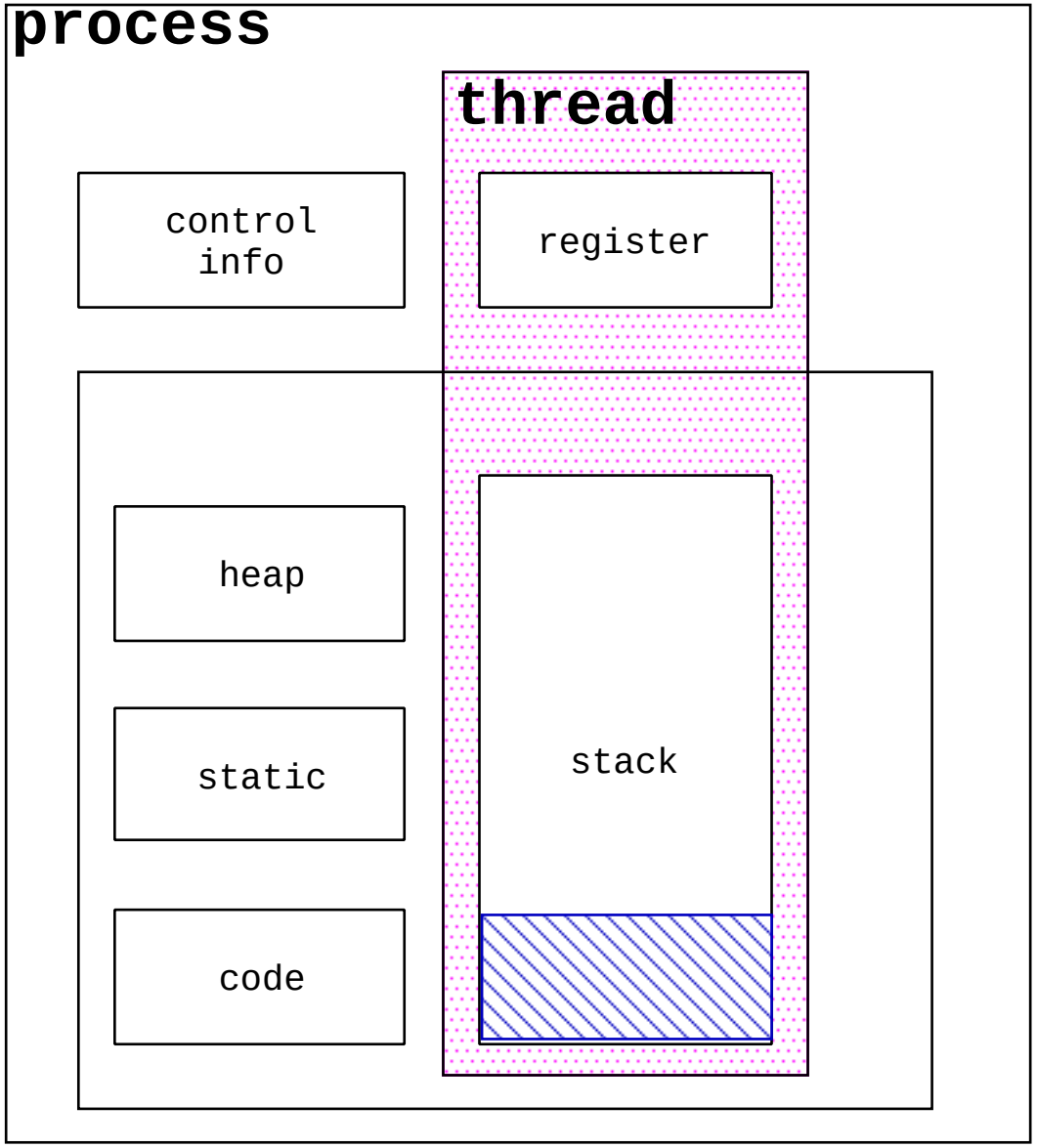
情報科学プロジェクト実験

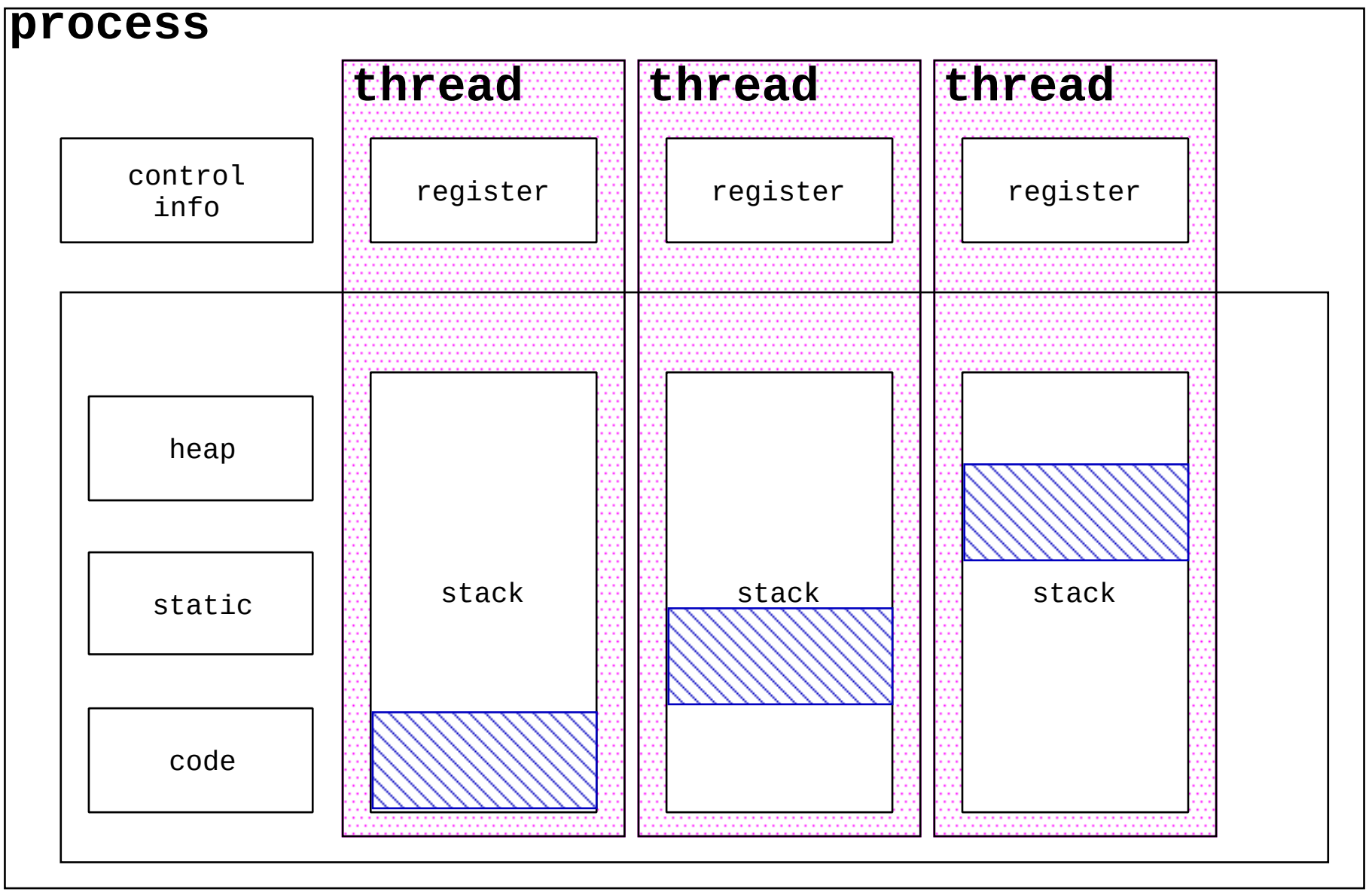
コンピュータシステム研究室

第13回目：スレッドプログラミング

スレッドとは何か？

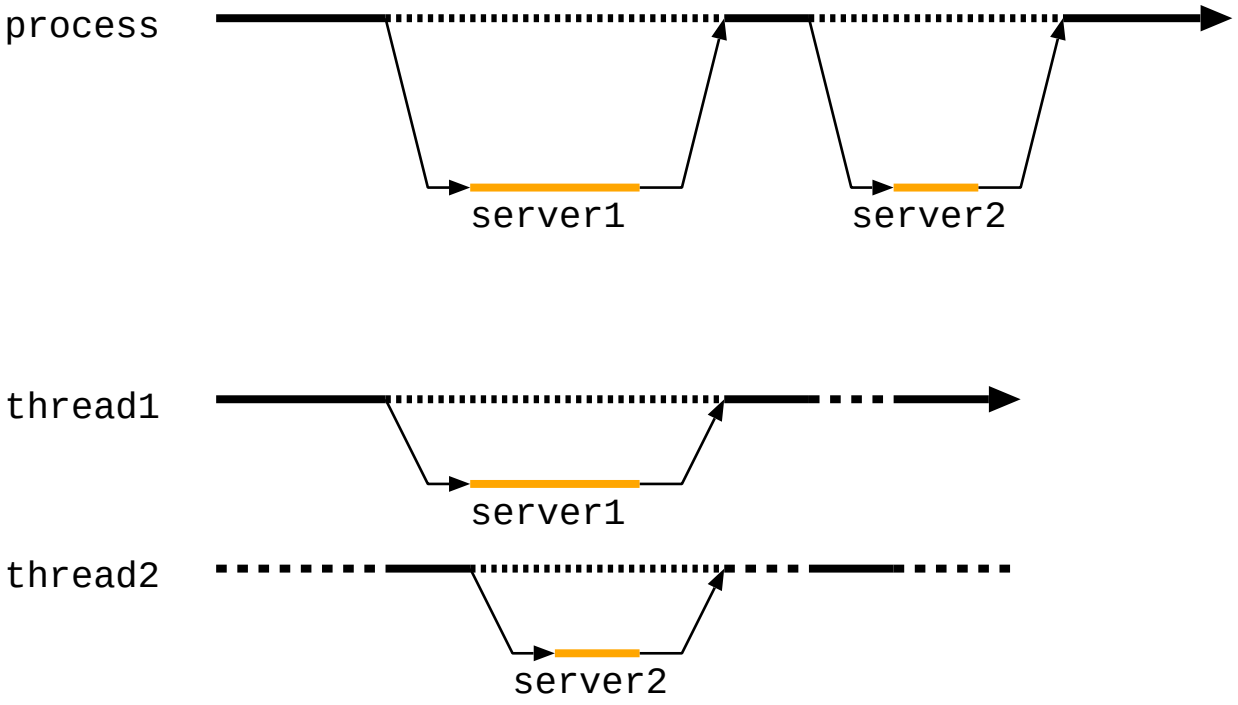
- UNIXプロセス内に存在する
  プログラムコードの独立した実行の並び.
- プロセス内で複数の制御スレッドが同時に
  アクティブになれる.
- 共有して使うもの
  大域変数, ヒープから取得したメモリ領域,
  ファイル記述子
- 専有して使うもの
  レジスタに保存される各種の情報(PC, SP,...)
  スタック領域は共有可能であるが,
  個別に使うこと想定


process
thread
control info
register
heap
static
code
stack


process
thread
thread
thread
control info
register
register
register
heap
static
code
stack
stack
stack

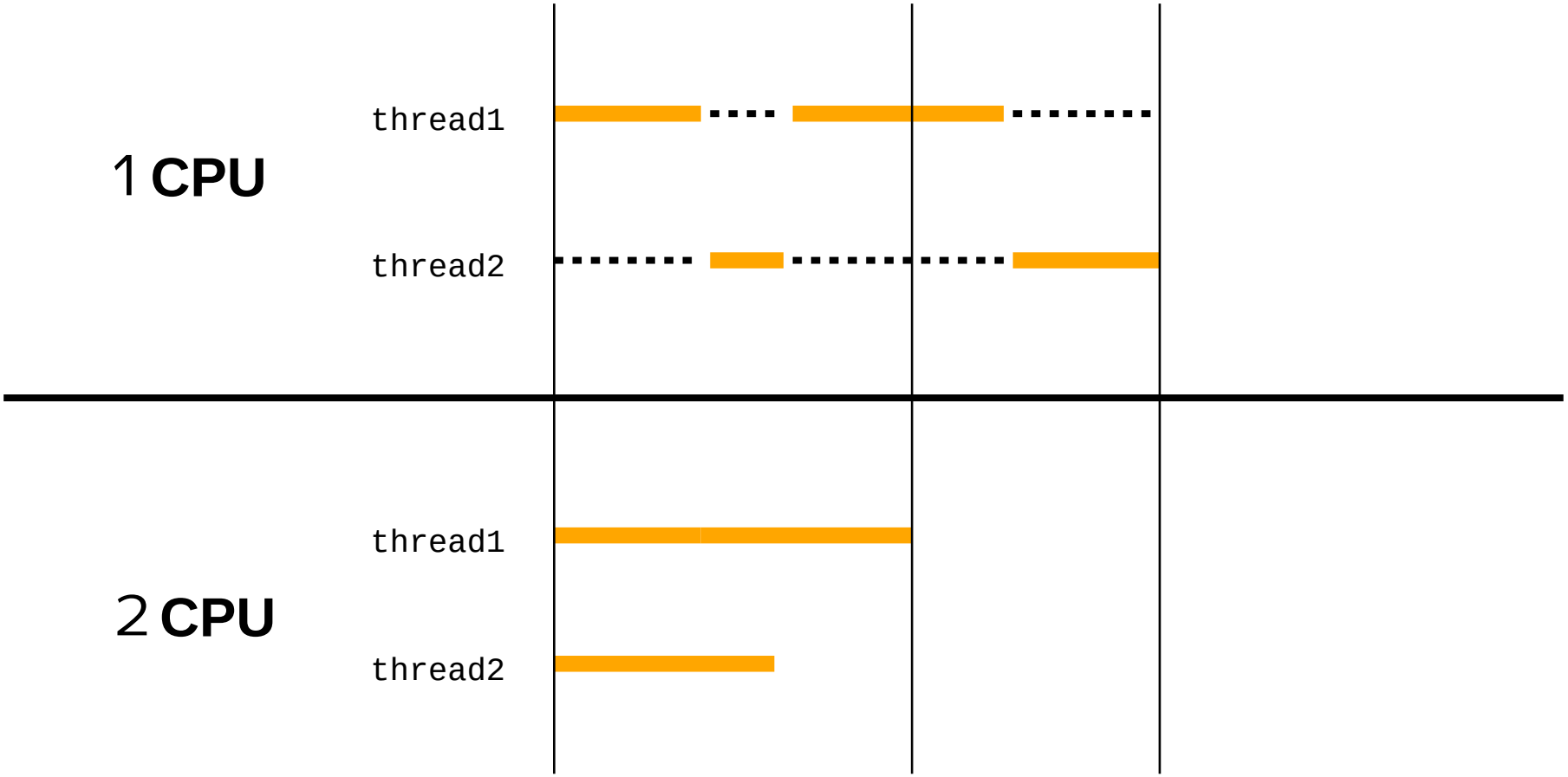
スループットの向上

- I/Oリクエストと他の処理のオーバラップ
- 複数の遠隔手続き呼び出し(RPC)



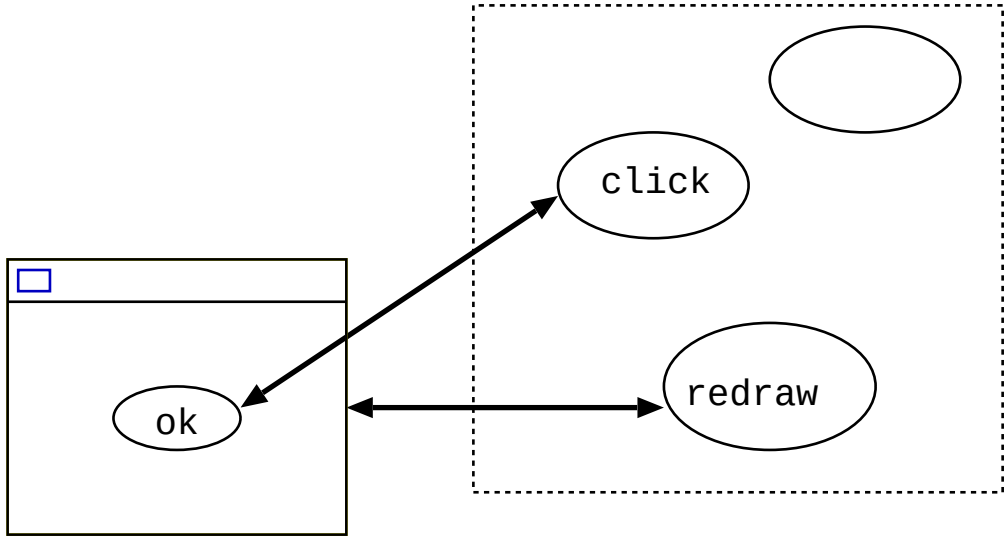
マルチプロセッサ・マルチコアによる並列処理

・複数の処理を同時に行う



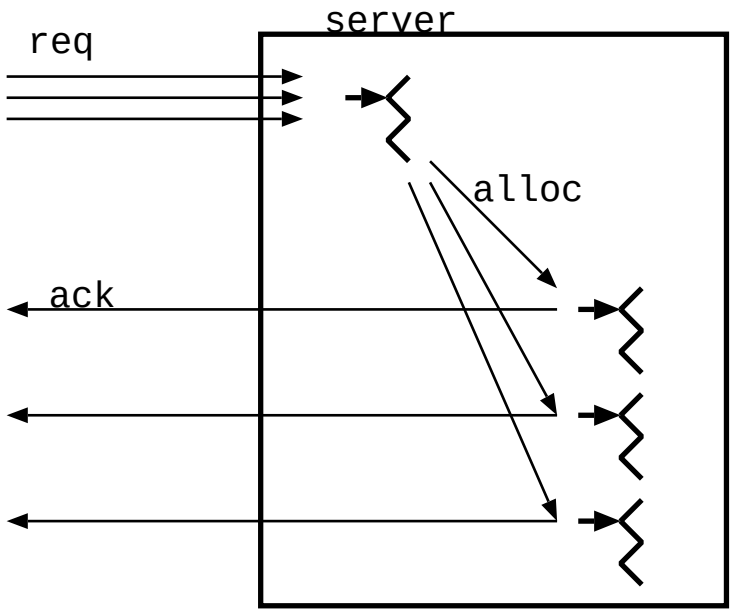
GUIにおける応答性の向上

ユーザ対応とバックグラウンド処理



C/S処理：サーバの応答性

・データベースサーバ
・Webサーバ



プログラム構造の単純化

独立性の高いモジュールを
スレッドとして設計できる
例：

・Windowプログラミング
　　インターフェース
　　バックグラウンド処理

・シミュレータ構築
　　構成要素ごとの処理

スレッドプログラミングの注意点

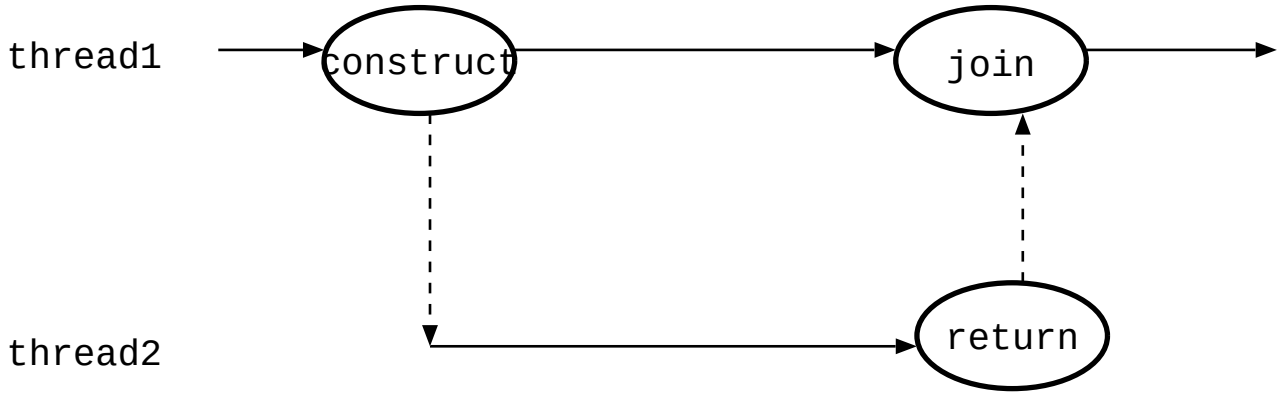
- 単一CPUで単一の計算が主体の場合には不要
- I/Oの並行性はシステムが提供している場合がある
- スレッドの数とCPU数は密接な関係にある
- 設計とデバッグが困難となる場合がある

c++ threadの基本関数

std::thread x(func):コンストラクタ：スレッドの生成
x.join(): スレッドの終了待ち
x.detach(): スレッドの切り離し
x.get_id(): スレッドIDの取得

std::this_thread::get_id():自スレッドIDの取得
std::this_thread::yield(): プロセッサを他に譲る

std::thread::hardware_concurrency():同時実行可能数の取得



2スレッドの例

```
#include <iostream>
#include <thread>
using namespace std;
int result = 0;

void func(int id)
{
        cout << "func " << id  << "\n";
        result = 1;
}

int main()
 {
        thread t(func, 0);
        cout << "main\n";
        t.join();
        cout << "done "<<result<<"\n";
}
```

コンパイル方法と実行例

```
$ g++  -pthread  -std=c++11  test.cpp
$ ./a.out
main
func 0
done 1
$
```

多数のスレッドの例

```
#include <iostream>
#include <thread>
#include <vector>
using namespace std;

void func(int id, double x)
{
	cout << "func "<< id <<":"<<x<< "\n";
}

int main()
{
	const int num = thread::hardware_concurrency();
	vector<thread> a;
	for (int i = 0; i < num; i++) a.push_back(thread(func, i, i*2.0));
	cout << "main\n";
	for (thread &t : a) t.join();
	cout << "done\n";
}
```

vectorの基本

```
#include <iostream>
#include <vector>
using namespace std;

int main()
{
   vector<int> v; // intを要素とするvector（要素数０）
   v.push_back(365);  // 要素の追加（要素数１）
   v.push_back(794);  // さらに追加（要素数２）
   v.push_back(931);

   for (size_t i = 0; i < v.size(); i++) // 要素数はsize()で分かる
      cout << v[i] << "\n";              // 配列の様に要素を読み出せる
   v[0] = 123;                          // 要素を書き換えても良い
   for (int x : v)                      // range for文でアクセスできる
      cout << x << "\n";
   return 0;
}
```

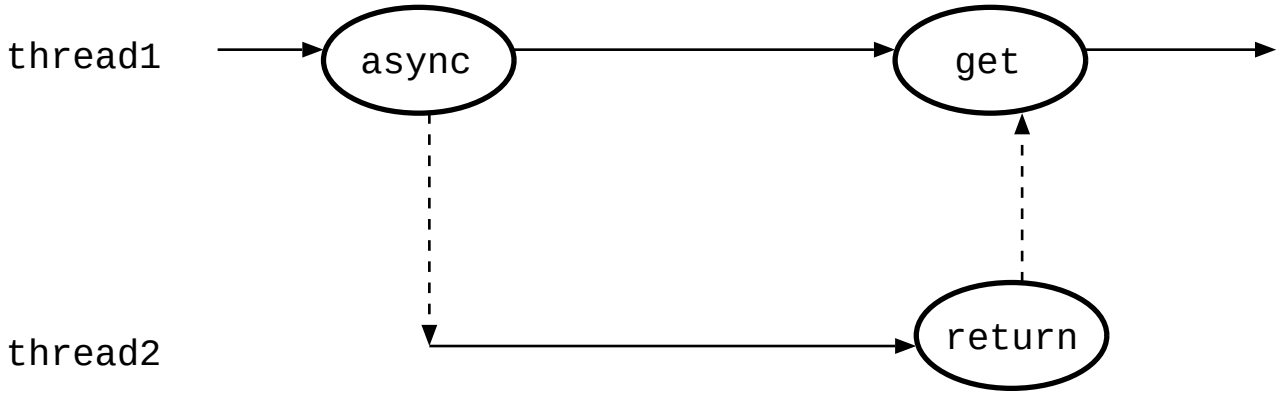
スレッドによる計算結果の利用

```
#include <iostream>
#include <thread>
#include <vector>
using namespace std;

void func(int id, double &result) { result = id * 2.5; }
int main()
{
	const int num = thread::hardware_concurrency();
	vector<thread> a;
	vector<double> res(num);
	for (int i = 0; i < num; i++)
		a.push_back(thread(func, i, ref(res[i])));
	for (thread &t : a) t.join();
	for (double &r : res) cout << r << " ";
	cout << "\n";
	return 0;
}
```

スレッドから戻り値を受け取る

std::threadはスレッドに指定された関数の戻り値を捨てる
スレッドから戻り値を受けとるにはstd::future<>クラスを利用する
　非同期操作の結果をget()で取り出す
　スレッドの終了を待つ
　取り出しは一回限り
std::future<>オブジェクトはstd::async()関数で作成できる
　futureに値を設定するスレッドを作成できる
　スレッド関数の戻り値が設定される



futureの例

```
#include <iostream>
#include <future>
#include <vector>
using namespace std;

double func(double x) { return x*2; }

int main()
{
   future<double> a = async(launch::async, func, 48);
   cout << a.get() << "\n";
}
```

futureの例

```
#include <iostream>
#include <future>
#include <vector>
using namespace std;

double func(double x) { return x*2; }

int main()
{
   const int num = 5;
   vector<future<double>> a;
   for (int i=0; i<num; i++)
      a.push_back(async(launch::async, func, i));

   double sum = 0;
   for(auto& x : a) sum += x.get();
   cout << sum << "\n";
}
```

時間を計る例

```
#include <iostream>
#include <chrono>

void do_something();

int main()
{
    auto start = std::chrono::high_resolution_clock::now();
    do_something();
    auto stop = std::chrono::high_resolution_clock::now();
    std::cout << "It took "
              << std::chrono::duration<double>(stop-start).count()
              << " seconds\n";
}
#include <unistd.h> // for sleep()
void do_something()
{
    sleep(std::chrono::seconds(5).count());
}
```

課題13

πの近似計算を行うために, xを[0,1]の範囲で
曲線4/(1+x*x)の下側の面積を数値積分で求める
プログラムがある.
これを複数の(2~8程度?)スレッドで
並列計算するように変更する.

std::threadを使う場合と
std::async()を使う場合を作成せよ.

変更前と後のプログラムを実行し,
実行時間を測定することで
プログラム実行の性能を調査する.

πの計算

```
#include <iostream>

const int intervals = 40000000;
const double width = 1.0 / intervals;
int main(int argc, char *argv[] )
{
      double sum = 0.0;
      for (int i = 0; i < intervals; i++ ) {
           const double x = (i + 0.5)*width;
           sum += 1/(1.0 + x*x);
      }
      sum *= 4.0*width;
      std::cout << "Estimation of pi is " << sum << "\n";
      return 0;
 }
```