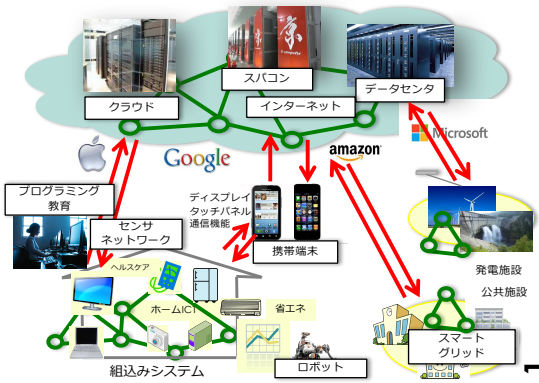
# システムプログラミング
## スレッドプログラミング

芝浦工業大学
情報工学科
菅谷みどり






## 授業計画

1. 9/27 イントロダクション, 歴史，シェルスクリプト
2. 10/4 システム管理, 開発環境プログラムのコンパイル，リンク
3. 10/11 ファイル（高水準なファイル操作）入出力ハードウエアと制御, ファイルシステム，ファイル構造
4. 10/18 プロセス
5. 10/25 プロセス2
6. 11/8 プロセス3
7. 11/15 , スレッド, デッドロック1，研究紹介1
8. 11/22, , スレッド, デッドロック2, 研究紹介2
9. 11/29 時刻, 割込み,シグナル
10. 12/1 ネットワークプログラミングI クライアントプログラム
11. 12/13 ネットワークプログラミングII サーバプログラム
12. 12/20 グループワーク
13. 1/10 中間報告
14. 1/17 グループワーク
15. 1/24 発表会，提出

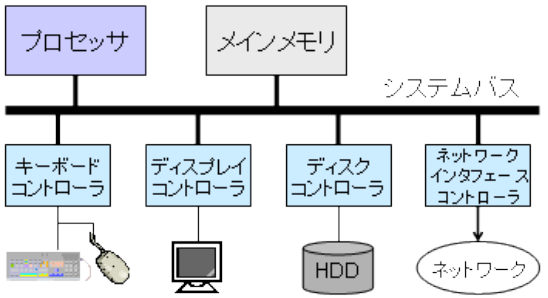


## 同期，非同期



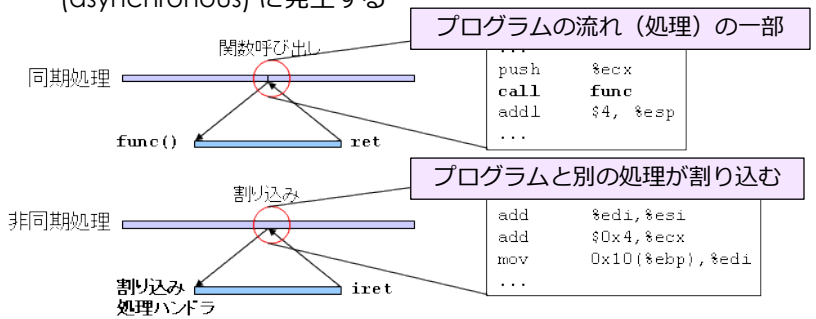
## (復習）入出力機器

- **割り込み**
  - 入出力機器からのイベント通知の仕組み
- **入出力機器**
  - PCの場合、コンピュータに I/O コントローラを通じて接続されている



# (復習）同期処理と非同期処理

- **関数呼び出しは同期処理**
  - 明示的に関数を呼び出し、呼び出しされた関数で処理が行われ、関数呼び出しから帰ってくる
- **割り込みは非同期処理**
  - 本来のプログラムの流れとは無関係に、非同期的(asynchronous) に発生する




いつ割り込まれるかわからない



# ブロッキング/ノンブロッキング

- **プロセスの終了**
  - カーネルは親にシグナルを送って通知
  - 子の終了は非同期（親の実行中いつでも起こり得る）<シグナルはいつでも送られる
  - 親は、実行中にシグナルを
    - 無視する：ブロックする
    - 受け取る：シグナルハンドラで受け取る

- **#include <sys/wait.h>**
- **1.pid_t wait(int *statloc);**
- **2.pid_t waitpid(pid_t pid, int *statloc, int options);**
  - 1: 子プロセスが終了するまで、呼び出し側をブロックする
  - 2: 正常終了は受け取る

- **ブロッキング：待つ**
- **ノンブロッキング：待たない（割り込み受ける）**



# 演習：

- **ブロッキング、ノンブロッキングの実装の性能を比較する**
  - 仕様：
    - wait(), waiitpid() 部分が異なる実装の forkプログラムを作成する
    - 例に従って比較して、考察を述べてみよう
- **例)**
  - time ./fork_wait
    - 0.016u 0.104s 0:00.28 39.2%  0+0k 0+0io 0pf+0w
  - time ./fork_waitpid 10
    - 0.000u 0.008s 0:00.12 0.0%  0+0k 0+0io 0pf+0w

- **real/user/sys**
  - キャッシュに乗ると早いので一度目と二度目以降の結果が異なるので注意



# システム設計

- **ハードが同一で、ソフト側で性能向上ができる**
  - ソフトウエア技術者にとって重要な技術
  - 特に応答性はインタラクションにとって重要な技術

- **ソフトウエアでの応答性向上**
  - アルゴリズムのチューニング
  - システム系のチューニング
    - システムコール
    - OS
    - 低レベルになればなるほど、貴重なエンジニアリング戦力

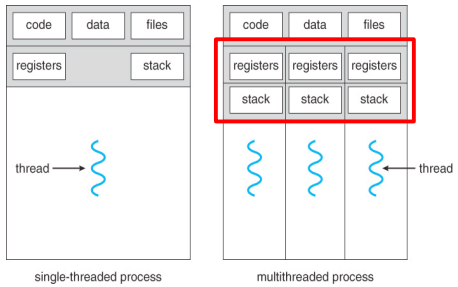
# なぜ性能が変わるのか？

- **割り込み処理**
  - ハードウエアの仕組み＋ソフトウエアの処理
- **ソフトウエアの処理において重要な点：割り込み応答性の実装**
  - プロセス（ユーザーモード）制御（APIの設計）
    - プロセスの終了ステータスの差
    - 実は安全性設計にも重要なポイント
    - ➔ ぜひ最終実装に反映してほしい
  - プリエンプティブ、ノンプリエンブティブkernel
    - ユーザーモードの割り込み応答性能 ➔ システムコールのAPI設計に依存
    - カーネルの割り込み応答性性能



# スレッド

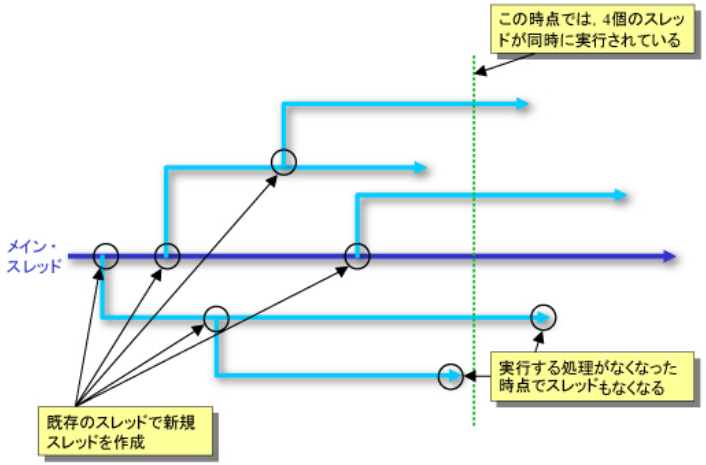
- **スレッド(thread) = 軽量プロセス(lightweight process)**
  - 1つの保護の単位としてのプロセス（タスク、あるいはアドレス空間）内部に含まれている並行処理（論理的な並列処理）の単位
- **シングルスレッドプログラム**
  - 1度に1つの手続き(Cの関数）しか動作しない．プロセスより軽量
- **マルチスレッドプログラム**
  - 1度にスレッドの数だけの手続きが論理的には同時に動作する(同期でブロックされているものも含む)






# マルチスレッドプログラム

- **複数スレッド = マルチスレッド**

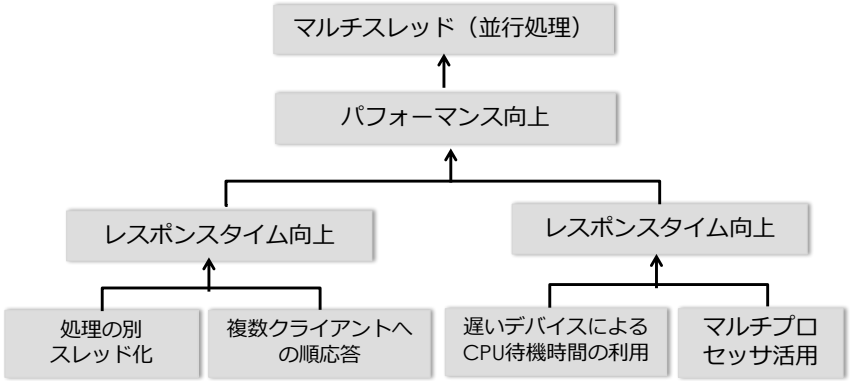



ITPro 268374より引用



# マルチスレッドの目的

- **複数タスク処理の際のパフォーマンス向上**



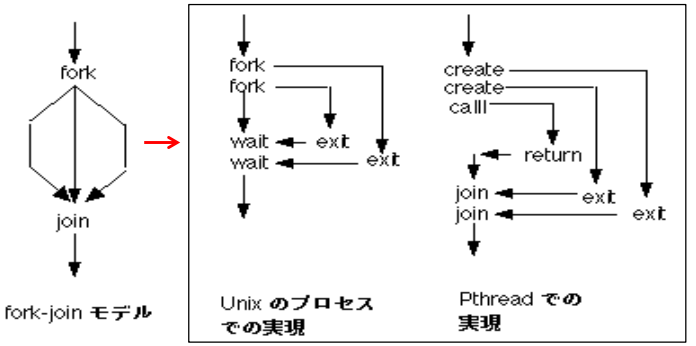
# 有用な例

- **複数のクライアント要求を受け付けるサーバの場合**
  - Web サーバなどのサーバプログラム
    - クライアントごとに処理時間が異なる，時間がかかるケースも
- **時間のかかる処理を行う場合**
  - GUI アプリケーション
    - ユーザとのインタラクション
  - ロボットアプリケーション
    - ハードウエアとのインタラクション
- **複数の処理を同時に実行したいアプリケーションの場合**
  - ゲーム
  - マルチメディアアプリケーション
  - ロボットアプリケーション
  - ➔ メインと切り離して，スレッドとして実行する



# Fork-join モデル

- **fork-join モデルの実現**




- 逐次処理（スレッド/プロセスが1つ) の状態から始まる
- 並列性が必要になった時, fork 命令で複数のスレッド/プロセスに分かれて並列処理を行う
- 並列に動作できる部分が終わると join 命令で再び逐次処理に戻る



# Fork-join モデルの実現

- **Unix の fork による実現**
  - fork() システムコールで、プロセスのコピーが生成される。Pthread とは異なり, 同期を行うためには, 子供は exit(), 親は wait () する
- **Pthread による実現**
  - Pthread では, コピーではなくcreate により新しいスレッドを作成する
  - 同じ関数を実行したい場合には, 直接 call を呼ぶ
    - 別の関数を実行するなら呼ばなくても良い
  - 子スレッドでは, pthread_exit() （トップの手続きからリターン）
    - 親は、pthread_join() する
  - 後で join する必要がない時は、pthread_detach() を使って切り離す（join しなくてもゾンビが残らない）

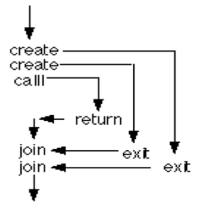


# Pthread システムコール

- **スレッドの生成**
  - int pthread_create(pthread_t * thread, pthread_attr_t * attr, void * (*start_routine)(void *), void * arg);
    - pthread_t *thread : 内部的にスレッドを識別するID
    - pthread_attr_t * attr : 作成したスレッドの属性
    - *start_routine: 生成されたスレッドが最初に処理をはじめる関数へのポインタ, void ポインタを引数に取り, void ポインタを返す
    - void * arg: 生成されたスレッドが, start_routine を実行する際に渡される引数
- **スレッドの終了**
  - void pthread_exit(void *retval);
    - スレッドを終了する

# Pthread システムコール

- **スレッドの終了待ち**
  - Int pthread_join (pthread_t th, void **thread_return);
    - th で指定されたスレッドが pthread_exit() 呼び出して終了するか、取り消しされて終了するのを待つ




Pthread での実現

- Join されるスレッド th は, 合流可能な状態でなければならない pthread_detach() を使用して、デタッチされていたり, death を与えられていたりしてはならない



# スレッドプログラミングをしてみよう

```
1  /* create two threads */
2  #include <stdio.h> /* printf() */
3  #include <pthread.h>
4  void func1(int x);
5  void func2(int x);
6  int main (void)
7  {
8  pthread_t t1;
9  pthread_t t2;
10 printf("in main()¥n");
11 pthread_create(&t1, NULL, (void *)func1, (void *)10);
12 pthread_create(&t2, NULL, (void *)func2, (void *)20);
13 pthread_join(t1, NULL);
14 pthread_join(t2, NULL);
15       }
```

```
16  void func1 (int x)
17 {
18    int i;
19 for (i = 0; i < 3; i++) {
20      printf("func1(%d): %d¥n", x, i);
21    }
22 }
23   void func2 (int x)
24 {
25 int i;
26 for (i = 0; i < 3; i++) {
27    printf("func2(%d): %d¥n", x, i);
28   }
29 }
```



# 実行例

- **コンパイルと実行**

```
$ gcc -o create create_join.c –lpthread
 ./create
in main()
func1(10): 0
func2(20): 1
func1(10): 2
func2(20): 0
func1(10): 1
func2(20): 2
```

- **この例では、次の３つのスレッドが作られる**
  - Main を実行するスレッド
  - Func2 から作られたスレッドt2



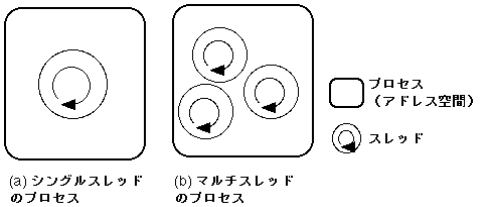
# プロセスとスレッドの関係を見よう

```
 1  /* create two threads */
 2  #include <stdio.h> /* printf() */
 3  #include <stdlib.h>
 4  #include <pthread.h>
 5  #include <unistd.h>
 6  #include <sys/types.h>
 7  void func1(int x);
 8  void func2(int x);
 9  int main (void)
10  {
11    int pid;
12    pthread_t t1;
13    pthread_t t2;
14
15    pid = getpid();
16    printf("in main() pid=%d¥n", pid);
17    pthread_create(&t1, NULL, (void *)func1, (void *)10);
18    pthread_create(&t2, NULL, (void *)func2, (void *)20);
```

```
19        pthread_join(t1, NULL);
20        pthread_join(t2, NULL);
21      }
22      void func1 (int x)
23      {
24        int i, pid;
25        pthread_t tid;
26        tid = pthread_self();
27        fprintf(stderr, "thread_func called¥n");
28        fprintf(stderr, " thread ID = %ld¥n", tid);
29        pid = getpid();
30        fprintf(stderr, " 2:pid=%d¥n", pid);
31        for (i = 0; i < 3; i++) {
32          printf("func1(%d): %d¥n", x, i);
33        }
<- func2 を作成する（コピー）
34      }
```

# 実行結果

```
$   gcc -o create_tid create_join_tid.c -lpthread
$ ./create_tid
in main() pid=2576
thread_func called
 thread ID = 1092409680
 2:pid=2576
func2(10): 0
func2(10): 1
func2(10): 2
thread_func called
 thread ID = 1083980112
 2:pid=2576
func2(20): 0
func2(20): 1
func2(20): 2
```




- **プロセスの中に，複数のスレッドが存在する**



# マルチスレッド

- **マルチスレッドプログラミング**
  - 複数スレッドを扱うプログラミング
  - Main も一つのスレッドが実行している
    - 初期スレッド、またはメインスレッド
  - Ex. Pthread_create()
    - POSIX, Pthread では、メインスレッド以外は pthread_create() にて作成される
- **スレッドの実行順序**
  - 決まっていない。スレッドは、もともと順番を決めないような処理、非同期的(asynchronous) な処理を表現するためのもの
  - どうしても、他のスレッドとの同期が必要な場合には、mutex や条件変数といった同期機能を使う



# 演習1：比較してみよう

- **スレッドを生成するプログラム**
  - include<pthread.h>
  - pthread_t td;
  - pthread_create(&td, NULL, (void *)func, (void *)val);
  - pthread_join(td, NULL);
- **引数で，作成するスレッドの個数を受け取り，スレッドを複数個作れるようにしよう**
  - ./threads [num]
- **プロセスと性能を比較しよう**
  - time ./fork 7
    - 0.004u 0.020s 0:00.04 50.0%  0+0k 0+0io 0pf+0w
  - Time ./thread 127
    - 0.000u 0.004s 0:00.00 0.0%  0+0k 0+0io 0pf+0w



# Mutex によるスレッド間の同期

- **複数のプロセス/スレッドで共有しなければならないもの**
  - プロセスの場合
    - ファイル、端末、ウィンドウ、主記憶、CPU時間
    - ➔ 主記憶やCPU時間などの資源は、横取りしても平気なので、あまり問題にならない
  - スレッドの場合
    - プロセス内のメモリを共有する
    - プログラミング言語の変数
    - 変更されない変数の値を読むだけなら、特に問題は起きない
    - それ以外の時、特にスレッドで値を読み書きする時に問題が起きる
- **問題が起きるケースとは？**
  - 共有資源をロックなしでアクセスすると何がおこるか？

# 演習2：スレッドで並列処理してみよう

- **一つのスレッドで、total 2M となる計算を行うプログラムを作成**
  - for (i = 0; i < 2000000; i++)
- **二つのスレッドをもちいて、並列処理できるようにする**
  - 1つのグローバル変数(shared_total)を二つのスレッドでカウントアップ
  - for (i = 0; i < 1000000; i++)
- **実行結果を比較**
  - 考察をする



# 相互排除（mutual exclusion)

- **相互排除 (MUTual EXclusion)**
  - ある資源をアクセスできるスレッドを、１つにする
- **クリティカルセクション**
  - 共有資源の操作など処理がきわどいところは、プログラム上では、クリティカルセクションと呼ばれる
  - クリティカルセクションでは相互排除を行う必要がある（そうしないと結果が保証されない）
- **Pthread での相互排除**
  - Mutex : Pthread で相互排除を行う仕組み
  - 相互排除を行いたい部分を lock と unlock で囲む

```
pthread_mutex_t mutex1 = PTHREAD_MUTEX_INITIALIZER;
        :
pthread_mutex_lock(&mutex1);
    < クリティカルセクション>
pthread_mutex_unlock(&mutex1);
```



# 演習3：

- **二つのスレッドをもちいて、 正しく、並列処理できるようにする**
  - 1つのグローバル変数(shared_total)を二つのスレッドでカウントアップ
  - for (i = 0; i < 1000000; i++)
- **実行結果を比較**
  - 考察をする



# 考察（ロック付き）

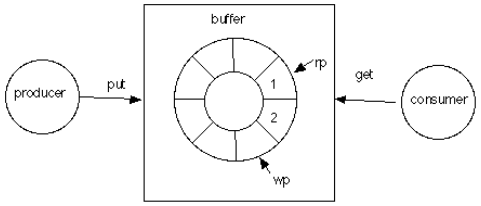
- このプログラムが、共有メモリ型マルチプロセッサで動いているとして、 動作を考える
  - 後から来た thread_B() は、他のスレッドが実行中は、待たさ れる。

```
Thread_A()
        :
23 pthread_mutex_lock(&mutex1)
24 x = shared_resource;
25 x = x + 1;
26 shared_resource = x;
27 pthread_mutex_unlock(&mutex1)
<実行終了>
        :
```

```
Thread_B()
        :
34 pthread_mutex_lock(&mutex1)
< 他のスレッドが実行中なので、この状態で他のスレッドが終わるまで待つ>
        :
        :
<実行再開>
35 x = shared_resource;
36 x = x + 1;
37 shared_resource = x;
38 pthread_mutex_unlock(&mutex1)
        :
```

# 条件変数によるスレッド間の同期

- **あるスレッドが別のスレッドの終了を待つ**
  - スレッドプログラミングを行っている時に行いたい動作
- **パイプと循環バッファ**
  - Unix のパイプのようなことを、スレッドを使って実行する
    **$ thread_A | thread_B**
  - 二つのスレッドの間はバッファを置く




- **環状バッファ、生産者スレッド、消費者スレッド**
  - バッファが空の時、thread_B() は, thread_A() が何かデータをバッファに入れるのを待つ.バッファがいっぱいの時、thread_A() は、thread_B() がバッファから何かデータを取り出すのを待つ。



# 条件変数の利用

- **条件変数(conditional variable) で、ある条件が発生するのを待つ**
- **条件変数の操作**
  - wait:
    - ある条件が満たされるまで待つ
  - signal:
    - ある条件が満たされたことを伝える、待っているスレッドが 1 つだけ起き上がる。
  - broadcast :
    - ある条件が満たされることを伝える. 待っているスレッドが全て起き上がる



# 条件変数を使ったプログラムを書いてみよう(1)

```
1    #include <stdio.h>
2    #include <stdlib.h>
3    #include <pthread.h>
4    void thread_A(), thread_B();
5    #define BUFFER_SIZE   4

6    struct circular_buffer
7    {
8      int rp;   /* read pointer */
9      int wp;   /* write pointer */
10     int used;  /* the number of elements in buffer */
11     int data[BUFFER_SIZE];   /* data space */
12     pthread_mutex_t mutex;   /* mutex */
13     pthread_cond_t not_full;  /* not full flag */
14     pthread_cond_t not_empty; /* not empty flag */
15   };
```



# 条件変数を使ったプログラムを書いてみよう(2)

```
16   void put (struct circular_buffer *cb, int x)
17   {
18     pthread_mutex_lock(&cb->mutex);
19    loop: if (cb->used == BUFFER_SIZE) {
20       pthread_cond_wait (&cb->not_full, &cb->mutex);
21       goto loop;
22     }
23     cb->data[cb->wp++] = x;
24     if (cb-> wp >= BUFFER_SIZE)
25       cb->wp = 0;

26     cb->used++;
27     pthread_cond_signal(&cb->not_empty);
28     pthread_mutex_unlock(&cb->mutex);
29   }
```

# 条件変数を使ったプログラムを書いてみよう(3)

```
30  int get (struct circular_buffer *cb) {
31    int x;
32    pthread_mutex_lock(&cb->mutex);
33   loop: if (cb->used == 0) {
34      pthread_cond_wait (&cb->not_empty, &cb->mutex);
35      goto loop;
36    }

37    x = cb->data[cb->rp++];
38    if (cb->rp >= BUFFER_SIZE)
39        cb->rp = 0;
40    cb->used--;
41    pthread_cond_signal(&cb->not_full);
42    pthread_mutex_unlock(&cb->mutex);
43    return(x);
44  }
```



# 条件変数を使ったプログラムを書いてみよう(4)

```
45  int main (void)
46  {
47    pthread_t t1;
48    pthread_t t2;
49    struct circular_buffer *cb;
50    cb = (struct circular_buffer *) malloc(sizeof(struct circular_buffer));
51    if (cb == NULL) {
52      perror("no empty for struct buffer¥n");
53      exit(-1);
54    }
55    cb->rp = 0;
56    cb->wp = 0;
57    cb->used = 0;
58
59    pthread_mutex_init(&cb->mutex, NULL);
60    pthread_cond_init(&cb->not_full, NULL);
61    pthread_cond_init(&cb->not_empty, NULL);
62    pthread_setconcurrency(2);
63    pthread_create(&t1, NULL, (void *) thread_A, (void *) cb);
64    pthread_create(&t2, NULL, (void *) thread_B, (void *) cb);
65
66    pthread_join(t1, NULL);
67    pthread_join(t2, NULL);
68  }
```

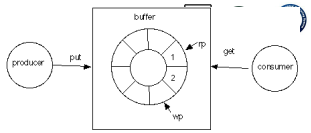


# 条件変数を使ったプログラムを書いてみよう(5)

```
69  void thread_A (struct circular_buffer *cb)
70  {
71    int i, x;
72    for (i = 0; i < 10; i++) {
73      x = i;
74      printf("thread_A(): put(%d)¥n", x);
75      put(cb, x);
76    }
77  }
78
79  void thread_B (struct circular_buffer *cb)
80  {
81    int i, x;
82    for (i = 0; i < 10; i++) {
83      x = get(cb);
84      printf("thread_B(): get() %d¥n", x);
85    }
86  }
87
```



# 解説




- **put () : 生産者**
  - バッファにデータを追加する時に使う手続き
    - Pthread_mutex_lock() し、出口で pthread_mutex_unlock() する
  - バッファが一杯の時には、条件変数 cb->not_full で一杯でないという条件が満たされるまで待つ
  - 待っている間, mutex のロックは解除される
  - Pthread_cond_wait() よりリターンしてくる時
    - もう一度、ロックされた状態に戻るが、待っている間、他の変数(rp, wp, data) が書き換えられている可能性があるので、もう一度最初から調べる
- **get() : 消費者**
  - バッファからデータを取り出す手続き。Put() とほぼ対称形
  - バッファが空の時に wait し、バッファがもはや一杯ではないことを signal する
  - thread_A() は、10回バッファにデータを書き込むスレッド、thread_B() は、10回バッファからデータを読み出すスレッド

# 実行結果

**$ gcc –o condv-buffer condv-buffer.c -lpthread**
**$ ./condv-buffer**

```
thread_A(): put(0)
thread_A(): put(1)
thread_B(): get() 0
thread_B(): get() 1
thread_A(): put(2)
thread_A(): put(3)
thread_A(): put(4)
thread_A(): put(5)
thread_A(): put(6)
thread_B(): get() 2
thread_B(): get() 3
thread_B(): get() 4
thread_B(): get() 5
thread_B(): get() 6
thread_A(): put(7)
thread_A(): put(8)
thread_B(): get() 7
thread_B(): get() 8
thread_A(): put(9)
thread_B(): get() 9
```



# 演習4

- **1. コード解説**
  - put/get でどのような処理をしているか、
  - スレッドの効果
  - について解説せよ
- **2. 実際にプログラムを作成し、比較し、考察した結果を述べる**
  - プログラムの実行結果を表示する
  - バッファを変化させた場合の結果を示す
  - 変化により生じた差を説明する
- **3. 発展課題**
  - それぞれ複数のスレッドがいた場合、どのように処理する必要があるか？
    - ヒント：スレッドキュー
- **上記を Kadai3.txt にまとめる**



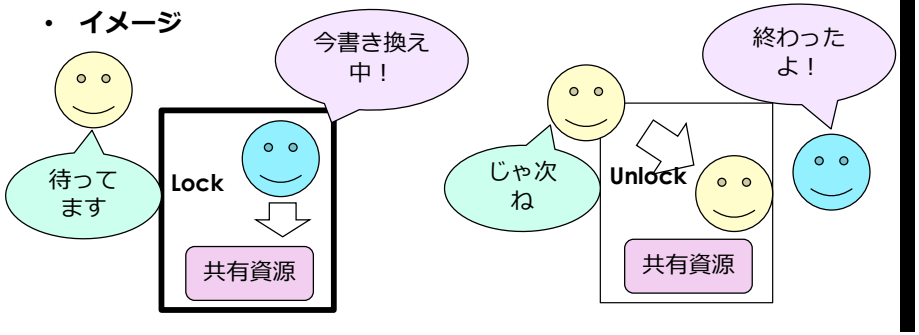
# 課題の提出

- **授業中に指示します**



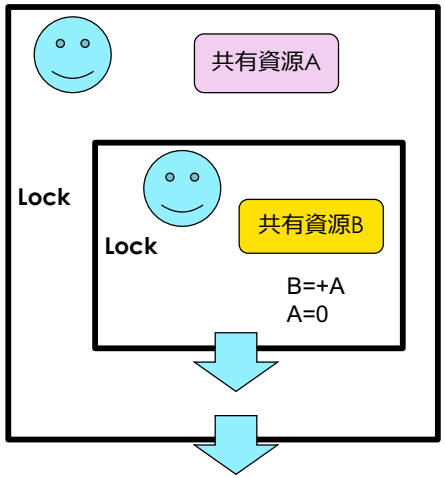
# 相互排除（ロック）

- **ロックが必要となる場合**
  - ある共有資源に二つのスレッドがアクセスする場合
    - 同時に２つのスレッドがアクセスし、値を書き換えてしまうと値がおかしくなる（➔ 前回の授業）
- **ロックによる相互排除**
  - ある共有資源にアクセスできるスレッドを一つにする
- **イメージ**



# 共有資源が二つの場合

- あるスレッドAが共有資源A, 共有資源Bを変更するクリティカルセクションに入る



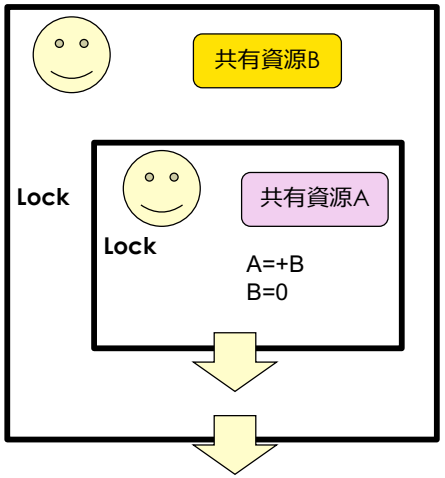

スレッド A の手順
1. CSA に入る
2. CSB に入る
3. B に A の値を加算
4. A に 0 を代入
5. CSB から出る
6. CSA から出る

クリティカルセクションA (CSA)
クリティカルセクションB (CSB)



# 共有資源が二つの場合

- あるスレッドBが共有資源A, 共有資源Bを変更するクリティカルセクションに入る

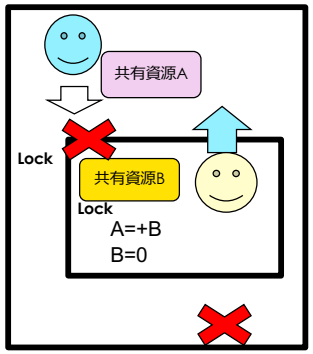



スレッド B の手順
1. CSB に入る
2. CSA に入る
3. A に B の値を加算
4. B に 0 を代入
5. CSA から出る
6. CSB から出る

クリティカルセクションA (CSA)
クリティカルセクションB (CSB)



# デッドロックが発生する場合




| スレッドA | スレッドB | CSA Lock | CSB Lock |
|---|---|---|---|
| スレッド発生 | | | |
| 処理A開始 | | | |
| **CSAに入る** | | スレッドA | |
| プロセススイッチ A->B | | | |
| 待機 | スレッド発生 | | |
| | 処理 B 開始 | | |
| | **CSB に入る** | | スレッドB |
| プロセススイッチ B->A | | | |
| CSB に入れない | 待機 | | |
| プロセススイッチ A->B | | | |
| CSB の解放を待つ | CSA に入ることに失敗 | | |
| | CSAの解放を待つ | | |
| デッドロックの発生 | | | |



# デッドロックが発生するプログラム

```
 1 /*
 2  * mutex deadlock
 3  */
 4 #include <stdio.h>
 5 #include <pthread.h>
 6 #include <unistd.h>
 7 void thread_A(), thread_B();
 8 int     shared_resourceA ;
 9 int     shared_resourceB ;
10 pthread_mutex_t mutexA ;
11 pthread_mutex_t mutexB ;
12
13 main() {
14     pthread_t t1 ;
15     pthread_t t2 ;
16       shared_resourceA = 1 ;
17       shared_resourceB = 100 ;
18       pthread_mutex_init( &mutexA, NULL );
19       pthread_mutex_init( &mutexB, NULL );
20
21       pthread_create( &t1, NULL, (void *)thread_A, 0 );
22       pthread_create( &t2, NULL, (void *)thread_B, 0 );
23       pthread_join( t1, NULL );
24       pthread_join( t2, NULL );
25       printf("main(): shared_resourceA == %d¥n", shared_resourceA );
26       printf("main(): shared_resourceB == %d¥n", shared_resourceB );
27 }
```

# デッドロックが発生するプログラム

```
29  void thread_A()
30  {
31
32      //printf("thread_A(): Try to enter CSA ... ¥n");
33      pthread_mutex_lock(&mutexA);
34      printf("thread_A(): Can enter CSA!  ¥n");

35      printf("thread_A(): Try to enter CSB ... ¥n");
36      pthread_mutex_lock(&mutexB);
37      printf("thread_A(): Can enter CSB! ¥n");
38      shared_resourceB += shared_resourceA;
39      shared_resourceA = 0;
40      pthread_mutex_unlock(&mutexB);
41      printf("thread_A(): CSB done. ¥n");

42      pthread_mutex_unlock(&mutexA);
43      printf("thread_A(): CSA done. ¥n");
44  }
```



# デッドロックが発生するプログラム

```
45  void thread_B()
46  {
47
48      //printf("thread_B(): Try to enter CSB ... ¥n");
49      pthread_mutex_lock(&mutexB);
50      printf("thread_B(): Can enter CSB! ¥n");
51
52      printf("thread_B(): Try to enter CSA ... ¥n");
53      pthread_mutex_lock(&mutexA);
54      printf("thread_B(): Can enter CSA! ¥n");
55      shared_resourceA += shared_resourceB;
56      shared_resourceB = 0;
57      pthread_mutex_unlock(&mutexA);
58      printf("thread_B(): CSA done. ¥n");

59      pthread_mutex_unlock(&mutexB);
60      printf("thread_B(): CSB done. ¥n");
61  }
```

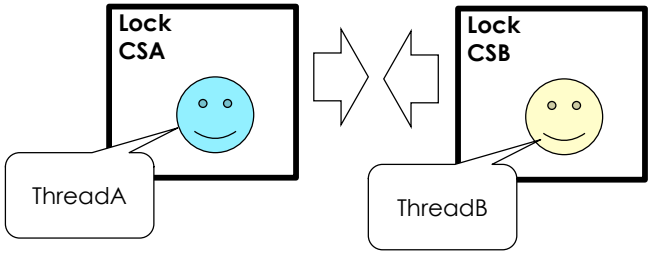


# 実行結果

gcc -o mutex-deadlock mutex-deadlock.c –lpthread

**./mutex-deadlock**

**thread_A(): Can enter CSA!** ← CSAに入ることができた!
thread_A(): Try to enter CSB ... ← CSB に入ろうとして、スイッチ
**thread_B(): Can enter CSB!** ← CBSに入ることができた！
thread_B(): Try to enter CSA ... ← CBAに入ろうとして、待機



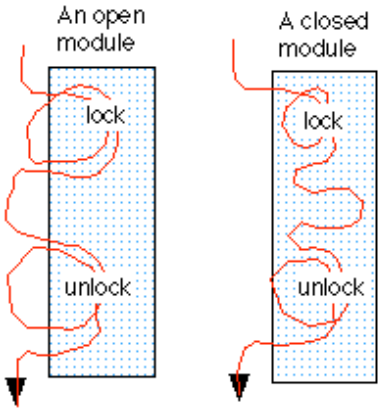



# 回避方法

- **様々な回避方法がある**
  - アトミック処理
    - “共有資源 A と共有資源 B にアクセスする” という mutex を用いる
  - クリティカルセクションに入れない場合は、既に入っているクリティカルセクションから出て処理を終了するようにする
  - Wait-free なデータ構造
    - Mutex を使って書いてあるアルゴリズムを、スタック、キュー、セット、マップにより操作できるようにする
      - 全ての要求を管理する別のスレッドを作り、そこに Wait-free のキューを作成して、キューから順次取り出して、値を更新する
  - その他
    - Compare-and-Swap など、様々な回避方法が

# 再入可能

- **リカーシブ mutex**
  - 一つのスレッドで、一つの mutex を複数回ロックしたい
    - 相互に呼び出しても良い






# 再入可能に書き換える

```
… 関数の追加
static int
my_pthread_mutex_init_recursive(pthread_mutex_t *mutex)
{
  pthread_mutexattr_t attr;
  int err;
  if (( err = pthread_mutexattr_init(&attr)) < 0)
    return (0);
  if (( err = pthread_mutexattr_settype(&attr, PTHREAD_MUTEX_RECURSIVE)) < 0)
    return (0);
  err = pthread_mutex_init(mutex, &attr);
  return (err);
}
… main プログラムの書き換え
        shared_resourceB = 100 ;
        my_pthread_mutex_init_recursive( &mutexA );
        my_pthread_mutex_init_recursive( &mutexB );


        pthread_create( &t1, NULL, (void *)thread_A, 0 );
…
```



# 実行してみよう

**$ gcc -o mutex-deadlock-recursive mutex-deadlock-recursive.c –lpthread -DPTHREAD_MUTEX_RECURSIVE=PTHREAD_MUTEX_RECURSIVE_NP**

```
[doly@yli007 12_Samples]$ ./mutex-deadlock-recursive
thread_A(): Try to enter CSA ...
thread_A(): Can enter CSA!
thread_A(): Try to enter CSB ...
thread_B(): Try to enter CSB ...
thread_A(): Can enter CSB!
thread_A(): CSB done.
thread_A(): CSA done.
thread_B(): Can enter CSB!
thread_B(): Try to enter CSA ...
thread_B(): Can enter CSA!
thread_B(): CSA done.
thread_B(): CSB done.
main(): shared_resourceA == 101
main(): shared_resourceB == 0
```

- ※タイミングによってはうまくいかないこともある



# Pthread とメモリ

- **ローカル（auto) 変数**
  - 各スレッドは、独立したスタックが割り当てられる。C言語のローカル（auto）変数は、スレッドごとにコピーが作られる
    - スレッド間でポインタを渡す場合には、スレッドの寿命にも注意する
    - ※ 関数内部で宣言され、宣言された関数の中でのみ使用可能
    - 関数実行中のみメモリ上のスタックに確保され、関数の実行が終了すると、メモリから削除される
- **静的（static) 変数**
  - static 変数は、プログラムのモジュール性を高めるために有効に使われてきた
    - マルチスレッドとは相性が悪い。Static 変数も extern 変数と同様に、複数のスレッドで共有される。変更する場合には、mutex でロックが必要になる

# スレッドセーフ

- **スレッドセーフ(thread-safe)**
  - 複数のスレッドで呼び出しても、きちんと動作すること
    - MT-Safe (multi-thread-safe)
    - Reentrant (再入可能)
  - Extern や static を使わず、auto 変数や malloc() のみを使っているような手続きは、スレッドセーフ

- **スレッドセーフではない手続きの場合**
  - 一つのスレッドだけからしか呼び出さないようにする
  - ロックを使う

  - ※一見無関係な手続きが内部で変数を共有している場合もある