

システムソフトウェア

2021年度

第6回 (10/21)

月曜7-8限・木曜7-8限(Zoom)

講義担当：渡部卓雄 (Takuo Watanabe)

<http://titech-os.github.io>

e-mail: takuo@c.titech.ac.jp

本日のメニュー

- 並行性制御(2)

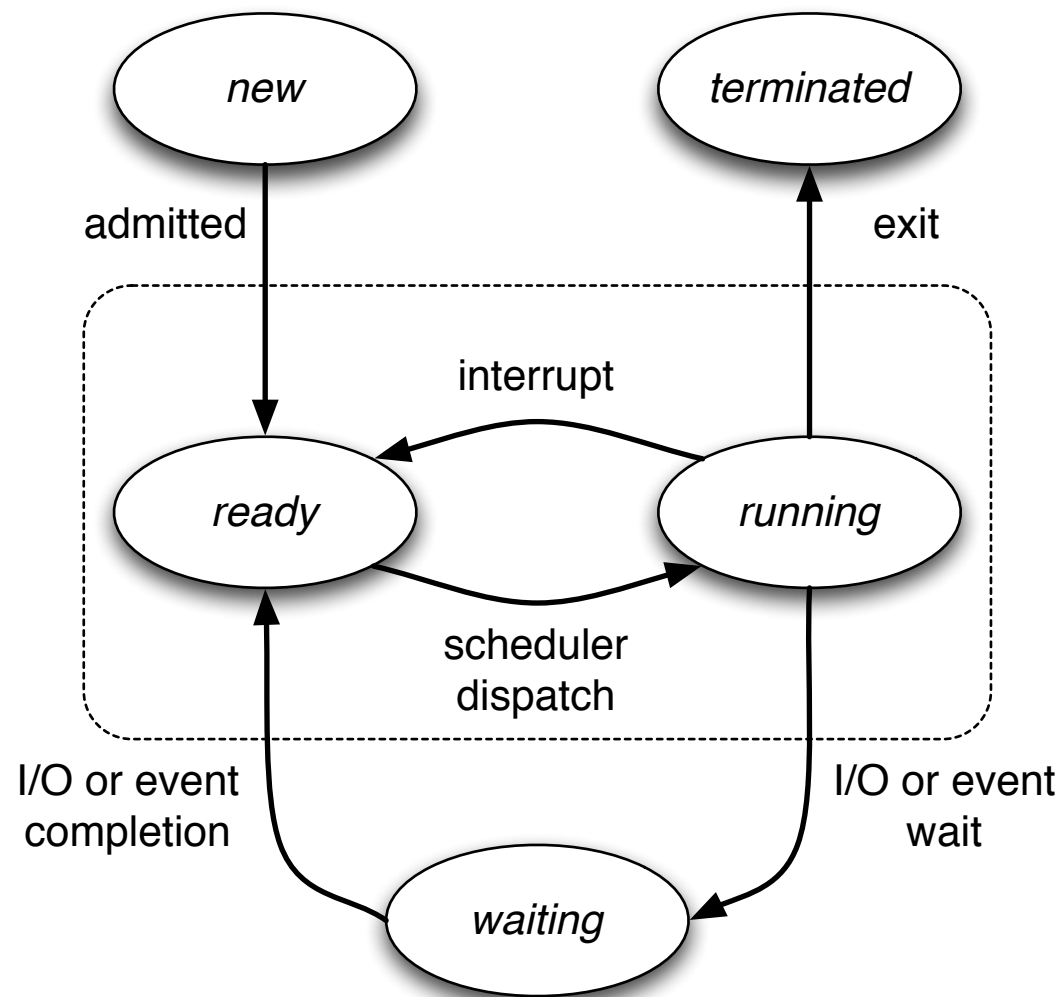
セマフォ (semaphore)

- E. W. Dijkstraが考案した同期機構
- 以下のデータからなるデータ構造
 - カウンタ(非負整数)
 - 寝ている(待ち状態にある)スレッドの有限集合
- セマフォに関する2つの操作
 - P操作 (downまたはwaitともいう)
 - オランダ語 *passeren*
 - V操作 (upまたはsignalともいう)
 - オランダ語 *verhoog*

セマフォの動作

- セマフォのカウンタの初期値を $k(>0)$ とする.
- P-操作
 - カウンタの値が0でなければカウンタを1減らす. 0ならばこの操作を行ったスレッドは寝る(待ち状態になる).
- V-操作
 - 寝ているスレッドがなければカウンタの値を1増やす. 寝ているスレッドがあったらその中から1つ選んで起こす.

スレッドの状態(再)



- *new*
 - スレッド生成時の状態
- *ready*
 - CPUの割り当て待ち状態
 - 論理的には「実行中」
- *running*
 - CPUが割り当てられている状態（命令の実行中）
 - この状態になれるスレッド数はCPU数以下
- *waiting*

セマフォの定義の準備(1)

- スレッドを表すデータ型 `thread` および以下に挙げる操作があるとする.
 - `void wait(void);`
 - この操作を実行したスレッドが待ち(`waiting`)状態になる.
 - `void notify(thread t);`
 - 引数`t`は待ち状態にあるスレッドとする. あるスレッドがこの操作を実行すると, `t`が実行可能(`ready`)状態となる.
 - `thread me(void);`
 - 返値はこの関数を呼び出したスレッドとする.

セマフォの定義の準備(2)

- 集合に関する記法
 - \emptyset : 空集合
 - $\{a\}$
 - 要素 a のみからなる集合
 - $X \cup Y$
 - 集合 X と集合 Y の和集合
 - $X \setminus Y$
 - 集合 X から集合 Y の要素を取り去った集合
 - $\text{choose}(X)$
 - 集合 X から 任意に 1つ選んだ要素

セマフォの構造 (C風疑似コード)

```
typedef struct {  
    unsigned int count;  
    thread_set waiting;  
} semaphore;
```

- スレッドを表す型

P-操作の定義（C風疑似コード）

```
atomic void P (semaphore s) {  
    if (s.count > 0)  
        s.count--;  
    else {  
        s.waiting = s.waiting u {me()};  
        wait();  
    }  
}
```

V-操作の定義（C風疑似コード）

```
atomic void V (semaphore s) {  
    if (s.waiting ==  $\emptyset$ )  
        s.count++;  
    else {  
        thread t = choose(s.waiting);  
        s.waiting = s.waiting \ {t};  
        notify(t);  
    }  
}
```

セマフォによる相互排除

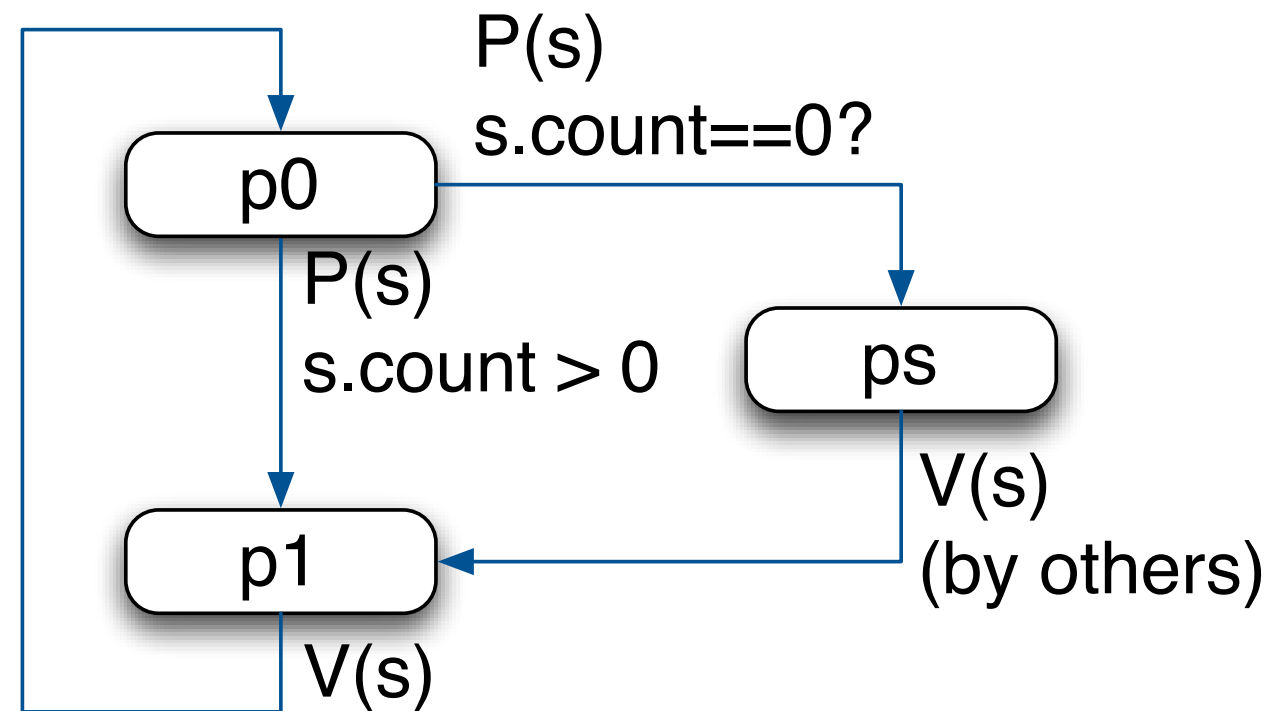
```
// shared variables  
semaphore s = { 1,  $\emptyset$  };
```

```
// thread p  
while (true) {  
    NC  
    P(s);  
    CS  
    V(s);  
}
```

```
// thread q  
while (true) {  
    NC  
    P(s);  
    CS  
    V(s);  
}
```

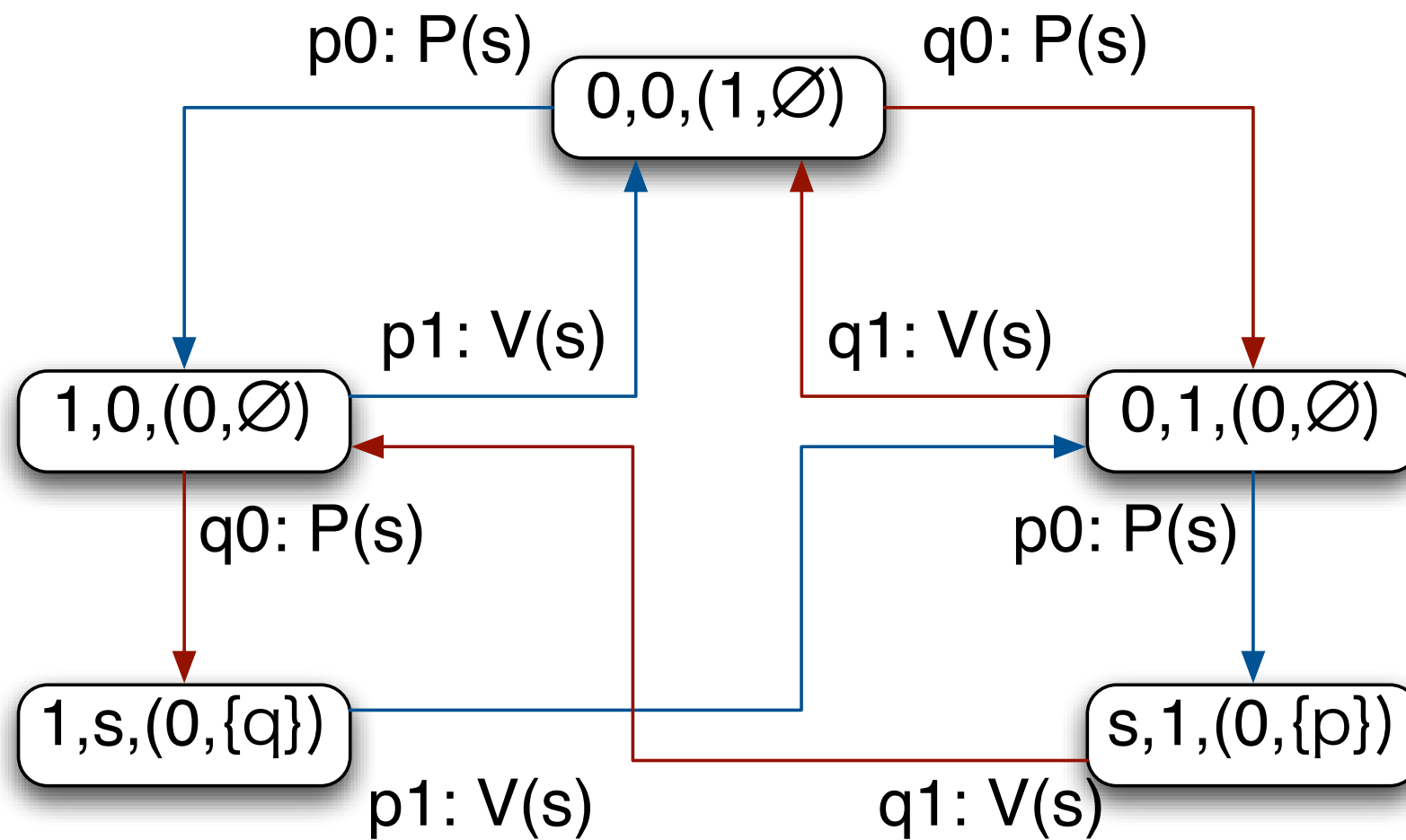
スレッドの状態

```
// thread p
while (true) {
  p0: P(s);
  p1: V(s);
}
```



プログラムの実行状態 `p0`, `p1` のほかに, セマフォ `s` に関する待ち状態 `ps` がある.

2つのスレッドからなる系



N個のスレッドの場合

```
// shared variables  
semaphore s = { 1,  $\emptyset$  };
```

```
// thread pi (i=0..N-1)  
while (true) {  
    NC  
    P(s);  
    CS  
    V(s);  
}
```

N個のスレッドの場合(2)

- 安全性は満たす.
 - － 同時にCSに入れるのは高々 1 個
- 活性はこのままでは満たさない.
 - － デッドロックは起らない.
 - － 飢餓状態は起こり得る.
 - 弱い公平性を仮定しても, choose が選ぶ要素については何も仮定がないため, たまたま 1 つのスレッドがいつまでも選ばれないという状況を排除できない.
 - － 公平性と choose は関係ないので.

キューを使ったセマフォ(1)

```
typedef struct {  
    unsigned int count;  
    thread_queue waiting;  
} semaphore;
```

- thread_queue
 - － スレッドのキューを表す型
 - － 操作
 - void enqueue(thread_queue, thread);
 - thread dequeue(thread_queue);
 - bool is_empty(thread_queue);
 - thread_queue new_queue(void);


```
atomic void P (semaphore s) {  
    if (s.count > 0)  
        s.count--;  
    else {  
        enqueue(s.waiting, me());  
        wait();  
    }  
}
```

```
atomic void V (semaphore s) {  
    if (is_empty(s.waiting))  
        s.count++;  
    else {  
        thread t = dequeue(s.waiting);  
        notify(t);  
    }  
}
```

キューを使ったセマフォ(2)

- 強いセマフォ (strong semaphore) と呼ばれるセマフォの一種で, 3個以上のスレッドにいても飢餓は起らない.
 - choose に何の仮定もないセマフォを弱いセマフォ (weak semaphore) という.
- 初期化
 - semaphore s = { 1, new_queue() };

セマフォの初期値

```
// semaphore  
semaphore s = { k, ∅ };
```

```
// thread pi  
while (true) {  
    NC  
    P(s);  
    CS  
    V(s);  
}
```

- $k > 0$ とする.
- この場合, 最大 k 個のスレッドが同時にCSに入ることができる.

バイナリセマフォと計数セマフォ

- バイナリセマフォ (binary semaphore)
 - s.countの初期値が1で, 1を超えない場合.
 - mutex と呼ばれてよく用いられる.
- 計数セマフォ (counting semaphore)
 - s.countの値が任意の非負整数をとる場合.

セマフォの性質(1)

- 定義
 - － $\#P(s)$
 - セマフォ s が作られてからの $P(s)$ の実行回数
 - － ただしwaiting状態にあるものはnotifyされるまでカウントしないものとする.
 - － $\#V(s)$
 - セマフォ s が作られてからの $V(s)$ の実行回数
 - － $\#CS$
 - クリティカルセクションにいるスレッドの数

セマフォの性質(2)

- s をセマフォ, k は $s.count$ の初期値とする. 以下は不変条件である.
 - $s.count \geq 0$
 - $s.count = k + \#V(s) - \#P(s)$

生産者・消費者問題

- 以下のような2つのスレッドの同期をとる.
 - － 生産者
 - データを生産してバッファ（キュー）に入れる.
 - バッファが一杯ならそこで待たされる.
 - － 消費者
 - データをバッファから取り出して消費する.
 - バッファが空ならそこで待たされる.

無限長バッファの場合

```
// shared variables  
semaphore ne = { 0, ∅ }; // not empty  
queue buff = new_queue(); // unbounded
```

```
// producer  
while (true) {  
    data x;  
    x = produce();  
    enqueue(buff, x);  
    V(ne);  
}
```

```
// consumer  
while (true) {  
    data x;  
    P(ne);  
    x = dequeue(buff);  
    consume(x);  
}
```


有限長バッファ (長さN)の場合

```
// shared variables
semaphore ne = { 0, ∅ }; // not empty
semaphore nf = { N, ∅ }; // not full
queue buff = new_queue(N);
```

```
// producer
while (true) {
    data x;
    x = produce();
    P(nf);
    enqueue(buff, x);
    V(ne);
}
```

```
// consumer
while (true) {
    data x;
    P(ne);
    x = dequeue(buff);
    V(nf);
    consume(x);
}
```

readers-writers問題

- Xを共有変数とする. Xを読むスレッド (reader)とXに書くスレッド (writer)がそれぞれ任意個あるとする.
- Xについて以下のようなCSを作りたい:
 - 同時にCSに入ることのできるwriterは高々 1 個で, readerがCSに入っているときはwriterは入ることはできない.
 - writerがCSに入っていない限り, readerは何個でもCSに入ることができる.

readers-writers問題の解(1)

```
// shared variables
semaphore wrt = { 1, ∅ };
semaphore mtx = { 1, ∅ };
int readers = 0;
```

```
// writers
while (true) {
    NC
    P(wrt);
    CS
    V(wrt);
}
```

readers-writers問題の解(2)

```
// readers
while (true) {
    NC
    P(mtx);
    readers++;
    if (readers == 1) P(wrt);
    V(mtx);
    CS
    P(mtx);
    readers--;
    if (readers == 0) V(wrt);
    V(mtx);
}
```

食事する哲学者の問題：解 1 (1)

```
semaphore fork[N];  
for (int i = 0; i < N; i++) {  
    fork[i].count = 1;  
    fork[i].waiting = 0;  
}
```

食事する哲学者の問題：解1(2)

```
// i : 0..N-1
void philosopher (int i) {
    while (true) {
        THINK
        P(fork[i]);
        P(fork[(i + 1) % N]);
        EAT
        V(fork[i]);
        V(fork[(i + 1) % N]);
    }
}
```

解1の性質

- 安全性：○
 - － どのフォークも高々一人の哲学者によって確保される.
- 活性：×
 - － デッドロックが起こり得る.
 - まず、各哲学者(i)が一斉に自分の左側のフォーク($\text{fork}[i]$)を手にとったとする.
 - 次に右側のフォーク($\text{fork}[(i+1)\%N]$)を取ろうとすると、それはすでに隣の哲学者によって取られている.

解2(1)

```
semaphore fork[N];  
for (int i = 0; i < N; i++) {  
    fork[i].count = 1;  
    fork[i].waiting = ∅;  
}  
semaphore room = { N - 1, ∅ };
```

同時に食堂(room)に入ることのできる人数をN-1までに制限することで、全員が一斉にフォークを持てないようにし、デッドロックが起きないようにする。

解2(2)

```
// i : 0..N-1
void philosopher (int i) {
    while (true) {
        THINK
        P(room);
        P(fork[i]);
        P(fork[(i + 1) % N]);
        EAT
        V(fork[i]);
        V(fork[(i + 1) % N]);
        V(room);
    }
}
```

解3：非対称解

```
void philosopher0 (void) {  
    while (true) {  
        THINK  
        P(fork[1]);  
        P(fork[0]);  
        EAT  
        V(fork[0]);  
        V(fork[1]);  
    }  
}
```

$i > 0$ については解1と同じ。ひとりだけフォークを取る順序を逆にして、全員が一斉に左のフォークを取らないようにする。

Javaにおけるセマフォ

- java.util.concurrent.Semaphoreクラス
 - － コンストラクタ
 - Semaphore(int k)
 - － 初期値 k の計数セマフォを作成する.
 - － メソッド
 - void acquire()
 - － P-操作
 - void release()
 - － V-操作

モニタ (monitor)

- Hoareによって提案された同期機構
 - － 原論文（古典．読むべし）
 - C. A. R. Hoare, Monitors: An Operating System Structuring Concept, CACM, Vol. 17, No. 10, pp. 549-557, 1974.
 - － セマフォと異なり構造化されているため使いやすく誤りが生じにくい.
 - － 様々なバリエーションが各種言語やライブラリで採用されている(例：Java).

モニタの定義(Java風疑似コード)

```
monitor Name {  
    monitor variable declarations  
    monitor procedure definitions  
}
```

- Javaのクラスに似た構造で, モニタ変数とモニタ手続きから構成される.
 - モニタ変数
 - モニタ外からは読み書きできない変数
 - モニタ手続き
 - アトミックに実行される手続きで, モニタ変数の読み書きができる.

モニタの基本動作

- 一つのモニタのモニタ手続きは、同時に一つのスレッドだけが実行できる。
 - － あるスレッドがモニタ手続きを実行している（モニタに入っている）間に、他のスレッドが同じモニタのモニタ手続きを実行しようとするのを待たされる。
- 同じモニタの別のモニタ手続きの場合でも
 - － モニタ手続きを実行中のスレッドがそのモニタ手続きの実行を終える（モニタから出る）と、待たされている別のスレッドが入ることができる。

例 1 : DoubleCounter(1)

```
monitor DoubleCounter {  
    int n = 0, m = 0;  
  
    public void inc () {  
        int p = n, q = m;  
        n = p + 1;  
        m = q + 2;  
    }  
}
```

例 1 : DoubleCounter(2)

```
// shared resources
DoubleCounter dc =
    new_monitor DoubleCounter();
```

```
// in each thread
while (true) {
    dc.inc();
}
```

- モニタ Counter のモニタ手続き inc はアトミックに実行されるため, 常に $n * 2 == m$ になる.

セマフォによる定義(1)(C風疑似コード)

```
typedef struct doubleCounter {  
    int n, m;  
    semaphore sem;  
} *doubleCounter;  
  
doubleCounter new_doubleCounter () {  
    doubleCounter dc =  
        malloc(sizeof(struct doubleCounter));  
    dc->n = dc->m = 0;  
    dc->sem.count = 1;  
    dc->sem.waiting = 0;  
    return dc;  
}
```

セマフォによる定義(2)(C風疑似コード)

```
void doubleCounter_inc (doubleCounter dc) {  
    P(dc->sem);  
    int p = dc->n, q = dc->m;  
    dc->n = p + 1;  
    dc->m = q + 2;  
    V(dc->sem);  
}
```

```
// shared resources  
doubleCounter dc = new_doubleCounter();
```

```
// in each thread  
while (true) {  
    doubleCounter_inc(dc);  
}
```

Javaによる定義

```
class DoubleCounter {  
    private int n = 0, m = 0;  
  
    public synchronized void inc () {  
        int p = n, q = m;  
        n = p + 1;  
        m = q + 2;  
    }  
}
```

モニタでセマフォを定義(1)

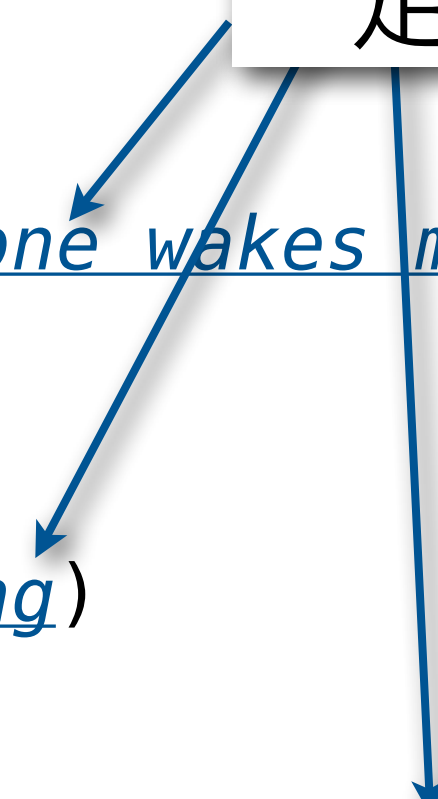
```
monitor Semaphore {  
    int counter;  
  
    // constructor  
    public Semaphore (int c) {  
        counter = c;  
    }  
  
    // P- and V-operations (next page)  
    public void P () { ... }  
    public void V () { ... }  
}
```

モニタでセマフォを定義(2)

```
public void P () {  
    if (counter > 0)  
        counter--;  
    else  
        wait until someone wakes me up  
}
```

```
public void V () {  
    if (no one is waiting)  
        counter++;  
    else  
        choose one and wake him/her up  
}
```

どのようにして
定義するか？

Three blue arrows originate from the question box. One arrow points to the wait until someone wakes me up line in the P() method. Another arrow points to the no one is waiting line in the V() method. A third arrow points down towards the choose one and wake him/her up line in the V() method.

条件変数(condition variable)

- モニタ内で以下のようにして宣言
 - **condition** c_1, c_2, \dots, c_k ;
- 条件変数(c)に関する操作
 - **wait**(c)
 - **notify**(c)
- これらの操作は c を宣言しているモニタ手続き内でのみ有効.

wait(c)

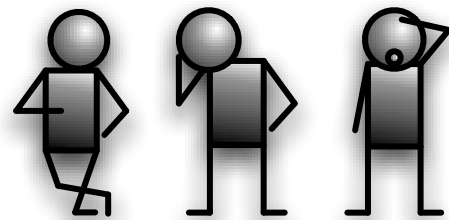
- モニタM内でこの操作を実行したスレッドは待ち状態(waiting状態)に入る。その際、Mには他のスレッドが一つ入れるようになる。
 - － これを実行して待ち状態にあるスレッドを、「cに関して待っている（待ち状態にある）」スレッドと呼ぶ。

notify(c)

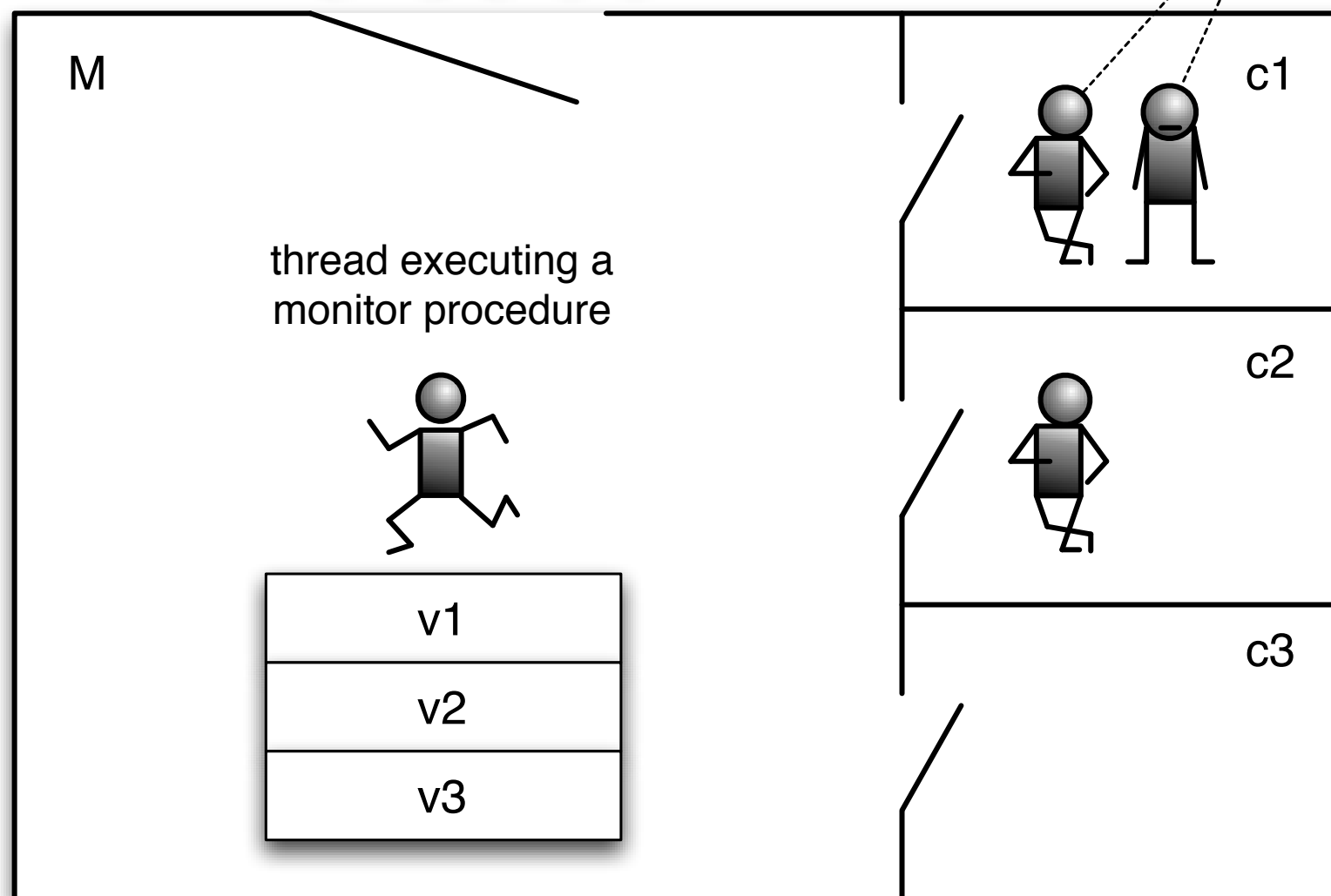
- cに関して待っているスレッドを一つ選んで実行可能(ready状態)にする.
 - 多くの実装では, 待っているスレッドはキューを構成するようになっている.
- empty(c)
 - cに関して待っているスレッドがないときに真となる述語

モニタのイメージ図

threads waiting to be allowed to enter M



threads waiting on the condition variable c1



モニタによるセマフォの定義(1)

```
monitor Semaphore {  
    int counter;  
    condition waiting;  
  
    public Semaphore (int c) { ... }  
  
    public void P () { ... }  
    public void V () { ... }  
}
```

モニタによるセマフォの定義(2)

```
public void P () {  
    if (!(counter > 0))  
        wait(waiting);  
    assert(counter > 0);  
    counter--;  
}
```

```
public void V () {  
    counter++;  
    assert(counter > 0);  
    notify(waiting);  
}
```

生産者・消費者問題

```
// shared resoruces  
PC pc = new_monitor PC();
```

```
// producer  
while (true) {  
    data x;  
    x = produce();  
    pc.put(x);  
}
```

```
// consumer  
while (true) {  
    data x;  
    x = pc.get();  
    consume(x);  
}
```

生産者・消費者問題の解

```
monitor PC {  
    Queue[data] buff =  
        new Queue[data](N);  
    condition notEmpty, notFull;  
  
    public void put (data v) { ... }  
    public data get () { ... }  
}
```

```
public void put (data v) {  
    if (!buff.is_not_full())  
        wait(notFull);  
    assert(buff.is_not_full());  
    buff.enqueue(v);  
    assert(buff.is_not_empty());  
    notify(notEmpty);  
}
```

```
public data get () {  
    if (!buff.is_not_empty())  
        wait(notEmpty);  
    assert(buff.is_not_empty());  
    data w = buff.dequeue();  
    assert(buff.is_not_full());  
    notify(notFull);  
    return w;  
}
```

Readers-Writers問題

```
monitor RW {  
    int readers = 0, writers = 0;  
    condition OKtoRead, OKtoWrite;  
  
    public void startRead () { ... }  
    public void endRead () { ... }  
    public void startWrite () { ... }  
    public void endWrite () { ... }  
}
```

```
public void startRead () {  
    if (!(writers == 0 && empty(OKtoWrite)))  
        wait(OKtoRead);  
    assert(writers == 0 && empty(OKtoWrite));  
    readers++;  
    assert(writers == 0 && empty(OKtoWrite));  
    notify(OKtoRead);  
}  
  
public void endRead () {  
    readers--;  
    if (readers == 0) {  
        assert(writers == 0 && readers == 0);  
        notify(OKtoWrite);  
    }  
}
```



```
public void startWrite () {  
    if (!(writers == 0 && readers == 0))  
        wait(OKtoWrite);  
    assert(writers == 0 && readers == 0);  
    writers++;  
}  
  
public void endWrite () {  
    writers--;  
    if (empty(OKtoRead)) {  
        assert(writers == 0 && readers == 0);  
        notify(OKtoWrite);  
    }  
    else {  
        assert(writers == 0 && empty(OKtoWrite));  
        notify(OKtoRead);  
    }  
}
```

食事する哲学者の問題

```
monitor DP {  
    final int N;  
    int[] nforks;  
    condition[] OKtoEat;  
  
    public DP (int n) {  
        N = n;  
        nforks = new int[N];  
        for (int i = 0; i < N; i++)  
            nforks[i] = 2;  
        OKtoEat = new condition[N];  
    }  
  
    public void startEat (int i) { ... }  
    public void endEat (int i) { ... }  
}
```

```

public void startEat (int i) {
    if (!(nforks[i] == 2))
        wait(OKtoEat[i]);
    nforks[(i + 1) % N]--;
    nforks[(i - 1 + N) % N]--;
}

public void endEat (int i) {
    nforks[(i + 1) % N]++;
    nforks[(i - 1 + N) % N]++;
    if (nforks[(i + 1) % N] == 2)
        notify(OKtoEat[(i + 1) % N]);
    if (nforks[(i - 1 + N) % N] == 2)
        notify(OKtoEat[(i - 1 + N) % N]);
}

```

notifyについて

- スレッド t が $\text{notify}(c)$ を実行すると、 c に関して待っていたスレッド t' が起こされる.
- この場合、 t と t' の2つが一つのモニタ内で同時に実行可能(ready)になってしまう.
- そこで、 t と t' に優先順位を設け、どちらかがモニタを出るまで待たされることにする.
 - Hoare式(IRR): $t < t'$
 - Mesa式(non-IRR): $t > t'$

IRR(Immediate Resumption Requirement)

- notify(c)で起こされたスレッドが、すぐに (notify(c)を実行したスレッドより優先的に) 実行できる方式.
 - Hoareのオリジナルなモニタ
 - 本資料のモニタも
- 利点
 - wait(c)で待っているスレッドは、cが表す条件が成り立つまで待っている. IRRは、起こされたときにその条件が成り立つことを保証している.

non-IRR

- notify(c)を実行したスレッドが、notify(c)で起こされたスレッドよりも優先する方式.
 - － 例：Mesa(言語), Javaのモニタ
- 利点
 - － notify(c)を実行したスレッドが余計に待たされず、多くのプログラムで並行性が出る.
- 欠点
 - － wait(c)で待っていたスレッドが起こされたときに、cが表す条件が成立しているとは限らない.

non-IRRの場合のセマフォの定義

```
public void P () {  
    while (!(counter > 0))  
        wait(waiting);  
    assert(counter > 0);  
    counter--;  
}
```

```
public void V () {  
    counter++;  
    assert(counter > 0);  
    notify(waiting);  
}
```

まとめ

- 同期と排他制御(2)
 - － 一般的な同期機構：セマフォ, モニタ