並列分散コンピューティング
(3)スレッドによる並行処理
Cスレッドプログラミング

大瀧保広

今日の内容

- 並行プログラミング
  - プロセスとスレッド
- C言語によるスレッドプログラミング
  - 生成と終了
- 時間の測り方

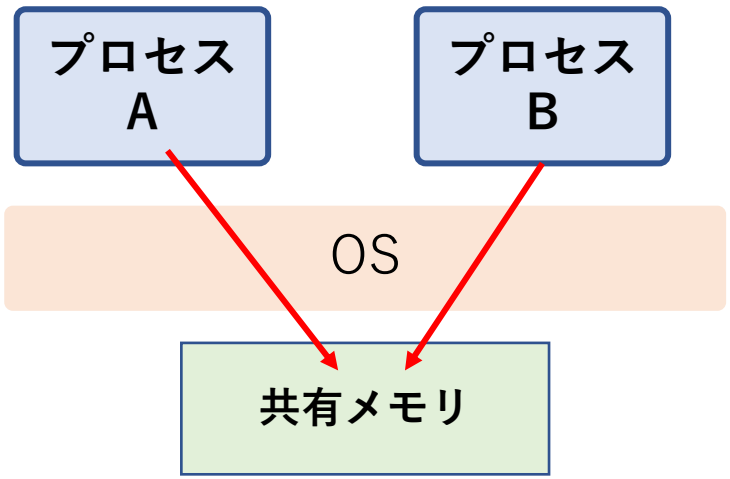
- レポート課題【予告?】

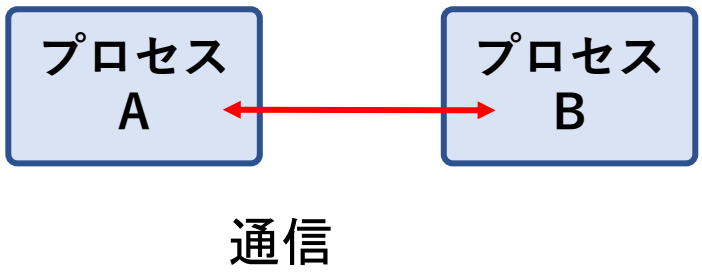
- おまけ（make, gprof）

復習：プロセスとスレッド

プロセスによる並行処理

- 個々のプロセスは、異なるコンピュータ上で実行してもよい。
- 各プロセスの間には直接 共有されるものがないので、
  OSの助け（共有メモリ機構やファイルシステムなど）を
  借りて情報を共有するか、通信によって連携を取ることになる。
- 一つのプロセスで致命的なエラーが起きても、
  そのプロセスだけが止まるようにできる。





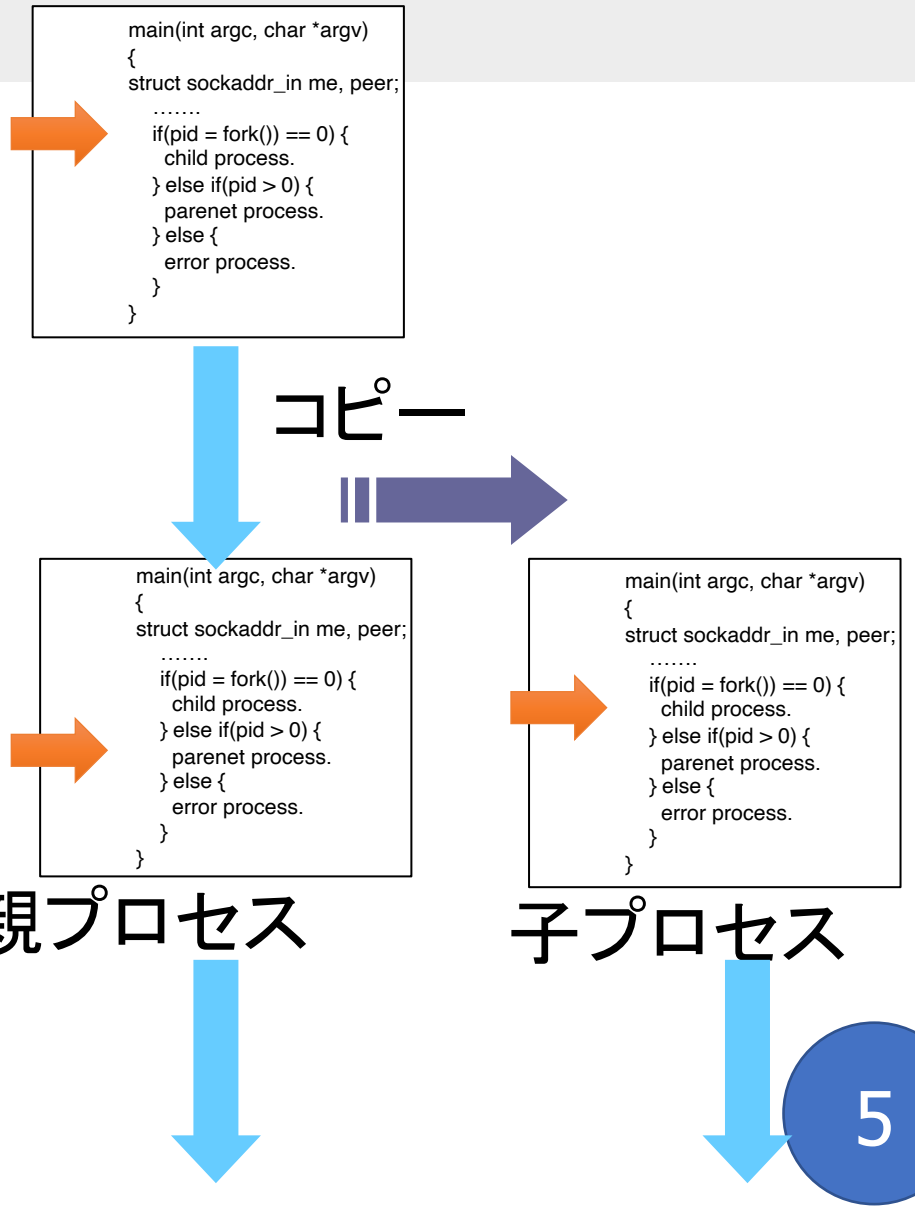
同じコンピュータ上でも
異なるコンピュータ上でもよい

プロセスの生成

プロセスの生成は、fork()システムコールによる。

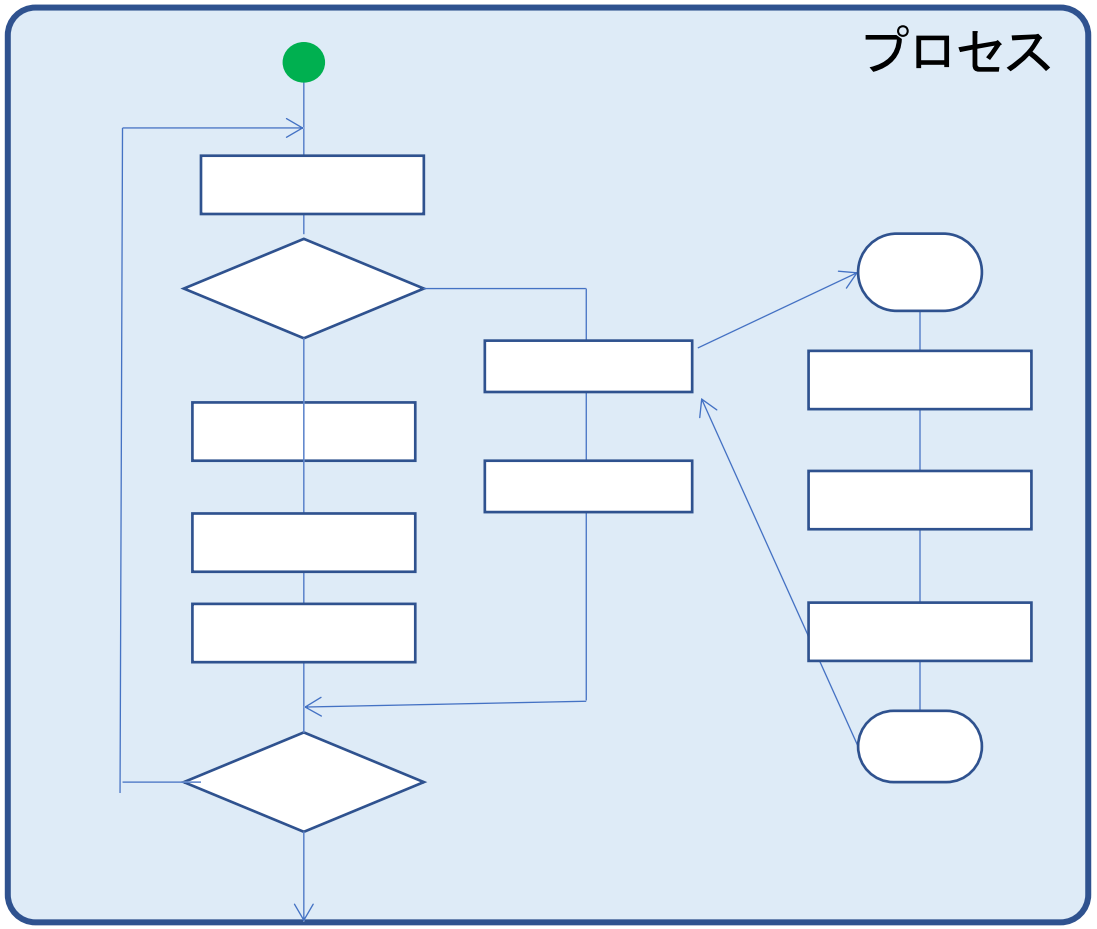
```
int pid;

if( (pid = fork()) == 0){
        /* child process */
} else if(pid > 0){
        /* parent process */
} else {
        perror("fork()");
}
```

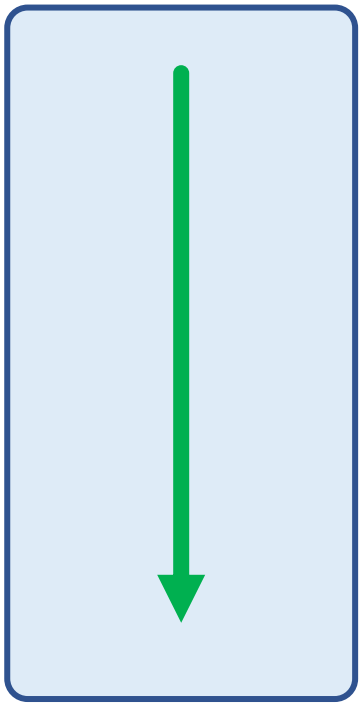
親プロセスが子プロセスを生成するとき、メモリ空間、すべてのディスクリプタ（ファイル識別子）が複製される。
→プロセス生成処理が重い、遅い



スレッド（Thread）

■スレッドとは、プロセスの中での処理の流れ。

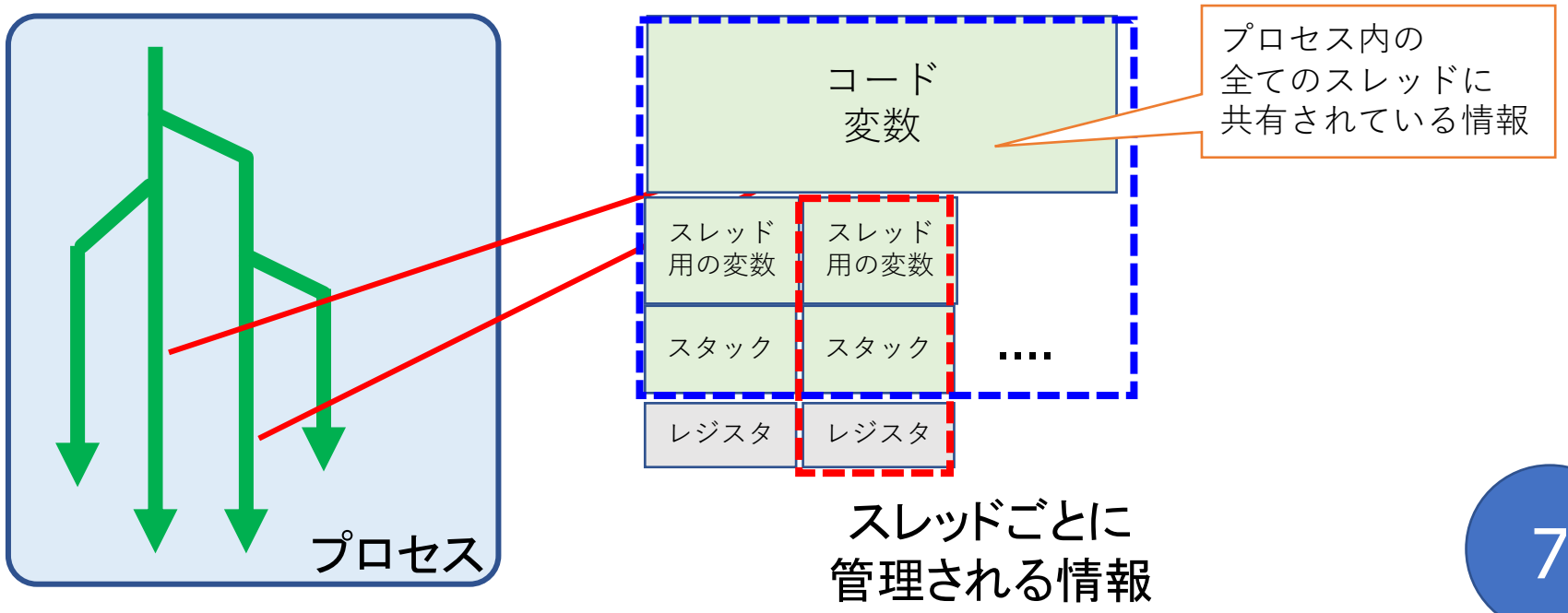




プロセス中のスレッド
（シングルスレッド）

スレッドによる並行処理

- 各スレッドが同じメモリ空間を共有しているので、そこを共有メモリとして利用することができる。
- 一つのスレッドで致命的なエラーが起きると、プロセス全体に影響がある。簡単に言えば、プロセス全体が止まる。



C言語による スレッドプログラミング

ここからは 実際に サンプルファイルを使いながら試すことを勧めます

スレッドの生成と終了

プログラムの実行開始時に生成されるメインスレッド以外のスレッドは、
pthread_createを呼び出して生成する。
スレッドで実行を開始する起点は関数単位。

pthread_create関数

```
#include <pthread.h>
pthread_create(pthread_t *tid, const pthread_attr_t *attr,
                                void *(*func)(void *), void *arg);
```

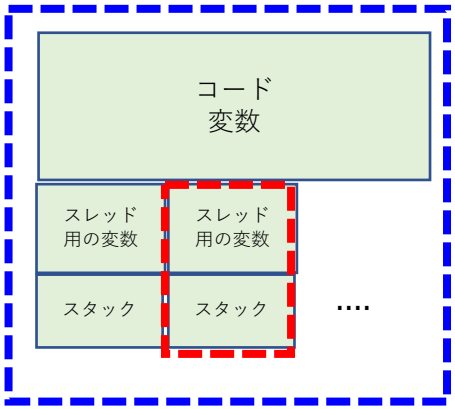
pthread_exit関数

```
pthread_exit ( void *value_ptr );
```

スレッドの後始末を行ない、join のための終了status をセットする。

スレッドの終了まち／解放

pthread_createでスレッドを作ると、スレッド用の領域が確保される。
threadの実行が終わったら必ずdetachするかjoinする。
さもないと領域が解放されないままになる恐れがある。



pthread_join関数：スレッドの終了待ち

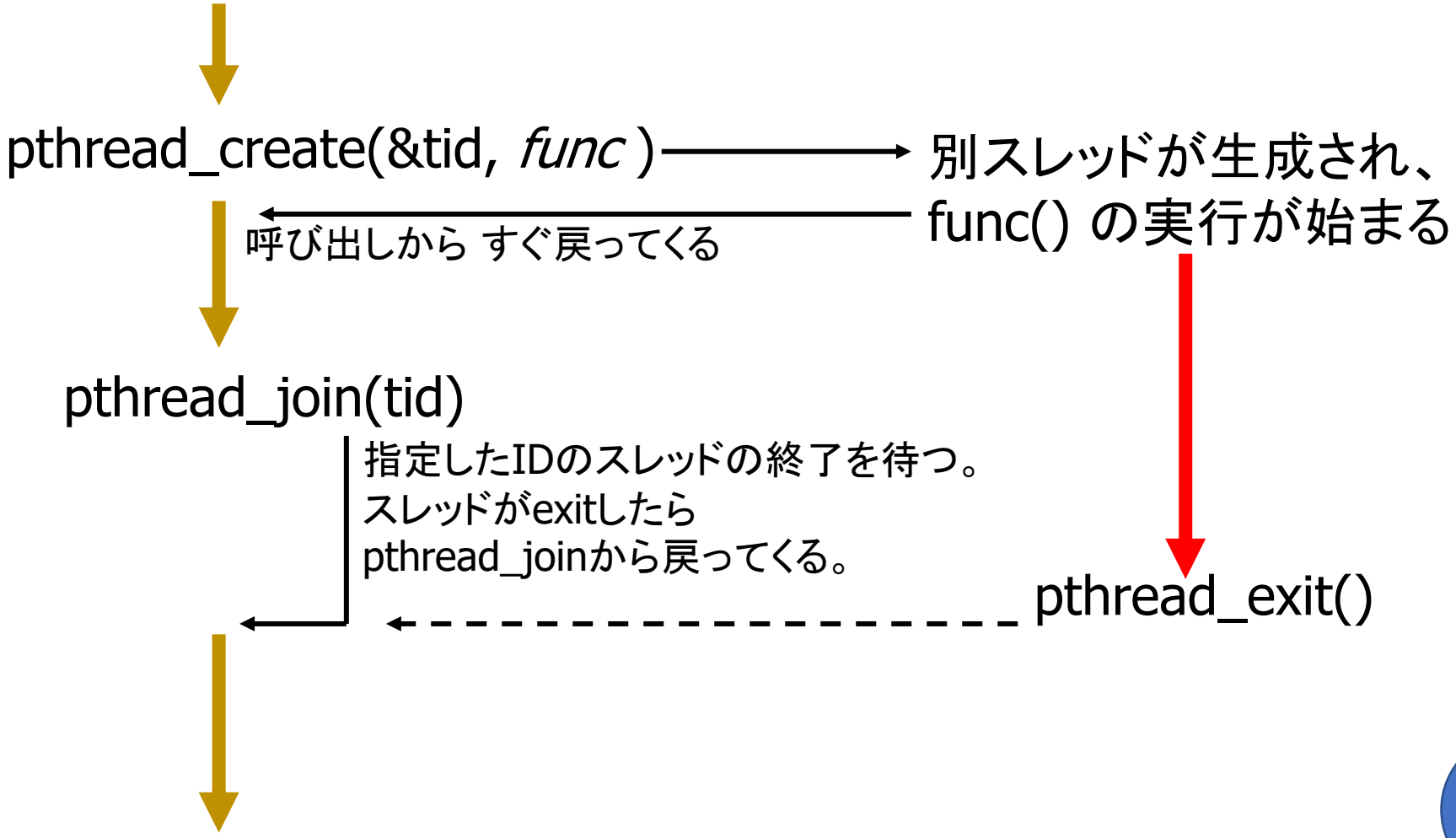
```
int pthread_join(pthread_t tid, void **status);
```

pthread_detach関数：スレッドの解放

（スレッドの終了ステータスは得られない）
スレッドの中のエラー処理で自分自身に対して使うことが多い

```
int pthread_detach(pthread_t tid);
```

create, exit, join の関係



スレッドの強制終了(cancel, kill)

スレッドを最後まで実行させずに、途中で停止することは簡単ではない。
処理を途中で打ち切ることによる不整合が生じないように、
帳尻をあわせる処理を記載しなければならない。

pthread_cancel関数：スレッドにおわって！っていう。
　すぐに終了するわけではなく、cancel ポイントで停止。

```
int pthread_cancel(pthread_t tid);
```

pthread_kill関数：スレッドにシグナルを送る。
　スレッドにシグナルハンドラを設定しておくことで
　実行に割り込んで停止処理を行うことができる。

```
int pthread_kill(pthread_t tid, int sig);
```

pthread_create関数の使用例

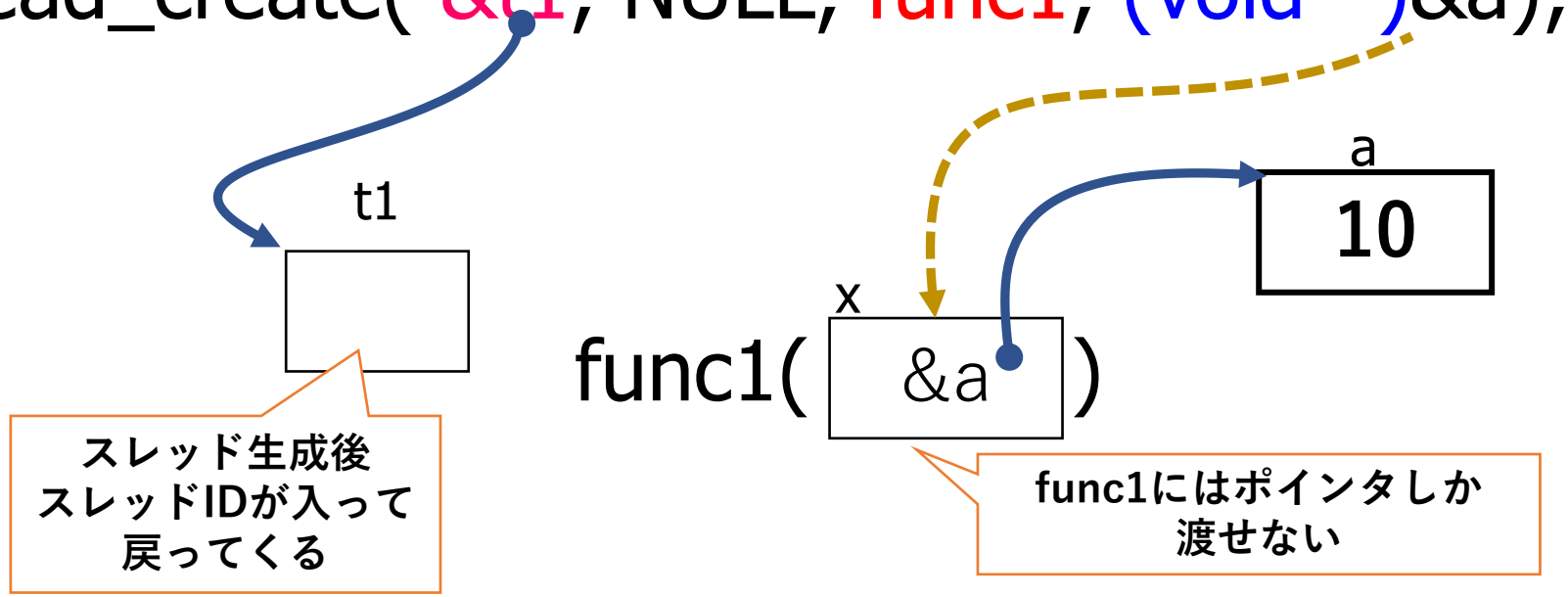
例えば、整数型のデータを渡してfunc1() を
新規スレッドで実行したい場合

```
void * func1(void *x);
```

func1 のプロトタイプ

プロトタイプに合わせる
ためのキャスト

```
pthread_create( &t1, NULL, func1, (void *)&a);
```



スレッドの記述例（1）

```
pthread_t t1;
int a=10;

/* スレッド の生成 */
pthread_create( &t1, NULL, func1, (void *)&a);
        :
        :
/* スレッドの終了を待つ */
pthread_join( t1, NULL );
```

```
/*スレッドとして実行される関数（関数のプロトタイプは固定！）*/
void *func1( void *x) {
        int b;
        b=*((int *)x);
        /* なんらかの処理 */
       pthread_exit(NULL);
}
```

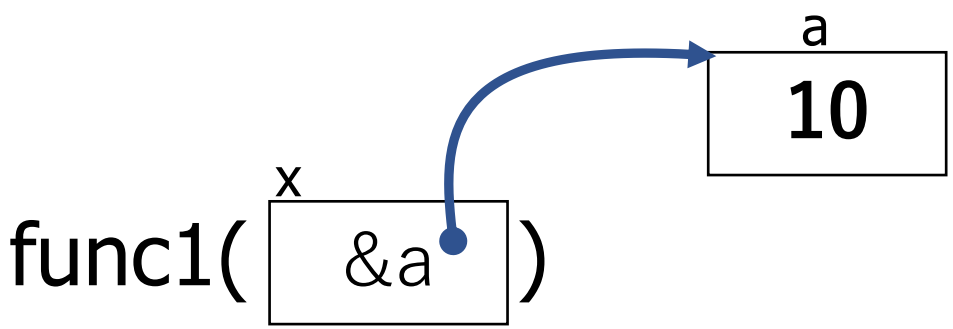
スレッドの後始末を行なう。
本来は、終了status を返す。

記述例の説明

func1() 側で、渡されたデータに正しくアクセスするには？

渡された値は &a (int型変数 aのアドレス)

func1() の仮引数は　x
xに格納されている値は &a (aへのポインタ)
xがポインタであるとき、ポインタが
指す先の値にアクセスする基本形は *x



pthread_createのプロトタイプ宣言によれば、
xの型は「void *」(void型へのポインタ)なので、このままでは
ポインタが指す先にある値の型がわからない。(=値を正しく取り出せない)

ポインタの先にあるデータを int型として取り出すには、
xを「int型へのポインタ」に型変換した上で参照する。→正しく値10が得られる。

```
b = *((int *)x);
```

スレッドの記述例（2）：引数2個以上／0個

```
pthread_t t1;

struct foo {
 int a;
 float  b;
} ;

struct foo s;
s.a=3; s.b=5.3;

pthread_create( &t1, NULL, func1, (void *)&s);
```

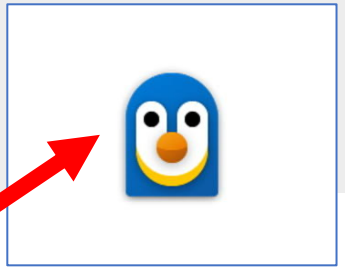
Int型とfloat型のデータの
2つを func1 に渡したいので、
pthread_createに渡すために
一つの構造体にまとめる。

```
void *func1( void *x) {
   struct foo *p;
   int r;
   float s;
  p=(struct foo *)x;
   r= p->a; s=p->b;
 /* なんらかの処理 */
  pthread_exit(NULL);
}
```

```
pthread_create( &t1, NULL, func1, NULL);
```

```
void *func1( void *x) {
  /* なんらかの処理 */
  pthread_exit(NULL);
}
```

演習用ファイルの入手



- WSLを起動する
  （Windowsのアプリケーションから こいつを起動）
- Linuxのコマンドプロンプトで以下のように入力する。

```
cd
wget http://nenya.cis.ibaraki.ac.jp/PDC.zip
unzip PDC.zip
```

ホームディレクトリにPDCというディレクトリが生成されて、
サンプルファイルが展開されます。

演習用のファイル：スレッドの生成と終了

- サンプルプログラム のディレクトリ:
  PDC/C-Thread/simple/

  cd PDC/C-Thread/simple
  ls

- プログラムの説明
  - simple1.c : 2つの新規スレッドを生成する。
    （メインスレッドと合わせて合計3つ）
    各スレッドでは値を3回ずつ出力する。
  - simple2.c : 同上。ただし引数2個以上や0個のサンプル。

コンパイルと実行

- コンパイル
  - コマンドラインなら
    `gcc  -o simple1 simple1.c -pthread`
  - Makefileを利用する場合
    - コンパイル
      make
    - 実行ファイルの削除
      make clean
- 実行
  ./simple1

「重箱の隅」
`-lpthread` ではない

演習用のファイル：スレッドの生成と終了

- simple1.c をコンパイル、実行してみよう。
  - 並行に実行されていない！？

（スレッドで実行する関数があまりにも小さすぎて、
スレッドのコンテキストスイッチが起こる前に
スレッドの実行が終了していると思われる。）

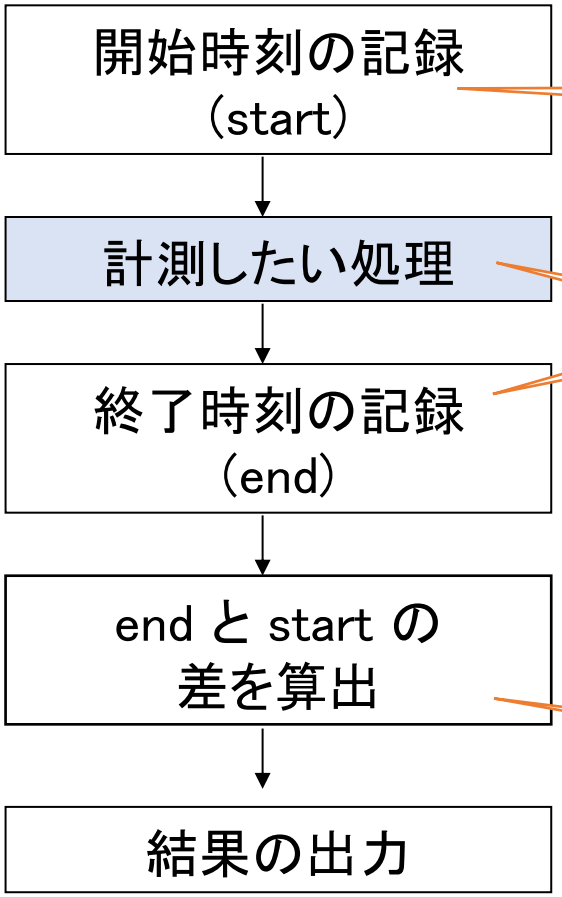
```
% ./simple1
main()
func1( 10 ): 0
func1( 10 ): 1
func1( 10 ): 2
func2( 20 ): 0
func2( 20 ): 1
func2( 20 ): 2
%
```

- 関数 func1,func2でprintfを実行するたびに
  1秒間スリープするように修正し、再度実行しよう。
  （ sleep(1); を挿入する ）

実行時間の計測

実行時間の計測

■処理の流れ



これらの「時刻」を
どの程度の精度で求められるか？

時刻の測定精度に対して
計測したい処理は十分に長いか？
(短すぎるなら、ここを10000 回繰り返すなどの工夫が必要になる)

値が近い場合、減算時の計算誤差を
小さくする工夫も必要

C言語での実行時間の計測

C言語で時刻を取得する方法はいくつかある

- 古くは gettimeofday関数

  システムの現在時刻をUTC時間(Universal Time, Coordinated)
  世界協定時の1970年1月1日0時0分0秒(the Epoch)からの
  経過時間で返す。

- clock関数
- clock_gettime関数

Linuxでのオンラインマニュアルを調べるときは
セクション3 (C Library)を指定する。

```
man -s3 clock
```

脱線：the Epoch（と2038年問題）

- UNIX系のOSでは、the Epoch と呼ばれる時点からの経過時間で時刻を管理している。
  - The Epochの時点（1970年1月1日0時0分0秒） は、UNIXで最初にシステムクロックが機能実装された時にキリがよかった過去の時刻であり、たまたま そう決めただけ。
- gettimeofday関数は時刻情報をtime_t型で返す。
  伝統的な実装では符号付き32ビット。
  →表せる最大値は $(2^{31} - 1) = 2{,}147{,}483{,}647$ 秒まで。
- The Epochから 2,147,483,647秒（≒ 68年）経過した
  2038年1月19日3時14分7秒 (UTC)
  を過ぎると、この値がオーバーフローし負の数となる。
  そのため、この時刻に依存した処理をするプログラムは誤作動する恐れがある（といわれている）。

C言語での時刻の計測

- clock関数

  clock_t clock(void);

  - **clock()** はプログラムが使用したCPU時間の近似値を返す。
  - 返り値は clock_t型のCPU時間である。
    「単位」を「秒」にするには、
    これの値を**CLOCKS_PER_SEC**で割る。

C言語での時刻の計測

- clock_gettime 関数

```
int clock_gettime(clockid_t clk_id,
                  struct timespec *tp);
```

  - clock_idに何を渡すかによって異なる「時刻」が取得できる
  - CLOCK_REALTIME：システムの現在時刻をナノ秒単位で取得する。しかし実際にその精度があるわけではない。
  - CLOCK_MONOTONIC：ある開始時点からの単調増加の時間で表現されるクロックが取得できる。開始時点がどの時点となるかは規定されていない。この時計は、システム時間の不連続な変化（例えば、システム管理者がシステム時間を手動で変更した場合など）の影響を受けない。

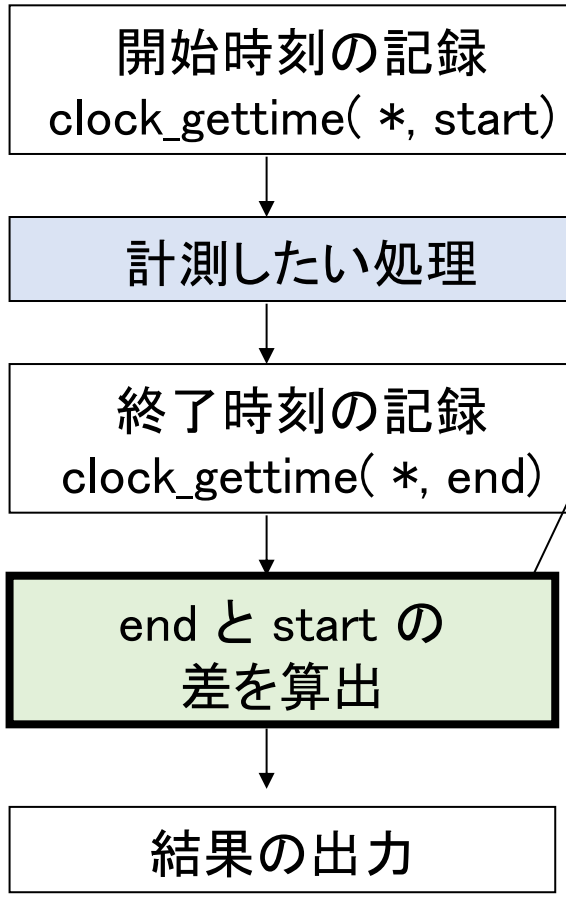
時刻を保持する構造体

- **int clock_gettime(clockid_t clk_id,**
  struct timespec *tp);

```
struct timespec {
      time_t   tv_sec;        /* seconds */
      long     tv_nsec;       /* nanoseconds */
};
```

実行時間の高精度計測

■処理の流れ



構造体同士を単純に引くことはできないので…

```
double elapsed;

elapsed = (double)(end.tv_sec - start.tv_sec);
elapsed += (double)(end.tv_usec - start.tv_usec) * 1e-9;
```

桁落ちの恐れがあるが、以下の方法でも可。

```
double start_s, end_s, elapsed;

start_s=(double)(start.tv_sec)
        +(double)start.tv_usec * 1e-9;

end_s  =(double)(end.tv_sec)
        +(double)(end.tv_usec) * 1e-9;

elapsed = end_s - start_s;
```

実行時間の高精度計測(C-Time)

■使用するファイル

PDC/Time/C-Time の下

■中身

- time0.c　：CLOCK_REALTIMEとCLOCK_MONOTONICの違いを見る
- time1.c　：10^5 マイクロ秒 = 0.1秒のスリープを10回
- time2.c　：エラトステネスのふるい1
- time3.c　：エラトステネスのふるい2

実行時間の高精度計測(C-Time)

- コンパイル
  - gccコマンドを手で打ち込むなら 以下のような感じ。

    gcc -o time0 time0.c
  - Makefile を用意しているので、makeコマンドが使用可能。
    - コンパイル

      make
    - 実行ファイルの削除

      make clean
- 実行
  - ./time0

復習項目（あとで各自でやってみること）

- time1.c のプログラムを実行して、実行時間を測ってみよう。
- time1.cのプログラムをよく見ると、
  計測範囲に計測対象ではない処理が含まれている。
  計測したい処理だけになるようにプログラムを修正し、
  実行時間を測ってみよう。差は感じられるか？

- time2.c を実行し、ふるいの実行時間を測ってみよう。

- time3.c はふるいの配列の使い方が少し異なる。
  time2.cとの実行時間の差は測れるか？

Javaでの時間の計測

■現在の時刻をミリ秒単位で得る

System.currentTimeMillis()

```
start = System.currentTimeMillis(); //     計測を開始
…
end = System.currentTimeMillis();  // 計測を終了
elapsed_time = end - start;    // 経過時間を計算
```

■現在の時刻をナノ秒単位で得る
System.nanoTime ()
（実装系によって利用できないことがある）

復習項目：実行時間の計測（JavaTime）

- Time/JavaTime の中身
  - Main.java
- **cd Time/JavaTime**
- コンパイル：
  make
- 実行：
  make run

- System.nanoTime() を利用した測定に変更してみよう。

今回紹介した計測方法の注意点(1/2)

- clock_gettime(CLOCK_REALTIME, *)や、
  clock_gettime(CLOCK_MONOTONIC, *)で計測される値は、
  実世界での時刻(Clock on the wall)に連動する値である。つまり
  厳密にいえば「そのプロセスの処理に要した時間」ではない。

  - 例えば、計測開始地点と計測終了地点の間に、
    システムが他のプロセスの実行を行なっている可能性が高く、
    その処理に要した時間も含まれている。
  - プログラムに変動要素がなく、全く同じ処理を計測している
    はずなのに測定結果がばらつくのは、これが大きな理由である。

今回紹介した計測方法の注意点(2/2)

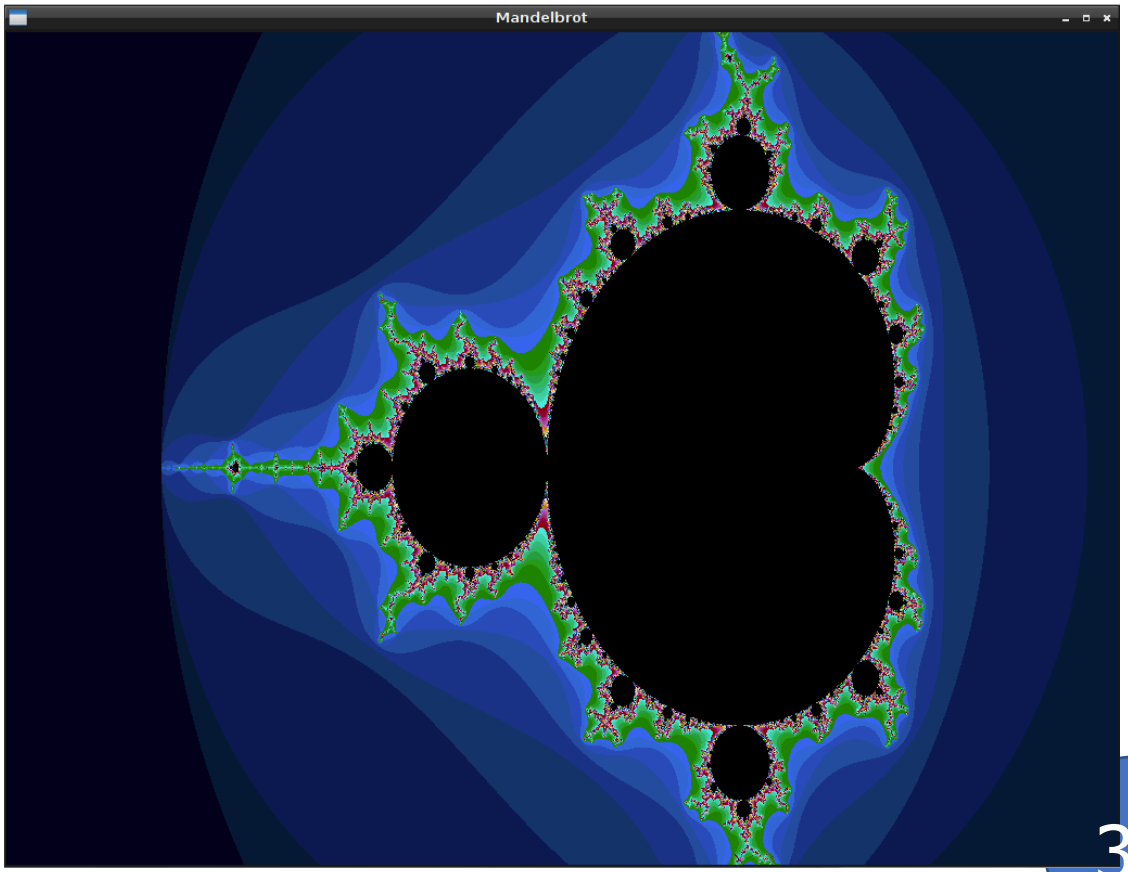
- 「処理に要した時間」の真の値に近づけるためには、外的要因をなるべく排除する工夫が必要。
- 真の「処理に要した時間」に最も近いのは、外的要因の影響が最も少ない、計測時間が最小のものである。平均を求めても意味がない。

- `clock`系の関数に渡す`clock_id`として **`CLOCK_PROCESS_CPUTIME_ID`** というのもあり、これを指定するとプロセスが消費した処理時間が求められるように見える。しかし、処理系によって実装されていないこともあるらしく、マニュアルにも詳しい説明がないので、ここでは紹介しないでおきます。

第1回レポート課題

- PDC/mandelbrot/mandelbrot.c
  - このプログラムは、マンデルブロ集合の図をOpenGLを用いて描画する。
  - わざと時間がかかるように作ってある。

- コンパイル→実行ができるか、早めに確認すること。



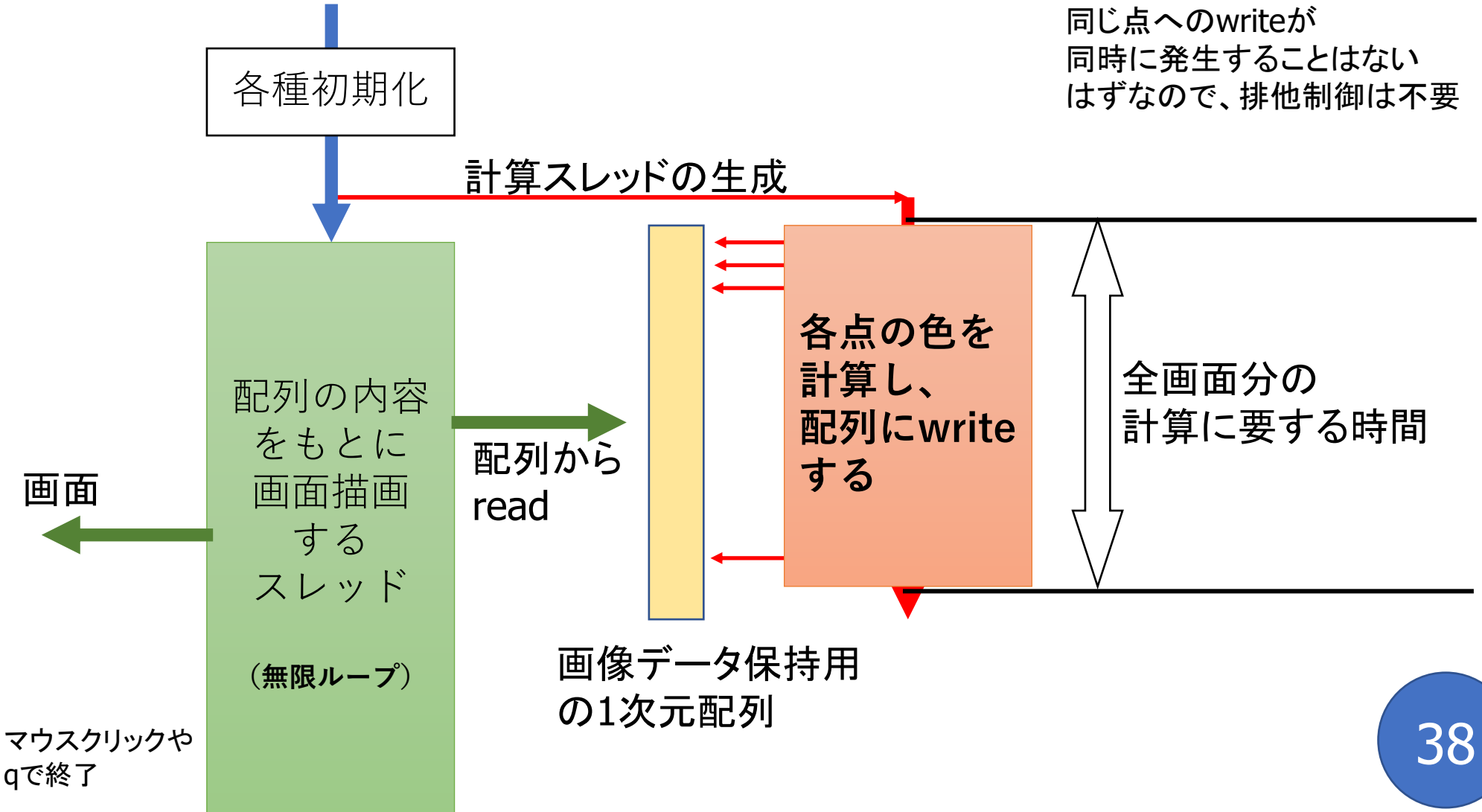
第1回レポート課題

1. 最初のプログラムで画像全体の計算に要する時間を計測せよ。
2. 画像全体の計算をする部分を
   複数スレッドを用いて並行処理し、高速化せよ。
   処理をどのように分割するか／スレッド数を幾つにするか などはお任せ。
3. 画像全体の計算に要する時間を計測し、
   元のプログラムに比べて どの程度 短縮できたか、考察せよ。

■提出期限と提出物

- ■2024年5月10日(金)までに manaba に提出
- ■提出物（2つ）
  - ■修正後のプログラムのソースファイル
  - ■レポート本体（説明用の文書）（PDFで）
    こちらのレポート中には全ソースコードを含める必要はない。

mandelbrot.cのプログラム構造の解説

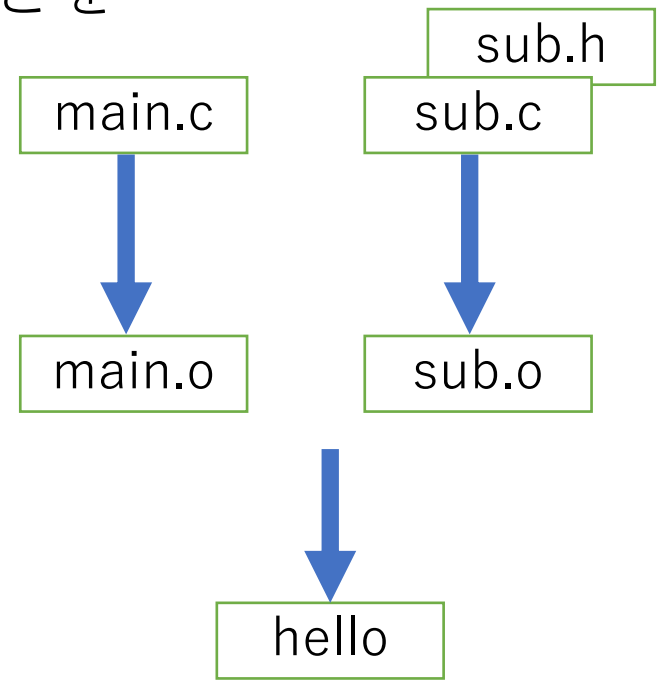


おまけ1：makeコマンドとmakefile

■分割コンパイルやオプションの指定などを
含めた古典的手順書

makefile

```
hello : main.o sub.o
	gcc -o hello main.o sub.o

main.o : main.c sub.h
	gcc -c -o main.o main.c

sub.o : sub.c sub.h
	gcc -c -o sub.o sub.c
```



ここは必ず TAB で落とす

ターゲットを表すのに $@
ソースを表すのに $<

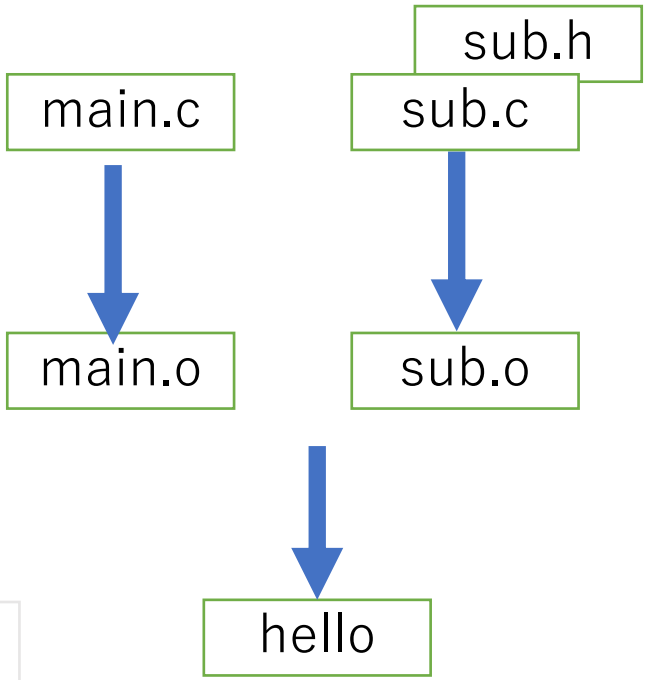
おまけ1：make

■ファイルではないターゲットもできる

makefile

```
.PHONY clean

clean :
	rm -f *.o hello
```

cleanというターゲットは
ファイルじゃない宣言



make clean
と入力すると、rmコマンドが実行される

おまけ2：プロファイラ

- プログラムの高速化や最適化を行なうときには、処理時間の大きい関数を集中的に最適化すると効果が大きい。
- そのためには まず関数ごとの実行頻度や実行時間などといった、「プログラムの詳細な振舞い」を測定する必要がある。
- プロファイラは そのためのツールである。ここでは gprof を紹介する。

gprofの使用法の概略

- gprof を使う準備
  - コンパイル時とリンク時に –pg オプションをつける。
- （ -pg を付けて作成した）プログラムを普通に実行する。
  - プログラムが正常終了すると、 カレントディレクトリに gmon.out というファイルが生成されている。
  - Ctrl-Cで止めたり、Segmentation Faultで異常終了した時には gmon.out は生成されない。
- 解析結果の出力
  - 「gprof 実行ファイル名 gmon.out」とすると、解析結果が標準出力に表示される。
  - 出力される分量が多いので、リダイレクションなどでファイルに保存すると良い。

サンプルプログラム (sample.c)

- sample.c
  - a() ：10000回 加算をするだけの関数
  - b() ： a()を4回呼ぶ関数
  - Main() ： a()とb()を10000回ずつ呼び、
    さらに　30000回　加算をする。

サンプルプログラム (sample.c)

```
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
        int i,j,g=0;
        for (i=0;i<10000;i++){
                a();
                b();
        }
        for (j=0;j<30000;j++){
                g+=j;
        }
        return EXIT_SUCCESS;
}
```

```
int a(void) {
        int i=0,g=0;
        for (i=0; i<100000; i++){
                g+=i;
        }
        return g;
}

int b(void) {
        int i=0,g=0;
        for (i=0; i<4; i++){
                g+=a();
        }
        return g;
}
```

実行と分析

- gcc -pg -o sample sample.c
  （ usvの場合：gcc -pg -no-pie -o sample sample.c）

- ./sample (実行時間は 10秒ちょっと)
- gprof sample gmon.out > sample.log

- less sample.log

gprofの出力（抜粋）

分解能は0.01秒
それほど高精度ではない

```
Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  us/call  us/call  name
 86.59     10.03    10.03    50000   200.53   200.53  a
  0.00     10.03     0.00    10000     0.00   802.13  b
```

各関数が消費する時間の割合

累計時間

関数の消費時間

関数呼び出し1回あたりの消費時間

関数の1回あたりの消費時間（サブ関数の実行時間を含む）

呼び出し関係の分析

```
index % time    self  children    called     name
                2.01    0.00   10000/50000       main [2]
                8.02    0.00   40000/50000       b [3]
[1]     100.0   10.03    0.00   50000          a [1]
-----------------------------------------------
                                               <spontaneous>
[2]     100.0    0.00   10.03                  main [2]
                0.00    8.02   10000/10000       b [3]
                2.01    0.00   10000/50000       a [1]
-----------------------------------------------
                0.00    8.02   10000/10000       main [2]
[3]      80.0   0.00    8.02   10000          b [3]
                8.02    0.00   40000/50000       a [1]
-----------------------------------------------
```

[1]とかは関数のID

a()は50000回呼ばれた。
そのうち
mainから10000回
bから40000回

b()は
main()から10000回呼ばれ、
a()を40000回呼んだ。