

並行処理と同期

田浦健次郎

目次

共有メモリと競合状態

同期

排他制御

バリア同期

条件変数

不可分更新命令

休眠待機 vs. ビジーウェイト (繁忙待機?)

同期の実装

(高度な話題) Futex の実装

デッドロック

Contents

共有メモリと競合状態

同期

排他制御

バリア同期

条件変数

不可分更新命令

休眠待機 vs. ビジーウェイト (繁忙待機?)

同期の実装

(高度な話題) Futex の実装

デッドロック

共有メモリ (復習)

- ▶ 同一プロセス内のスレッドはメモリを共有している
- ▶ 例えば以下のプログラムは何を表示する？

```
1  /* 大域変数 */
2  int g = 0;
3
4  /* スレッドの開始関数 */
5  void * f(void * arg) {
6      g += 100;
7      return 0; }
8
9  int main() {
10     g = 200;
11     /* スレッドを作る */
12     pthread_t child_thread_id;
13     pthread_create(&child_thread_id, 0, f, 0);
14     /* 終了待ち */
15     pthread_join(child_thread_id, 0);
16     printf("g = %d\n", g); /* ここで表示されるのは? */
17     return 0;
18 }
```

次のプログラムは?

- ▶ 違い: 2つの子スレッドを作る (f は先と同じ)

```
1  int main() {
2      int err;
3      g = 200;
4      /* スレッドを作る */
5      pthread_t child_thread_id[2];
6      for (int i = 0; i < 2; i++)
7          pthread_create(&child_thread_id[i], 0, f, 0);
8      /* 終了待ち */
9      for (int i = 0; i < 2; i++)
10         pthread_join(child_thread_id[i], 0);
11     printf("g = %d\n", g);
12     return 0;
13 }
```

望む結果 (400) にならない実行系列

| | 子スレッド A | 子スレッド B |
|-----|-----------------|-----------------|
| A1: | $t_0 = g(200);$ | |
| B1: | | $t_1 = g(200);$ |
| B2: | | $g = 300;$ |
| A2: | $g = 300;$ | |

- ▶ 他にも様々なケースが考えられる
 - ▶ e.g., A1; B1; A2; B2
- ▶ うまく行くのは以下の2ケースのみ
 - ▶ A1; A2; B1; B2;
 - ▶ B1; B2; A1; A2;
- ▶ 「うまく行く」 \iff 「片方の $g += 100$ の間にもう一方のスレッドによって g が書き換えられていない」

注: 問題の根本

- ▶ `g += 100` が「実は」

```
t = g;
```

```
g = t + 100;
```

の二つの文に分かれて実行されていると聞いて、「他にもやり方はあるかも知れない, なぜそのように決めつけるのだ?」と思うのは正しい疑問

- ▶ より本質的な問題の記述は以下

- ▶ CPU が「一度に」行えるメモリに対する操作は, read (load 命令), write (store 命令) **そのどちらか**である
- ▶ そのもとで, ある変数に対する加算を複数のスレッドが「確実に」行うには?

用語: 競合状態

定義: 以下のような状態を「競合状態」という

1. 複数のスレッドが
同じ場所 (変数, 配列, 構造体の要素 etc.) を, 並行して
アクセスしている
2. うち少なくとも1つは書き込みである

- ▶ 競合状態があるプログラムは,
 - ▶ ほとんどの場合, **非決定的**: 実行のタイミングによって結果が異なる
 - ▶ ⇒ (ほとんどの場合, そのうち一部しか「望ましい」結果ではないため) 正しくないプログラム
- ▶ 用語: **際どい領域, クリティカルセクション (critical section)**: コード上で, 競合状態が発生している領域

競合状態を大雑把に分類

- ▶ 不可分性 (atomicity) の崩れ:
 - ▶ 「一度にできない一連の操作」の途中で, 他の処理 (更新) が挟まるために, 意図した操作が行えなくなる状態
 - ▶ 「一度にできない一連の操作」の例
 - ▶ 1つの変数を読み出し～変更
 - ▶ 2つ以上の変数を読み出し, または書き込み
 - ▶ 用語「一度に出来る」≈ 他の処理が挟まらないことが保証されている ≡ 不可分 (atomic) に実行できる
- ▶ 順序, 依存関係 (dependency) の崩れ:
 - ▶ 複数スレッド間で, 読み書きにある順序 (例えば書いてから読む) を保証しなくてはならないが, それが崩れるために意図した値の通信が行えなくなる状態

Contents

共有メモリと競合状態

同期

排他制御

バリア同期

条件変数

不可分更新命令

休眠待機 vs. ビジーウェイト (繁忙待機?)

同期の実装

(高度な話題) Futex の実装

デッドロック

同期 (synchronization)

- ▶ 一般的な意味: 歩調を合わせる, タイミングをそろえる
- ▶ 並行処理という文脈での意味: 競合状態を避けるための, タイミングの制御

同期の種類

- ▶ 排他制御 (目的: 不可分性の保証)
- ▶ バリア同期 (多スレッド間で順序を保証)
- ▶ 条件変数 (目的: 順序の保証含め, 「何か起きるまで待ってから実行」をするための汎用機構)
- ▶ ... (他にも色々)

Contents

共有メモリと競合状態

同期

排他制御

バリア同期

条件変数

不可分更新命令

休眠待機 vs. ビジーウェイト (繁忙待機?)

同期の実装

(高度な話題) Futex の実装

デッドロック

排他制御 (mutual exclusion; mutex)

- ▶ 排他制御 ≈ 一人しか入れない部屋 (個室トイレ)
- ▶ 排他制御に対する操作
 - ▶ **lock** ≈ トイレが空いていれば入ってカギをかける; 空いていなければ空くまで待つ
 - ▶ **unlock** ≈ カギを開けてトイレを空ける
- ▶ 典型的な使い方 (不可分に行いたい操作を lock/unlock ではさむ)

```
1 lock(...);  
2   操作  
3 unlock(...);
```



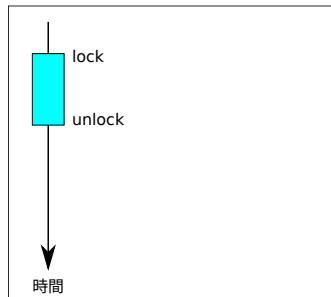
Pthread の mutex API

- ▶ `#include <pthread.h>`
- ▶ `pthread_mutex_t m; /* 排他制御オブジェクト */`
- ▶ `pthread_mutex_init(&m, attr);`
- ▶ `pthread_mutex_destroy(&m);`
- ▶ `pthread_mutex_lock(&m); /* lock */`
- ▶ `pthread_mutex_try_lock(&m);`
- ▶ `pthread_mutex_unlock(&m); /* unlock */`

Mutex API によって保証されること

- ある mutex m に対して, `pthread_mutex_lock(m)` と `pthread_mutex_unlock(m)` に挟まれた (前者が return してから後者を呼び出すまでの) 時間帯を「 m による排他区間」と呼ぶことにする

```
1 pthread_mutex_lock(m);  
2 ...  
3 pthread_mutex_unlock(m);
```

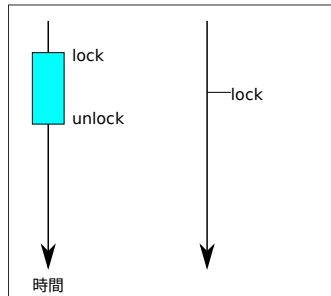


- 保証: 同一の m に対する排他区間は時間的に重ならない

Mutex API によって保証されること

- ある mutex m に対して, `pthread_mutex_lock(m)` と `pthread_mutex_unlock(m)` に挟まれた (前者が return してから後者を呼び出すまでの) 時間帯を「 m による排他区間」と呼ぶことにする

```
1 pthread_mutex_lock(m);  
2 ...  
3 pthread_mutex_unlock(m);
```

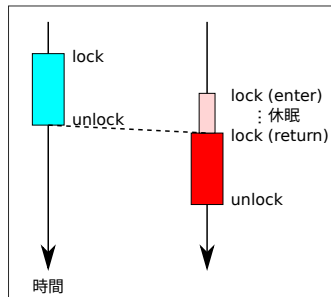


- 保証: 同一の m に対する排他区間は時間的に重ならない

Mutex API によって保証されること

- ある mutex m に対して, `pthread_mutex_lock(m)` と `pthread_mutex_unlock(m)` に挟まれた (前者が return してから後者を呼び出すまでの) 時間帯を「 m による排他区間」と呼ぶことにする

```
1 pthread_mutex_lock(m);  
2 ...  
3 pthread_mutex_unlock(m);
```



- 保証: 同一の m に対する排他区間は時間的に重ならない

最初の間違いの修正

```
1 void * f(void * arg) {  
2     g += 100;  
3     return 0; }
```

⇒

```
1 pthread_mutex_t m;  
2  
3 void * f(void * arg) {  
4     pthread_mutex_lock(&m);  
5     g += 100;  
6     pthread_mutex_unlock(&m);  
7     return 0;  
8 }  
9  
10 int main() {  
11     pthread_mutex_init(&m, attr);  
12     ...  
13 }
```

同期を隠ぺいしたデータ構造

- ▶ データをスレッドで更新する必要性が生ずるたびに、mutex を別途定義するのは煩わしい・間違いの元
- ▶ データ + それを保護する mutex (または今後述べる同期のためのデータ) をひとつのデータ構造に隠ぺいするのが通常 (スレッドセーフなデータ構造・関数)
- ▶ 例

```
1 typedef struct {
2     int x;                // 更新される変数
3     pthread_mutex_t m; // xを守るための mutex
4 } counter_t;
5 int counter_inc(counter_t * c, int dx) {
6     pthread_mutex_lock(&c->m);
7     int x = c->x;
8     c->x = x + dx;
9     pthread_mutex_unlock(&c->m);
10    return x;
11 }
```

Contents

共有メモリと競合状態

同期

排他制御

バリア同期

条件変数

不可分更新命令

休眠待機 vs. ビジーウェイト (繁忙待機?)

同期の実装

(高度な話題) Futex の実装

デッドロック

バリア同期

- ▶ バリア ≈ 出走馬のゲート (発馬機)
- ▶ 全員がそろそろまで待ち, そろったら一斉に開ける



<https://commons.wikimedia.org/w/index.php?curid=6317140>

PthreadのバリアAPI

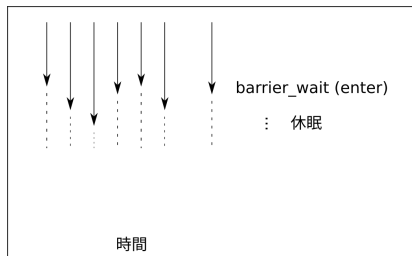
- ▶ `#include <pthread.h>`
- ▶ `pthread_barrier_t b; /* バリアオブジェクト */`
- ▶ `pthread_barrier_init(&b, attr, count);`
`/* count=参加するスレッド数 */`
- ▶ `pthread_barrier_destroy(&b);`
- ▶ `pthread_barrier_wait(&b);`
`/* 同期点に到達; 他のスレッドを待つ */`

バリア API によって保証されること

b : `count = n` で初期化されたバリアオブジェクト

- ▶ n スレッドが `pthread_barrier_wait(b)` を呼び出すまで、どの呼び出しもリターンしない
- ▶ 実践的な意味: `pthread_barrier_wait(b)` を
 - ▶ 呼び出す前 (A) の書き込み → リターン後 (B) の読み込みに伝わる
 - ▶ 呼び出す前 (A) の読み込み → リターンした後 (B) の値は読まない (前の値が潰されている心配はない)

```
1  ... (A) ...  
2  barrier_wait(b);  
3  ... (B) ...  
4
```

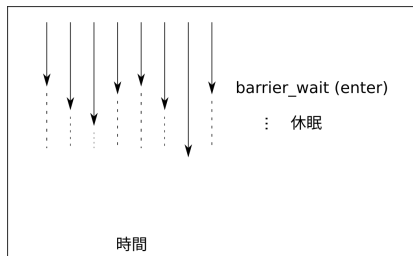


バリア API によって保証されること

b : `count = n` で初期化されたバリアオブジェクト

- ▶ n スレッドが `pthread_barrier_wait(b)` を呼び出すまで、どの呼び出しもリターンしない
- ▶ 実践的な意味: `pthread_barrier_wait(b)` を
 - ▶ 呼び出す前 (A) の書き込み → リターン後 (B) の読み込みに伝わる
 - ▶ 呼び出す前 (A) の読み込み → リターンした後 (B) の値は読まない (前の値が潰されている心配はない)

```
1  ... (A) ...  
2  barrier_wait(b);  
3  ... (B) ...  
4
```

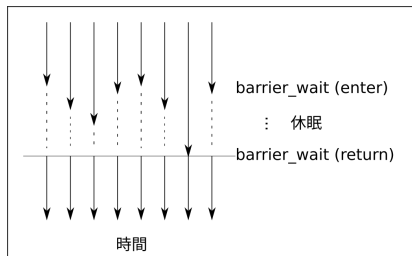


バリア API によって保証されること

b : count = n で初期化されたバリアオブジェクト

- ▶ n スレッドが `pthread_barrier_wait(b)` を呼び出すまで、どの呼び出しもリターンしない
- ▶ 実践的な意味: `pthread_barrier_wait(b)` を
 - ▶ 呼び出す前 (A) の書き込み → リターン後 (B) の読み込みに伝わる
 - ▶ 呼び出す前 (A) の読み込み → リターンした後 (B) の値は読まない (前の値が潰されている心配はない)

```
1  ... (A) ...  
2  barrier_wait(b);  
3  ... (B) ...  
4
```



Contents

共有メモリと競合状態

同期

排他制御

バリア同期

条件変数

不可分更新命令

休眠待機 vs. ビジーウェイト (繁忙待機?)

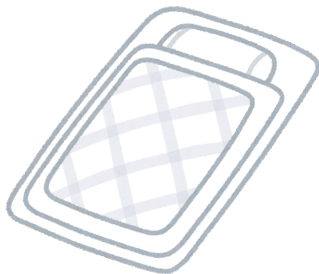
同期の実装

(高度な話題) Futex の実装

デッドロック

条件変数 (condition variable)

- ▶ 条件変数 ≈ 布団
- ▶ 「ある条件が整うまで待つ」, 「待っているスレッドを起こす」ための汎用同期機構



Pthread の条件変数 API

- ▶ `#include <pthread.h>`
- ▶ `pthread_cond_t c;`
- ▶ `pthread_cond_init(&c, attr);`
- ▶ `pthread_cond_destroy(&c);`
- ▶ `pthread_cond_wait(&c, &m); /* 寝る */`
 - ▶ `pthread_mutex_t m;`
- ▶ `pthread_cond_broadcast(&c); /* 全員起こす */`
- ▶ `pthread_cond_signal(&c); /* 誰か一人起こす */`

pthread_cond_wait(c , m) の動作

- ▶ pthread_cond_wait(c , m) を呼び出したスレッドが m をロックしていることが前提. その上で,

pthread_cond_wait(c , m) の動作

- ▶ pthread_cond_wait(c , m) を呼び出したスレッドが m をロックしていることが前提. その上で,
 - ▶ m を unlock する
 - ▶ c の上で寝る (中断・ブロックする)
- を不可分に行う

pthread_cond_wait(c , m) の動作

- ▶ pthread_cond_wait(c , m) を呼び出したスレッドが m をロックしていることが前提. その上で,
 - ▶ m を unlock する
 - ▶ c の上で寝る (中断・ブロックする)
- を不可分に行う
- ▶ 目覚めてリターンする際はまた m を lock している

pthread_cond_wait(c , m) の動作

- ▶ pthread_cond_wait(c , m) を呼び出したスレッドが m をロックしていることが前提. その上で,
 - ▶ m を unlock する
 - ▶ c の上で寝る (中断・ブロックする)を不可分に行う
 - ▶ 目覚めてリターンする際はまた m を lock している

- ▶ この一見複雑な動作の意味は後にわかる

例: 飽和するカウンタ

- ▶ 例として以下のような API を実現することを考える

```
1 typedef struct { ... } scounter_t;
2
3 /* 初期化. 飽和する値capacity を指定. カウンタの初期値を 0 にセット */
4 int scounter_init(scounter_t * s, int capacity);
5
6 /* +1; ただし capacity を超える場合は (誰かが dec してくれるのを) 待つ */
7 int scounter_inc(scounter_t * s);
8
9 /* -1; 簡単のため 0 を下回ってもよい */
10 int scounter_dec(scounter_t * s);
```

- ▶ inc, dec とも複数のスレッドが並行に呼び出し得る (それでも正しく動作する. i.e., スレッドセーフ)

飽和なし版

▶ データ構造 (mutex を隠ぺい)

```
1 typedef struct {  
2     pthread_mutex_t m;  
3     int x;  
4     int capacity;  
5 } scounter_t;
```

▶ inc (足し算を不可分に)

```
1 int scounter_inc(scounter_t * s) {  
2     pthread_mutex_lock(&s->m);  
3     int x = s->x;  
4     s->x = x + 1;  
5     pthread_mutex_unlock(&s->m);  
6     return x;  
7 }
```

飽和あり版 概要

▶ 飽和していたら待機

```
1 int scounter_inc(scounter_t * s) {  
2     pthread_mutex_lock(&s->m);  
3     int x = s->x;  
4     if (x == s->capacity) {  
5         ... 待機 ...  
6     }  
7     s->x = s->x + 1;  
8     pthread_mutex_unlock(&s->m);  
9     return x; }
```

- ▶ 上記の「待機」をするのが `pthread_cond_wait`
- ▶ ただし一般には目覚めたら `s->x < s->capacity` という保証はない
- ▶ → 目覚めたらまたやり直すコード (ループ) に変形

飽和あり版 概要

```
1 int scounter_inc(scounter_t * s) {
2     pthread_mutex_lock(&s->m);
3     int x;
4     while (1) {
5         x = s->x;
6         if (x < s->capacity) break;
7         pthread_cond_wait(&s->c, &s->m); /* 待機 */
8     }
9     /* 仕事が可能になった */
10    s->x = x + 1;
11    pthread_mutex_unlock(&s->m);
12    return x;
13 }
```

- ▶ -1 したことで飽和状態が解消されたら、寝ている人を(いれば)起こす

```
1 int scounter_dec(scounter_t * s) {
2     pthread_mutex_lock(&s->m);
3     int x = s->x;
4     s->x = x - 1;
5     if (x == s->capacity) {
6         pthread_cond_broadcast(&s->c);
7     }
8     pthread_mutex_unlock(&s->m);
9     return x;
10 }
11
```

pthread_cond `_signal` vs `_broadcast`

- ▶ `broadcast(c)` : c 上で寝ている人を全員起こす
- ▶ `signal(c)` : c 上で寝ている人のうちどれか一人を起こす
- ▶ 「誰が起きてもいい」と確信出来る場合は `signal`, そうでなければ `broadcast` を使う

条件変数の使い方テンプレート

- ▶ 「条件 C が成り立つまで待って A をする」

```
1 pthread_mutex_lock(&m);
2 while (1) {
3     C = ...; /* 条件評価 */
4     if (C) break;
5     pthread_cond_wait(&c, &m);
6 }
7 /* C が成り立っているのでここで何かをする */
8 A
9 /* 寝ている誰かを起こせそうなら起こす */
10 ...
11 pthread_mutex_unlock(&m);
```

pthread_cond_wait(*c*, *m*) の動作

- ▶ pthread_cond_wait が条件変数だけでなく, (lock されたことが前提の) mutex を引数に取るという仕様は一見わかりにくい (必然性がわからない)
- ▶ しかしよく考えると実によく出来ている

```
1 pthread_mutex_lock(&m);
2 while (1) {
3     C = ...; /* 条件評価 */
4     if (C) break;
5     pthread_cond_wait(&c, &m);
6 }
7     ...
8 pthread_mutex_unlock(&m);
```

pthread_cond_wait(*c*, *m*) の動作

```
1 pthread_mutex_lock(&m);
2 while (1) {
3     C = ...; /* 条件評価 */
4     if (C) break;
5     pthread_cond_wait(&c, &m);
6 }
7 ...
8 pthread_mutex_unlock(&m);
```

1. 通常データ構造は、どのみち何らかの mutex で保護されているので、条件 *C* を判定する際に mutex を lock しているのは自然

pthread_cond_wait(*c*, *m*) の動作

```
1 pthread_mutex_lock(&m);
2 while (1) {
3     C = ...; /* 条件評価 */
4     if (C) break;
5     pthread_cond_wait(&c, &m);
6 }
7 ...
8 pthread_mutex_unlock(&m);
```

1. 通常データ構造は、どのみち何らかの mutex で保護されているので、条件 *C* を判定する際に mutex を lock しているのは自然
2. しかも自分がブロックする際はその mutex を開放しないと、他のスレッドがデータを変更できない (pthread_cond_wait がそれをやってくれているのは親切)

pthread_cond_wait(*c*, *m*) の動作

```
1 pthread_mutex_lock(&m);
2 while (1) {
3     C = ...; /* 条件評価 */
4     if (C) break;
5     pthread_cond_wait(&c, &m);
6 }
7     ...
8 pthread_mutex_unlock(&m);
```

1. 通常データ構造は、どのみち何らかの mutex で保護されているので、条件 *C* を判定する際に mutex を lock しているのは自然
2. しかも自分がブロックする際はその mutex を開放しないと、他のスレッドがデータを変更できない (pthread_cond_wait がそれをやってくれているのは親切)
3. しかしそれでも、「unlock」と「寝る」は別の API でも良いのではという疑問が残る

これでは何がいけないか?

- ▶ `pthread_cond_wait*(c)` : `c` 上で寝るだけの (架空の)API

```
1 pthread_mutex_lock(&m);
2 while (1) {
3     C = ...; /* 条件評価 */
4     if (C) break;
5     pthread_mutex_unlock(&m); // unlock して
6     pthread_cond_wait*(&c);   // 寝る
7     pthread_mutex_lock(&m);   // 目覚めたらまたlock
8 }
9 ...
10 pthread_mutex_unlock(&m);
```

Lost wake up 問題

- ▶ 本来起こされるべき (ブロックしてはならない) 人がブロックしてしまう問題

Lost wake up 問題

- ▶ 本来起こされるべき (ブロックしてはならない) 人がブロックしてしまう問題



mutex



cond

Lost wake up 問題

- ▶ 本来起こされるべき (ブロックしてはならない) 人がブロックしてしまう問題



Lost wake up 問題

- ▶ 本来起こされるべき (ブロックしてはならない) 人がブロックしてしまう問題



Lost wake up 問題

- ▶ 本来起こされるべき (ブロックしてはならない) 人がブロックしてしまう問題



mutex



cond



Lost wake up 問題

- ▶ 本来起こされるべき (ブロックしてはならない) 人がブロックしてしまう問題



mutex



cond

Lost wake up 問題

- ▶ 本来起こされるべき (ブロックしてはならない) 人がブロックしてしまう問題
- ▶ unlock とブロックが「不可分に」行われれば起きないことに注意



mutex



cond

Lost wake up 問題

注: 関数名は適宜短縮している (pthread_mutex_lock → lock など)

```
1 inc(s) {
2   lock(&s->m);
3   while (1) {
4     if (s->x < s->capacity) break;
5     unlock(&s->m);
6     cond_wait*(&s->c, &s->m);
7     lock(&s->m); }
8   ...
```

```
1 dec(s) {
2   lock(&s->m);
3   int x = s->x; s->x = x - 1;
4   if (x == s->capacity) {
5     cond_broadcast(&s->c);
6   }
7   unlock(&s->m);
```

| s->x == capacity | |
|---|--|
| inc | dec |
| 5: if (s->x < s->capacity) ...; 6: unlock(&s->m) | 2: lock(&s->m) 3: int x = s->x; s->x = x - 1; 4: if (x == s->capacity) { 5: cond_broadcast(&s->c) |
| 7: wait*(&s->m) | |

- ▶ cond_wait の仕様 (m の開放 + 休眠を「不可分に」行う) は、まさにこうならないことを保証している

Contents

共有メモリと競合状態

同期

排他制御

バリア同期

条件変数

不可分更新命令

休眠待機 vs. ビジーウェイト (繁忙待機?)

同期の実装

(高度な話題) Futex の実装

デッドロック

不可分更新命令

- ▶ CPU が不可分に行えるのは, 64 bit までの load, store のどちらか (*)
- ▶ それ以上の処理の不可分性を保証するための基本は排他制御

```
1 lock(m);  
2   ...  
3 unlock(m);
```

実は...

- ▶ 一変数に対する読み書きを不可分に行ういくつかの命令がある (つまり (*) は正確ではない)
 - ▶ ad-hoc な命令
 - ▶ fetch&add
 - ▶ test&set
 - ▶ swap
 - ▶ 汎用命令
 - ▶ compare&swap

ad-hoc な不可分更新命令

それぞれ以下を不可分に行う命令

- ▶ `test&set p` (0 だったら 1 にする)

```
1  if (*p == 0) {  
2      *p = 1; return 1;  
3  } else {  
4      return 0;  
5  }
```

- ▶ `fetch&add p,x`

```
1  *p = *p + x;
```

- ▶ `swap p,r`

```
1  x = *p;  
2  *p = r;  
3  r = x;
```

GCC: https://gcc.gnu.org/onlinedocs/gcc/_005f_005fsync-Builtins.html

compare&swap (CAS)

- ▶ compare&swap p, r, s

```
1 x = *p;  
2 if (x == r) {  
3     *p = s;  
4     s = x;  
5 }
```

- ▶ 自分が読んだ値が書き換わっていないことを確かめながら、書き込むのに使える
- ▶ GCC の関数 2 種類
 - ▶ `bool __sync_bool_compare_and_swap(type *p, type r, type s)` (swap が起きたかどうかを返す)
 - ▶ `type __sync_val_compare_and_swap(type *p, type r, type s)` ($*p$ に入っていた値を返す)

compare&swap (CAS)

ad-hoc なプリミティブを包含

▶ test&set(p) \equiv

```
1 return __sync_bool_compare_and_swap(p, 0, 1);
```

▶ fetch&add(p, x) \equiv

```
1 while (1) {  
2     type r = *p;  
3     if (__sync_bool_compare_and_swap(p, r, r + x)) return r;  
4 }
```

▶ swap(p, r) \equiv

```
1 while (1) {  
2     type o = *p;  
3     if (__sync_bool_compare_and_swap(p, o, r)) return o;  
4 }
```

compare&swap テンプレート

- ▶ $*p = f(*p)$; という更新を不可分に行う

```
1 while (1) {  
2     x = *p;  
3     y = f(x);  
4     if (__sync_bool_compare_and_swap(p, x, y)) break;  
5 }
```

- ▶ 一つの場所 (変数, 配列, 構造体要素) に対する, 「読み出し~計算~書き込み」を不可分に行うのに排他制御よりも高速
- ▶ 排他制御を実装するために有用 (必要) ⇒ 次節
- ▶ 二つ以上の場所に対しては適用困難

Contents

共有メモリと競合状態

同期

排他制御

バリア同期

条件変数

不可分更新命令

休眠待機 vs. ビジーウェイト (繁忙待機?)

同期の実装

(高度な話題) Futex の実装

デッドロック

飽和カウンタ再掲

▶ inc

```
1 int scounter_inc(scounter_t * s) {
2     pthread_mutex_lock(&s->m);
3     int x;
4     while (1) {
5         x = s->x;
6         if (x < s->capacity) break;
7         pthread_cond_wait(&s->c, &s->m); /* 待機 */ }
8     s->x = x + 1;
9     pthread_mutex_unlock(&s->m);
10    return x; }
```

▶ dec

```
1 int scounter_dec(scounter_t * s) {
2     pthread_mutex_lock(&s->m);
3     int x = s->x;
4     s->x = x - 1;
5     if (x == s->capacity) {
6         pthread_cond_broadcast(&s->c); /* 起こす */ }
7     pthread_mutex_unlock(&s->m);
8     return x; }
```

飽和カウンタの別解? (1)

- ▶ inc: cond_wait で寝たり起きたりは何のため? これじゃ何がダメ?

```
1  int scounter_inc(scounter_t * s) {
2      pthread_mutex_lock(&s->m);
3      int x;
4      while (1) {
5          x = s->x;
6          if (x < s->capacity) break;
7          pthread_mutex_unlock(&s->m); pthread_mutex_lock(&s->m); }
8      s->x = x + 1;
9      pthread_mutex_unlock(&s->m);
10     return x; }
```

- ▶ dec:

```
1  int scounter_dec(scounter_t * s) {
2      pthread_mutex_lock(&s->m);
3      int x = s->x;
4      s->x = x - 1;
5      /* cond_broadcast 不要 */
6      pthread_mutex_unlock(&s->m);
7      return x; }
```

飽和カウンタの別解? (2)

- ▶ inc: 競合状態は `s->x` だけなので, 不可分更新命令を使えばなぞめいた mutex unlock/lock も不要

```
1 int scounter_inc'(scounter_t * s) {
2     while (1) {
3         int x = s->x;
4         if (x < s->capacity) {
5             if (compare_and_swap(&s->x, x, x + 1)) {
6                 return x;
7             }
9 }
```

- ▶ dec: `s->x` を不可分命令で減らすだけ. 起こす必要もない

```
1 int scounter_dec'(scounter_t * s) {
2     return fetch_and_add(&s->x, -1);
3 }
```


飽和カウンタの別解? (2)

- ▶ inc: 競合状態は `s->x` だけなので, 不可分更新命令を使えばなぞめいた mutex unlock/lock も不要

```
1 int scounter_inc'(scounter_t * s) {
2     while (1) {
3         int x = s->x;
4         if (x < s->capacity) {
5             if (compare_and_swap(&s->x, x, x + 1)) {
6                 return x;
7             }
9 }
```

- ▶ dec: `s->x` を不可分命令で減らすだけ. 起こす必要もない

```
1 int scounter_dec'(scounter_t * s) {
2     return fetch_and_add(&s->x, -1);
3 }
```

一般に

- ▶ もう少し一般には、「条件 C が成り立つまで待つ」を、条件変数を用いたテンプレート

```
1 pthread_mutex_lock(&m);
2 while (1) {
3     C = ...; /* 条件評価 */
4     if (C) break;
5     pthread_cond_wait(&c, &m);
6 }
7 ...
8 pthread_mutex_unlock(&m);
```

- ▶ のかわりに、

```
1 pthread_mutex_lock(&m);
2 while (1) {
3     C = ...; /* 条件評価 */
4     if (C) break;
5     pthread_mutex_unlock(&m); pthread_mutex_lock(&m);
6 }
7 ...
8 pthread_mutex_unlock(&m);
```

では何がいけないか?

休眠待機 vs. ビジーウェイト

- ▶ 休眠待機 (blocking wait)
 - ▶ 表面的には, `futex`, `cond_wait` などの API を呼んで待つ
 - ▶ 本質的には, OS に「CPU を割り当てなくて良い」とわかるように待つ (ブロックする, 実行可能でなくなるようなシステムコールを呼ぶ)
 - ▶ 前スライドの `lock`, `scounter_inc`
- ▶ ビジーウェイト・スピンウェイト (busy wait, spin wait)
 - ▶ 表面的には, `while (!条件) { 何もしない }` みたいな待ち方
 - ▶ 本質的には, OS に「CPU を割り当てなくて良い」ことがわからない待ち方 (実行可能であり続ける)
 - ▶ 前スライドの `lock'`, `scounter_inc'`

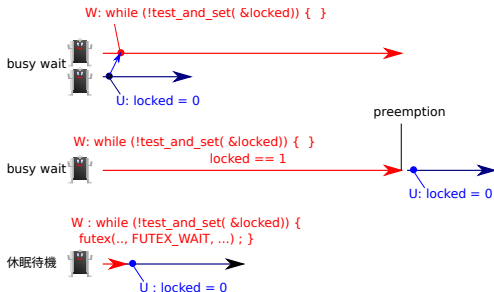
ビジーウェイトの問題点

条件を待っているスレッド (W) も, 次の preemption まで (数 ms のオーダー), CPU を使い続ける \Rightarrow

1. 消費電力増加
2. 他のアプリケーションの邪魔をする
3. (重要・気づきにくい) ブロックして待てば「条件」を成立させるスレッド (U) がすぐに CPU を得られたかも知れないが, その機会を (次の preemption まで) 奪う \Rightarrow 条件が成り立つまでの時間の増大 \Rightarrow 重大な性能低下

ビジーウェイトの使いみち

- ▶ 基本は使わない
- ▶ すべてのスレッドが「同時に」実行されている (常に CPU が割り当てられている) と仮定できる場合には, 高速な同期の手段になりうる
- ▶ 多数のスレッドが中断/復帰する場合 (例: バリア同期) には有効



スピンロック (ビジーウェイトで排他制御)

- ▶ ビジーウェイトによる排他制御は特にスピンロックと呼ばれて、短いクリティカルセクションを不可分に実行するのに使われる
- ▶ 効用は mutex と同様 (違いはビジーウェイトであること)
- ▶ Pthread API
 - ▶ `#include <pthread.h>`
 - ▶ `pthread_spinlock_t s; /* スピンロックオブジェクト */`
 - ▶ `int pthread_spin_init(&s, attr);`
 - ▶ `int pthread_spin_lock(&s); /* lock */`
 - ▶ `int pthread_spin_trylock(&s);`
 - ▶ `int pthread_spin_unlock(&s); /* unlock */`

Contents

共有メモリと競合状態

同期

排他制御

バリア同期

条件変数

不可分更新命令

休眠待機 vs. ビジーウェイト (繁忙待機?)

同期の実装

(高度な話題) Futex の実装

デッドロック

同期はどのように実装されている？

- ▶ 多くの、同期のためのデータ構造は「排他制御 + 条件変数」で実現できる
 - ▶ 有限バッファ, バリア同期, セマフォ
- ▶ では排他制御や条件変数は？

排他制御の実装

▶ たとえば mutex (pthread_mutex_lock) の概要

```
1 int lock(mutex_t * m) {  
2     while (1) {  
3         if (ロックされていない) {  
4             ロックする;  
5             break;  
6         } else {  
7             ブロックする;  
8         }  
    }
```

▶ 具体化 (3~4行目)

```
1 typedef struct {  
2     int locked;  
3     ...  
4 } mutex_t;
```

```
1 int lock(mutex_t * m) {  
2     while (1) {  
3         if (m->locked == 0) {  
4             m->locked = 1;  
5             break;  
6         } else {  
7             ブロックする;  
8         }  
    }
```

ここでも競合状態!

```
1 int lock(mutex_t * m) {
2   while (1)
3     if (m->locked == 0) {
4       m->locked = 1;
5       break;
6     } else {
7       ブロックする;
8   }}
```

| スレッド A | スレッド B |
|--------------------------|--------------------------|
| 3: if (m->locked == 0) { | 3: if (m->locked == 0) { |
| 4: m->locked = 1; | 4: m->locked = 1; |

- ▶ これを不可分に行うのに (当然!) 排他制御は使えない
- ▶ ⇒ 不可分更新命令

不可分更新命令

```
1 int lock(mutex_t * m) {
2     while (1) {
3         if (test_and_set(&m->locked)) {
4             break;
5         } else {
6             ブロックする;
7         }
8     }
9     int unlock(mutex_t * m) {
10        m->locked = 0;
11        ブロックしているスレッドを起こす;
12    }
```

- ▶ 3~4行目はこれで解決
- ▶ 「ブロックする」をどう実現するか? ⇒ (Linux) futex
- ▶ 今度は lost wake up 問題!

| スレッド A | スレッド B |
|------------------------------------|--------------------|
| 3: if (test_and_set(&m->locked)) { | 11: m->locked = 0; |
| 6: ブロックする; | 12: 寝ているスレッドを起こす; |

- ▶ 「if (*p == v) ブロックする」を不可分に実行

```
1 futex(p, FUTEX_WAIT, v, 0, 0, 0);
```

- ▶ p上でブロックしているスレッドをn個まで起こす

```
1 futex(p, FUTEX_WAKE, n, 0, 0, 0);}
```

- ▶ 注: futexというシステムコールはあるが, C言語から呼べる関数 (glibc wrapper) は定義されていない. 以下を自分で定義 (参照: man futex)

```
1 #include <sys/syscall.h>
2 #include <linux/futex.h>
3 #include <sys/time.h>
4 int futex(int * uaddr, int futex_op, int val,
5           const struct timespec *timeout,
6           int *uaddr2, int val3) {
7     return syscall(SYS_futex, uaddr, futex_op,
8                   val, timeout, uaddr2, val3);
9 }
```

不可分更新命令 + futex による排他制御実装

```
1 int lock(mutex_t * m) {
2     while (!test_and_set(&m->locked)) {
3         /* m->locked == 1 だったらブロック */
4         futex(&m->locked, FUTEX_WAIT, 1, 0, 0, 0);
5     } }
6 int unlock(mutex_t * m) {
7     m->locked = 0;
8     futex(&m->locked, FUTEX_WAKE, 1, 0, 0, 0);
9 }
```

- ▶ 4行目 ; 7行目 ⇒ ブロック; 起こす;
- ▶ 7行目 ; 4行目 ⇒ ブロックしない
- ▶ 扱っている問題は条件変数とほとんど同じ
- ▶ 他のデータ構造も, 「排他制御 + 条件変数」の代わりに, 「不可分更新命令 + futex」で実現することも可能な場合が多い
- ▶ じゃあ, futex が lost wakeup しないことはどう保証する (短い答え: ビジーウェイト)?

Contents

共有メモリと競合状態

同期

排他制御

バリア同期

条件変数

不可分更新命令

休眠待機 vs. ビジーウェイト (繁忙待機?)

同期の実装

(高度な話題) Futex の実装

デッドロック

(再掲) futex

- ▶ `futex(int * ua, FUTEX_WAIT, int v, 0, 0, 0);`
 - ▶ 「if (*ua == v) ブロックする」を不可分^{atomic}に実行
 - ▶ 以降これを `futex_wait(int * ua, int v)` と表記
- ▶ `futex(int * ua, FUTEX_WAKE, int n, 0, 0, 0);`
 - ▶ `ua` 上でブロックしているスレッドを n 個まで起こす
 - ▶ 以降これを `futex_wake(int * ua, int n)` と表記

futex_wait が「不可分」の意味

- ▶ \approx 以下のようなケースを心配する必要がない

| スレッド A (<code>futex_wait(ua, v)</code>) | スレッド B (<code>futex_wake(ua, 1)</code>) |
|---|---|
| A1: <code>if (*ua == v)</code> | B2: <code>*ua = v' ($\neq v$)</code> |
| A2: <code>ブロックする;</code> | |

futex_wait が「不可分」の意味

- ▶ 言い換え: 以下のようなコードで「lost wake up」がない
 - ▶ 初期状態: `*ua == v`
 - ▶ スレッド A:
`A1: futex_wait(ua, v);`
 - ▶ スレッド B:
`B1: *ua = v'; ($\neq v$)`
`B2: futex_wake(ua, 1);`
- ▶ lost wake up がない理由
 - ▶ A1 が B1 より先に実行 \Rightarrow A はブロック; B2 で起こされる
 - ▶ A1 が B1 より後に実行 \Rightarrow A はブロックしない
- ▶ A1 の実行の途中で B1 が実行みたいなケースを考えなくて良いというのが「`futex_wait(ua, v)` が不可分」という「仕様」の意味

futex を用いた mutex の実装 (再掲)

```
1 int lock(mutex_t * m) {
2     while (!test_and_set(&m->locked)) {
3         /* m->locked == 1 だったらブロック */
4         L1: futex_wait(&m->locked, 1);
5     } }
6 int unlock(mutex_t * m) {
7     U1: m->locked = 0;
8     U2: futex_wake(&m->locked, 1);
9 }
```

- ▶ L1 が U1 の前に実行 ⇒ ブロックして U2 で起こされる
- ▶ L1 が U1 の後に実行 ⇒ ブロックしない

それでも残る疑問

- ▶ 「futex_wait(ua, v); が不可分」という「仕様」をどう実装するのか?
- ▶ 安直に, 文字通り,

```
1 futex_wait(int * ua, int v) {  
2     if (*ua == v) ブロックする;  
3 }
```

ではダメなのはすでに見たとおり

| | スレッド A (futex_wait(ua, v)) | スレッド B (futex_wake(ua, 1)) |
|----|----------------------------|-----------------------------|
| A1 | if (*ua == v) { | B2: ua 上で wait しているスレッドを起こす |
| A2 | ブロックする; | |

futex_wait の実装 (0次近似)

- ▶ 一連の処理をロックを使って不可分に. ただし spinlock で (!)

```
1 futex_wait(int * ua, int v) {  
2     spinlock_lock(...);  
3     if (*ua == v) ブロックする;  
4     spinlock_unlock(...);  
5 }
```

```
1 futex_wake(int * ua, int n) {  
2     spinlock_lock(...);  
3     ブロックしているすれっどを最大n 個起こす  
4     spinlock_unlock(...);  
5 }
```

- ▶ もう少し詳細化 (正確に)

「ブロックする」って？

- ▶ ブロックする ≈ そのスレッドを「中断」させ、他の「実行可能」なスレッドに実行を切り替える
- ▶ 「中断」させる際、後で起こしてもらえるよう、スレッドの情報をどこか (後に起こす人がその情報を見つけられる場所) に書き込むことが必要
- ▶ ⇒ futex の第一引数 (ua) に紐付いた構造体が必要. e.g.,

```
1 typedef struct {  
2     spinlock_t s;  
3     waiting_thread_queue_t q;  
4 } futex_t;
```

futex_wait の実装 (1次近似)

```
1  futex_wait(int * ua, int v) {
2      /* ua に紐付いた futex_t 構造体 */
3      futex_t * f = find_futex_struct(ua);
4      spinlock_lock(&f->s);
5      if (*ua == v) { /* ブロックする */
6          pthread_t self = このスレッド;
7          self を run queue から削除;
8          self を f->q に挿入;
9          spinlock_unlock(&f->s);
10         next = run queue から次のスレッドを取り出し;
11         switch_to(next);
12     }
13 }
```

```
1  futex_wake(int * ua, int n) {
2      /* ua に紐付いた futex_t 構造体 */
3      futex_t * f = find_futex_struct(ua);
4      spinlock_lock(&f->s);
5      f->q から n 個まで取り出し, run queue に挿入;
6      spinlock_unlock(&f->s);
7  }
```

Contents

共有メモリと競合状態

同期

排他制御

バリア同期

条件変数

不可分更新命令

休眠待機 vs. ビジーウェイト (繁忙待機?)

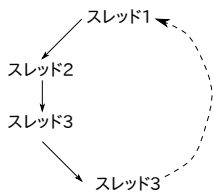
同期の実装

(高度な話題) Futex の実装

デッドロック

デッドロック

- ▶ 日常での意味: 行き詰まり, 膠着状態 (例: 与野党の対立で国会がデッドロックしている)
- ▶ 並行処理における定義: 同期のための待機状態が循環作ってどのスレッド・プロセスも (永遠に) ブロックしたままの状態
- ▶ 症状: プログラムが止まったまま前進しない



- ▶ 「スレッド $P \rightarrow$ スレッド Q 」 \equiv
 P が Q (のアクション) を待っている
- ▶ 例:
 - ▶ Q がロックしている mutex を P もロック (しようと) している
 - ▶ P が wait している条件を Q がやがて満たす

デッドロックの生ずる例(1) — 二つ(以上)の排他制御

スレッド 1

```
1 lock(A);  
2 lock(B);  
3 unlock(B);  
4 unlock(A);
```

スレッド 2

```
1 lock(B);  
2 lock(A);  
3 unlock(A);  
4 unlock(B);
```

| スレッド 1 | スレッド 2 |
|-------------|-------------|
| 1: lock(A); | 1: lock(B); |

スレッド1

スレッド2

デッドロックの生ずる例(1) — 二つ(以上)の排他制御

スレッド 1

```
1 lock(A);  
2 lock(B);  
3 unlock(B);  
4 unlock(A);
```

スレッド 2

```
1 lock(B);  
2 lock(A);  
3 unlock(A);  
4 unlock(B);
```

| | スレッド 1 | スレッド 2 | |
|----|----------|-------------|------|
| 1: | lock(A); | 1: lock(B); | |
| 2: | lock(B); | | ブロック |



デッドロックの生ずる例(1) — 二つ(以上)の排他制御

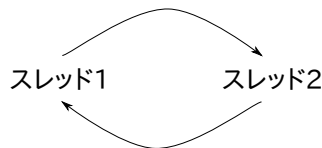
スレッド 1

```
1 lock(A);  
2 lock(B);  
3 unlock(B);  
4 unlock(A);
```

スレッド 2

```
1 lock(B);  
2 lock(A);  
3 unlock(A);  
4 unlock(B);
```

| | スレッド 1 | スレッド 2 | |
|----|----------|-------------|--------------|
| 1: | lock(A); | 1: lock(B); | |
| 2: | lock(B); | 2: lock(A); | ブロック ブロック |



デッドロックの生ずる例(2) — 送受信バッファ

スレッド1

```
1 while (...) {  
2   send(...) or recv(...);  
3 }
```

スレッド2

```
1 while (...) {  
2   send(...) or recv(...);  
3 }
```



- ▶ バッファが両方向満杯のときにお互いが send をしようとする状況
- ▶ 注: バッファが両方空のときにお互いが recv をしようとする状況も同様
- ▶ デッドロックしない方法
 - ▶ recv できる (空でない) ときは recv する
 - ▶ send できる (空でない) ときは send する

デッドロックの生ずる例(2) — 送受信バッファ

スレッド1

```
1 while (...) {  
2   send(...) or recv(...);  
3 }
```

スレッド2

```
1 while (...) {  
2   send(...) or recv(...);  
3 }
```



- ▶ バッファが両方向満杯のときにお互いが send をしようとする状況
- ▶ 注: バッファが両方空のときにお互いが recv をしようとする状況も同様
- ▶ デッドロックしない方法
 - ▶ recv できる (空でない) ときは recv する
 - ▶ send できる (空でない) ときは send する

デッドロックの生ずる例(2) — 送受信バッファ

スレッド1

```
1 while (...) {  
2   send(...) or recv(...);  
3 }
```

スレッド2

```
1 while (...) {  
2   send(...) or recv(...);  
3 }
```



- ▶ バッファが両方向満杯のときにお互いが send をしようとする状況
- ▶ 注: バッファが両方空のときにお互いが recv をしようとする状況も同様
- ▶ デッドロックしない方法
 - ▶ recv できる (空でない) ときは recv する
 - ▶ send できる (空でない) ときは send する

デッドロックの生ずる例(2) — 送受信バッファ

スレッド1

```
1 while (...) {  
2   send(...) or recv(...);  
3 }
```

スレッド2

```
1 while (...) {  
2   send(...) or recv(...);  
3 }
```



- ▶ バッファが両方向満杯のときにお互いが send をしようとする状況
- ▶ 注: バッファが両方空のときにお互いが recv をしようとする状況も同様
- ▶ デッドロックしない方法
 - ▶ recv できる (空でない) ときは recv する
 - ▶ send できる (空でない) ときは send する

デッドロックの生ずる例(2) — 送受信バッファ

スレッド1

```
1 while (...) {  
2   send(...) or recv(...);  
3 }
```

スレッド2

```
1 while (...) {  
2   send(...) or recv(...);  
3 }
```



- ▶ バッファが両方向満杯のときにお互いが send をしようとする状況
- ▶ 注: バッファが両方空のときにお互いが recv をしようとする状況も同様
- ▶ デッドロックしない方法
 - ▶ recv できる (空でない) ときは recv する
 - ▶ send できる (空でない) ときは send する

デッドロックの回避

- ▶ 一般的な予防は困難
- ▶ 問題の始まりは、「誰かを待たせながら誰かを待つ」こと
- ▶ 複数の排他制御によって生ずるデッドロックの防ぎ方
 1. mutex をひとつだけにする (giant lock)
 2. 一つのスレッドは、二つの mutex を同時にロックしない (ある mutex を lock している状態で別の mutex の lock を呼ばない)
 3. すべての mutex に順序をつけ、すべてのスレッドはその全順序の順でしか lock をしない
 4. 不可分更新をするのに排他制御を使わない
 - 4.1 不可分更新命令 (一つのアドレスに対する更新)
 - 4.2 トランザクショナルメモリ (時間があれば)

まとめ: 並行プログラミングは大変

- ▶ 競合状態 (mutex, 条件変数)
- ▶ ビジーウェイト
- ▶ デッドロック
- ▶ 1 アドレスに対する不可分更新 (compare and swap)
- ▶ 2 アドレス以上は mutex またはトランザクショナルメモリ