムの中では、task1 と task2 がスレッドとして同時に起動し、task1 が cash から balance 一定値を移動し、task2 が balance から cash に一定値を移動している。移動は即座に実行されず読み出しと変更・書き込みの間にはランダムな遅延が生じる。プログラムは最後に balance と cash の和の値が 10000 になるように期待されて作成したものであるが、task1、task2 のグローバル変数へのアクセスのタイミングがずれているために、期待通りの結果とならない。終了時の balance と cash の和の表示が 10000 になるように、mutex.c に mutex を利用した排他処理を追加せよ。

（本課題は 2009 年度 TA 鈴木君の協力によるものである。）

2.4　参考：POSIX スレッド

この演習では、スレッドプログラミングにおいて、もっとも標準的に使われる POSIX スレッドを使用する。POSIX（ポジックス、Portable Operating System Interface）とは、異なる実装の UNIX ベースの OS が備えるべき最低限の仕様のセットとして、IEEE が策定したアプリケーションインタフェイス規格である。各社の UNIX 互換 OS にはそれぞれ独自の拡張や仕様の変更が施され、互換性が失われてしまったため、各 OS 間で最低限の互換性を確保するために定義された。アプリケーションソフトが OS の提供する機能を呼び出すための方法（システムインターフェース）などを定義している。スレッドのインターフェースも用意されており POSIX スレッド（pthread）として、UNIX 互換 OS でほぼ使用可能な汎用性の高いものになっている。

2.5　参考：スレッドを利用するときのその他の注意点

スレッドは並列処理でありながら、同一のプロセス内で動作するためにメモリ領域を共有している。そのためにスレッド間のデータ共有を容易に行うことができる。逆の言い方をすれば、あるスレッドが他のスレッドの領域を犯し、動作に悪影響を与えることもあるため注意が必要となる。

また、マルチスレッドは、グローバル変数とスタティック変数に注意を払う必要がある。スレッドが複数同時に処理され、各スレッドで同一のグローバル変数をアクセスするということは、マルチプロセスとは異なり本当に同じ変数をアクセスすることになる。この仕組みをうまく利用すると共通のパラメータや状態を共有できるが、注意をしないとスレッド間での衝突が発生する。また、グローバル変数よりさらに深刻な問題が、各関数で持っているスタティック変数で発生する。ローカル変数は別々のスタックで確保され独立しているが、スタティック変数はプロセスとして 1 つの領域に確保され複数のスレッドで参照される。例えば、strtok() はトークンの切り出しに便利な関数であるが、複数のスレッド内で同時に使用される場合、strtok 内のスタティック領域にコピーした内容、ポインタが他のスレッドからの strtok() で上書きされてしまう。そのため、スタティック変数を使用する関数は並列スレッド内では使用できない。POSIX では、これらの関数の代替えとして、スレッドセーフ関数が定義されている。その代替インタフェースは、元の関数のうしろに“ _r ”を付加した名前になっている。スレッド同士がメモリやその他のプロセスリソースを操作している間に競合することがないようにスレッドを同期するための仕組みが用意されている。

3 OpenMP

3.1 OpenMP とは

OpenMP（Open Multi-Processing）は共有メモリ型計算機における並列処理をサポートする API（Application Program Interface）である。C、C++、Fortran で利用可能。OpenMP ではプラグマディレクティブ（#pragma）といった指示文を用いて記述する。この指示文を既存のプログラムに挿入するだけで、既存のプログラムを容易に並列処理に拡張することができる。この演習では最低限の使い方のみを紹介するので詳しくは OpenMP のホームページ（http://www.openmp.org/）を参照されたい。

3.2 OpenMP を使ってみよう

まず「Hello World!」と出力するプログラムを書いてみよう。これを hello.c として保存する。

```
#include <stdio.h>
int main()
{
  printf("hello world!\n");
}
```

コンパイルしてみよう。

```
$ gcc -o hello hello.c
```

実行してみよう。

```
$ ./hello
hello world!
```

このように「Hello World!」が 1 つ出力される。

次に OpenMP を使って hello.c を修正しよう。**printf 文を「#pragma omp parallel{}」で囲む**。OpenMP では「#pragma omp parallel{}」で囲んだ部分の処理を行う複数のスレッドが自動的に作成され、並列処理される。修正したプログラムを hello_omp.c とする。

```
#include <stdio.h>
int main()
{
  #pragma omp parallel
  {
    printf("hello world!\n");
  }
  return 0;
}
```

コンパイルしてみよう。**コンパイルオプションに「-fopenmp」をつける。**

```
$ gcc -fopenmp -o hello_omp hello_omp.c
```

実行してみよう。

```
$ ./hello_omp
hello world!
hello world!
```

計算機に搭載された CPU のコア数に応じて「Hello World!」が複数回出力される。アーキテクチャによっては、物理コア数と論理コア数が異なり、1 つの物理コアを見かけ上 2 つのプロセッサとして扱うこともある。「Hello World!」は論理コアの数だけ出力される。計算機の CPU コア数を調べるには、下記を実行して CPU(s) と Thread(s) per core を見ればよい。

```
$ less /proc/cpuinfo
```

課題 4

hello_omp.c を自ら実装して、コンパイル、実行してみよう。また、計算機の論理コア数を調べてみよう。

3.3 for ループの並列化

OpenMP を利用すると for ループを自動的に並列化してくれる。プログラム例を以下に示す。このプログラム名を forloop_omp.c とした。これは a と b の配列の和を c の配列に入れるプログラムである。for ループを OpenMP で並列化するには **for 文の直前に「#pragma omp parallel for」を挿入**する。

```
#include <omp.h>
#include <stdio.h>
#define N 10

int main ()
{
  int i, a[N], b[N], c[N];

  for (i=0;i<N;i++){/* 初期化 */
    a[i]=b[i]=i;
    c[i]=0;
  }

#pragma omp parallel for
  for (i=0; i<N; i++){
    c[i] = a[i] + b[i];
    printf("Thread %d: c[%d]= %d\n",omp_get_thread_num(),i,c[i]);
  }
  return 0;
}
```

コンパイルしてみよう。**コンパイルオプションに「-fopenmp」をつける。**

```
$ gcc -fopenmp -o forloop_omp forloop_omp.c
```

実行してみよう。

```
$ ./forloop_omp
Thread 1: c[5]= 10
Thread 1: c[6]= 12
Thread 1: c[7]= 14
Thread 1: c[8]= 16
Thread 1: c[9]= 18
Thread 0: c[0]= 0
Thread 0: c[1]= 2
Thread 0: c[2]= 4
Thread 0: c[3]= 6
Thread 0: c[4]= 8
```

10 回繰り返し計算される訳だが、この実行例ではインデックスの 0 から 4 までをスレッド 0 に、5 から 9 までがスレッド 1 に自動的に割り当てられることが分かる。プログラム内で omp_get_thread_num() 関数が出てくるが、これはスレッド番号を自動的に取得してくれる関数である。この関数を利用するために **omp.h をインクルードしている。**

課題 5

forloop_omp.c を自ら実装して、コンパイル、実行してみよう。

3.4　セクションを用いた並列化

つぎのようなプログラム section.c を考えてみよう。

```
#include <stdio.h>
#define N 5

int main ()
{
  int i,j,a[N], b[N], c[N], d[N];
  for (i=0; i<N; i++) a[i] = b[i] = i;

  for (i=0; i<N; i++){
    c[i] = a[i] + b[i];
    printf("c[%d]= %d\n",i,c[i]);
  }

  for (j=0; j<N; j++){
    d[j] = a[j] * b[j];
    printf("d[%d]= %d\n",j,d[j]);
  }

  return 0;
}
```

これは a と b の配列の和を c の配列に入れるタスク 1 と、a と b の配列の積を d の配列に入れるタスク 2 が含まれるプログラムである。タスク 1 とタスク 2 は独立であるのでこれらを並列化することができる。並列化するには次の手順を踏む。

1. 並列化する部分を「#pragma omp parallel {}」で囲う。
2. さらに並列化したい部分を「#pragma omp sections {}」で囲う。
3. スレッドに分割したい部分をそれぞれ「#pragma omp section {}」で囲う。

section.c を OpenMP を利用して並列化したプログラム例を以下に示す。プログラム名を section_omp.c とする。

```
#include <omp.h>
#include <stdio.h>
#define N 5

int main ()
{
  int i,j,a[N], b[N], c[N], d[N];
  for (i=0; i<N; i++) a[i] = b[i] = i;
#pragma omp parallel
  {
#pragma omp sections
    {
#pragma omp section
      {
        for (i=0; i<N; i++){
          c[i] = a[i] + b[i];
          printf("Thread %d: c[%d]= %d\n",omp_get_thread_num(),i,c[i]);
        }
      }
#pragma omp section
      {
        for (j=0; j<N; j++){
            d[j] = a[j] * b[j];
            printf("Thread %d: d[%d]= %d\n",omp_get_thread_num(),j,d[j]);
        }
```

```
      }
    }
  }
  return 0;
}
```

コンパイルしてみよう。**コンパイルオプションに「-fopenmp」をつける。**

```
$ gcc -fopenmp -o section_omp section_omp.c
```

実行してみよう。

```
$ ./section_omp
Thread 1: d[0]= 0
Thread 1: d[1]= 1
Thread 1: d[2]= 4
Thread 1: d[3]= 9
Thread 1: d[4]= 16
Thread 0: c[0]= 0
Thread 0: c[1]= 2
Thread 0: c[2]= 4
Thread 0: c[3]= 6
Thread 0: c[4]= 8
```

このように、和を計算する部分がスレッド 0 に割り当てられ、積を計算する部分がスレッド 1 に自動的に割り当てられているのが分かる。

課題 6

section_omp.c を自ら実装して、コンパイル、実行してみよう。

3.5　プライベート変数、共有変数

forloop_omp.c をもう一度考えてみる。

```
#include <omp.h>
#include <stdio.h>
#define N 10

int main ()
{
  int i, a[N], b[N], c[N];

  for (i=0;i<N;i++){
    a[i]=b[i]=i;
    c[i]=0;
  }

#pragma omp parallel for
```

```
  for (i=0; i<N; i++){
    c[i] = a[i] + b[i];
    printf("Thread %d: c[%d]= %d\n",omp_get_thread_num(),i,c[i]);
  }
  return 0;
}
```

「#pragma omp parallel」で囲まれた部分の変数は基本的に共有変数として扱われる。共有変数とは、それぞれのスレッドが同時に読み書きできる変数のことである。forloop_omp.c の例であれば、a、b、c が共有変数として各々のスレッド間で情報共有可能なようになる。**サンプルプログラム mutex.c 内の大域変数（flag, balance, cash）のようなものと理解できる。**共有変数を明示的に指示することも可能である。例えば a、b、c を明示的に共有変数と指示するには、**parallel 指示文の後に shared(a,b,c) と書く。**

スレッド間で共有しない変数をプライベート変数という。プライベート変数はスレッド本体内のみで有効な変数であり、各スレッド内だけで有効な変数のコピーが作成される。プライベート変数は初期値が不定であるために、初期化する必要がある。**サンプルプログラム mutex.c の task1、task2 関数内のローカル変数（t, balanceread）のようなものと理解できる。**「#pragma omp parallel」で囲まれた部分の変数は原則的に共有変数として扱われるが、例外として**「#pragma omp parallel for」直後の for 文のインデックスはプライベート変数として扱われる。**forloop_omp.c ではインデックス i がプライベート変数として扱われる。プライベート変数として明示的に指示することも可能であり、例えばインデックス i では、**parallel 指示文の後に private(i) と書く。**

forloop_omp.c の変数を明示的に共有変数とプライベート変数に記述したプログラムを以下に示す。このプログラム名を forloop_var_omp.c とする。ここで、”#pragma omp parallel for” が指示文であり、shared() と private() のみが足されたことに注意。

```
#include <omp.h>
#include <stdio.h>
#define N 10

int main ()
{
  int i, a[N], b[N], c[N];

  for (i=0;i<N;i++){
    a[i]=b[i]=i;
    c[i]=0;
  }

#pragma omp parallel for shared(a,b,c) private(i) /* for までが元々の指示文 */
  for (i=0; i<N; i++){
    c[i] = a[i] + b[i];
    printf("Thread %d: c[%d]= %d\n",omp_get_thread_num(),i,c[i]);
  }
  return 0;
}
```

課題 7

サンプルプログラムに含まれる section_omp2.c をコンパイルし実行せよ。このプログラムは section_omp.c とほぼ同じであるが、積の計算のインデックス j がインデックス i に変更されている。この実行結果は section_omp の実行結果と一致しない。一致しない理由を考え、private()、shared() といった指示のみを用いて修正せよ。1 行の修正で済む。

3.6 reduction

次のような配列の中身の和を計算するプログラムを考える。このプログラム名を sum.c とする。

```
#include <stdio.h>
#define N 10000

int main ()
{
  int i, a[N];
  int sum=0;

  for (i=0; i<N; i++) a[i] = i+1;

  for (i=0; i<N; i++){
    sum += a[i];
  }
  printf("sum = %d\n",sum);

  return 0;
}
```

1 から 10000 までの和を計算するの実行結果は、50005000 となる。和の計算部分を並列に計算させるために下記のプログラムのように変更したがうまく動作しない。sum_omp_bad.c とする。

```
#include <stdio.h>
#define N 10000

int main ()
{
  int i, a[N];
  int sum=0;

  for (i=0; i<N; i++) a[i] = i+1;

#pragma omp parallel for
  for (i=0; i<N; i++){
    sum += a[i];
  }
  printf("sum = %d\n",sum);

  return 0;
}
```

なぜなら sum は共有変数であり、分割したスレッドが変数 sum を奪い合うことで sum の値がおかしくなる。それでは sum をプライベート変数にすればいいかというと、これもうまくいかない。プライベート変数として宣言するとスレッドが終了するとそれぞれのスレッド内でせっかく足し算した結果が破棄されるため、スレッド間の sum の総和を計算できない。つまり、**sum は各スレッドでプライベート変数として扱われるが、計算終了後に各スレッドの sum 変数同士の和を計算する必要がある。**この操作を実現するのが reduction 指示句である。具体的には、**#pragma omp parallel for の後に reduction(+:sum) と書く。**スレッド終了時に各スレッドで計算された sum の和を計算するため**「+:sum」**となっている。reduction で指示された変数はプライベート変数のように振る舞うが、初期値は演算子に応じた値に自動的に決まる。和（+）であれば変数の初期値は 0 と決められている。

```
#include <stdio.h>
#define N 10000

int main ()
{
  int i, a[N];
  int sum=0;

  for (i=0; i<N; i++) a[i] = i+1;

#pragma omp parallel for reduction(+:sum)
  for (i=0; i<N; i++){
    sum += a[i];
  }
  printf("sum = %d\n",sum);

  return 0;
}
```

課題 8

サンプルプログラムに含まれる sum_omp_bad.c をコンパイル、実行せよ。適切に和が計算できないことを確認せよ。さらに reduction 指示句を用いることで適切に和が計算されるように修正した sum_omp.c を作成制よ。

課題 9

サンプルプログラムに含まれる innerprod.c コンパイル、実行せよ。これはベクトル同士の内積を計算するプログラムである。このプログラムを OpenMP を用いて並列化した innerprod_omp.c を作成せよ。

課題 10

サンプルプログラムに含まれる calcpi.c をコンパイル、実行せよ。これは数値計算により円周率を求めるプログラムである。このプログラムを OpenMP を用いて並列化した calcpi_omp.c を作成せよ。並列化によって精、度が同一のまま処理速度が短縮されることを確認せよ。$\pi = 3.14159265358979323846264338327950288\cdots$

ここで TA による進捗チェックを受け、さらに次の発展課題に取り組め。

発展課題 1

行列同士の積を計算するプログラムを書け。さらに OpenMP を用いて並列化せよ。

4 （参考）共有メモリによるプロセス間通信

プロセス間では、メモリ空間が異なるので、単純に情報のやりとりをすることができない。マルチスレッドの場合は、同一プロセス内での処理となるのでメモリ空間が共有されており、グローバル変数で情報の共有が可能であったが、プロセス間ではこれは不可能である。そこで、共有メモリを用いて情報共有を行うことにする。

4.1 共有メモリのサンプルプログラム

まず、以下の課題を行い共有メモリの動作を見てみよう。

（参考）課題 1

1. sharedmemory.tar.gz を展開、make せよ。
2. shm-server を実行後に shm-client を実行し、サーバから a から z までの文字が共有メモリに書き込まれ、それがクライアントが読み出せること、クライアントが'*' を共有メモリに書き込むことでサーバが終了することを確認せよ。

簡単にプログラムの内容を見ていくことにする。共有メモリを使うためには、はじめに、お互いに読み書きできる掲示板のような役割を作る必要がある。掲示板自体が共有メモリにあたる。共有メモリを作るには shmget 関数（システムコール）を実行する。shmget 関数の使い方は次に示す通り。

```
int shmget(key_t key, size_t size, int shmflg);
```

共有メモリの領域獲得に成功すると共有メモリの ID を返し、失敗すると-1 を返す。

key_t key 共有メモリセグメントの数字名

size_t size 共有メモリセグメントのバイト数の大きさ

int shmflg 共有メモリを作成したいときに IPC_CREAT を指定。アクセスパーミッションも指定可能。例えば、shmflg に IPC_CREAT と IPC_EXCL を指定した場合、key がそれに対応する共有メモリをすでに確保している場合エラーを返す。

共有メモリは作成しただけでは利用できず、プログラムがその共有メモリをあたかも自分のメモリ領域のように扱うための準備をしてはじめて利用可能となる。共有メモリセグメントへの読み書きを許可する操作は、shmat 関数を使用する。shmat 関数の詳細は以下に示す通りである。

```
void *shmat(int semid, void *shmaddr, int shmflg);
```

この関数は、アドレスを含む文字ポインタを返す。失敗した場合には (char *)-1 が返される。

int semid shmget によって獲得された共有メモリの ID

void *shmaddr 一般に NULL を与える。NULL 以外は特殊なアプリケーションで利用される。

int shmflg SHM_RDONLY が指定されれば読取り専用となる。それ以外であれば、セグメントは読み書きが可能となる。通常、読み取り専用でなければ通常 0 を指定する。

共有メモリセグメントを使い終えたなら、プログラムはそのセグメントを取り外す。これには shmdt 関数を使用する。具体的な利用方法を以下に示す。

```
int shmdt(void *shmaddr);
```

セグメントの取り外しが成功すれば 0 を返し、失敗すれば-1 を返す。

void *shmaddr shmat によって結び付けられた共有メモリの開始アドレス

このプログラムには利用されていないが、共有メモリセグメントに関する情報の調査および変更が可能である。その関数として shmctl がある。このシステムコールの詳細を以下に示す。

```
int shmctl(int semid, int cmd, struct shmid_ds *buf);
```

返り値が 0 なら正常終了、-1 なら異常終了。

int semid 共有メモリ ID

int cmd コマンド。共有メモリセグメントの情報を得たいときは IPC_STAT を指定する。共有メモリセグメントに情報を設定したいときは IPC_SET を利用する。共有メモリセグメントの破棄をしたいときは、IPC_RMID を指定する。このとき buf には NULL を指定する。

struct shmid_ds *buf コマンドへ（から）の付加的な情報。

4.2　プロセス間の同期、セマフォ

マルチスレッドの場合には、mutex によりスレッド間の同期をとった。同様にマルチプロセスにおいても第 2.3 節であげた問題が生じるために同期を行う必要がある。ここではセマフォを用いてプロセス間の同期をとることにする。セマフォは複数のセマフォを同時に扱える仕組みを持ち高機能なため、その理解が難しくなっている。そこで、セマフォの最も単純な利用によりメモリの排他制御を行うことを目指す。（注意：ここで説明するセマフォの機能は非常に限定的なものである。）

セマフォを作成 セマフォを作成するには semget システムコールを利用する。このシステムコールの利用方法は次の通りである。

```
int semget(key_t key, int nsems, int semflg);
```

key_t key セマフォ識別子であり、semget の key と同じ意味を持つ。

int nsems 作成するセマフォの個数であり、複数指定できるが、今回は 1 とする。

int semflg セマフォを作成したい場合には、IPC_PRIVATE を指定する。同時にアクセス許可の定義も指定できる。

セマフォを初期化 セマフォを初期化するには semctl システムコールを利用する。この利用方法は次の通りである。

```
int semctl(int semid, int semnum, int cmd, union semun arg);
```

int semid semget システムコールの戻り値を設定する。

int semnum 何番目のセマフォを操作するかを指定する。最初のセマフォは 0 となる。

int cmd セマフォに対しどのような操作を行うかを指定する。初期化には値を設定する必要があるので、SETVAL を利用する。後に述べるセマフォの削除には IPC_RMID を指定する。

union semun arg SETVAL を用いた場合、セマフォの値を指定する。セグメントに入り込んだプロセスがセマフォの値を足したり、引いたりするための初期値。

排他制御 排他制御を実際に行うには semop システムコールを利用する。

```
int semop(int semid, struct sembuf sops, unsigned nsops);
```

int semid semget システムコールの戻り値を指定する。

struct sembuf sops sumbuf 構造体は次のようになっている。

```
struct sembuf{
   short sem_num; /*セマフォの番号*/
   short sem_op;  /*セマフォの操作*/
   short sem_flg; /*操作のフラグ*/
};
```

sem_num はアクセス権を要求するセマフォの番号を指定する。

sem_op この値が正ならば、セマフォの値に sem_op の値を加算し、処理はすぐに返る。この値が負なら、セマフォの値から sem_op の値（絶対値）を減算しようとする。減算結果が負なら、semop はその値が 0 以上になるまでブロックされる。0 以上になれば、semop から制御が返る。例えば、セマフォの値が 1 で sem_op に-1 を指定した場合、セマフォの値が減算され 0 となり、制御が返る。セマフォの値が 0 で sem_op に-1 を指定した場合、セマフォの値が 1 以上になるまでブロックされる。これより、簡単には、アクセス権を要求するには、sem_op に-1 を設定し、処理の終了時に sem_op に 1 を設定すればよい。

sem_flg に、例えば IPC_NOWAIT を指定すると、プロセスがブロックされる状況にあってもブロックされずに、semop によってエラーコードがすぐに返される。

unsigned nsops sembuf 構造体配列の数。

セマフォの削除 セマフォの削除には semctl で IPC_RMID を指定する。第 4 引数は特に指定しなくて良い。

（参考）課題 2

1. semaphore.tar.gz を展開、make せよ。
2. sem-server を実行後に sem-client を実行せよ。sem-client は共有メモリからデータを読み出そうとするがセマフォによりブロックされて読み込めない。sem-server で何らかのキー入力を行うことでブロックが外れて、sem_client が共有メモリにアクセス可能となる様子を確認せよ。