ソースファイルが1ファイル

```
#include <stdio.h>      /*ヘッダーファイル*/

#defineN         100 /* マクロの定義 */

struct point{                    /* 構造体の宣言 */
   int x,y;
};

typedef long SInt32     /* 型定義 */

int func( int x); /* 関数プロトタイプ宣言 */

int main(void)
{
   /*関数本体*/
}
```

ヘッダーファイルに記述することが多い

関数のプロトタイプ宣言

関数本体を関数の
使用部分より上に書く

```
int func(void){
  printf(“hello\n”);
}

int main(void)
{
  func();
}
```

関数本体を関数の
使用部分より下に書く

```
/*プロトタイプ宣言*/
int func(void);

int main(void)
{
  func();
}

int func(void){
  printf(“hello\n”);
}
```

通常は，プロトタイプ宣言を必ず書く

１つのファイル

main.c

```
 1:  /* ヘッダーファイルの組み込み */
 2:  #include <stdio.h>
 3:
 4:  /* マクロの定義 */
 5:  #define              N          100
 6:  #define              M          5000
 7:
 8:  /* 関数プロトタイプ宣言 */
 9:  int sum( int x , int y );
10:  void hyoji ( int kekka );
11:
12:  void main( void )
13:  {
14:     int   kekka;
15:
16:     kekka  = sum( N , M );
17:     hyoji( kekka );
18:  }
```

```
19:  /* x から yまでの合計を求める */
20:  int sum( int x , int y )
21:  {
22:     int   i , kekka;
23:
24:     kekka = 0;
25:     for(i=x;i<=y;i++) kekka +=i;
26:
27:     return kekka;
28:  }
29:
30:  /* 計算結果を表示する */
31:  void hyoji ( int kekka )
32:  {
33:     printf ("kekka = %d¥n”, kekka);
34:  }
```

２つのファイルに分割

main.c

```
 1: /* ヘッダーファイルの組み込み */
 2: #include <stdio.h>
 3:
 4: /* マクロの定義 */
 5: #define          N          100
 6: #define          M          5000
 7:
 8: /* 関数プロトタイプ宣言 */
 9: int sum( int x , int y );
10: static void hyoji ( int kekka );
11:
12: void main( void )
13: {
        略
18: }
19:
20: /* 計算結果を表示する */
21: static void hyoji ( int kekka )
22: {
23:   printf ("kekka = %d¥n”, kekka);
24: }
```

sum.c

```
 1: /* 関数プロトタイプ宣言 */
 2: int sum( int x , int y );
 3:
 4: /* x から yまでの合計を求める */
 5: int sum( int x , int y )
 6: {
 7:    int   i , kekka;
 8:
 9:    kekka = 0;
10:    for(i=x;i<=y;i++) kekka +=i;
11:
12:   return kekka;
13: }
```

static と extern

別ファイルの関数を呼び出すためには，
関数を「extern」で宣言する.

extern int sum(int x, int y);

この場合のexternは省略できるが，省略しない方が
外部関数の定義であることがわかりやすい.

「static」をつけて宣言した関数は，他のファイルから
呼び出すことはできない.

static void hyoji(int kekka);

ヘッダーファイルの作成

main.c

```
 1: /* ヘッダーファイルの組み込み */
 2: #include <stdio.h>
 3: #include “sum.h”
 4:
 5: /* マクロの定義 */
 6: #define          N          100
 7: #define          M          5000
 8:
 9: /* 関数プロトタイプ宣言 */
10: static void hyoji ( int kekka );
11:
12: void main( void )
13: {
                    略
18: }
19:
20: /* 計算結果を表示する */
21: static void hyoji ( int kekka )
22: {
23:    printf ("kekka = %d¥n”, kekka);
24: }
```

sum.h

```
1: /* 関数プロトタイプ宣言 */
2: int sum( int x , int y );
```

sum.c

```
 1: /* ヘッダーファイルの組み込み */
 2: #include “sum.h”
 3:
 4: /* x から yまでの合計を求める */
 5: int sum( int x , int y )
 6: {
 7:    int   i , kekka;
 8:
 9:    kekka = 0;
10:    for(i=x;i<=y;i++) kekka +=i;
11:
12:    return kekka;
13: }
```

複数人でのプログラムの作成法

1. 共通な部分（ヘッダーファイル）を作成する

2. 個々で独立にプログラムを作成し，テストを行う.

3. 途中で1で作成したものでは都合が悪いことが
   判明した場合には，もう一度作成しなおして，
   全員に通知する.

4. 全員が作成したプログラムをつなげて
   プログラムを完成させる.

B君のプログラム

test_main.c

```
 1:  /* ヘッダーファイルの組み込み */
 2:  #include <stdio.h>
 3:  #include “sum.h”
 4:
 5:  void main( void )
 6:  {
 7:    int   a , b;
 8:
 9:    /* 2つの整数を読み込む */
10:    scanf( “%d”, &a );
11:    scanf( “%d”, &b );
12:
13:    /* a から bまでの合計を計算する */
14:    kekka  = sum( a , b );
15:
16:    /* 結果を表示する */
17:    printf( “%d\n”, kekka );
18:  }
```

sum.h

```
 1:  /* 関数プロトタイプ宣言 */
 2:  int sum( int x , int y );
```

sum.c

```
 1:  /* ヘッダーファイルの組み込み */
 2:  #include “sum.h”
 3:
 4:  /* x から yまでの合計を求める */
 5:  int sum( int x , int y )
 6:  {
 7:    int   i , kekka;
 8:
 9:    kekka = 0;
10:    for(i=x;i<=y;i++) kekka +=i;
11:
12:    return kekka;
13:  }
```

A君のプログラム

main.c

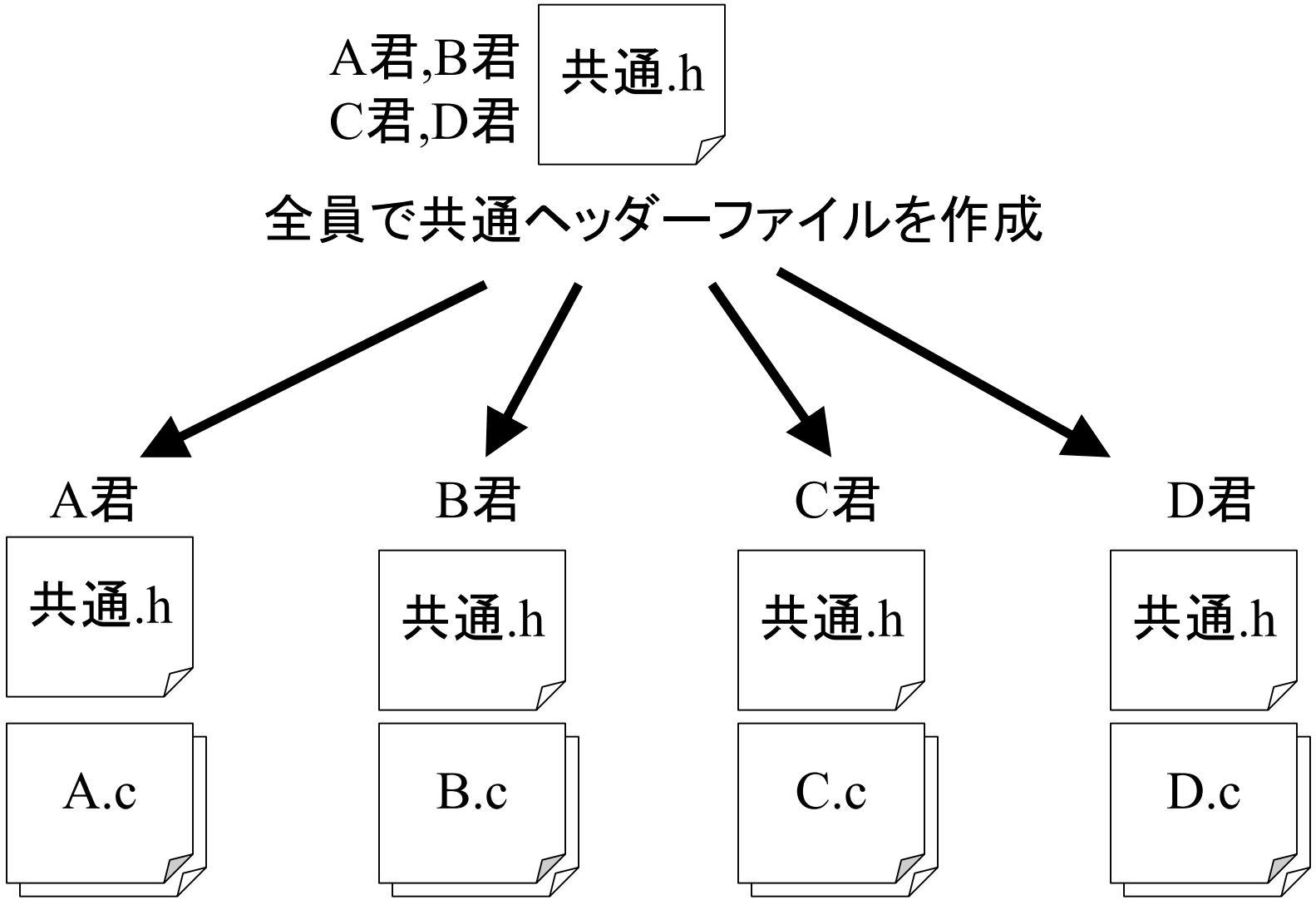
```
 1: /* ヘッダーファイルの組み込み */
 2: #include <stdio.h>
 3: #include “sum.h”
 4:
 5: /* マクロの定義 */
 6: #define              N          100
 7: #define              M          5000
 8:
 9: /* 関数プロトタイプ宣言 */
10: static void hyoji ( int kekka );
11:
12: void main( void )
13: {
14:   int   kekka;
15:
16:   kekka  = sum( N , M );
17:   hyoji( kekka );
18: }
19:
20: /* 計算結果を表示する */
21: static void hyoji ( int kekka )
22: {
23:   printf ("kekka = %d¥n”, kekka);
24: }
```

sum.h

```
1: /* 関数プロトタイプ宣言 */
2: int sum( int x , int y );
```

test_sum.c

```
 1: /* ヘッダーファイルの組み込み */
 2: #include “sum.h”
 3:
 4: /* x から yまでの合計を求める */
 5: int sum( int x , int y )
 6: {
 7:   printf(“%d から %d の合計を求める\n”, x , y );
 8:
 9:   return x+y;          /* 適当です */
10: }
```


A君,B君
C君,D君
共通.h
全員で共通ヘッダーファイルを作成
A君
B君
C君
D君
共通.h
共通.h
共通.h
共通.h
A.c
B.c
C.c
D.c
各人が別々に受け持ちのモジュール，ファイルを作成
全員がファイルを持ち寄って，プログラムを完成させる

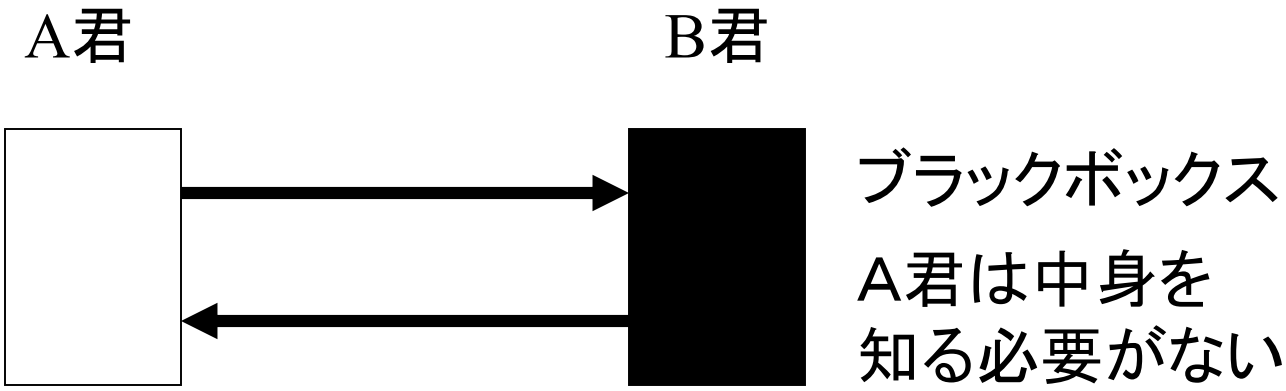
B君の合計値の求め方を変更

1 ～ n までの合計を求める式

$$sigma(n) = \frac{1}{2} n(n+1)$$

x ～ y までの合計を求める式

$$sigma(\mathrm{y}) - sigma(\mathrm{x} - 1)$$



入力と出力（外部関数の引数と返り値）は変更なし

B君のプログラムを変更

main.c

```
 1: /* ヘッダーファイルの組み込み */
 2: #include <stdio.h>
 3: #include “sum.h”
 4:
 5: /* マクロの定義 */
 6: #define          N          100
 7: #define          M          5000
 8:
 9: /* 関数プロトタイプ宣言 */
10: static void hyoji ( int kekka );
11:
12: void main( void )
13: {
        略
18: }
19:
20: /* 計算結果を表示する */
21: static void hyoji ( int kekka )
22: {
23:   printf ("kekka = %d¥n”, kekka);
24: }
```

sum.h

```
1: /* 関数プロトタイプ宣言 */
2: int sum( int x , int y );
```

sum.c

```
 1: /* ヘッダーファイルの組み込み */
 2: #include “sum.h”
 3:
 4: static int sigma(int n);
 5:
 6: /* x から yまでの合計を求める */
 7: int sum( int x , int y )
 8: {
 9:   int   kekka;
10:
11:   kekka = sigma(y)- sigma(x-1);
12:
13:   return kekka;
14: }
15:
16: /* 1 から n までの合計を求める */
17: static int sigma(int n)
18: {
19:  return 1/2*n*(n+1)
20: }
```

1ファイル内で有効なグローバル変数

main.c

```
 1: /* ヘッダーファイルの組み込み */
 2: #include <stdio.h>
 3: #include “sum.h”
 4:
 5: /* マクロの定義 */
 6: #define          N          100
 7: #define          M          5000
 8:
 9: /* グローバル変数の定義 */
10: static int gKekka = 0 ;
11:
12: /* 関数プロトタイプ宣言 */
13: static void hyoji (void);
14:
15: void main( void )
16: {
17:   gKekka = sum( N , M );
18:   hyoji();
19: }
20:
21: /* 計算結果を表示する */
22: static void hyoji (void)
23: {
24:   printf ("kekka = %d¥n”, gKekka);
25: }
```

sum.h

```
1: /* 関数プロトタイプ宣言 */
2: int sum( int x , int y );
```

sum.c

```
 1: /* ヘッダーファイルの組み込み */
 2: #include “sum.h”
 3:
 4: static int sigma(int n);
 5:
 6: /* x から yまでの合計を求める */
 7: int sum( int x , int y )
 8: {
 9:   int   kekka;
10:
11:   kekka = sigma(y)- sigma(x-1);
12:
13:   return kekka;
14: }
15:
16: /* 1 から n までの合計を求める */
17: static int sigma(int n)
18: {
19:   return 1/2*n*(n+1)
20: }
```

全ファイルで有効なグローバル変数

main.c

```
 1: /* ヘッダーファイルの組み込み */
 2: #include <stdio.h>
 3: #include “sum.h”
 4:
 5: /* マクロの定義 */
 6: #define          N          100
 7: #define          M          5000
 8:
 9: /* グローバル変数の定義 */
10: int gKekka = 0 ;
11:
12: /* 関数プロトタイプ宣言 */
13: static void hyoji (void);
14:
15: void main( void )
16: {
17:    sum( N , M );
18:    hyoji();
19: }
20:
21: /* 計算結果を表示する */
22: static void hyoji (void)
23: {
24:    printf ("kekka = %d¥n”, gKekka);
25: }
```

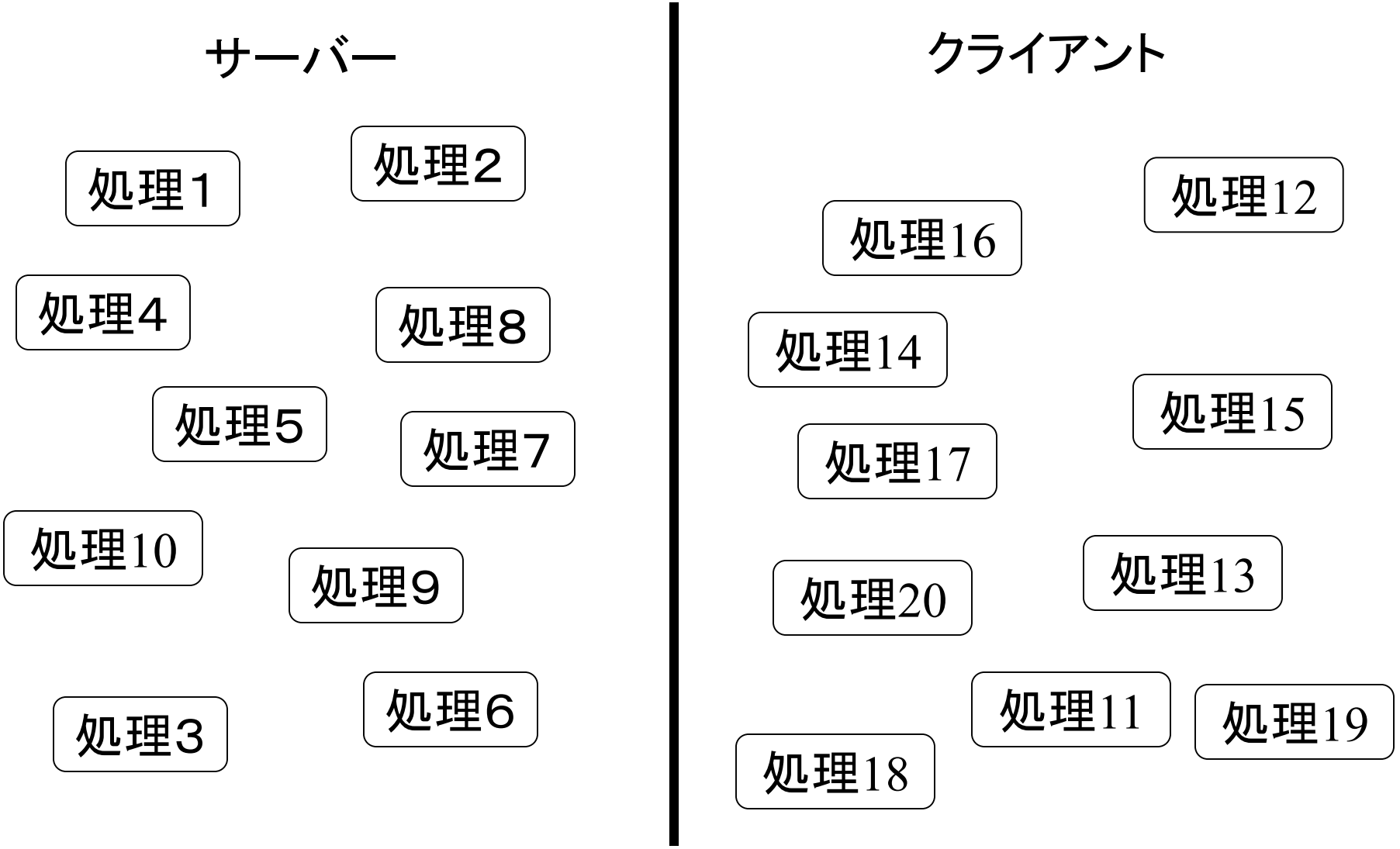
sum.h

```
1: /* グローバル変数の宣言 */
2: extern int gKekka ;
3:
4: /* 関数プロトタイプ宣言 */
5: void sum( int x , int y );
```

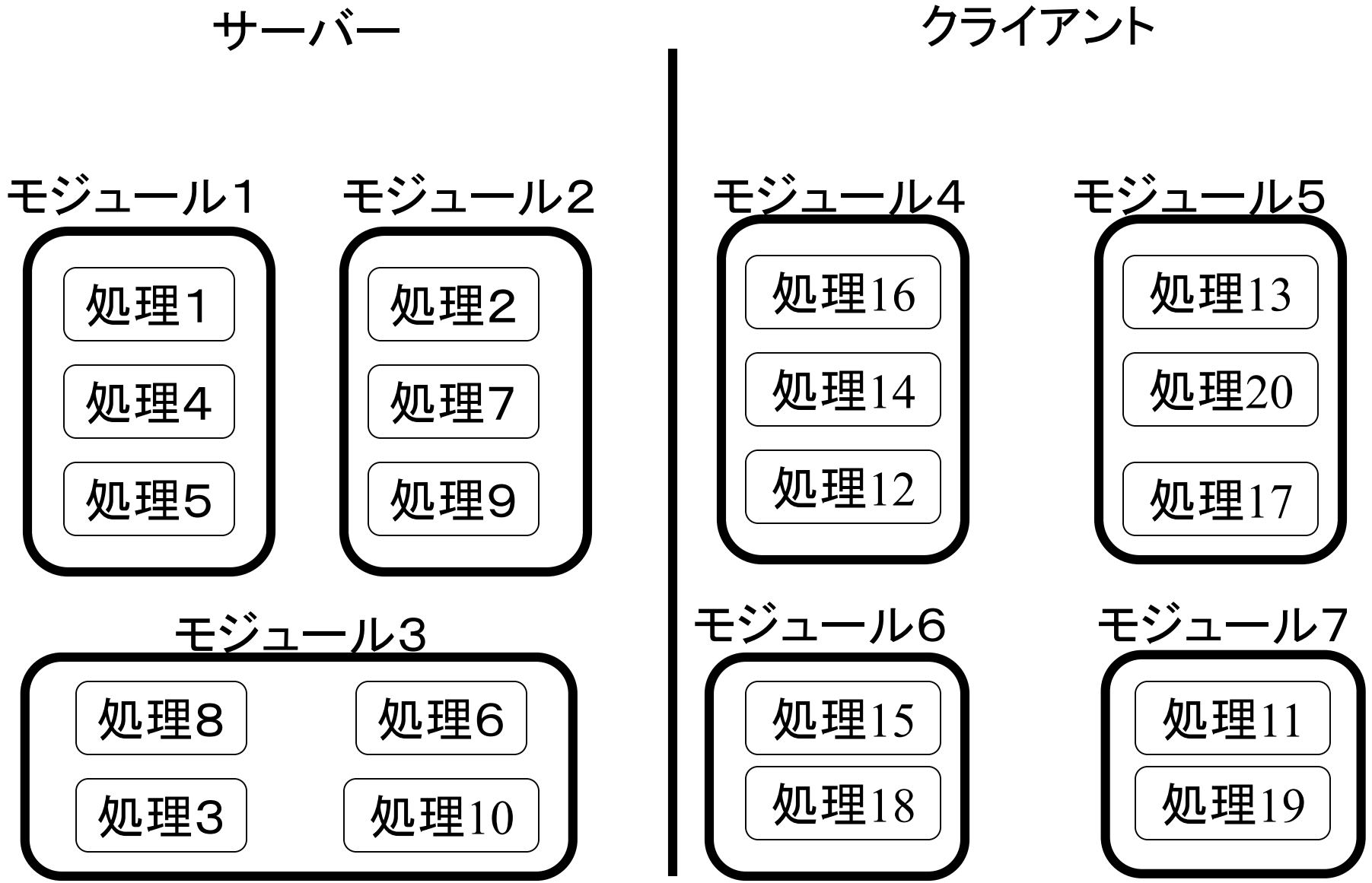
sum.c

```
 1: /* ヘッダーファイルの組み込み */
 2: #include “sum.h”
 3:
 4: static int sigma(int n);
 5:
 6: /* x から yまでの合計を求める */
 7: void sum( int x , int y )
 8: {
 9:    gKekka  = sigma(y)- sigma(x-1);
10: }
11:
12: /* 1 から n までの合計を求める */
13: static int sigma(int n)
14: {
15:    return 1/2*n*(n-1)
16: }
```

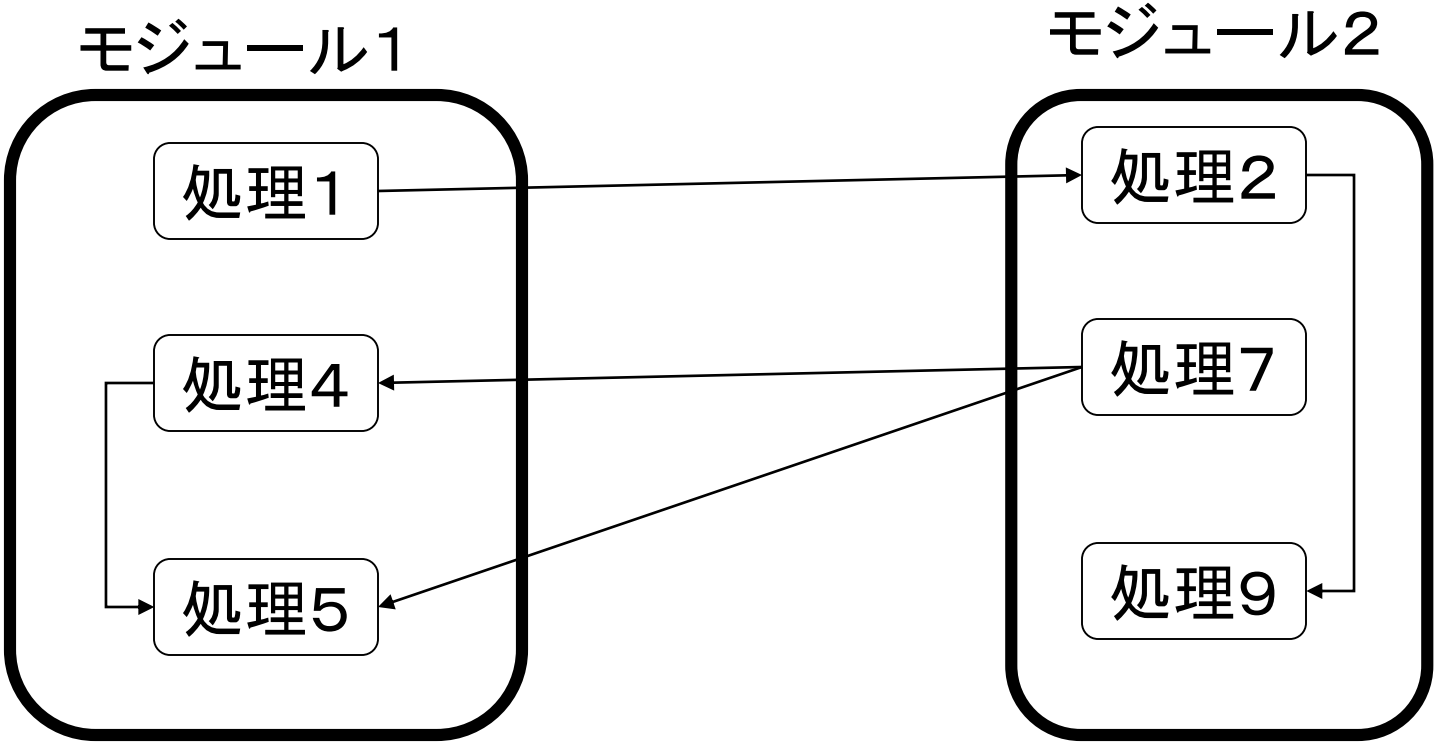
モジュール化１


サーバー
処理１
処理２
処理４
処理８
処理５
処理７
処理10
処理９
処理６
処理３
クライアント
処理12
処理16
処理14
処理15
処理17
処理13
処理20
処理11
処理19
処理18

モジュール化2

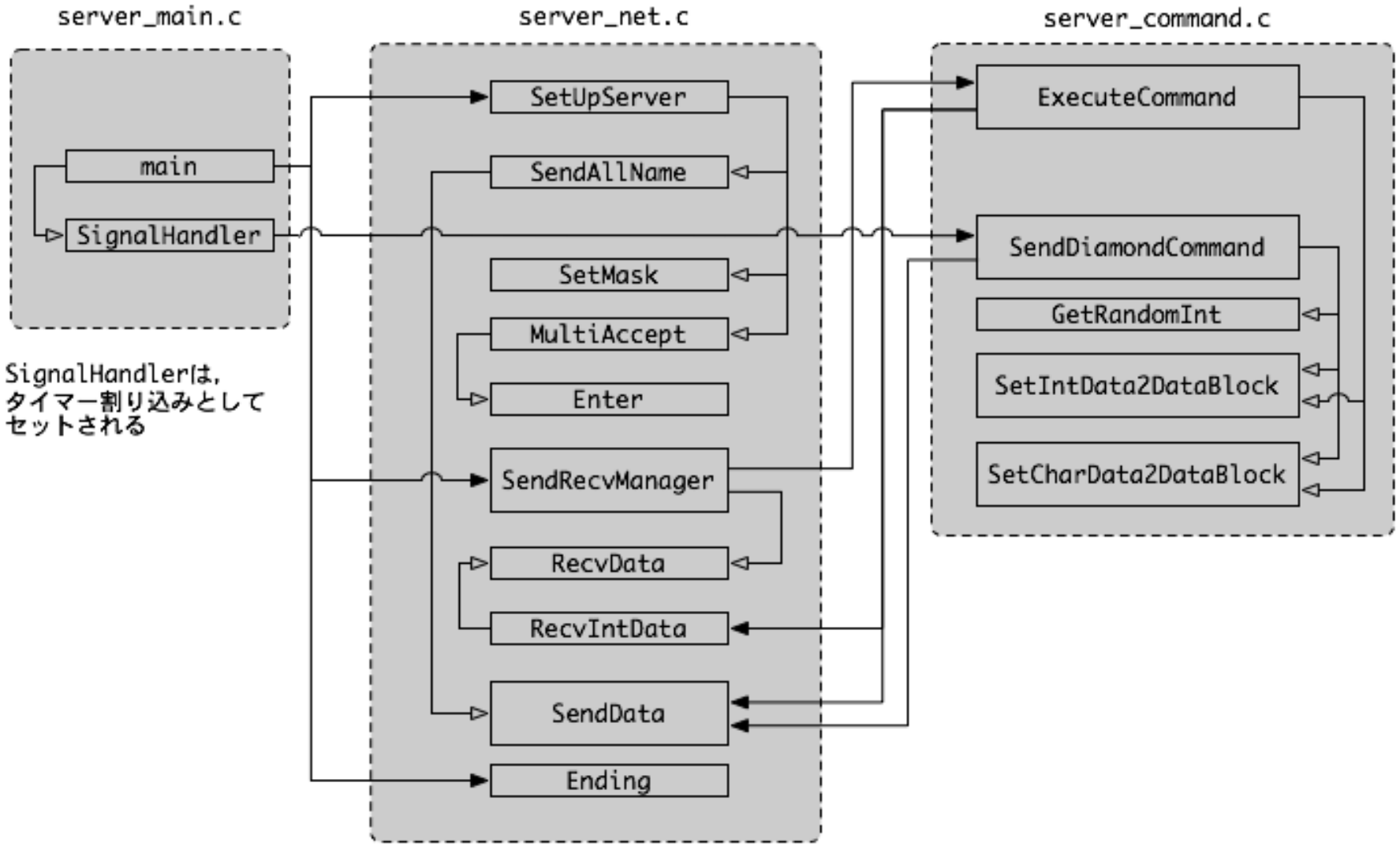

サーバー
クライアント
モジュール1
処理1
処理4
処理5
モジュール2
処理2
処理7
処理9
モジュール3
処理8
処理6
処理3
処理10
モジュール4
処理16
処理14
処理12
モジュール5
処理13
処理20
処理17
モジュール6
処理15
処理18
モジュール7
処理11
処理19

モジュール化3

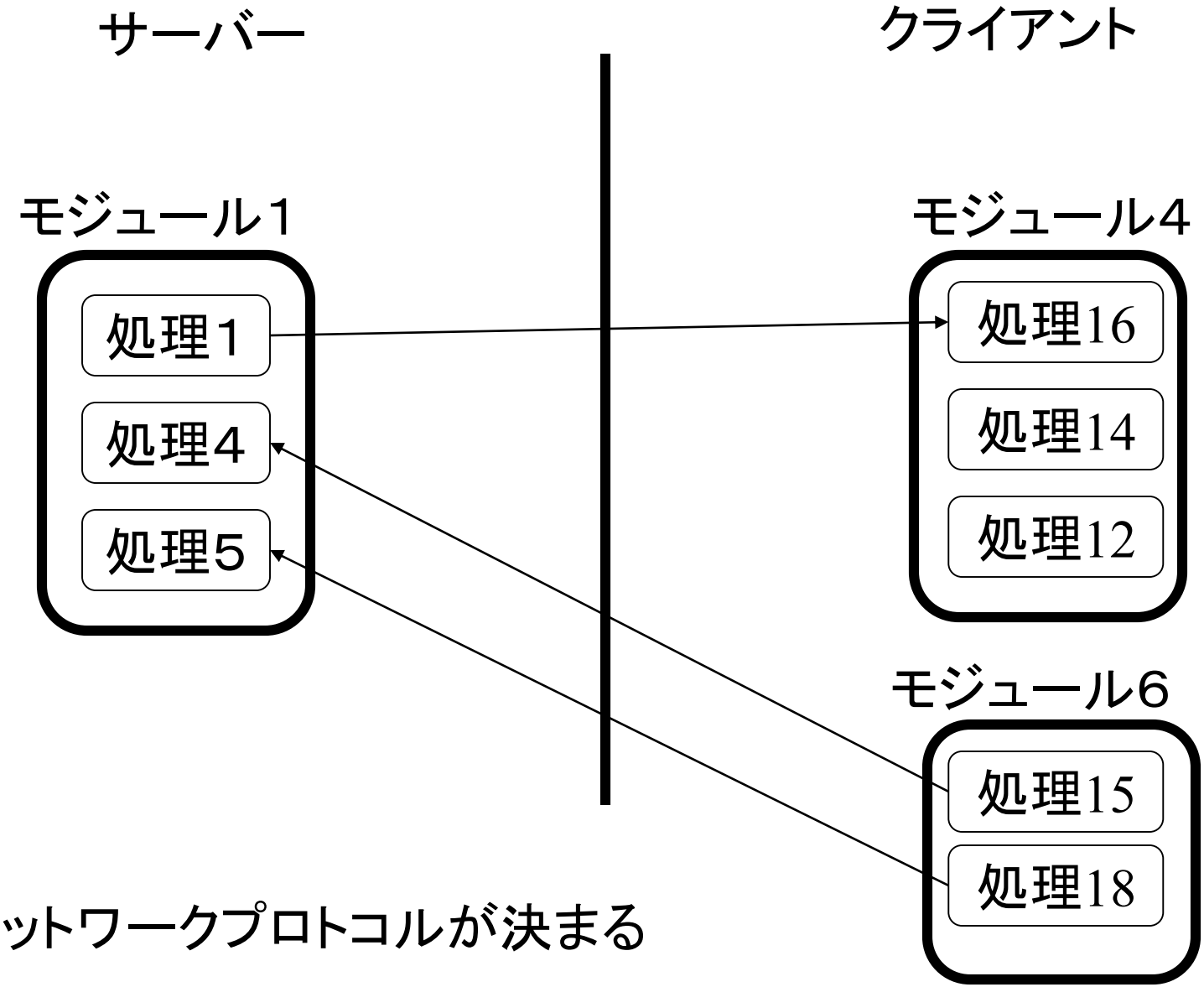


外部関数，データ構造などが決まる

サーバーの関数呼び出し関係



モジュール化４



ネットワークプロトコルが決まる

クライアントのメインイベントループ

```
main()
{
  /* 初期化 */

   endFlag = 1;
   while(endFlag){
       WindowEvent();
       endFlag = SendRecvManager();
   }

   /* 終了処理 */
}
```

```
WindowEvent()
{
    SDL_PollEvent();
    /* イベント処理 */
}
```

```
SendRecvManager ()
{
    select();
    /* イベント処理 */
}
```

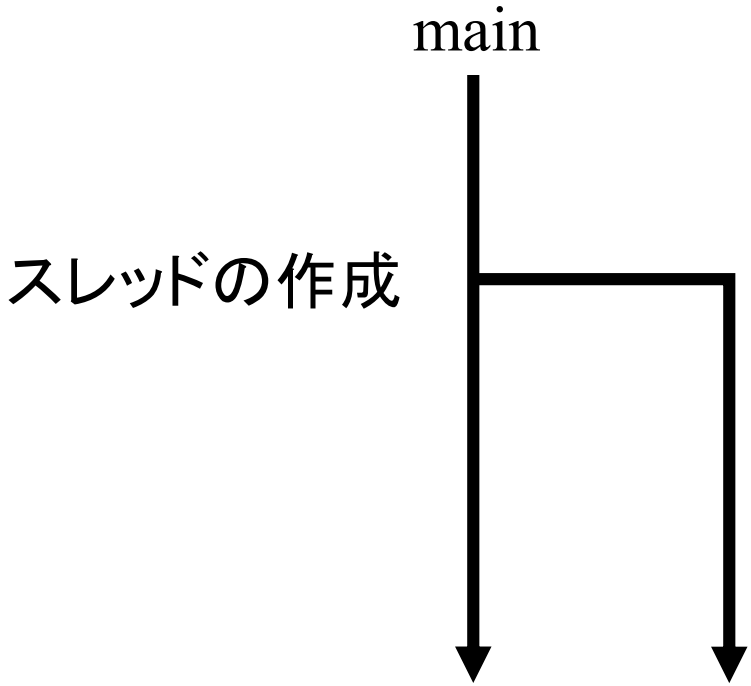
SDLのGUI と ネットワークを使ったプログラムでは,
両方のイベントを調べて処理する必要がある.

どちらかで時間のかかる処理が実行されると困る

スレッド

シングルスレッド

マルチスレッド



マルチスレッドプログラムでは，
2つ以上の処理を同時に実行できる
複雑な時間のかかる処理などを，別スレッドで実行させる

スレッドの作成例

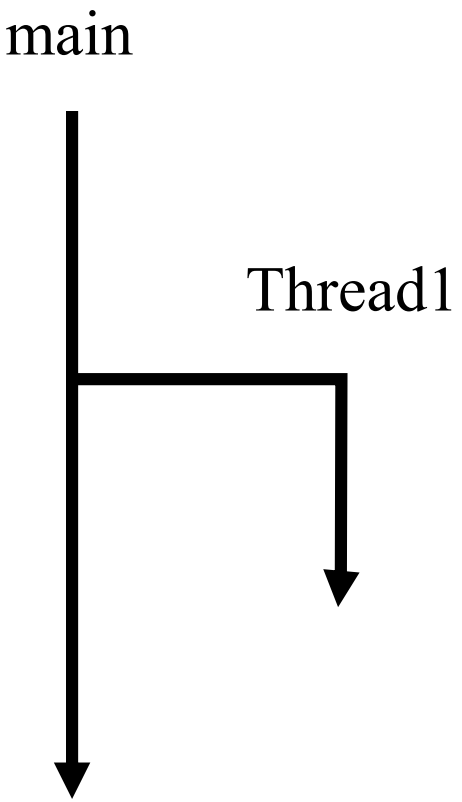
```
#include<pthread.h>

int *Thread1(void *data)
{
  /* 処理 */
  return 0; //スレッドの終了
}
int main()
{
  SDL_Thread thr;  //スレッドID
  /*スレッドの作成*/
  thr = SDL_CreateThread(&Thread1, NULL);
  / *  メインの処理 */

  SDL_WaitThread(thr, NULL); //終了を待つ

  return 0;
}
```

mainとThread1が
同時に処理される



スレッドを使ったイベントループ

```
main()
{
   /* 初期化 */
   endFlag = 1;
    thr = SDL_CreateThread(
          &NetworkEvent, &endFlag);
   while(endFlag){
      endFlag = WindowEvent();
   }
   /* 終了処理 */
}
void NetworkEvent(void *data)
{
 int *endFlag=(int*)data;
  while(*endFlag ){
     *endFlag  = SendRecvManager();
  }
}
```

```
WindowEvent()
{
  SDL_PollEvent();
  /* イベント処理 */
}
```

```
SendRecvManager ()
{
  select();
  /* イベント処理 */
}
```

select と割り込み処理

p.6右 server_net.c　L.105

```
if(select(gWidth,&readOK,NULL,NULL,NULL) < 0){
  /* エラーがおこった */
  return endFlag;
}
```

select の第５引数にNULLを指定すると，timeoutがないので，データが来るまで，プログラムはここでストップ

ストップしているときに割り込み処理がおこると，
select 関数はエラーとして，負の値を返して終了する.

このエラー処理がないと，データが届いていないのに，
データが届いているとみなされ，プログラムが実行される
sleep などの関数も同じことが起こる

Wiiリモコンとネットワーク

Wiiリモコンのイベントを取得する関数wiimote_updateは,
Wiiリモコンからのイベントがあるまで, プログラムが
ストップしてしまいます.
Wiiリモコンを使ったネットワークプログラムを作る場合,
ネットワークの処理が行われない.

解決法１）
Wiiリモコンとネットワークを別スレッドにする

解決法２）
加速度センサを使用する（wiimote.mode.acc = 1 ）
加速度センサの値が常に更新されるので,
wiimote_updateで処理が中断しない

#ifdef assert NDEBUG

p.15右 client_command.c L.85

```
assert(0 <= pos && pos<MAX_CLIENTS);

#ifndef NDEBUG
  printf ・・・・
#endif
```

引数の値が正しいかチェックをしている

p.1左 Makefile L.5

```
CFLAGS = -c -DNDEBUG
```

← #define NDEBUG と同じ

この「 -DNDEBUG」を消せば，assertと
#ifndef NDEBUG が有効になるので，デバッグ中は消す．
デバッグが終われば，「 -DNDEBUG」を書く．

ヘッダーの２重インクルード１

sample.c

```
#include”a.h”
#include”b.h”

main()
{
}
```

a.h

```
struct pos{
   int x, y;
};
```

b.h

```
#include ”a.h”
```

includeを展開すると

```
struct pos{
   int x, y;
};
struct pos{
   int x, y;
};

main()
{
}
```

同じ名前の構造体が
２回定義されており，
エラーとなる

ヘッダーの２重インクルード２

a.h

```
#ifndef _A_H_
#define _A_H_
struct pos{
   int x, y;
};
#endif
```

b.h

```
#include ”a.h”
```

```
#ifndef _A_H_
#define _A_H_
struct pos{
   int x, y;
};
#endif

#ifndef _A_H_
#define _A_H_
struct pos{
   int x, y;
};
#endif
```

_A_H_が定義されていないので，
実行される

_A_H_が定義されているので，
実行されない

２重定義にならない！

課題

GUI とネットワークを使った「じゃんけんゲーム」を
作成しなさい.

・２クライアントで対戦
・画面に「グー」「チョキ」「パー」を表すボタン（３個）を表示
・サーバーで勝敗を判定する

グー

チョキ

パー

サーバー

それぞれのクライアントが選択した手を保持
両クライアントからデータが届いたら判定
結果をクライアントに報告

クライアント1

グー　チョキ　パー

クライアント2

グー　チョキ　パー

レポートに関する注意

- 実行に必要な全てのファイルを提出する
  アカウント番号の名前のディレクトリを圧縮して提出
  tar zcvf c50xxxxxxx.tgz c50xxxxxxx

- 作成したプログラムの内容，操作方法，
  新たに作成した外部関数，コマンドプロトコルなどを
  必ず記述すること　（付録A．１節～A．５節を参照）
  これがないと，再レポートになります

- ボーナスポイント用に改良した点をレポート内に記述すること