並行プログラミング入門と
Rustを用いた低レベルプログラミング

組込みマルチコアサミット EMS2021

高野 祐輝

大阪大学 特任准教授

目次

- 自己紹介
- 並行プログラミング
- 同期処理
- Rustによる低レベルプログラミング
- Rustとその現状

自己紹介

- 名前：高野 祐輝
- 所属：大阪大学 特任准教授
- 専門分野：システムソフトウェア、サイバーセキュリティ、コンピュータネットワーク、オペレーティングシステム、プログラミング言語

『並行プログラミング入門』オライリー・ジャパンより発売中

- マルチコアプログラミングに必須となる技術を網羅的に解説
- 内容：アトミック命令、Mutexなどの同期処理、async/await、グリーンスレッド、アクターモデル計算などの計算モデル、AArch64による実装
- 本日は、こちらの序盤を説明予定

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：同じ時刻で、複数のプロセスが実行状態となる
- 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

OSは一般用語の「プロセス」を専門用語としてつかっているけれど、本スライドでは一般用語として利用するニャ！

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

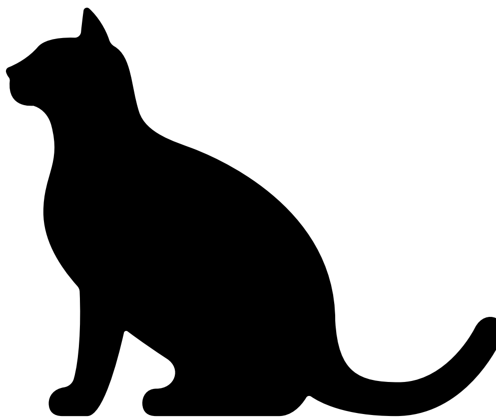
今

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

並行プログラミングには、本質的な難しさと、基本技術すらまとまった情報が無いという、2つの問題があるニャ。後者は『並行プログラミング入門』である程度解決するニャ！

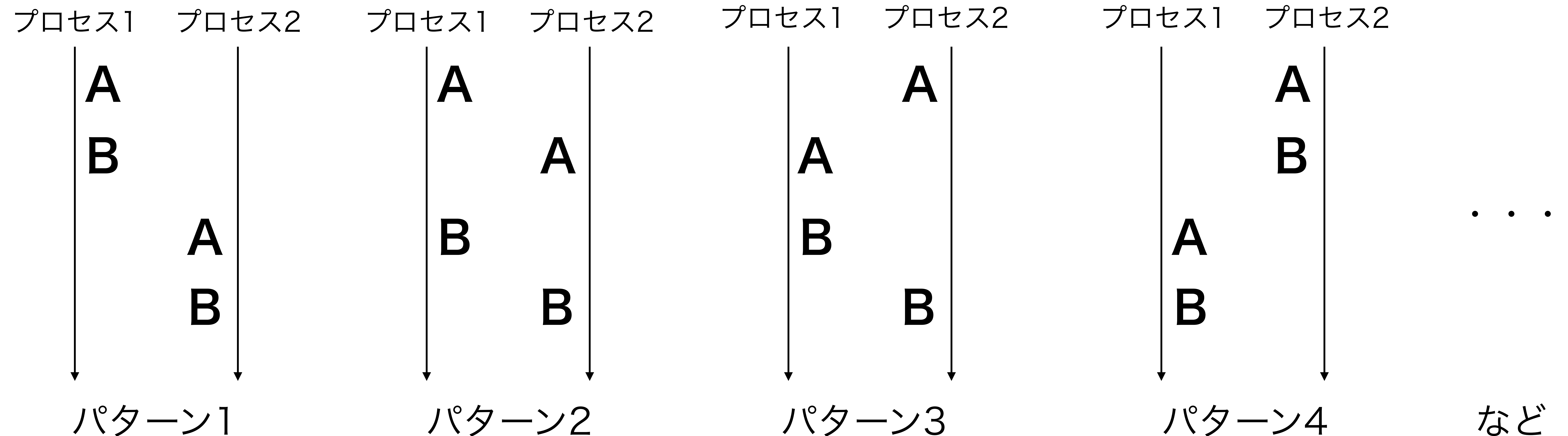


プロセスの実行パターン

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



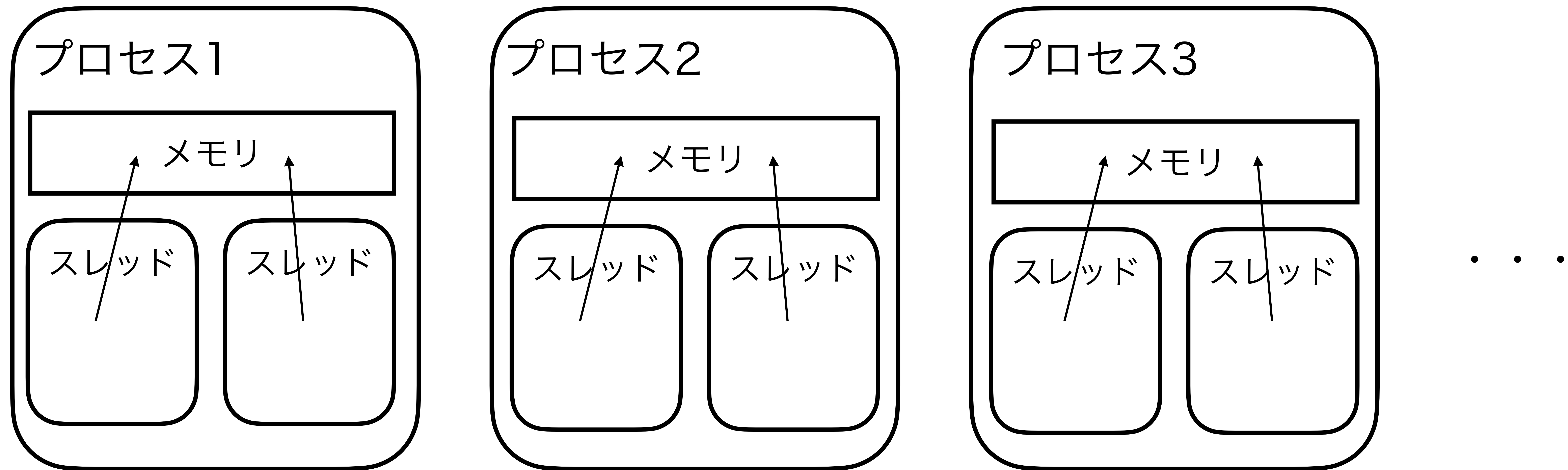
計算パターン数の爆発

- 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

目次

- アトミック処理
- レースコンディション
- 排他制御
    - ミューテックス
    - セマフォ
    - スピンロック
- まとめ

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観測することが出来ず
    - 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

アトム（分子）は古代ギリシアの哲学者デモクリトスの発明した概念で、彼は、この世界はこれ以上分割できないアトムから出来ているという考えたニャ！

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）といった命令もアトミックだが、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）

- AMD64の場合、xchg命令にコンパイルされる
- xchg命令は該当メモリのキャッシュラインをExclusiveに設定して実行されることが保証されている（排他的に実行される）

```
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_bool_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

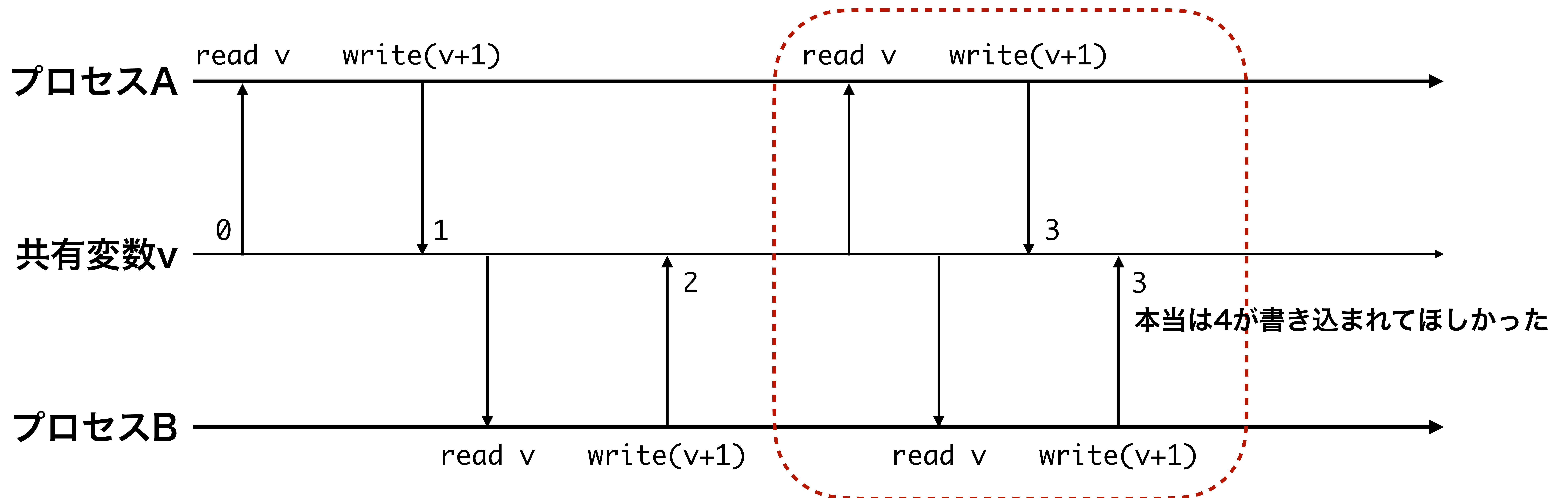
```
if (*p == val) {
    *p = newval; return True;
} else {
    return False;
}
```

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
- 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）、排他的実行の略

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

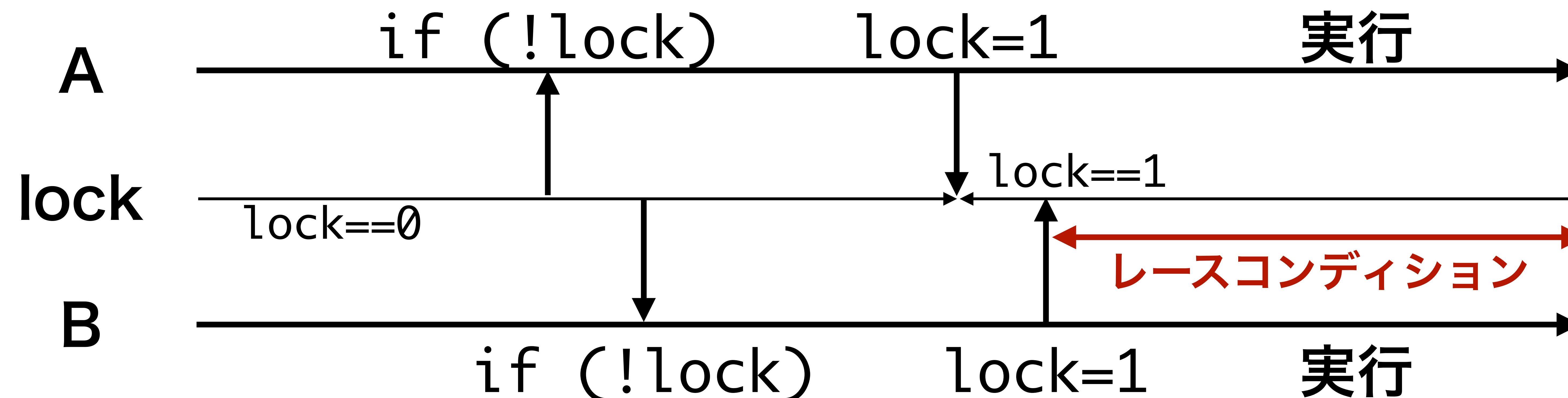
単純なミューテックス（誤った実装）

```
// 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）

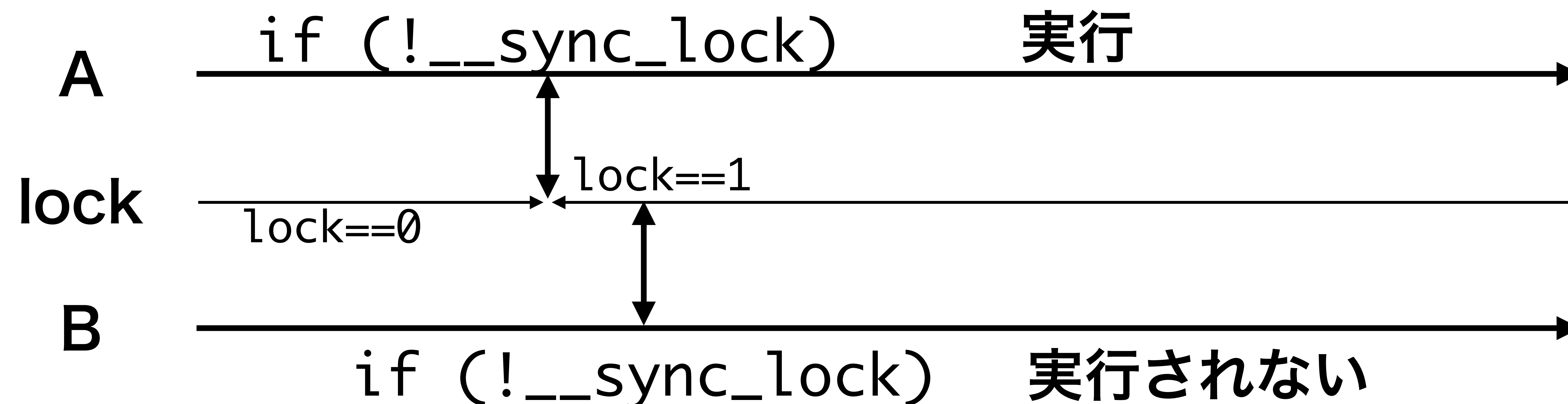


改良版ミューテックス

```
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);
}
```

まとめ

- アトミック処理から同期処理の基本までを説明した
- 現代は、CPUのアトミック処理を元にしてミューテックスなどを実装している
- これまで、CPUのアトミック処理や、同期処理についてまとまった書籍はなかったが（分散しては情報あったが体系化されていなかった）、『並行プログラミング入門』ではそられについても記載

Rustによる
低レベルプログラミング
OS実装の事例から

目次

- 背景・目的とモチベーション
- 既存研究と何故重要なのか？
- Rustとno_std
- DWARF Based Stack Unwinding (WIP)
- まとめ

背景・目的とモチベーション

- オペレーティングシステムなどの低レイヤを、安全に実装したい！
- ここでいう安全とは何か？
  - **メモリ関連のバグや脆弱性が起きない（メモリ安全性）**
    - **バッファオーバーランやぶら下がりポインタが起きないことの保証**
    - **セグメンテーションフォルトが起きないことの保証**
  - **不正な実行をしない：確実にエラーハンドリングをしているか、エラーの場合停止する**
- Rust言語では上記をある程度達成できる、と言われている
  - では、どこまで達成できるのか？その課題は何か？
  - モノリシックな伝統的な、実験的OSはRustで実装したが、よりRustを活かすにはどすれば良いか？

RustによるOSの既存研究

- Theseus（OSDI 2020）：state-spillを防ぐOSをRustで実装
- ReadLeaf（OSDI 2020）：Rust + 独自言語でゼロコピーなfault isolationを実現
- NrOS（OSDI 2021）：改良版のマルチカーネルをRustで実装
- Linux Kernel (2022年？)：mainlineに導入予定でドライバがRustで実装可能に
- Chrome OS：ファイルシステムにRustが利用されている

何故重要なのか？

- 問：LinuxとWindowsがあれば、新たなOSなんて必要ないのでは？何故今更研究するのか？
- こたえ
  - C言語とアセンブリー辺倒だったシステムソフトウェア実装が、より安全なRustに置き変わる可能性がある
  - OSの実装が変わっていくと言うことは、将来的には組み込みソフトウェアや、IoT機器などへも適用可能になる
  - 組み込みやIoT向けのソフトウェアが、メモリ安全になる！

メモリ安全は低レイヤの宿願ニャ！

Rustとno_std

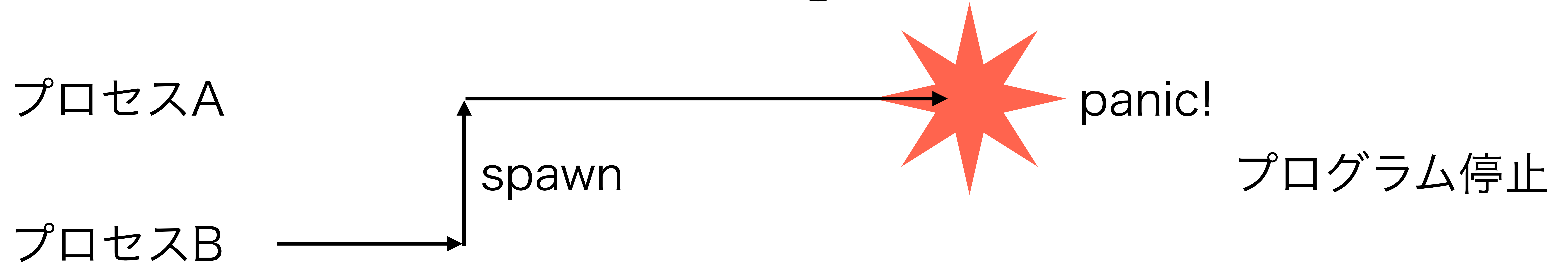
- Rustはno_std（C言語で言うところのlibc無し）でもコンパイル可能であり、そのためOS実装が可能になっている
- no_stdの制約
  - mallocなどを自前で実装する必要がある
  - **Stack Unwindingが無い**（stdだとある）
- TheseusのようにRustのみでプロセス隔離を実現するためには、Stack Unwindingが必要（なかなかハードルが高い）
- 現在、Rust no_stdのStack Unwindingについて調査中。今日はこれについて話す

Stack Unwinding

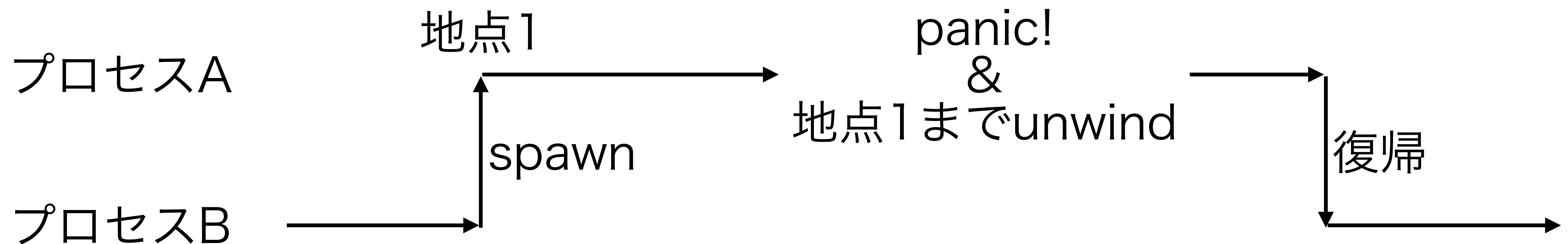
- Stack Unwindingは、呼び出し元の関数までスタックを巻き戻すこと
- 各種レジスタの値を元に戻す必要がある
- （実用上は、ヒープメモリなどの解放も必要）

Theseusでは、no_stdでStack Unwindingを実現したと簡単に言っているけれど、実はそう簡単ではないニャ！

Stack Unwindingの概要



panicによるプログラムの停止



Stack Unwindingによる復帰

Reliable and Fast DWARF-Based Stack Unwinding （OOPSLA19, 既存研究）

- デバッグ情報DWARFをもとにした、Stack Unwinding手法
- Theseusではこれをベースにしている
- 具体的には、DWARF中の.eh_frameと、Control Flow Graph（CFG）を利用して、unwind tableを生成

DWARFとはつまり、実行ファイル中に含まれているデバッグ情報ニャ！

Work-in-Progress
AArch64を対象とした、DWARFベースのStack Unwinding

- 既存実装はx86-64のみ。B4の学生と一緒に、検証と実装を実施中
- DWARF情報の検証 → done
  - 元の論文では、llvmにバグがあり、正しい.eh_frameが出力されていなかった（修正済み）
  - AArch64でも正しい.eh_frameかを再検証した結果、修正済みであることがわかった
- Stack Unwindingの実装
  - .eh_frameの取得と解析 → done
  - CFGの取得 → done
  - Stack Unwindingの実装 → 実装中

.eh_frameの例（x86-64）

Location of Content
各命令のアドレス

Canonical Frame Address
いわゆるフレームポインタ

callee-save registersが保存されている場所

return address

LOC	CFA	rbx	rbp	r12	r13	r14	r15	ra
0084950	rsp+8	u	u	u	u	u	u	c-8
0084952	rsp+16	u	u	u	u	u	c-16	c-8
0084954	rsp+24	u	u	u	u	c-24	c-16	c-8
0084956	rsp+32	c-32	u	u	u	c-24	c-16	c-8
0084958	rsp+40	c-32	c-40	u	u	c-24	c-16	c-8
0084959	rsp+144	c-32	c-40	u	u	c-24	c-16	c-8
0084962	rsp+40	c-32	c-40	u	u	c-24	c-16	c-8
0084964	rsp+32	c-32	c-40	u	u	c-24	c-16	c-8
0084966	rsp+24	c-32	u	u	u	c-24	c-16	c-8

Control Flow Graphの例（x86-64）

アセンブリ	CFAとreturn address
subq $0x10, %rsp	rsp+8 c-8
...	rsp+24 c-8
cmp %rax, %rbx	rsp+24 c-8
jne .L5	rsp+24 c-8

F rsp+24 c-8 T rsp+24 c-8

アセンブリ	CFAとreturn address
cmp %rdx, %rbx	rsp+24 c-8
je .L1	rsp+24 c-8

F rsp+24 c-8 T rsp+24 c-8

アセンブリ	CFAとreturn address
subq $0x8, %rsp	rsp+24 c-8
...	rsp+32 c-8
addq $0x8, $rsp	rsp+32 c-8
jmp .L5	rsp+24 c-8

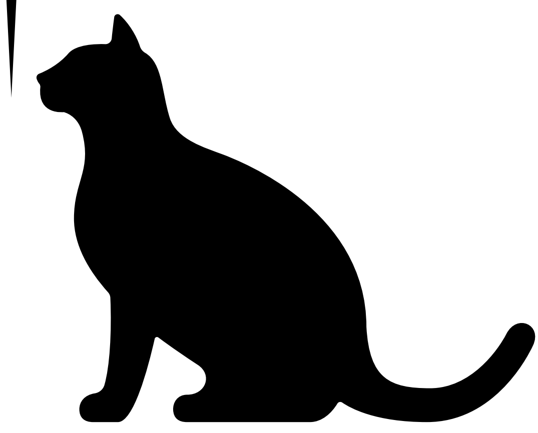
rsp+24 c-8

アセンブリ	CFAとreturn address
addq $0x10, %rsp	rsp+24 c-8
ret	rsp+8 c-8

アセンブリ CFAとreturn address

各フローのマージポイントに着目すると、CFAが同じ値になるニャ！

つまり、ジャンプがあっても、フレームポインタは変わらないニャ！



TODO: Unwind Tableの生成

- 次は、.eh_frameとCFGから、unwind table
- 生成はまだ出来ていない

論文中のアルゴリズムを、
AArch64で実装・検証中ニャ！

x86-64とAArch64の違い

- x86-64ではreturn addressはスタックに保存されるが、AArch64ではx30レジスタに保存される
- Linuxの場合、プログラムカウンタの値がバイナリと異なる
  - 32ビットモードとの互換性のため
  - 64ビットモードでは、4GiB以上のメモリ領域に配置される
    https://github.com/torvalds/linux/blob/5bfc75d92efd494db37f5c4c173d3639d4772966/arch/arm64/include/asm/elf.h#L131

実行時のPCの値が違っていて、
混乱したニャ！

まとめ

- RustでOSを実装する上での利点と課題を述べた
  - OSは組み込みや、IoT機器にも繋がる重要な課題
  - Rustを用いると、ある種の安全性が担保できる
- 一番大きな課題としてStack Unwindingが無いことを述べた
- DWARFベースのStack Unwinding手法をAArch64に適用中
  - DWARFの.eh_frame、CFGを利用してunwind tableを生成
  - .eh_frameの検証、および、.eh_frameとCFGの取得まで出来ている
  - 次は、unwind tableの生成を行う予定

Rustとその現状

Rustは何故良いのか？

- 安全である
  - メモリ安全、型安全、ヌル安全、並行プログラミングでの安全
  - 代数的データ型によるエラーハンドリング忘れ防止
- 低レイヤに強い
  - GCがないので、OSや組み込み等でも利用可能
  - システムソフトウェアを書きやすい
- 高速
- エコシステムが良い（rust-analyzer、cargoなど）

実際のところ、Rustはどこまで使えるのか？

- サーバなど、システムソフトウェアの実装としてはほぼ困らない。どんどん使っていける
- 組み込みや、OSまわりはまだまだこれから
  - 必要なライブラリがない
  - CPUアーキテクチャが未対応
- 誰が使うべきか？
  - 組み込みで業務として使うにはまだはやいか？
  - 5年後、10年後の未来を創造したい人は是非使ってみて欲しい

本当に組み込みにRustは来るのか？

- **現状ではまだわからない**
- 組み込みは、昔はアセンブリで書くのが常識だったが、いまではC言語が主流になっている（今でもアセンブリの場合も多いし、逆にArduinoのようにPythonもあるが）
- 同じように、Rustが主流になるかも？現状、一番可能性があるのがRust

組み込みでRustを使えるとどうなるか

- 各種安全性が担保される
  - IoTなどネットワークに繋がる機器は、安全性が特に重要
- async/awaitを利用可能になるかもしれない
  - JavaScriptなどの高級言語で利用可能だった非同期メカニズムのasync/awaitをベアメタルプログラミングで利用可能に
  - 実際、実験的なOS実装では実績あり
- WASMなどWeb系などでRustを利用していた人材が、組み込み業界にもリーチ可能になる

デメリット

- プログラミング言語のパラダイムが違うので、習得に時間がかかる可能性がある
    - ×難しい、○パラダイムが違う
    - 一旦慣れると、使うのは簡単。CやC++の方が難しく感じる
- 使いたいライブラリがない場合があり、ゼロから実装する必要がある可能性がある
- 使える人が少ない

発表のまとめ

発表のまとめ

- 並行プログラミングの基本から、Rustによる低レベルプログラミング、Rustとその現状までを説明
- 並行プログラミング自体は、弊著『並行プログラミング入門』がおすすめ！
- Rustは、現状ではサーバなどのシステムソフトウェア用途にはほぼ問題ない
- Rustの低レベルプログラミング、特にOSや組み込みはまだまだこれから
    - ポテンシャルは非常に大きい！
    - 今後の研究開発に期待

Rustの活躍はこれからだニャ！