実践情報セキュリティと
アルゴリズム 5
並行プログラミングとRust

大阪大学　大学院工学研究科　電気電子情報通信工学
王　イントウ
wang@comm.eng.osaka-u.ac.jp

並行プログラミング

並行と並列

- プロセス：時間的な広がりを持つ存在者
- 並行：ある時刻で、複数のプロセスが同時に実行しているように見える
- 並列：同じ時刻で、複数のプロセスが時間的に同時に実行される

並行・並列プログラミングの重要性

- 並行の重要性
  - プロセスが並行に動作しなければ、非常に不便
  - Webを見ている最中に、プッシュ通知を受け取ったりできるのも並行プログラミングのおかげ
  - 並行でないと、あるタスクが完了するまで、別のタスクを始めることが出来ない
- 並列の重要性
  - 半導体微細化の限界に近づいてきて、CPUの動作周波数を上げにくくなっている
  - 現在はCPUの動作周波数を上げるのではなく、CPUコアを複数並べる設計になってきている
  - 並列プログラミングを行わないとパフォーマンス向上が難しい

CPUの昔と今

昔

動作周波数:	10MHz	40MHz	1.4GHz
Intel CPU:	8086	i386	Pentium III
	1978	1985	1999

今

コア数:	4	6	8	8
動作周波数:	3.33GHz	3.5GHz	3GHz	3.8GHz
Intel Core i7:	975	3970x	5960x	9800x
	2009	2011	2014	2018

並行・並列プログラミングの問題点

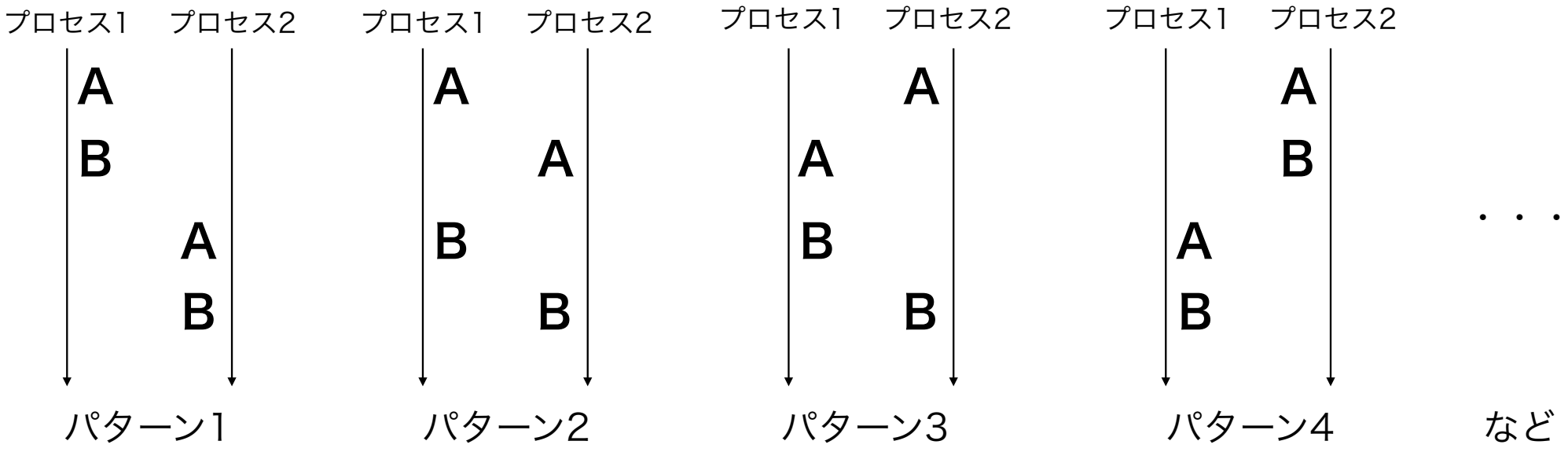
- とにかく難しい
  - デッドロック、スタベーション
  - ロック忘れ、ロック開放忘れ
- 実行タイミングを完全に把握することが困難

プロセスの実行パターン

2つのプロセスが次の2つの処理を実行するときのパターン

Aを実行

Bを実行



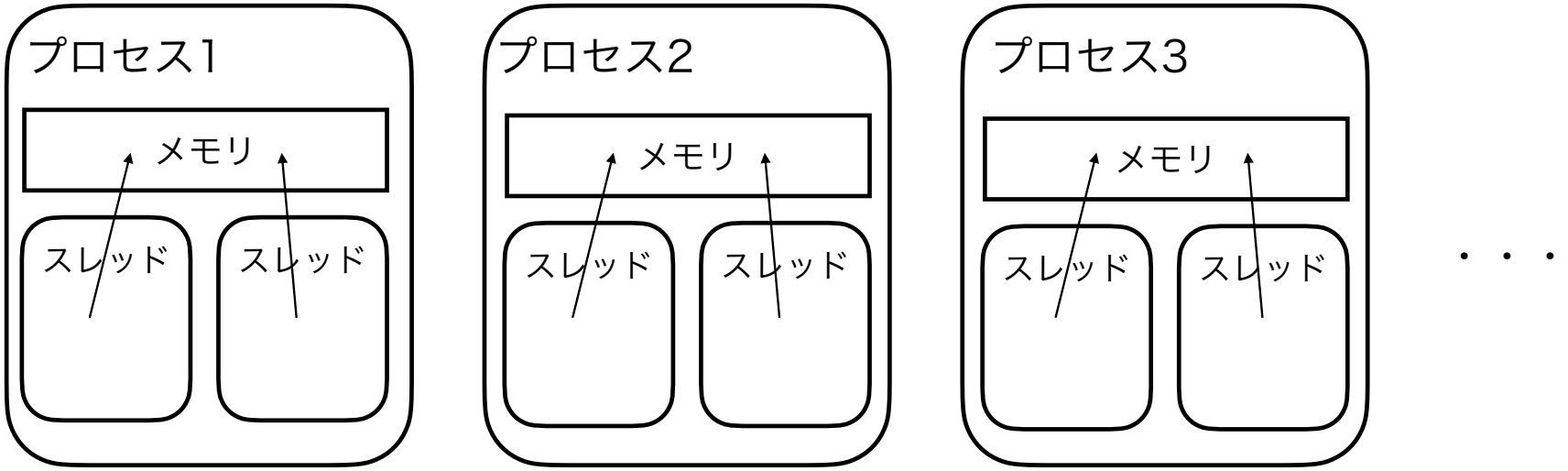
計算パターン数の爆発

- プロセスの数が増えて、処理内容も増えたときに、全パターンを把握するのは（人間には）難しい
- 膨大なパターンのうち、どれかにバグが有る場合に、再現性の低く修正の難しいバグとなってしまう

OSプロセスとスレッド

- プロセス：この講義では、プロセスと言ったとき計算実行主体という、より広い意味であると定義する。下記のOSプロセスも、スレッドもプロセス
- OSプロセス：OSからみたプログラムの実行単位。基本的に、OSプロセス毎のメモリ空間は分離されている
- スレッド：OSプロセス内で動作するプロセス。基本的に、プロセスのメモリ空間をスレッドごとに共有

OSプロセスとスレッドのメモリ空間



同期処理

アトミック処理（1/2）

- それ以上分割不可能な処理
- ある処理がアトミックであるならば
  - その処理の途中状態は他のプロセスから観測できない
  - かつ、もし処理が失敗した場合は完全に処理前の状態に戻る

アトミック処理（2/2）

- 一般的には、ハードウェア（CPU）的に提供される、特殊な処理をアトミック処理と呼ぶ
- 加算（add）や乗算（mul）といったCPUの命令もアトミック命令
- 特に、複数回のメモリアクセスが必要な命令をアトミック処理と呼ぶ

Test-and-Set (TAS)

- テストして値をセットする命令
- 意味的には左のようなソースコードになる
- アセンブリレベルでも（右のソースコード）複数命令となっている

```
int testAndSet(int *p) {
    if (*p) { // *pが真（非ゼロ）か？
        return 1;
    } else {
        *p = 1;
        return 0;
    }
}
```

```
    movl  $1, %eax
    cmpl  $0, (%rdi)
    je LBB0_1
    retq
LBB0_1:
    movl  $1, (%rdi)
    xorl  %eax, %eax
    retq
```

__sync_lock_test_and_set（1/3）

- GCCやClangでは、__sync_lock_test_and_setという、アトミックにTASを行うための組み込み関数が用意されている
- ただ、この関数は正確にはTASではなく、値を交換する関数となっている
- 以下が、組み込みTASとそれに対応するリリース関数の意味

```
type __sync_lock_test_and_set(type *p, type val) {
    type tmp = *p;
    *p = val;
    return tmp;
}

void __sync_lock_release(type *p) {
    *p = 0;
}
```

__sync_lock_test_and_set（2/3）

- __sync_lock_test_and_setの第2引数に1を渡すと、TASと同じ意味に

	呼び出し時の*p	実行後の*p	返り値
TAS	0	1	0
	1	1	1
組み込みTAS（第2引数=1）	0	1	0
	1	1	1

```
int testAndSet(int *p) {
    if (*p) { // *pが真（非ゼロ）か？
        return 1;
    } else {
        *p = 1;
        return 0;
    }
}

type __sync_lock_test_and_set(type *p, type val) {
    type tmp = *p;
    *p = val;
    return tmp;
}
```

__sync_lock_test_and_set（3/3）

- x64アーキテクチャの場合、xchg命令にコンパイルされる
- xchg命令は当該メモリのキャッシュラインを排他的(Exclusive)に設定して実行されることが保証されている

```
int testAndSet(int *p) {
    return __sync_lock_test_and_set(p, 1);
}
```

```
movl    $1, %eax.      # eax（つまりrax）レジスタに1を代入
xchgq   %rax, (%rdi)   # raxレジスタの値と(%rdi)の値を交換
retq                   # raxレジスタの値をリターン
```

__sync_lock_release

- __sync_lock_test_and_setで獲得したロックを開放するための関数
- 実態は、単純に0を代入しているのみとなる

その他のアトミック処理

- __sync_fetch_and_add(type *p, type val) : *pへ値valをアトミックに加算
- __sync_fetch_and_sub(type *p, type val) : *pへ値valをアトミックに減算
- __sync_bool_compare_and_swap(type *p, type val, type newval)
  以下のような意味

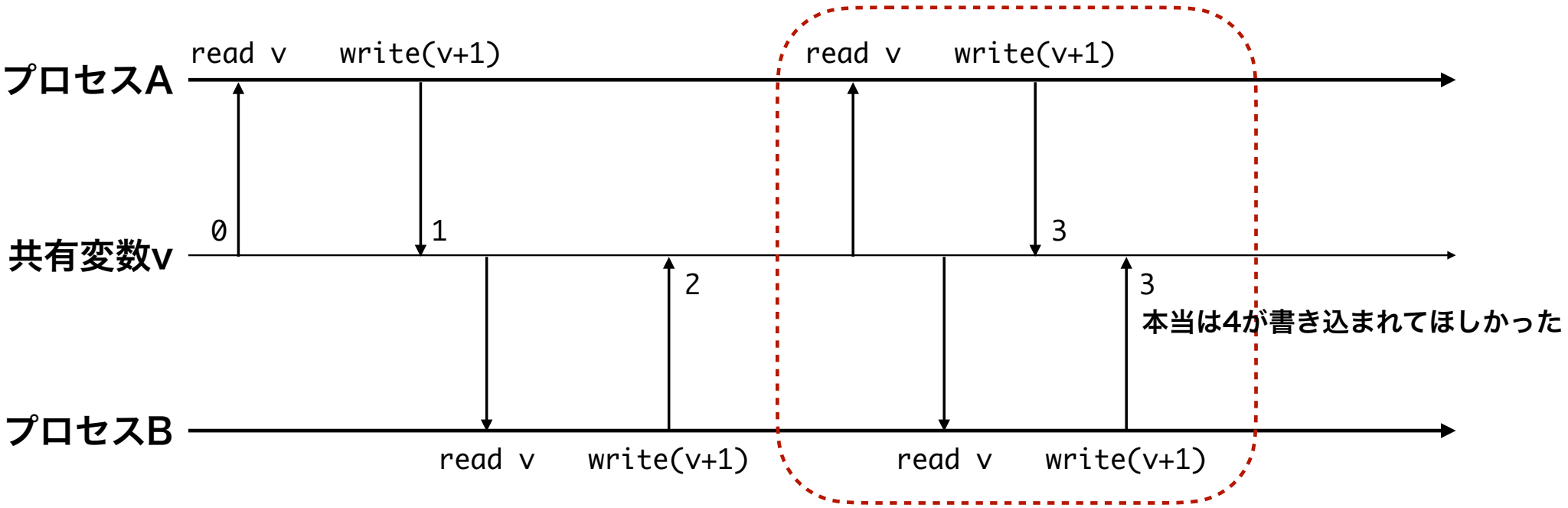
```
if (*p == val) {
    *p = newval; return True;
} else {
    return False;
}
```

レースコンディション

- 複数のプロセスやスレッドが、予期せぬ依存関係により、不具合が起きるような状態
- いわゆる並行プログラミングにおけるバグなどの要因となる
- レースコンディションを引き起こすようなプログラムコードの部分を**クリティカルセクション**と呼ぶ

レースコンディションの例

- 2つのプロセスが共有変数をメモリから読み込んでインクリメントする例



排他制御

- クリティカルセクションを実行可能なプロセスの数を制限するような制御方法
- ミューテックス、セマフォ、Readers-Writerロックがある

ミューテックス

- 最大1つのプロセス（あるいはスレッド）のみが、ロックを獲得可能な排他制御
- MUTual EXclusion（mutex）、相互排他の略
- アクセス制限を課す動作を「ロックする」、「ロックを取得する」、flagとも呼ばれる。

セマフォ

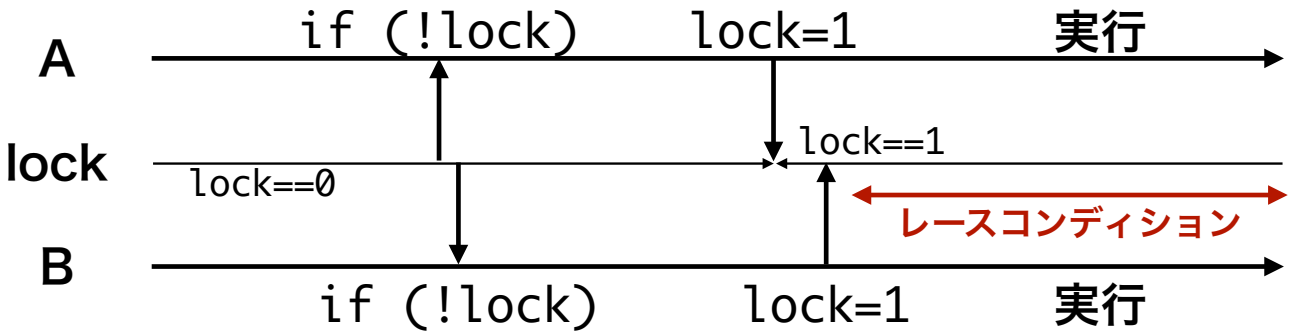
- 最大N個のプロセスのみがロックを獲得可能な排他制御
- Nは任意に設定可能
- N=1の場合がミューテックスで、セマフォはミューテックスを一般化したアルゴリズム

単純なミューテックス（誤った実装）

```
// lock変数が0のときに誰もロックを獲得しておらず、1のときに獲得
volatile int lock = 0; // 共有変数、初期値は0、volatileで最適化を抑止

void thread() { // 複数のスレッドがこの関数を同時に実行
    if (!lock) {
        lock = 1;
        // 排他実行したい
    }
}
```

タイミングによって同時に実行される可能性がある！
（レースコンディション）

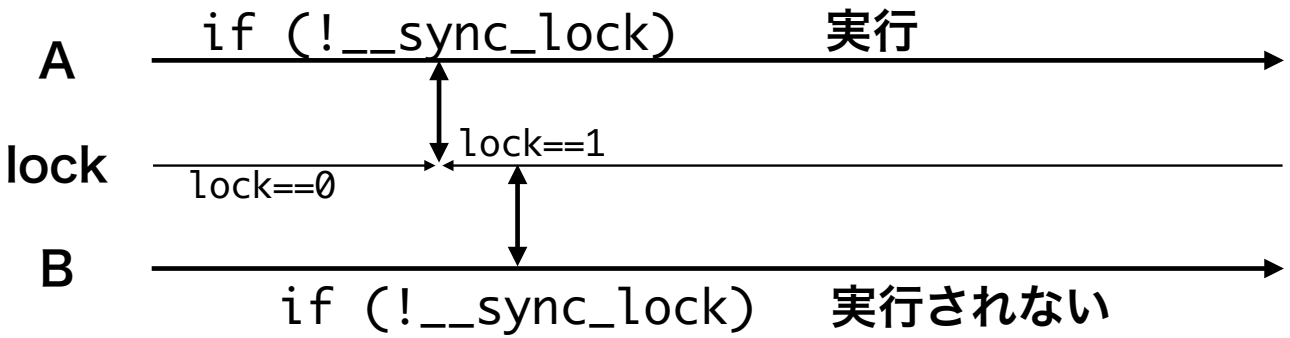


改良版ミューテックス

```
volatile int lock = 0; // 共有変数

void thread() { // 複数のスレッドがこの関数を同時に実行
    if (!__sync_lock_test_and_set(&lock, 1)) {
        // 排他実行
    }
}
```

アトミックに読み書きするため必ず排他的に実行される



スピンロックのアルゴリズム

```
int lock = 0; // 初期値0の共有変数を用意


void spinlockAqcuire(int *lock) {
    while (__sync_lock_test_and_set(&lock, 1));
    // ロックが獲得できるまでループしている
    // これがスピンに見える
}

void spinlockRelease(int *lock) {
    __sync_lock_release(&lock);
}
```

スピンロックの使い方

```
int lock = 0; // 共有変数
```

```
// 事前処理
spinlockAqcuire(&lock);

// クリティカルセクション


spinlockRelease(&lock);
// 終了処理
```

スレッド1

```
// 事前処理
spinlockAqcuire(&lock);

// クリティカルセクション


spinlockRelease(&lock);
// 終了処理
```

スレッド2

おまけ：修正版スピンロック

アトミック処理は重いため、まずは普通の方法で値を検査してから
TASで値を検査。

```
void spinlockAqcuire(volatile int *lock) {
    for (;;) {
        while(*lock);
        if (!__sync_lock_test_and_set(&lock, 1)) // 処理が重い
            break;
    }
}

void spinlockRelease(int *lock) {
    __sync_lock_release(&lock);
}
```

Pthreads

Pthreadsとは

- POSIX標準APIに準拠したスレッドライブラリ
- Linux、BSD、MacなどのUNIX系OSで標準的に利用可能
- Windowsは外部ライブラリを用意する必要あり

pthread_create/pthread_join

- pthread_create：スレッドを生成するための関数
- pthread_join：スレッドの終了を待機、リソース解放するための関数

pthread_create/pthread_joinの例

```c
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

#define NUM_THREADS 10

// スレッド用関数
void *thread_func(void *arg) {
    int id = (int)arg;
    for (int i = 0; i < 5; i++) {
        printf("id = %d, i = %d\n", id, i);
        sleep(1);
    }

    return "finished!";
}
```

```c
int main(int argc, char *argv[]) {
    pthread_t v[NUM_THREADS];
    // スレッド生成
    for (int i = 0; i < NUM_THREADS; i++) {
        if (pthread_create(&v[i], NULL, thread_func, (void *)i) != 0) {
            perror("pthread_create");
            return -1;
        }
    }

    // スレッドの終了を待機
    for (int i = 0; i < NUM_THREADS; i++) {
        char *ptr;
        if (pthread_join(v[i], (void **)&ptr) == 0) {
            printf("msg = %s\n", ptr);
        } else {
            perror("pthread_create");
            return -1;
        }
    }

    return 0;
}
```

pthread_mutex_lock, unlock

- 排他制御を行うための関数
- PTHREAD_MUTEX_INITIALIZERで初期化し、pthread_mutex_destroyでリソース解放

pthread_mutex_lock, unlockの例

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

pthread_mutex_t mut = PTHREAD_MUTEX_INITIALIZER;

void* some_func(void *arg) {
    if (pthread_mutex_lock(&mut) != 0) {
        perror("pthread_mutex_lock"); exit(-1);
    }

    // クリティカルセクション

    if (pthread_mutex_unlock(&mut) != 0) {
        perror("pthread_mutex_unlock"); exit(-1);
    }

    return NULL;
}
```

```c
int main(int argc, char *argv[]) {
    // スレッド生成
    pthread_t th1, th2;
    if (pthread_create(&th1, NULL, some_func, NULL) != 0) {
        perror("pthread_create"); return -1;
    }

    if (pthread_create(&th2, NULL, some_func, NULL) != 0) {
        perror("pthread_create"); return -1;
    }

    // スレッドの終了を待機
    if (pthread_join(th1, NULL) != 0) {
        perror("pthread_join"); return -1;
    }

    if (pthread_join(th2, NULL) != 0) {
        perror("pthread_join"); return -1;
    }

    // ミューテックスオブジェクトを開放
    if (pthread_mutex_destroy(&mut) != 0) {
        perror("pthread_mutex_destroy"); return -1;
    }

    return 0;
}
```

問題点・ロック獲得忘れ

- ミューテックスでロックすべき箇所をロックしていない

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

pthread_mutex_t mut = PTHREAD_MUTEX_INITIALIZER;

void* some_func(void *arg) {
    if (pthread_mutex_lock(&mut) != 0) {
        perror("pthread_mutex_lock"); exit(-1);
    }

    // クリティカルセクション

    if (pthread_mutex_unlock(&mut) != 0) {
        perror("pthread_mutex_unlock"); exit(-1);
    }

    return NULL;
}
```

ロック獲得を忘れて問題発生

問題点・ロック解放忘れ

- ロックしたあとに、ロック解放し忘れる

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

pthread_mutex_t mut = PTHREAD_MUTEX_INITIALIZER;

void* some_func(void *arg) {
    if (pthread_mutex_lock(&mut) != 0) {
        perror("pthread_mutex_lock"); exit(-1);
    }

    // クリティカルセクション

    if (pthread_mutex_unlock(&mut) != 0) {
        perror("pthread_mutex_unlock"); exit(-1);
    }

    return NULL;
}
```

ロックの解放を忘れると、次のロック獲得時に実行が進まなくなる（デッドロック）

Rust言語のマルチスレッド

Rust言語のMutexの特徴

- Pthreadsで問題となっていたことを解消
  - ロック獲得忘れ
  - ロック解放忘れ
  - 安全でないデータの共有方法の防止
- 型システムにより実現しているため、コンパイル時にエラーを発見可能

thread::spawn

- スレッドを生成するための関数
- クロージャを渡すことでスレッドが生成される
- デタッチスレッドのため、joinしなくて良いし、joinも出来る

Arc

- 参照カウントベースでマルチスレッドセーフなスマートポインタ
- **Syncトレイトを実装した型でないと利用できない**
- 安全でないデータ共有方法を防止

Mutex

- ミューテックスを行うための型
- **保護対象のデータを保持する**

MutexGuard

- ロック中に保護対象データにアクセスするための型
- **ロックしないと保護対象データにアクセスできない**

RustのMutexの例（1/7）

```rust
use std::sync::{Arc, Mutex};
use std::thread;

fn some_func(lock: Arc<Mutex<u64>>) {
    loop {
        // ロックしないとMutex型の中の値は参照不可
        let mut val = lock.lock().unwrap();
        *val += 1;
        println!("{}", *val);
    }
}
```

```rust
fn main() {
    // Arcはスレッドセーフな参照カウンタ型のスマートポインタ
    let lock0 = Arc::new(Mutex::new(0));

    // 参照カウンタがインクリメントされるのみで
    // 中身はクローンされない
    let lock1 = lock0.clone();

    // スレッド生成
    // クロージャ内変数へmove
    let th0 = thread::spawn(move || {
        some_func(lock0);
    });

    // スレッド生成
    // クロージャ内変数へmove
    let th1 = thread::spawn(move || {
        some_func(lock1);
    });

    // 待ち合わせ
    th0.join().unwrap();
    th1.join().unwrap();
}
```

RustのMutexの例（2/7）

```rust
use std::sync::{Arc, Mutex};
use std::thread;

fn some_func(lock: Arc<Mutex<u64>>) {
    loop {
        // ロックしないとMutex型の中の値は参照不可
        let mut val = lock.lock().unwrap();
        *val += 1;
        println!("{}", *val);
    }
}
```

保護対象データがu64型の値のMutexを作成

スマートポインタのArc型で管理

```rust
fn main() {
    // Arcはスレッドセーフな参照カウンタ型のスマートポインタ
    let lock0 = Arc::new(Mutex::new(0));

    // 参照カウンタがインクリメントされるのみで
    // 中身はクローンされない
    let lock1 = lock0.clone();

    // スレッド生成
    // クロージャ内変数へmove
    let th0 = thread::spawn(move || {
        some_func(lock0);
    });

    // スレッド生成
    // クロージャ内変数へmove
    let th1 = thread::spawn(move || {
        some_func(lock1);
    });

    // 待ち合わせ
    th0.join().unwrap();
    th1.join().unwrap();
}
```

RustのMutexの例（3/7）

```rust
use std::sync::{Arc, Mutex};
use std::thread;

fn some_func(lock: Arc<Mutex<u64>>) {
    loop {
        // ロックしないとMutex型の中の値は参照不可
        let mut val = lock.lock().unwrap();
        *val += 1;
        println!("{}", *val);
    }
}
```

Mutexをスレッド間で共有するためにクローン
ただし、実際には参照カウントをインクリメントするだけ

```rust
fn main() {
    // Arcはスレッドセーフな参照カウンタ型のスマートポインタ
    let lock0 = Arc::new(Mutex::new(0));

    // 参照カウンタがインクリメントされるのみで
    // 中身はクローンされない
    let lock1 = lock0.clone();

    // スレッド生成
    // クロージャ内変数へmove
    let th0 = thread::spawn(move || {
        some_func(lock0);
    });

    // スレッド生成
    // クロージャ内変数へmove
    let th1 = thread::spawn(move || {
        some_func(lock1);
    });

    // 待ち合わせ
    th0.join().unwrap();
    th1.join().unwrap();
}
```

RustのMutexの例（4/7）

```rust
use std::sync::{Arc, Mutex};
use std::thread;

fn some_func(lock: Arc<Mutex<u64>>) {
    loop {
        // ロックしないとMutex型の中の値は参照不可
        let mut val = lock.lock().unwrap();
        *val += 1;
        println!("{}", *val);
    }
}
```

スレッド生成

```rust
fn main() {
    // Arcはスレッドセーフな参照カウンタ型のスマートポインタ
    let lock0 = Arc::new(Mutex::new(0));

    // 参照カウンタがインクリメントされるのみで
    // 中身はクローンされない
    let lock1 = lock0.clone();

    // スレッド生成
    // クロージャ内変数へmove
    let th0 = thread::spawn(move || {
        some_func(lock0);
    });

    // スレッド生成
    // クロージャ内変数へmove
    let th1 = thread::spawn(move || {
        some_func(lock1);
    });

    // 待ち合わせ
    th0.join().unwrap();
    th1.join().unwrap();
}
```

RustのMutexの例（5/7）

```rust
use std::sync::{Arc, Mutex};
use std::thread;

fn some_func(lock: Arc<Mutex<u64>>) {
    loop {
        // ロックしないとMutex型の中の値は参照不可
        let mut val = lock.lock().unwrap();
        *val += 1;
        println!("{}", *val);
    }
}
```

排他ロック

```rust
fn main() {
    // Arcはスレッドセーフな参照カウンタ型のスマートポインタ
    let lock0 = Arc::new(Mutex::new(0));

    // 参照カウンタがインクリメントされるのみで
    // 中身はクローンされない
    let lock1 = lock0.clone();

    // スレッド生成
    // クロージャ内変数へmove
    let th0 = thread::spawn(move || {
        some_func(lock0);
    });

    // スレッド生成
    // クロージャ内変数へmove
    let th1 = thread::spawn(move || {
        some_func(lock1);
    });

    // 待ち合わせ
    th0.join().unwrap();
    th1.join().unwrap();
}
```

RustのMutexの例（6/7）

```rust
use std::sync::{Arc, Mutex};
use std::thread;

fn some_func(lock: Arc<Mutex<u64>>) {
    loop {
        // ロックしないとMutex型の中の値は参照不可
        let mut val = lock.lock().unwrap();
        *val += 1;
        println!("{}", *val);
    }
}
```

排他ロック後でないと、保護対象データにアクセス出来ない

```rust
fn main() {
    // Arcはスレッドセーフな参照カウンタ型のスマートポインタ
    let lock0 = Arc::new(Mutex::new(0));

    // 参照カウンタがインクリメントされるのみで
    // 中身はクローンされない
    let lock1 = lock0.clone();

    // スレッド生成
    // クロージャ内変数へmove
    let th0 = thread::spawn(move || {
        some_func(lock0);
    });

    // スレッド生成
    // クロージャ内変数へmove
    let th1 = thread::spawn(move || {
        some_func(lock1);
    });

    // 待ち合わせ
    th0.join().unwrap();
    th1.join().unwrap();
}
```

RustのMutexの例（7/7）

```rust
use std::sync::{Arc, Mutex};
use std::thread;

fn some_func(lock: Arc<Mutex<u64>>) {
    loop {
        // ロックしないとMutex型の中の値は参照不可
        let mut val = lock.lock().unwrap();
        *val += 1;
        println!("{}", *val);
    }
}
```

変数valがスコープ外となると、
自動でロック解放

```rust
fn main() {
    // Arcはスレッドセーフな参照カウンタ型のスマートポインタ
    let lock0 = Arc::new(Mutex::new(0));

    // 参照カウンタがインクリメントされるのみで
    // 中身はクローンされない
    let lock1 = lock0.clone();

    // スレッド生成
    // クロージャ内変数へmove
    let th0 = thread::spawn(move || {
        some_func(lock0);
    });

    // スレッド生成
    // クロージャ内変数へmove
    let th1 = thread::spawn(move || {
        some_func(lock1);
    });

    // 待ち合わせ
    th0.join().unwrap();
    th1.join().unwrap();
}
```

レポート課題

レポート課題１

・以下の意味について解説せよ

・クリティカルセクション

・レースコンディション

・アトミック処理

・ミューテックス

レポート課題2

・Rust言語のミューテックスは、Pthreadsのミューテックスよりも幾つかの点で安全であると言える。それはどのような点で、なぜ安全だと言えるのか解説せよ。

レポート課題３(情報通信工学演習のレポート)：スピンロック

- 以下のRustのスピンロック実装で、それぞれ何を行っているか解説せよ
  chap4/4.7/ch4_7_barrier/src/main.rs (https://drive.google.com/drive/folders/1PdUY6XqoMVNQ2y18i0DBzLQowJHhTcjl?usp=sharing)
- 最低でも、以下については説明すること
  - SpinLockGuard、AtomicBool型は一体何のためにあるのか？
  - compare_exchange_weakは何をする関数か？
  - SyncとSendトレイトは何のためにあるのか？
  - 何故この実装がスピンロックとなるのか?

締め切り

- 2024年2月7日 23:50（JST）