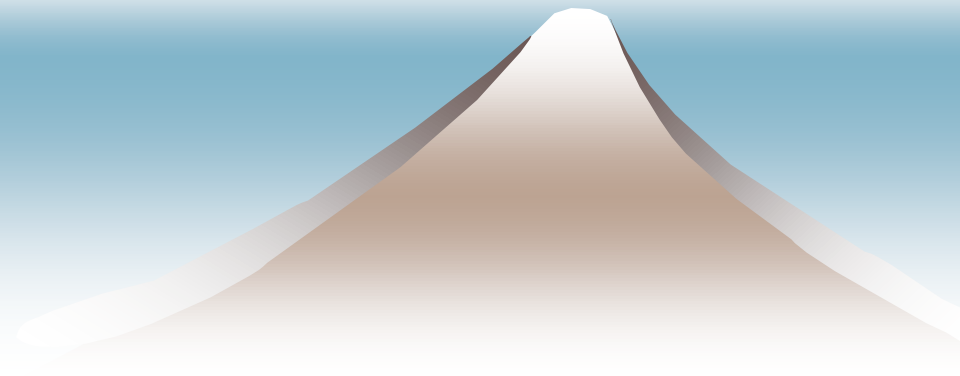
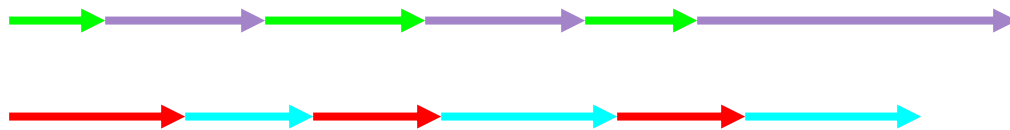


# 並行プログラムと同期



# スレッドとプロセス

- ◆ CPUの数だけ同時に実行
- ◆ CPU数を越えるスレッド・プロセスはOSによって交互に実行
- ◆ 2CPUの場合の図:



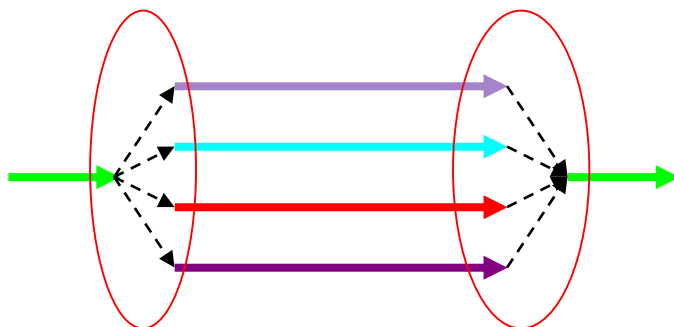
\_\_\_\_\_→  $t$

# スレッド・プロセスの利用目的 性能と記述性の向上

- ◆ **並列処理:** マルチプロセッサ(複数CPUを持つ計算機), マルチコアプロセッサでの性能向上
- ◆ **I/O遅延隠蔽:** I/Oによってブロックするスレッドを複数実行してCPU利用率を向上
  - サーバのリクエスト処理
- ◆ **記述の簡潔化:** 論理的にほぼ独立なタスクを簡便に記述
  - ブラウザの描画スレッドとダウンロードスレッド

# スレッド・プロセス間の協調

- ◆ 問題データや計算結果の受け渡しのための通信



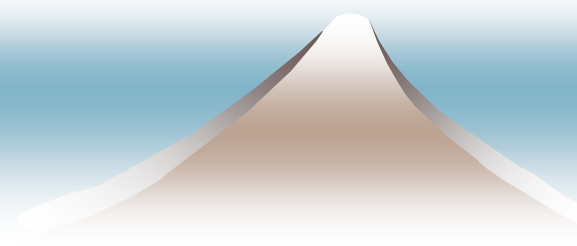
- ◆ 提供されている通信方法

- プロセス間: ファイル, パイプ, ソケット, etc.
- スレッド間: 上記 + 共有メモリ

以降の主なテーマ

# 共有メモリ

- ◆ 一つのプロセスの中の複数のスレッド間でメモリが共有される
- ◆ 仕組み
  - CPU (ハードウェア)が提供している共有メモリ
    - あるアドレスに書き込んだ結果はどのCPUからでも読みだせる
  - OSや言語処理系は(ほとんど)何もしていない



# 復習を兼ねて

- ◆ 

```
int main() {  
    int a[1]; pthread_t tid[2];  
    pthread_create(&tid[0], 0, f, a);  
    pthread_create(&tid[1], 0, g, a);  
    pthread_join(tid[0], 0);  
    pthread_join(tid[1], 0);  
}
```
- ◆ として二つのスレッドを作る. 次のうちスレッド間で通信してる(同じアドレスを触ってる)のはどれ?

◆ (あ)

```
int x;
f (void* a) {
    x = 100;
}
g (void* a) {
    ... = x ;
}
```

◆ (い)

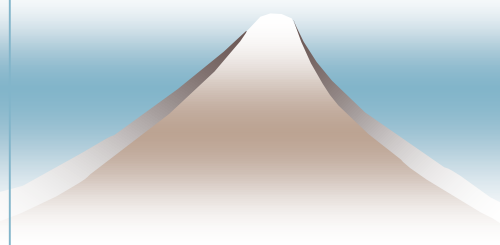
```
f (void* a) {
    int x;
    x = 100;
}
g (void* a) {
    int x;
    ... = x ;
}
```

◆ (う)

```
int x;
f (void * a) {
    int * x = a;
    *x = 100;
}
g (void * a) {
    int * x = a;
    ... = *x ;
}
```

◆ (え)

```
int x;
f (void * a) {
    int * x = malloc(sizeof(int));
    *x = 100;
}
g (void* a) {
    int * x = malloc(sizeof(int));
    ... = *x ;
}
```



# 簡単な例題： マンデルブロ集合の面積

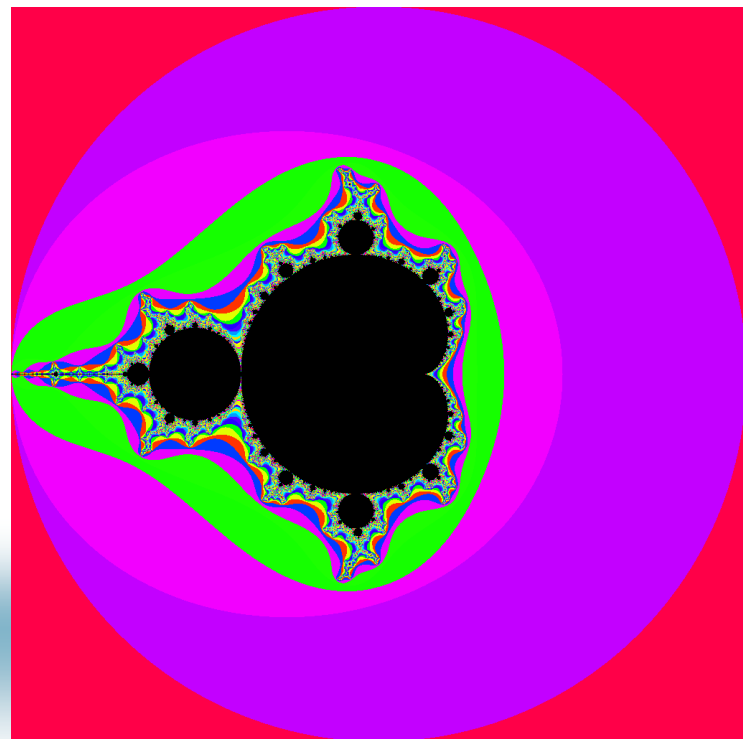
- ◆ マンデルブロ集合: 複素平面上で複素数列:

$$z_1 = 0;$$

$$z_{n+1} = z_n * z_n + c$$

が無限大に発散しないような $c$ の集合(黒部分)

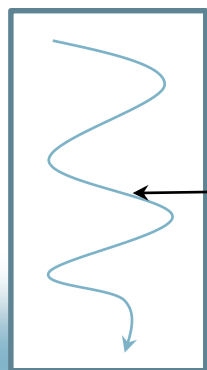
- ◆ 問題: その面積の近似値



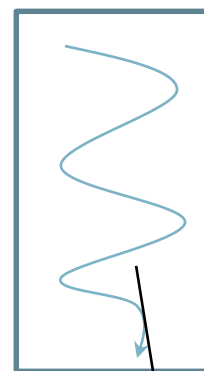


- ◆ 複数の計算スレッドを生成し、各々別々の領域内の面積を計算
- ◆ 結果がそろったらそれらを足し算

足し算スレッド



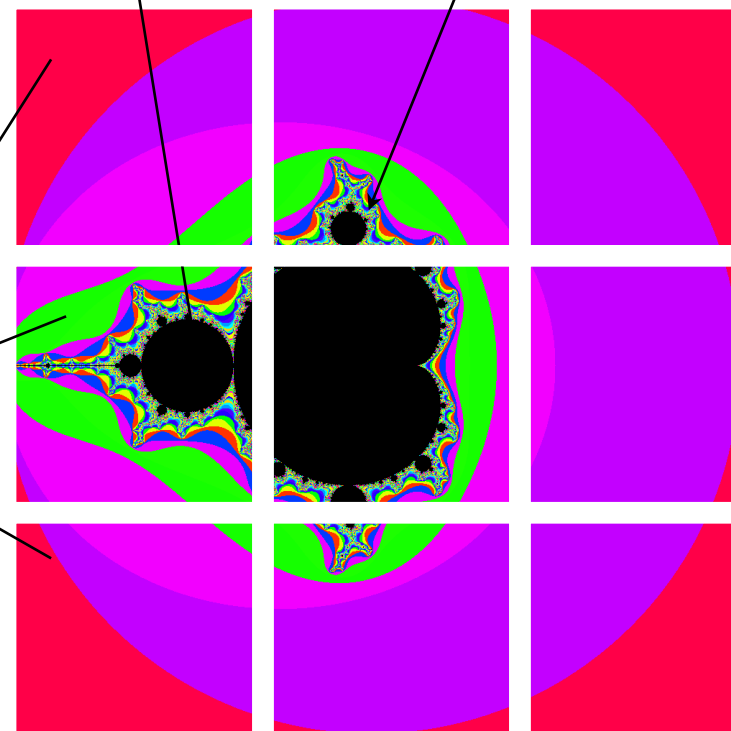
+



IN[1][0] = 48;

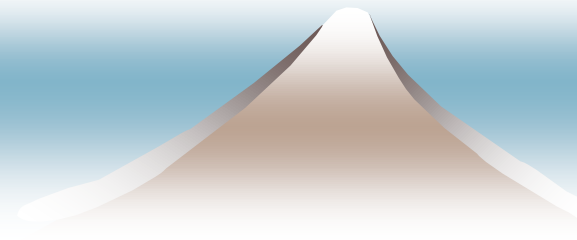


IN[0][1] = 13;



# 競合状態(Race Condition)の概念

- ◆ 並行して実行されているスレッド・プロセス間の「危ない」相互作用
  - 共有メモリの更新と読み出し
  - 共有されたファイルの更新と読み書き
- ◆ 共有メモリ(ファイル)を用いてスレッド(プロセス)の間で「正しく」通信をすることは自明ではない
- ◆ 以降は共有メモリを介して通信をするスレッド間の協調に話を絞る

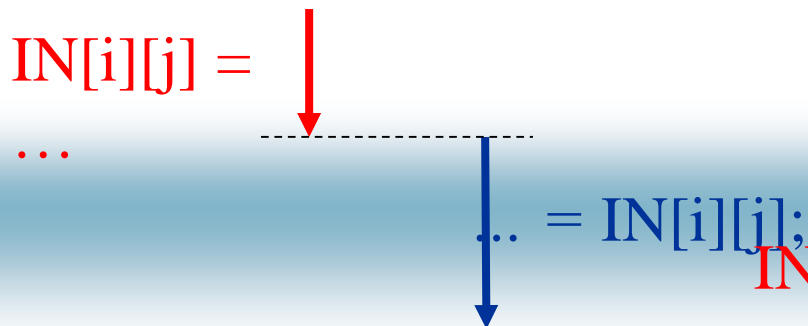


# 例1: 生産者消費者同期

- ◆ 足し算スレッドが「正しく」計算された結果を読むには?
- ◆ 計算スレッド:  $\{ \dots \text{IN}[i][j] = \dots \}$   
足し算スレッド:  $\{ \dots = \text{IN}[i][j]; \dots \}$

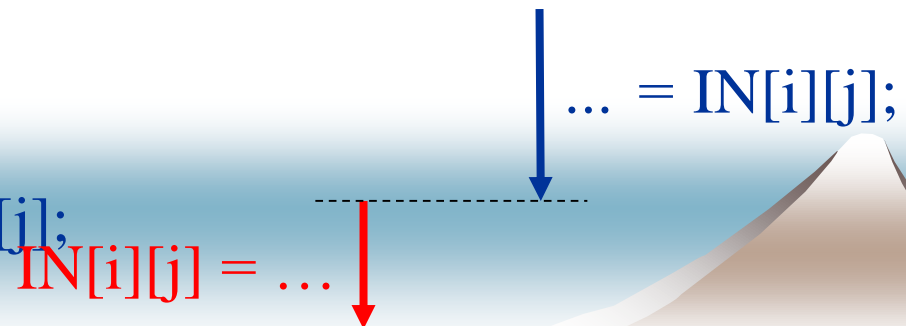
幸運なスケジュール

A B



不運なスケジュール

A B



# 何が間違っていたか？

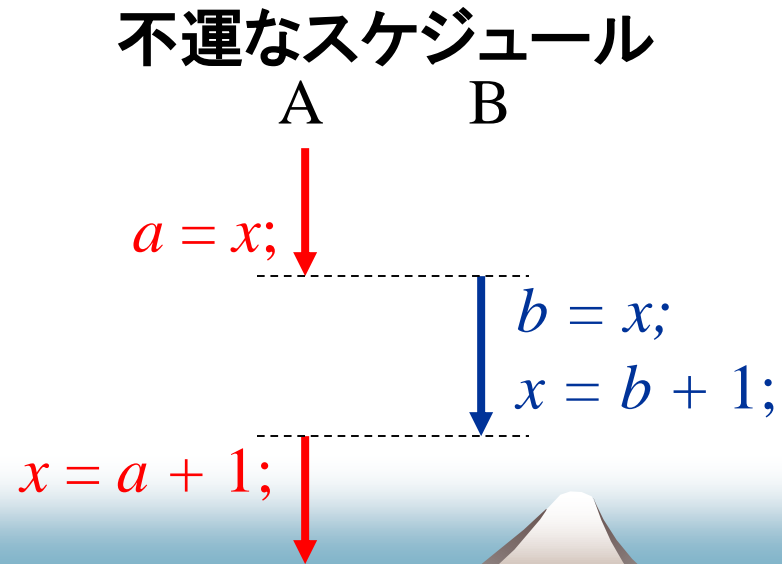
- ◆ イベント間の**順序(依存関係)**の崩れ
  - $IN[i][j]$ への代入(書き込み)は $IN[i][j]$ からの読み出しに先行しなければならない

$IN[i][j] = \dots$   
↓  
 $\dots = IN[i][j]$   
↓  
 $\dots;$

$\dots = IN[i][j] \dots;$   
↓  
 $IN[i][j] = \dots$   
↓

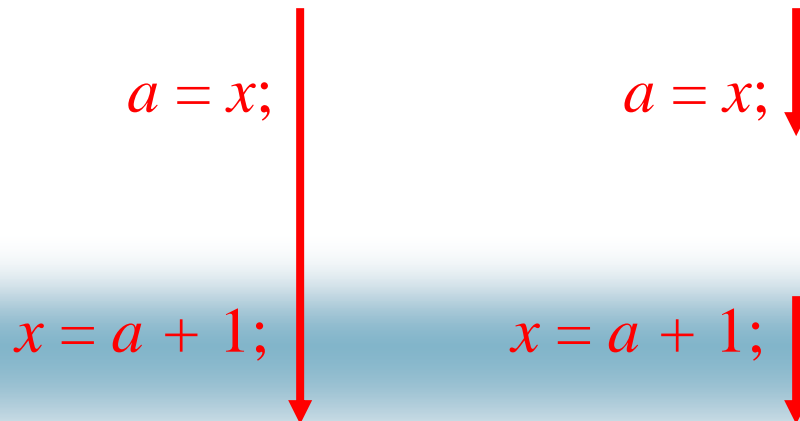
## 例2: 排他制御

- ◆ スレッドA, Bが変数 $x$ をincrement
- ◆ `int x; /* グローバル変数 */`  
スレッドA: { ...  $x++$ ; ... }  
スレッドB: { ...  $x++$ ; ... }



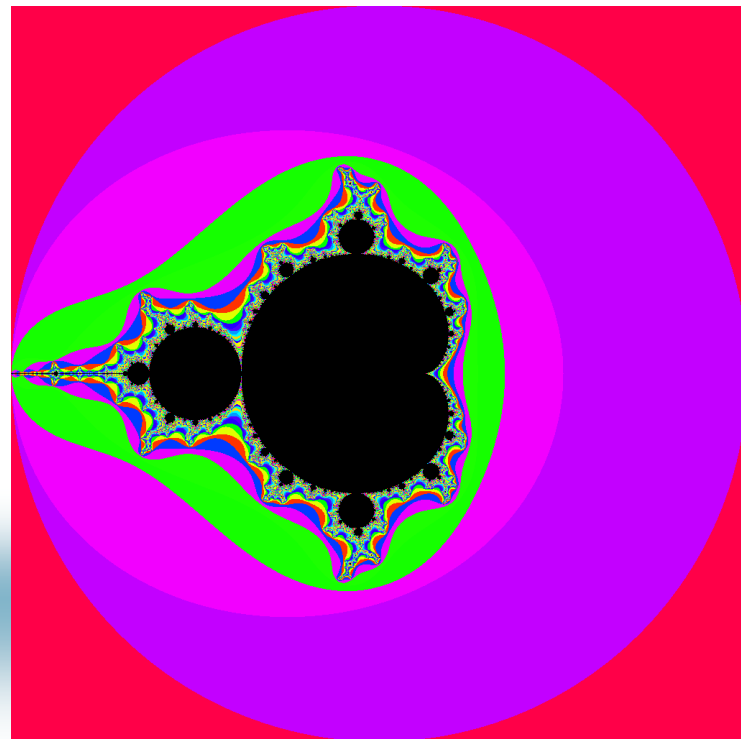
# 何が間違っていたか？

- ◆ 一連の処理の**不可分性, 原子性**  
(**atomicity**)の崩れ
  - 1命令の不可分性は(通常)ハードウェアが保証している
  - 2命令以上の列の不可分性は保証しない



# 例題で排他制御が 必要となる場合

- ◆ どの計算スレッドがどの点を計算するかの割り当てをどうやるか?
- ◆ 単純には: 縦 or 横方向にスレッド数で分割
- ◆ **動的な割り当て:** 「まだ計算されていない領域」を保持して少しずつ動的に割り当て

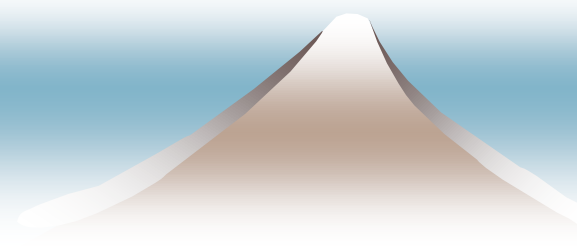


# 競合状態: 別の例

## ◆ 二つの変数の入れ替え

- ```
int A, B;  
void swap() {  
    int x = A;  
    int y = B;  
    A = y;  
    B = x;  
}
```

## ◆ これを二つのスレッドが行うと?



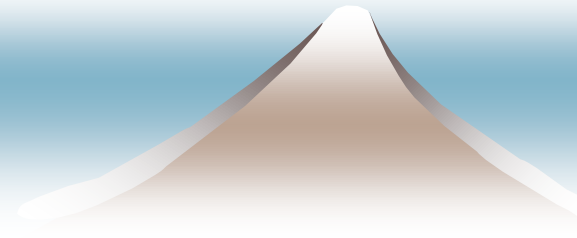


## さらなる例...

- ◆ `std::vector<int> v;`  
`void push_vals() {`  
    `for (i = 0; i < 1000000; i++) v.push_back(i);`  
`}`
- ◆ “push\_back” の中で(たぶん)データを書き換えている
- ◆ これをやっていいかどうかは “push\_back” 関数の中身に依存(ライブラリの仕様として明示されるべきもの)

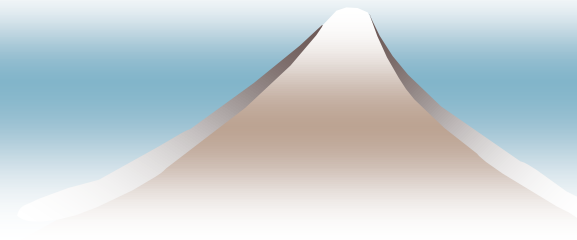
# 実践的なメッセージ

- ◆ 複数のスレッドが利用している場所への読み書きで、少なくとも一方が書き込みの時は「要注意」
  - 必要な順序が保たれているか？
  - 不可分性が崩れていないか



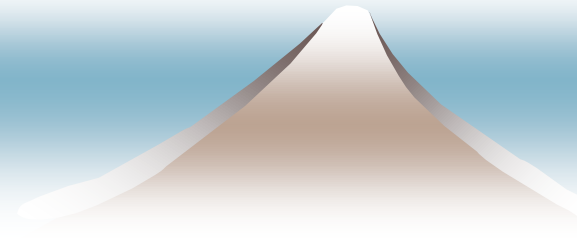
# いくつかの用語

- ◆ 競合状態(race condition)
  - スレッド間の「危ない相互作用」が起きる状態
  - $\approx$  同じ領域を複数スレッドが並行にアクセスし、少なくともどちらかは書いている
- ◆ クリティカルセクション(critical section)
  - 競合が発生している(時間的)区間



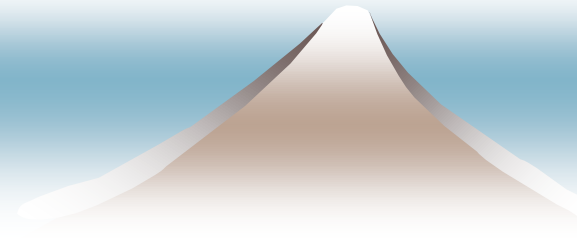
# 同期

- ◆ 同期: 競合状態を解消するための「タイミングの制御」
- ◆ 勝手なスケジューリングを許さないための「制約」
  - 不可分性の保証
  - 順序制約の保証
  - その他様々な「実行可能条件」の保証



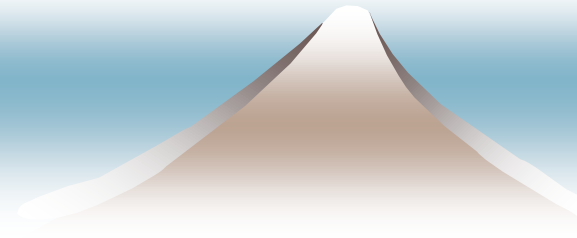
# 以降の流れ

- ◆ 同期のための代表的API
  - 有用であると考えられているメタファ
- ◆ 同期プリミティブの実現
  - ハードウェア(CPU)の提供するプリミティブ
  - 排他制御の実現
  - 排他制御無しの不可分更新
- ◆ デッドロック



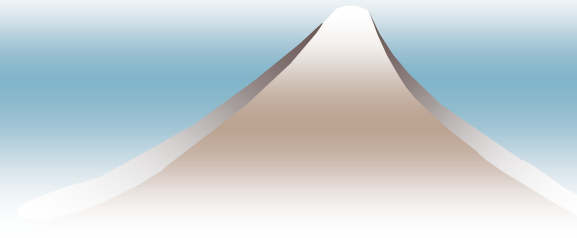
# 代表的API

- ◆ Win32スレッド
- ◆ Pthread (Unix系OSの共通API)
- ◆ Java, Python, などの各言語
- ◆ どれも概念は共通, APIの語彙は類似
  - 以降はWin32, PthreadのAPIを主に説明する



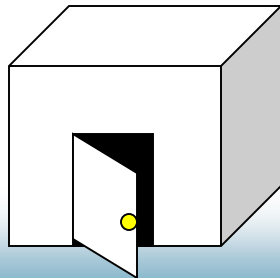
# よく使われる同期プリミティブ

- ◆ 排他制御(mutual exclusion): mutex
- ◆ イベント(event)
- ◆ セマフォ(semaphore)
- ◆ 条件変数(condition variable)

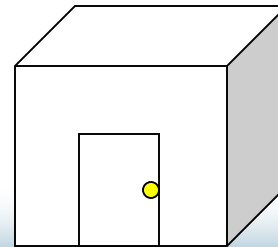


# 1. Mutex (排他制御)

- ◆ メタファ: 一人しか入れない部屋
- ◆ 操作
  - **lock**: 部屋に入る. すでに入っているならばそいつが出るまで待つ
  - **unlock**: 部屋から出る



unlockされた状態

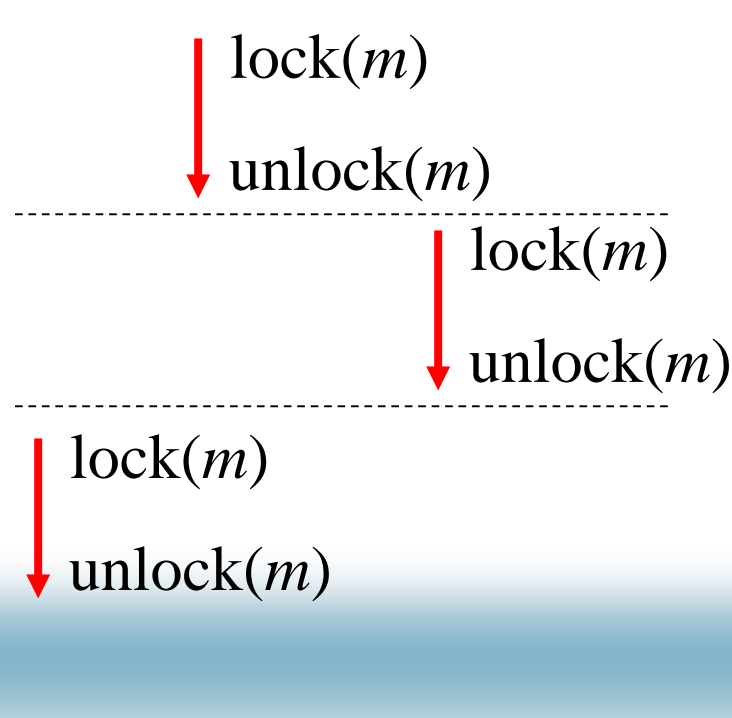


lockされた状態



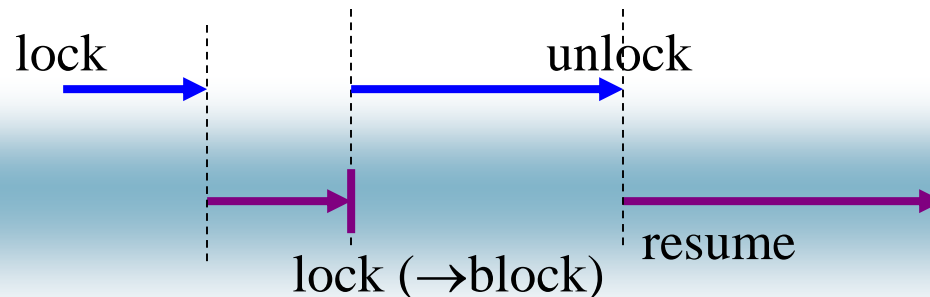
# Mutexによって保証されること

- ◆ ひとつのロックが「ロックされた状態」はinterleaveしない



# Mutex実際の動作(擬似コード)

- ◆  $\text{lock}(m)$  {  
    if ( $m$  is not locked)  $\text{mark\_as\_locked}(m)$ ;  
    else  $\text{block\_until\_unlocked}(m)$ ; }
- ◆  $\text{unlock}(m)$  {  
     $\text{mark\_as\_unlocked}(m)$ ;  
     $\text{resume\_if\_one\_is\_sleeping}(m)$ ; }



# Mutex利用場面

## ◆ 一連の操作

多くの場合, 他のスレッドと共有しているデータの読み出しから更新まで

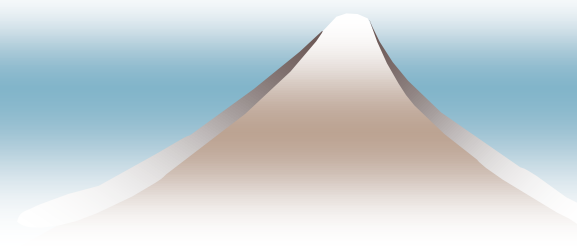
を不可分(途中で邪魔されず)に行いたい

## ◆ 例:

- `lock(m);`  
  `x ++;`  
  `unlock(m);`

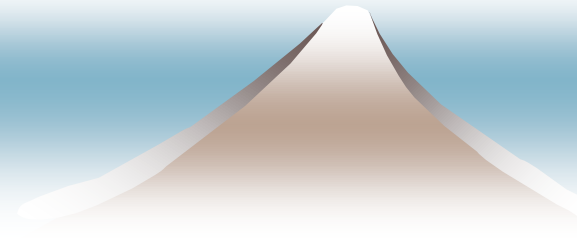
# Mutex API : Pthread

- ◆ `#include <pthread.h>`
- ◆ `pthread_mutex_t m;`
- ◆ `pthread_mutex_init(&m, NULL);`
- ◆ `pthread_mutex_lock(&m);`     `/* lock */`
- ◆ `pthread_mutex_unlock(&m);` `/* unlock */`
- ◆ `pthread_mutex_trylock(&m);`



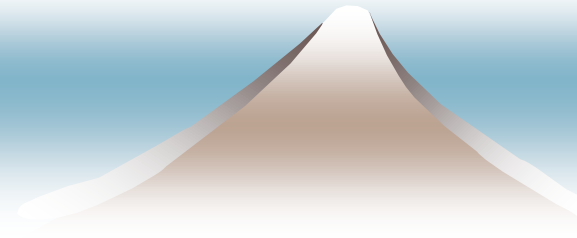
# Mutex API : Win32

- ◆ HANDLE  $m$  = CreateMutex(...);
- ◆ /\* lock \*/
  - WaitForSingleObject( $m$ );
  - WaitForMultipleObjects(...);
- ◆ /\* unlock \*/
  - ReleaseMutex( $m$ );



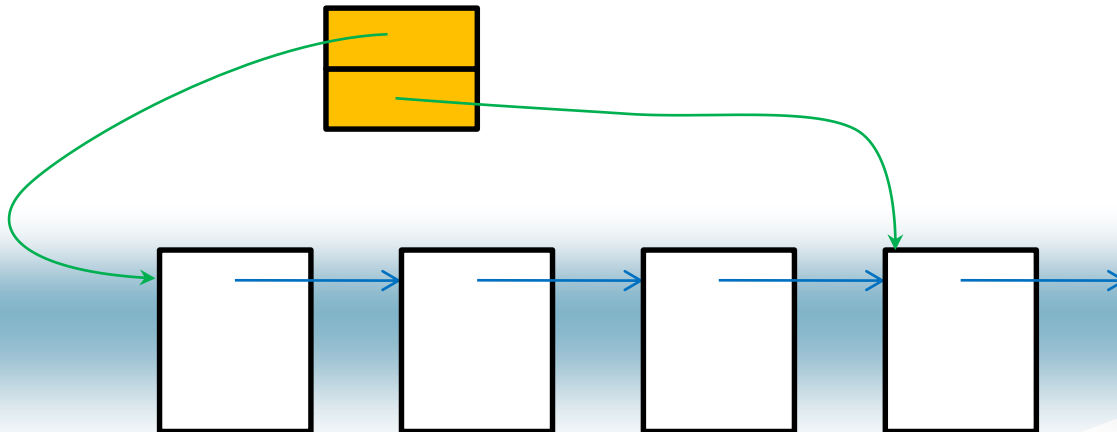
# 例題:FIFOキュー

- ◆  $\text{enq}(q, x)$ :
  - $x$ をFIFOキュー $q$ に入れる
- ◆  $x = \text{deq}(q)$ :
  - FIFOキュー $q$ から先頭要素を取り出して返す
  - 空ならEMPTYを返す



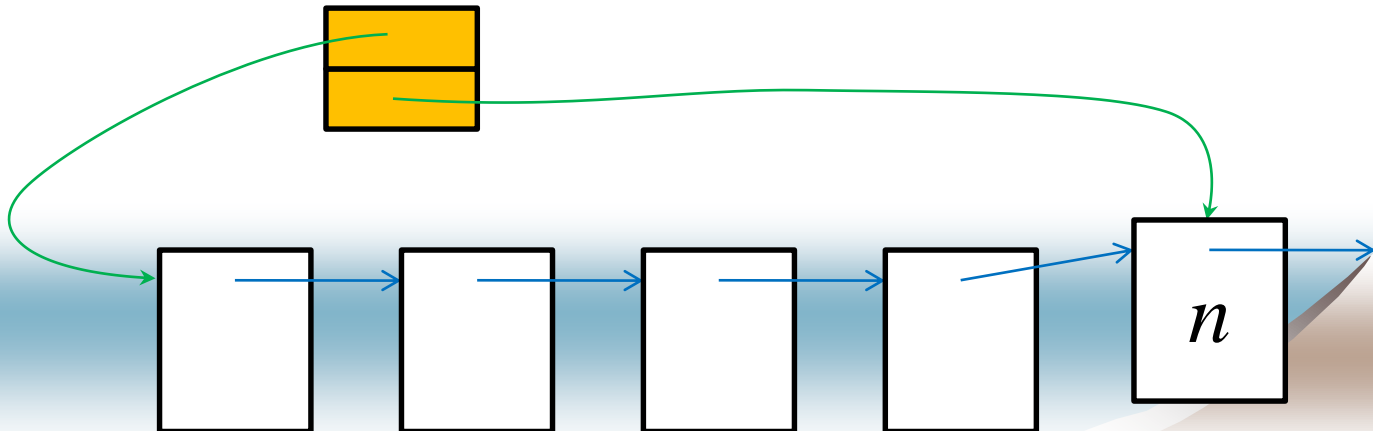
# 逐次コード (データ定義)

- ◆ typedef struct node {  
    struct node \* next; int val; } \* node\_t;
- ◆ typedef struct queue {  
    node\_t head; node\_t tail; } \* queue\_t;



# 逐次コード (enq)

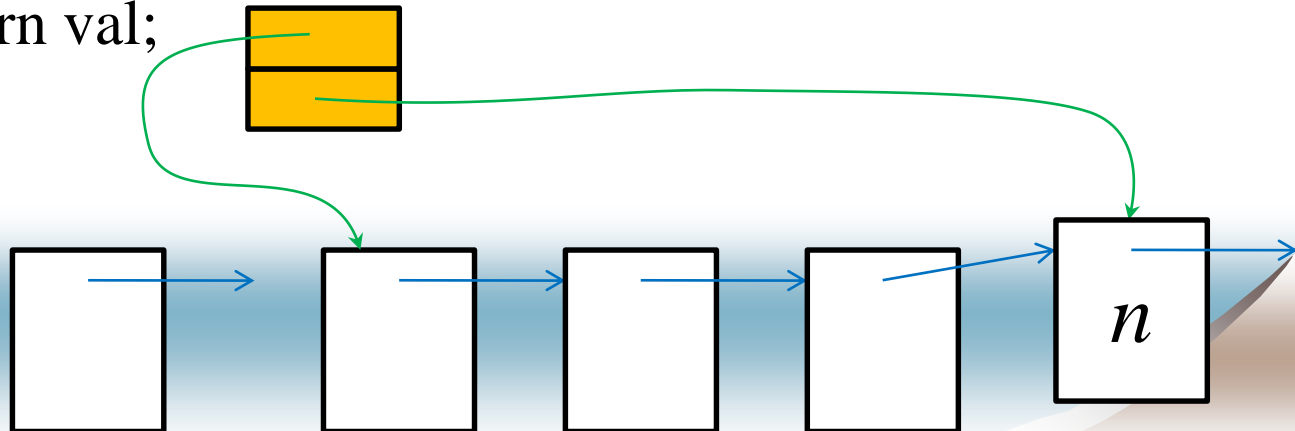
```
◆ void enq(queue_t q, int x) {  
    queue_node_t n = (node_t)malloc(sizeof(struct node));  
    n->next = NULL;  n->val = x;  
    if (q->tail) { q->tail->next = n; }  
    else { q->head = n; }  
    q->tail = n;  
}
```





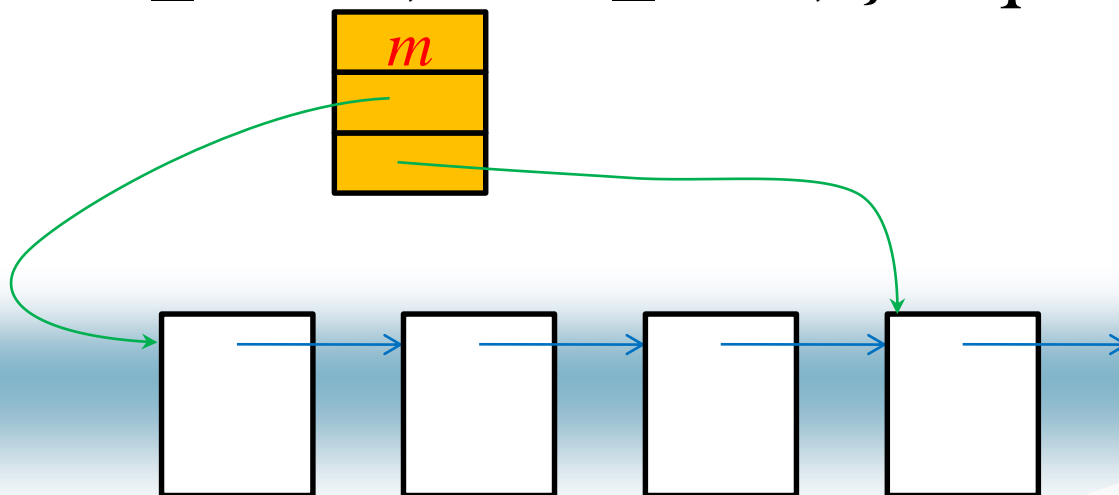
# 逐次コード (deq)

```
◆ int deq(queue_t q) {  
    int val;  
    queue_node_t h = q->head;  
    if (h) { /* 空じゃない場合 */  
        q->head = h->next;  
        if (h->next == NULL) q->tail = NULL;  
        val = h->val;  
    } else { val = EMPTY; /* 空の場合 */ }  
    return val;  
}
```



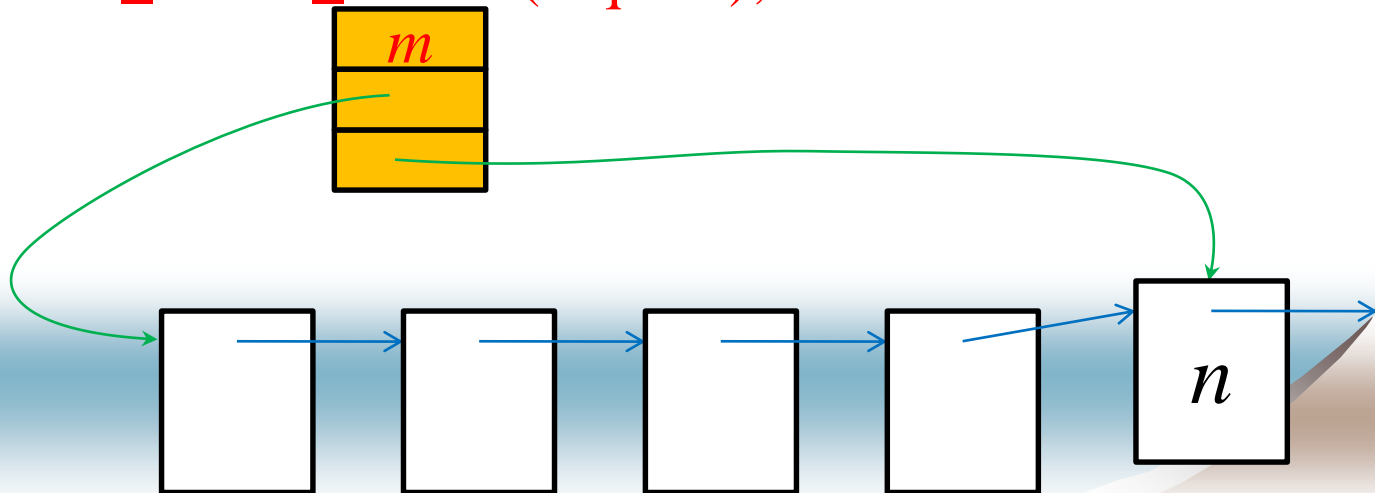
# Mutexによる排他制御 (データ定義)

- ◆ typedef struct node {  
    struct node \* **next**; int val; } \* node\_t;
- ◆ typedef struct **queue** {  
    **pthread\_mutex\_t** m;  
    node\_t **head**; node\_t **tail**; } \* queue\_t;



# Mutexによる排他制御 (enq)

```
◆ void enq(queue_t q, int x) {  
    queue_node_t n = (node_t)malloc(sizeof(struct node));  
    n->next = NULL;  n->val = x;  
    pthread_mutex_lock(&q->m);  
    if (q->tail) { q->tail->next = n; }  
    else { q->head = n; }  
    q->tail = n;  
    pthread_mutex_unlock(&q->m);  
}
```

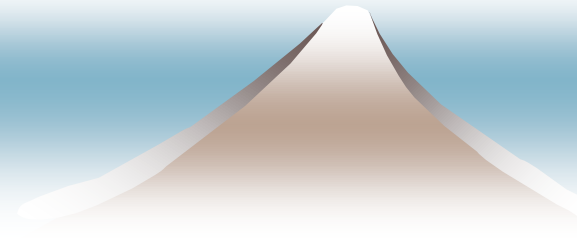


# Mutexによる排他制御 (deq)

```
◆ int deq(queue_t q) {  
    int val;  
    pthread_mutex_lock(&q->m);  
    queue_node_t h = q->head;  
    if (h) { /* 空じゃない場合 */  
        q->head = h->next;  
        if (h->next == NULL) q->tail = NULL;  
        val = h->val;  
    } else { val = EMPTY; /* 空の場合 */ }  
    pthread_mutex_unlock(&q->m);  
    return val;  
}
```

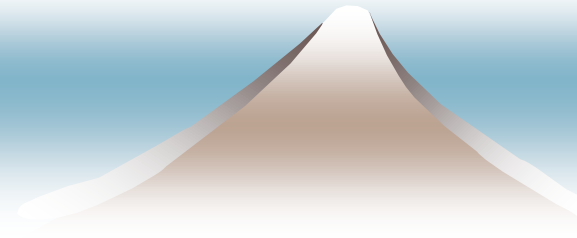
## 2. イベント

- ◆ メタファ：郵便(メッセージ)
- ◆ 操作
  - Set 郵便を届ける(お知らせ)
  - Wait 郵便が来るのを待って取り除く



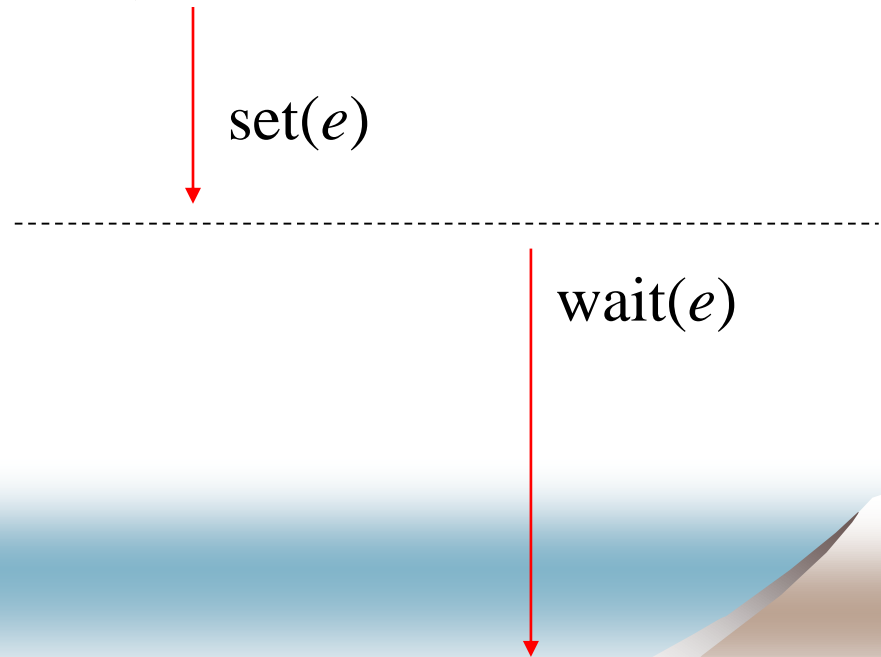
# Win32 API

- ◆ HANDLE  $e$  = CreateEvent(...);
- ◆ Set: SetEvent( $e$ );
- ◆ Wait:
  - WaitForSingleObject( $e$ , ...);
  - WaitForMultipleObjects(...);



# Eventによって保証されること

- ◆ あるひとつのeventオブジェクトに対するsetは, waitに先行する(それぞれ一回のみ発行されていると仮定)



# イベントを用いた生産者消費者同期

- ◆ 初期状態:  $e = \text{CreateEvent}(\dots);$

スレッドA:

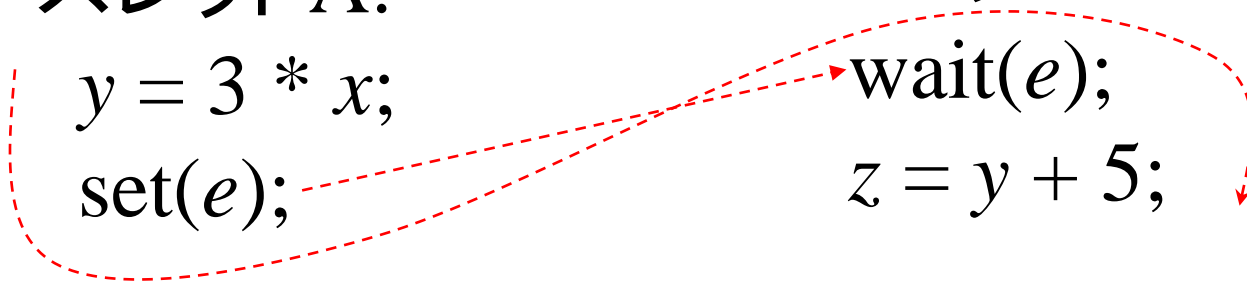
$y = 3 * x;$

$\text{set}(e);$

スレッドB:

$\text{wait}(e);$

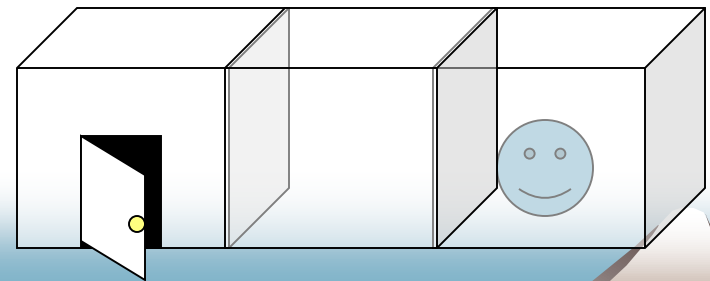
$z = y + 5;$





# 3. Semaphore

- ◆ メタファ: 銀行のATMの列(複数の機械. 列が1個)
- ◆ 操作
  - **signal**: ATM使用終了
  - **wait**: どれかの機械があくまで待って, あいたら使う

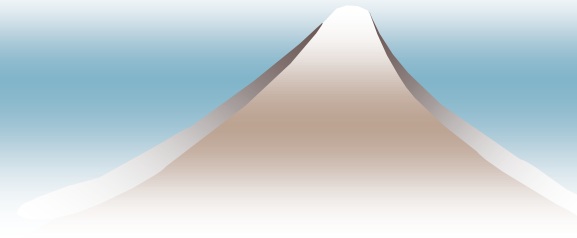


# Win32 API

- ◆ HANDLE  $s$  = CreateSemaphore(...,  $m$ ,  $n$ , ...);
  - $n$  : 機械の数
  - $m$  : 初期状態の空き機械の数
- ◆ Signal : ReleaseSemaphore(...);
- ◆ Wait :
  - WaitForSingleObject( $s$ , ...);
  - WaitForMultipleObjects(...);

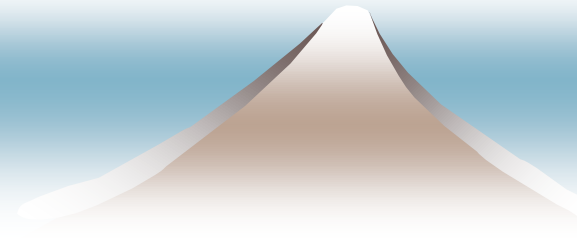
# Semaphore API : POSIX

- ◆ `sem_t * s = sem_open(..., O_CREAT);`
- ◆ `sem_init(s, 0, n);`
- ◆ `sem_post(s);`
- ◆ `sem_wait(s);`



## 4. 条件変数

- ◆「ある条件が成立するまで待つ」ための汎用的な同期機構

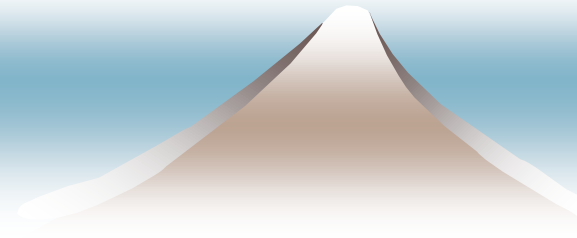


# 条件変数の典型的利用場面

- ◆ ある条件 (例:  $C(x, y, z)$ ) が成り立つまで待つ;  
 $x, y, z$  などを更新;
  - 例1:  
queueが空でなくなるのを待つ;  
queueから要素を一個取り出す;
  - 例2:  
 $x < y$  が成り立つまで待つ;  
 $y$  から10を引き,  $x$  に10を足す;

# 例題:FIFOキュー

- ◆  $\text{enq}(q, x)$ :
  - $x$ をFIFOキュー $q$ に入れる
- ◆  $x = \text{deq}(q)$ :
  - FIFOキュー $q$ から先頭要素を取り出して返す
  - ただし空の場合, リターンせずにenqされるまで待つ



# 何が問題か

```
◆ int deq(queue_t q) {  
    int val;  
    pthread_mutex_lock(&q->m);  
    queue_node_t h = q->head;  
    if (h) { /* 空じゃない場合 */  
        q->head = h->next;  
        if (h->next == NULL) q->tail = NULL;  
        val = h->val;  
    } else { val = EMPTY; /* 空の場合 */ }  
    pthread_mutex_unlock(&q->m);  
    return val;  
}
```

つまりここでどうするかが問題  
このために条件変数がある

# 概念的な解

```
◆ int deq(queue_t q) {  
    int val;  
    pthread_mutex_lock(&q->m);  
    while (1) {  
        queue_node_t h = q->head;  
        if (h) { /* 空じゃない場合 */  
            q->head = h->next;  
            if (h->next == NULL) q->tail = NULL;  
            val = h->val; break;  
        } else { お休みなさい, で, いい感じになったら起こして; }  
    }  
    pthread_mutex_unlock(&q->m);  
    return val;  
}
```

これがまさに条件変数

{ お休みなさい, で, いい感じになったら起こして; }

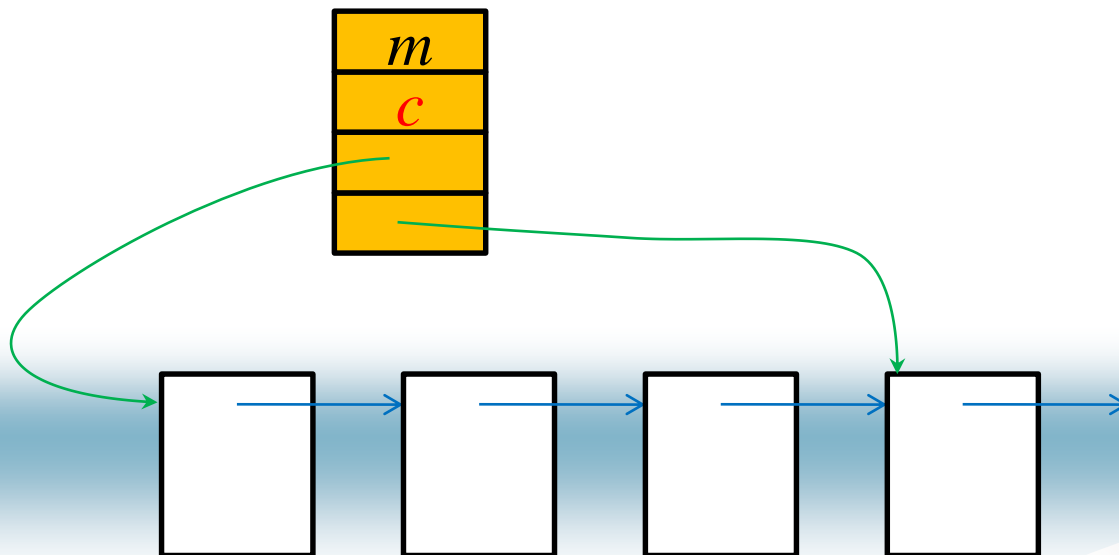


# Pthreads API

- ◆ 基本的にmutexとセットで使う
- ◆ `pthread_cond_t c; pthread_mutex_t m;`
- ◆ `pthread_cond_init(&c);`
- ◆ `pthread_cond_wait(&c, &m);`
  - *m*をunlockし, *c*の上でブロックする. 起こされたらまた*m*をlockする
- ◆ `pthread_cond_broadcast(&c);`
  - *c*でブロックしているスレッド全員を起こす

# 条件変数を用いたFIFOキュー (データ定義)

- ◆ typedef struct **queue** {  
    pthread\_mutex\_t m;  
    **pthread\_cond\_t** c;  
    node\_t **head**; node\_t **tail**; } \* queue\_t;



# 条件変数を用いたFIFOキュー (deq)

```
◆ int deq(queue_t q) {  
    int val;  
    pthread_mutex_lock(&q->m);  
    while (1) {  
        queue_node_t h = q->head;  
        if (h) { /* 空じゃない場合 */  
            q->head = h->next;  
            if (h->next == NULL) q->tail = NULL;  
            val = h->val; break;  
        } else { pthread_cond_wait(&q->c, &q->m); }  
    }  
    pthread_mutex_unlock(&q->m);  
    return val;  
}
```

mをunlockし, c上で寝る  
おこされたらまたmをlock

# 条件変数を用いたFIFOキュー (enq)

```
◆ void enq(queue_t q, int x) {  
    queue_node_t n = (node_t)malloc(sizeof(struct node));  
    n->next = NULL;  n->val = x;  
    pthread_mutex_lock(&q->m);  
    if (q->tail) { q->tail->next = n; }  
    else { q->head = n; pthread_cond_broadcast(&q->c); }  
    q->tail = n;  
    pthread_mutex_unlock(&q->m);  
}
```

c上で寝ているスレッドを全部起こす

# 条件変数の使い方テンプレート

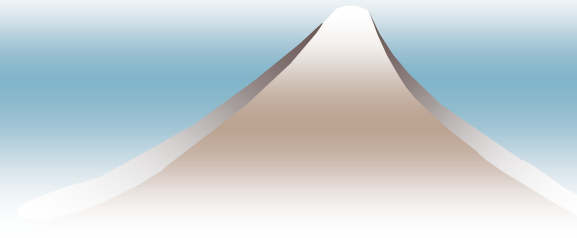
- ◆  $C(x, y, z)$ が成立するまで待ちたいとする
- ◆ `lock(m);`  
    `while (!C(x, y, z)) {`  
        `cond_wait(c, m);`  
    `}`  
  
    `/* C(x, y, z)が成り立ったら実行したいこと */`  
     $x, y, z$ などをいじる;  
  
    必要なら, 他の待ってる人たちを起こす;  
    `unlock(m);`

# 間違い探し(1)

```
◆ int deq(queue_t q) {  
    int val;  
    pthread_mutex_lock(&q->m);  
    while (1) {  
        queue_node_t h = q->head;  
        if (h) { /* 空じゃない場合 */  
            q->head = h->next;  
            if (h->next == NULL) q->tail = NULL;  
            val = h->val; break;  
        } else { /* 何もしない */ }  
    }  
    pthread_mutex_unlock(&q->m);  
    return val;  
}
```

# 答

- ◆ mutex ( $m$ )を解放しないままループしても無限ループに陥るだけ



## 間違い探し(2)

```
◆ int deq(queue_t q) {  
    int val;  
    while (1) {  
        pthread_mutex_lock(&q->m);  
        queue_node_t h = q->head;  
        if (h) { /* 空じゃない場合 */  
            q->head = h->next;  
            if (h->next == NULL) q->tail = NULL;  
            val = h->val; break;  
        } else { pthread_mutex_unlock(&q->m); }  
    }  
    pthread_mutex_unlock(&q->m);  
    return val;  
}
```

いったんlockを解放してあげる



# 答

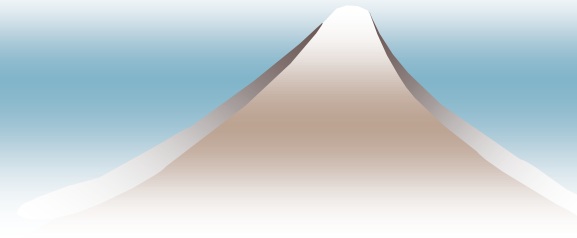
- ◆ この書き方では、スレッドBが同期不成立の間もCPUを消費する(頻忙待機; busy wait)
  - 次に、そのほかの理由(例: タイマ割り込み)でスレッドの切り替え(reschedule)が起きるまで、Aがスケジュールされない
- ◆ 一方、同期プリミティブは同期不成立時にブロックする(休眠待機; blocking/sleeping wait)

# 頻忙待機

- ◆ 条件が成り立つまで中断(suspend)せずに待つ待機方法
- ◆ 典型的なテンプレート
  - while (!条件) { /\* 何もしない \*/ }
- ◆ 「そのうち」他のスレッドが「条件」を書き換えてくれることを期待
- ◆ OSから見ると「実行可能」に見えるため、待機中のスレッドにCPUが割り当てられる

# 休眠待機

- ◆ 自分は「待機」している事をOSに伝え、「中断」する
  - 適切なAPIの呼び出しによっておこる
  - read, recv, pthread\_mutex\_lock, pthread\_cond\_waitなど
  - 再開するまでCPU割り当ての対象としない



# 頻忙待機 vs 休眠待機

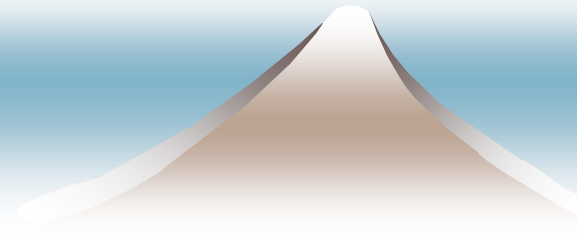
## ◆ 休眠待機

- CPUを「無駄にする」事がない(安全)
- こちらが基本フォーム(推奨)

## ◆ 頻忙待機

- 実行可能スレッド数  $\leq$  CPU数が確実ならオーバーヘッド小
  - 1 CPUでは絶対に×
  - スレッド数未知の場合も×
  - 長時間待つことになるかもしれない場合は×
- 実際の並列処理( $n$  CPU,  $n$ スレッド)では使われる事もある

- ◆ 同期プリミティブの実現
- ◆ CPUに備わる同期のための命令



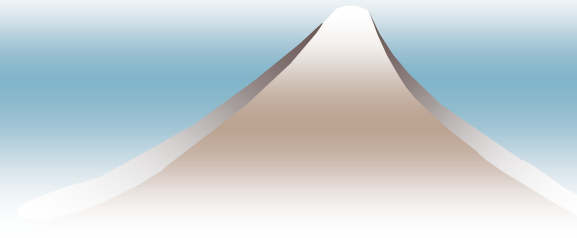
# 同期プリミティブの実現

## ◆ 例: mutexをどう実現する?

- `typedef struct mutex { ... } mutex;`
- `lock(m) {`  
    if ( $m$  がロックされていない) `mark_as_locked(m);`  
    else  $m$ がアンロックされるまでブロック; `}`
- `unlock(m) {`  
    `mark_as_unlocked(m);`  
     $m$  上で誰かブロックしていたら起こす;  
    `}`

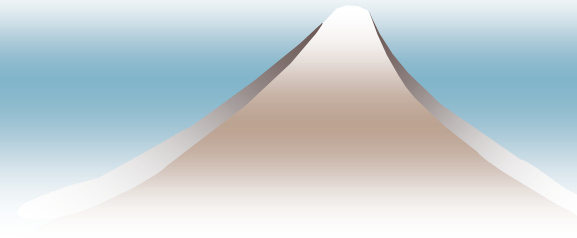
# 考えられるmutex構造体の中身

◆ typedef struct mutex {  
    int locked;     /\* lockされていれば1 \*/  
    queue q;       /\* blockしているスレッド \*/  
} mutex;



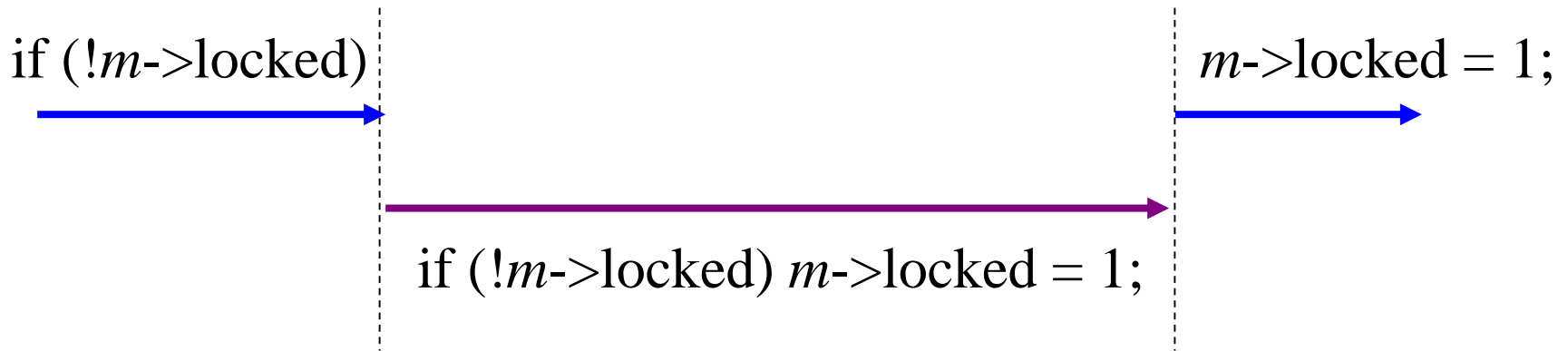
# 考えられるlock( $m$ )の実現???

- ◆ lock(mutex \*  $m$ ) {  
    if (! $m$ ->locked) { /\*  $m$ がロックされていなければ... \*/  
         $m$ ->locked = 1; /\*  $m$ をロックする \*/  
    } else { ... }  
}
- ◆ これでOK?





# またしても競合状態...



◆ lock(mutex \*  $m$ ) {  
    if (! $m$ ->locked) {  
         $m$ ->locked = 1  
    } else { ... }  
}

} 不可分に実行される必要がある

# Semaphoreの例

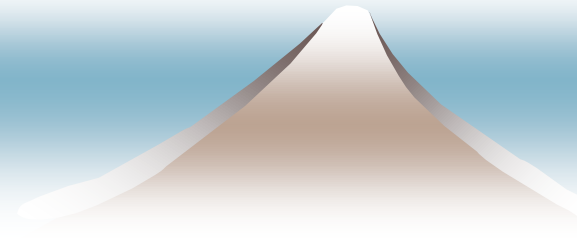
- ◆ typedef struct semaphore {  
    int n\_resources;       /\* 合計の資源の数 \*/  
    int n\_free\_resources; /\* 現在あいている資源の数 \*/  
    queue q;  
} semaphore;
  - ◆ semaphore\_acquire(semaphore \* s) {  
     $n = s \rightarrow n\_free\_resources$ ;  
    if ( $n > 0$ ) {  
         $s \rightarrow n\_free\_resources = n - 1$ ;  
    } else { ... }  
}
- ここでも同じ問題

# ハードウェアの持つ同期命令

- ◆ ほとんどのCPUで, いくつかのメモリアクセスを不可分(原子的; atomic)に行う命令が用意されている
- ◆ test-and-set  $p$ 
  - $\text{if } (*p == 0) \{ *p = 1; \text{succeeded} = 1; \}$   
   $\text{else } \{ \text{succeeded} = 0; \}$
- ◆ fetch-and-add  $p, x$ 
  - $*p += x$
- ◆ swap  $p, r$ 
  - $t = *p; *p = r; r = t;$

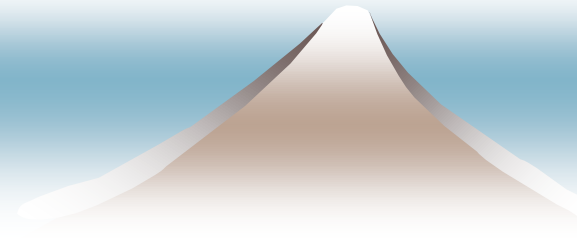
# test-and-setを用いた排他制御

- ◆ `lock(mutex * m) {`  
    `test-and-set(&m->locked);`  
    `if (succeeded) { /* OK */ }`  
    `else { ... }`  
}



# 汎用的なprimitive

- ◆ compare-and-swap  $p$   $r$   $s$ 
  - if ( $*p == r$ ) swap  $*p$  and  $s$ ;
  - これをatomicに行う

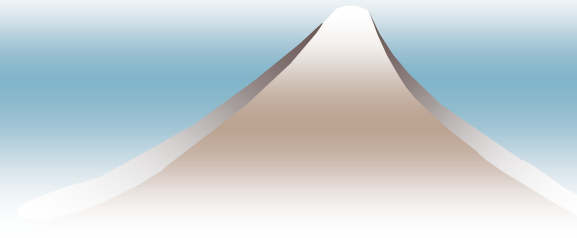


# compare-and-swapを用いたtest-and-set

- ◆ test-and-set( $p$ ) {  
     $s = 1$ ;  
    compare-and-swap  $p$  0  $s$ ;  
    if ( $s == 0$ ) succeeded = 1;  
    else succeeded = 0;  
}
- ◆ つまりcompare-and-swapは排他制御実現の道具として使える

# compare-and-swapを用いたfetch-and-add

- ◆ 例: スレッドA, B: {  $*p = *p + 1$ ; }
- ◆ while (1) {  
     $r = *p$ ;  
     $s = r + 1$ ;  
    compare-and-swap  $p$   $r$   $s$ ;  
    if ( $s == r$ ) break;  
}



# 一般的な不可分read-modify-write

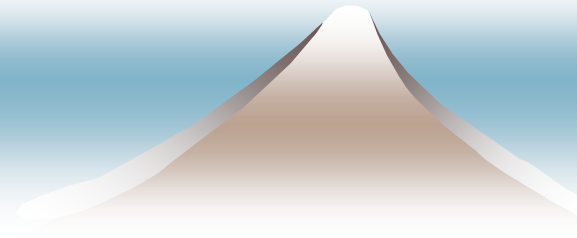
- ◆ “ $*p = \text{compute}(*p)$ ”を不可分に行う方法
- ◆ 

```
while (1) {  
     $r = *p$ ;  
     $s = \text{compute}(r)$ ;  
    compare-and-swap  $p$   $r$   $s$ ;  
    if ( $s == r$ ) break;  
}
```
- ◆ 一つのアドレスに対する, 短い更新(read-modify-write)を行う万能な方法



# デッドロック

- ◆ 競合状態がないように同期をかければ正しいプログラムとは限らない
- ◆ デッドロック
  - すべてのスレッドが意図せずブロックしてしまった状態
  - 安易に同期をかけた結果生じる問題



# 例: オンライン買い物サイト

- ◆ 多数の品物があり, ユーザは買い物かごに商品を入れて最後に支払いをする
- ◆ 簡単のため商品は2品まで. つまり, 以下のような「買い物リクエスト」が多数やってくる
  - $\text{buy}(a, b)$
  - $a, b$ の在庫が1以上あれば成立. さもないとエラー(どちらも売らない)

- ◆ `stock[N_ITEMS];`  
`buy(a, b) {`  
    `if (stock[a] > 0 && stock[b] > 0) {`  
        `stock[a]--;`  
        `stock[b]--;`  
    `}`  
`}`

- ◆ `buy`が多数のスレッドで実行されているときの問題は?

## 単純な解

```
◆ int stock[N_ITEMS]; mutex_t m;  
  buy(a, b) {  
    lock(&m);  
    if (stock[a] > 0 && stock[b] > 0) {  
      stock[a]--;  
      stock[b]--;  
    }  
    lock(&m);  
  }
```

問題点: すべてのリクエストは逐次的に実行される(並列度1)

# 改良版?

ひとつの品物に一つ  
のロック

```
◆ int stock[N_ITEMS]; mutex_t m[N_ITEMS];  
  buy(a, b) {  
    lock(&m[a]); lock(&m[b]);  
    if (stock[a] > 0 && stock[b] > 0) {  
      stock[a]--;  
      stock[b]--;  
    }  
    unlock(&m[b]); unlock(&m[a]);  
  }
```

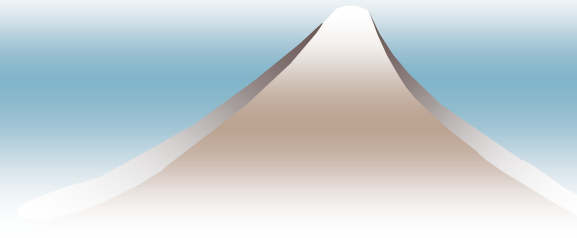
mutexを「分割して」必要最小限の排他制御をしているつもり

# 不運なシナリオ

- ◆ A : buy(1, 2) と B : buy(2, 1) が同時にやってきた
- ◆ A : lock(m[1]);  
B : lock(m[2]);  
A : lock(m[2]); /\* blocked \*/  
B : lock(m[1]); /\* blocked \*/
- ◆ デッドロック

# より複雑なシナリオ

- ◆  $\text{buy}(1, 2), \text{buy}(2, 3), \text{buy}(3, 4), \text{buy}(4, 1)$



# デッドロックの定義

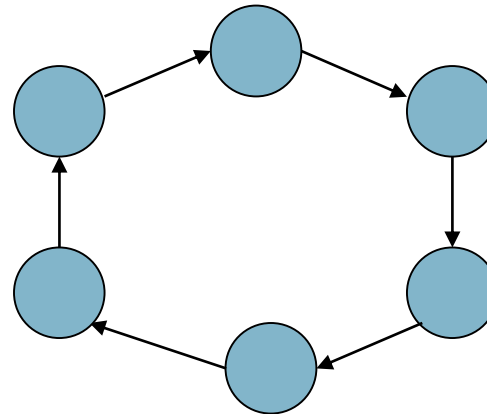
- ◆ スレッドの集合  $T = \{ T_1, \dots, T_n \}$  が以下の状態に陥ること
  - 各  $T_i$  は中断(ブロック)している
  - $T_i$  が中断から復帰するには  $T$  のどれかのスレッドが先へ進む必要がある
- ◆ 明らかにこの状態では  $T$  に属するスレッドが永遠にブロックし続ける



# デッドロックの生ずる条件

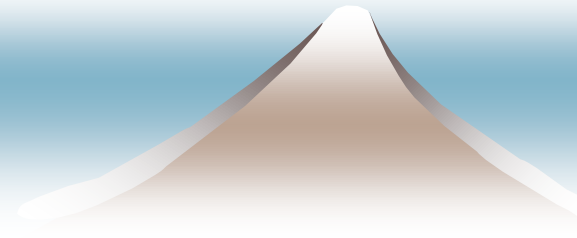
## ◆「待機」関係がサイクルを生ずる

- AがBを待つ
- BがCを待つ
- CがDを待つ
- ...
- XがAを待つ



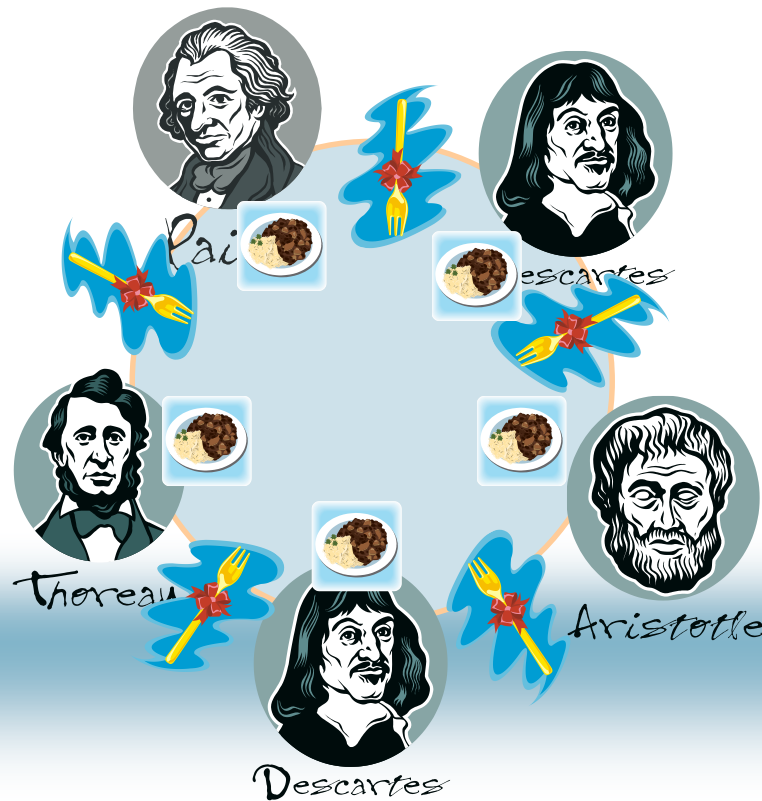
# 例題の解決策

- ◆ 解1: ロックを全部で一つだけにする
  - 欠点: 並列性が失われる
- ◆ 解2: ロックを確保する順番を統一する
  - $x < y$  ならば `lock(m[x]); lock(m[y]);` の順

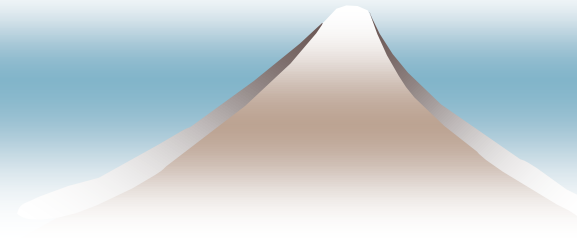


# 古典的な例題: 食事をする哲学者(Dining Philosopher)の問題

- ◆ 本質的には今の例題と全く同じだがあまりにも有名な例題なので一応出しておく

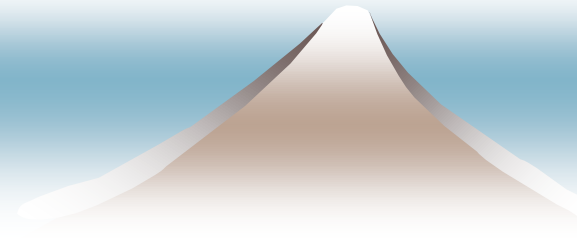


◆ Philosopher() {  
    while (1) {  
        think();  
        get\_right\_fork();    get\_left\_fork();  
        eat();  
        release\_left\_fork(); release\_right\_fork();  
    }  
}



# デッドロックしないための指針

- ◆ ロック以外の同期を行わないならば,「一つのプロセスは2個のロックを同時に保持しない」を順守する
- ◆ より強く「ロックは保持して一瞬で開放する」のが並列性を損なわないためにもよい指針となる



# まとめ 正しい同期

- ◆ 競合状態がない
  - Atomicityの保証
  - 実行の順序関係の保証
  - そのほか一般の「実行可能条件」の保証
- ◆ デッドロックがない
  - 成立すべき同期がすべて成立する
  - $\approx$ 「待機関係」がサイクルを作らない
- ◆ 頻忙待機しない

