システムソフトウェア

2020年度

第6回 (10/26)

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(1)

同期(synchronization)

- 適当な条件をみたすまでプロセスやスレッドを待たせること.
- 同期の目的
  - 協調(cooperation)
  - 排他(exclusion)

競合状態

- 複数のスレッドが動作する際，動作のタイミングによって実行結果が異なってしまうこと
  - 主に共有変数の読み書きのタイミングによる.
  - 単一スレッドのプログラムでも，割り込み(シグナル)によって競合状態が起ることがある.
    - 割り込みは並行動作の一種
- 一般に競合状態の検出やデバッグは難しい
  - 再現性が低い.
  - 観測によって動作が変わることがある.

クリティカルセクション

- プログラム中で，共有データへのアクセスを伴う部分のことをいう.
- ２つ以上のプログラムが同時にクリティカルセクションに入ることで競合状態が起る.

- 相互排除(mutual exclusion)
  - 複数のプログラムが同時にクリティカルセクションに入らないようにすること

相互排除の要件

- 安全性(safety)
  - クリティカルセクションに同時に入れるスレッドは高々ひとつ
- 活性(liveness, progress)
  - デッドロックがない(freedom from deadlock)
    - いくつかのスレッドがクリティカルセクションに入ろうとした場合，少なくとも一つは入ることができる.
  - 飢餓がない(freedom from starvation)
    - クリティカルセクションに入ろうとしたスレッドはいつかは必ず入ることができる.

相互排除問題の定式化

```
while (true) {
    NC
    CS
}
```

- 各スレッドは，クリティカルセクションではない部分(NC)とクリティカルセクションの部分(CS)を交互に繰り返すものとする.
- CSの実行は有限時間内に必ず終了するものとする.
- NCの実行は終了しないこともあり得る(ある時点からはずっとCSに入ってこないスレッドも存在する).

相互排除アルゴリズムの形式

```
while (true) {
     NC
     EnterCS
     CS
     ExitCS
}
```

- CSに入る際にEnterCS，出る際にExitCSを実行する.
- EnterCS/ExitCSはプログラムから独立している.
  - EnterCS/ExitCSでは，NC/CSで使われている変数は用いないものとする.
  - 逆にNC/CSでは，EnterCS/ExitCSで使われている変数は用いないものとする.

アトミックな文(atomic sentence)

- 文Sがアトミック(不可分)であるとは，Sの実行途中の状態が(他のスレッドによって)観測・干渉されないことをいう.
- アトミックな文S1, S2を両方実行した結果は，S1;S2 あるいは S2;S1 のいずれかの順に逐次実行したものと等しくなる.
- 多くの場合，機械語の 1 命令はアトミックとみなしてよい.
  - マルチプロセッサの場合は必ずしもそうでない場合もある.

高級言語の場合

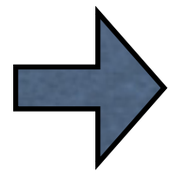
- 文や式は一般にアトミックではない.
  - コンパイラやプロセッサに依存する.
- 例：x++;
  - 読み出し-変更-書き込み(read-modify-write)を行っているかも知れない.
    - r1 = x;
    - r1 = r1 + 1;
    - x = r1;
  - たとえ1命令でも(x86のincl), マルチプロセッサ環境ではアトミックにならないことがある.

アトミック性についての仮定

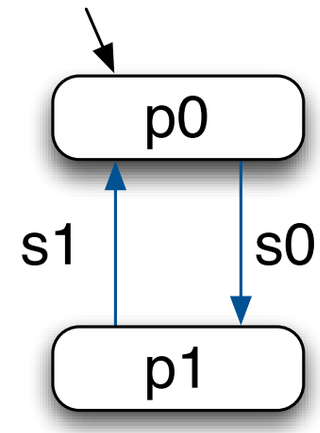
- 共有変数への代入 x = E; は以下の場合にアトミックであるとする.
  - 変数xの大きさが，プロセッサがアトミックに書きこめるサイズ以内であり，かつ
  - 式Eが共有変数を含まない場合
    - 例えば定数や，スレッド内で局所的な変数で構成された式の場合.
- 共有変数 x の読み出しは，xのサイズがアトミックに読み出せるサイズ以内の場合にアトミックであるとする.

スレッドの状態遷移図による表現

```
while (true) {
    s0
    s1
}
```



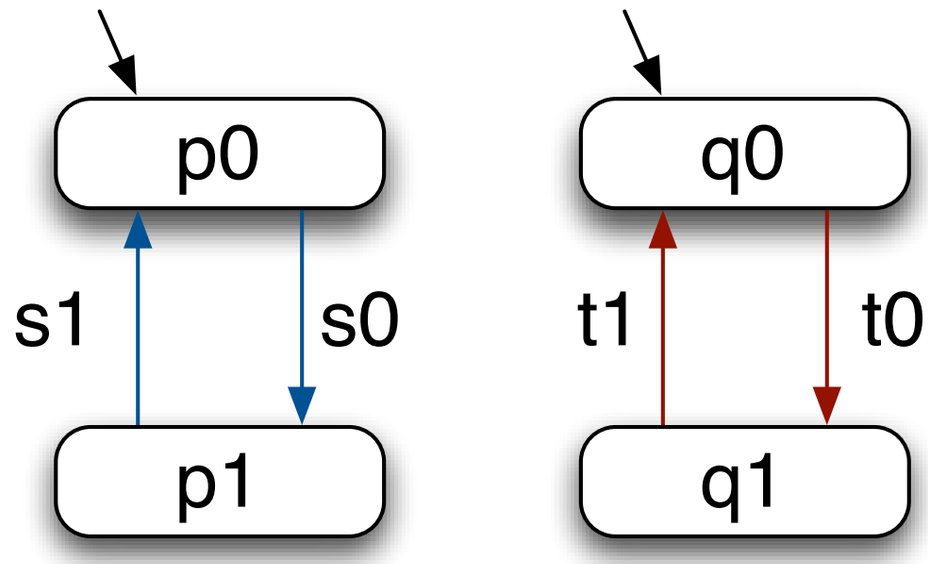
```
while (true) {
    p0: s0
    p1: s1
}
```



- アトミックな文s0, s1からなる上左のようなスレッドを考える.
- その各文にラベル p0, p1 をつける.
- これを右のような状態遷移図として表す.
  - ラベルp0,p1はs0,s1の実行前の状態を表す.

マルチスレッドの表現(1)

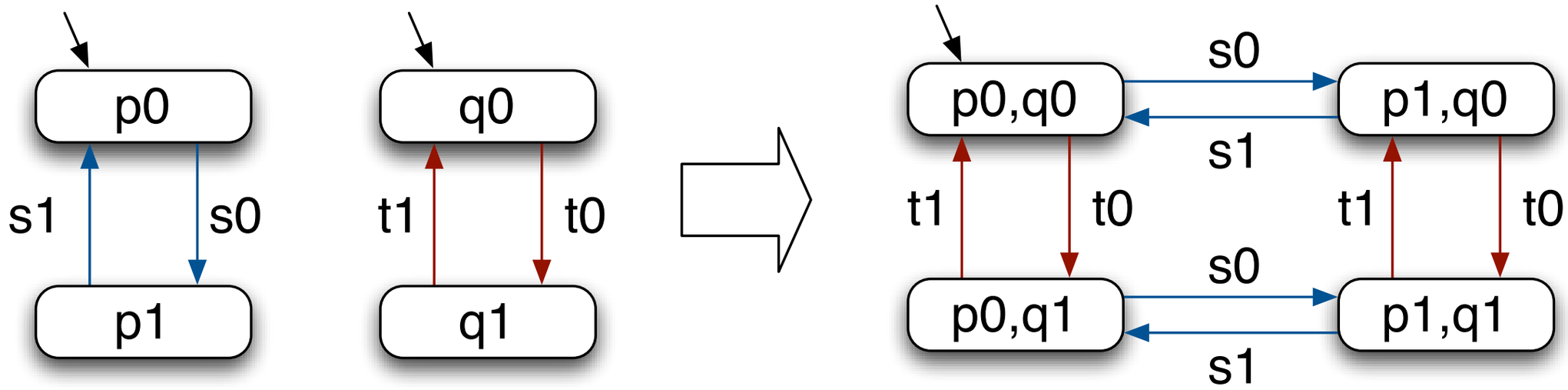
```
while (true) {
     p0: s0
     p1: s1
}
```




```
while (true) {
     q0: t0
     q1: t1
}
```

2つのスレッドはそれぞれ上のような状態遷移図として表現できる.

マルチスレッドの表現(2)



- 複数スレッドの状態遷移図を合成して，全ての可能な実行を表す状態遷移図を作ることができる.
  - インターリービング(interleaving)による表現

アルゴリズム０

```
// shared variable
bool in_use = false;
```

```
// thread p
while (true) {
  p0: NC
  p1: while (in_use);
  p2: in_use = true;
  p3: CS
  p4: in_use = false;
}
```

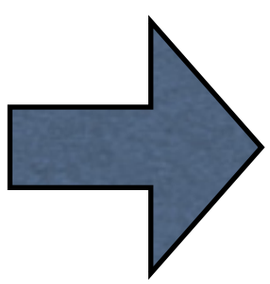
```
// thread q
while (true) {
  q0: NC
  q1: while (in_use);
  q2: in_use = true;
  q3: CS
  q4: in_use = false;
}
```

- EnterCS: p1, p2 (q1, q2)
- ExitCS: p4 (q4)

アルゴリズムOの単純化

```
while (true) {
   p0: NC
   p1: while (in_use);
   p2: in_use = true;
   p3: CS
   p4: in_use = false;
}
```

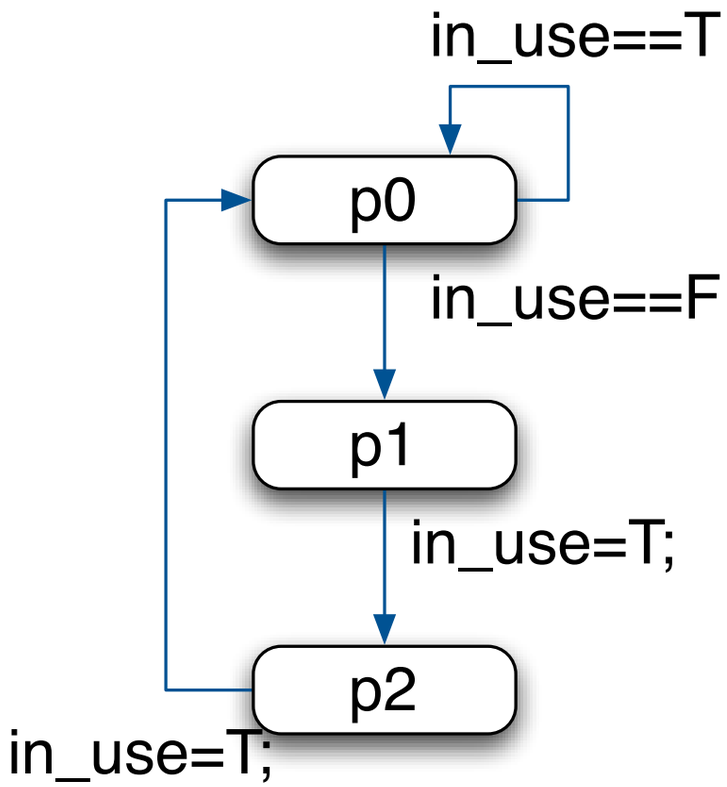
本質的でない
状態を削除



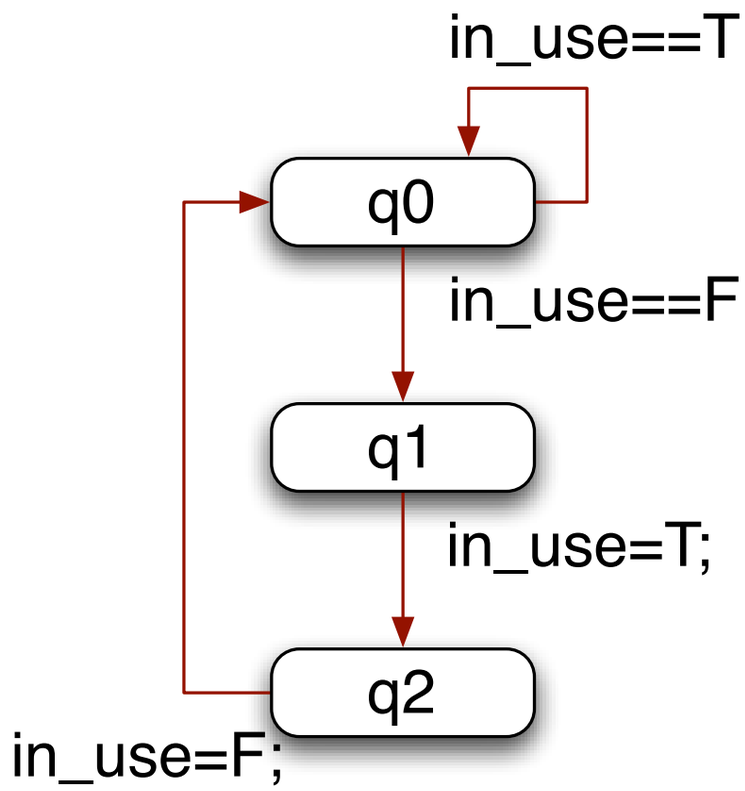
```
while (true) {
   p0: while (in_use);
   p1: in_use = true;
   p2: in_use = false;
}
```

2つのスレッドの表現

```
while (true) {
   p0: while (in_use);
   p1: in_use = true;
   p2: in_use = false;
}
```



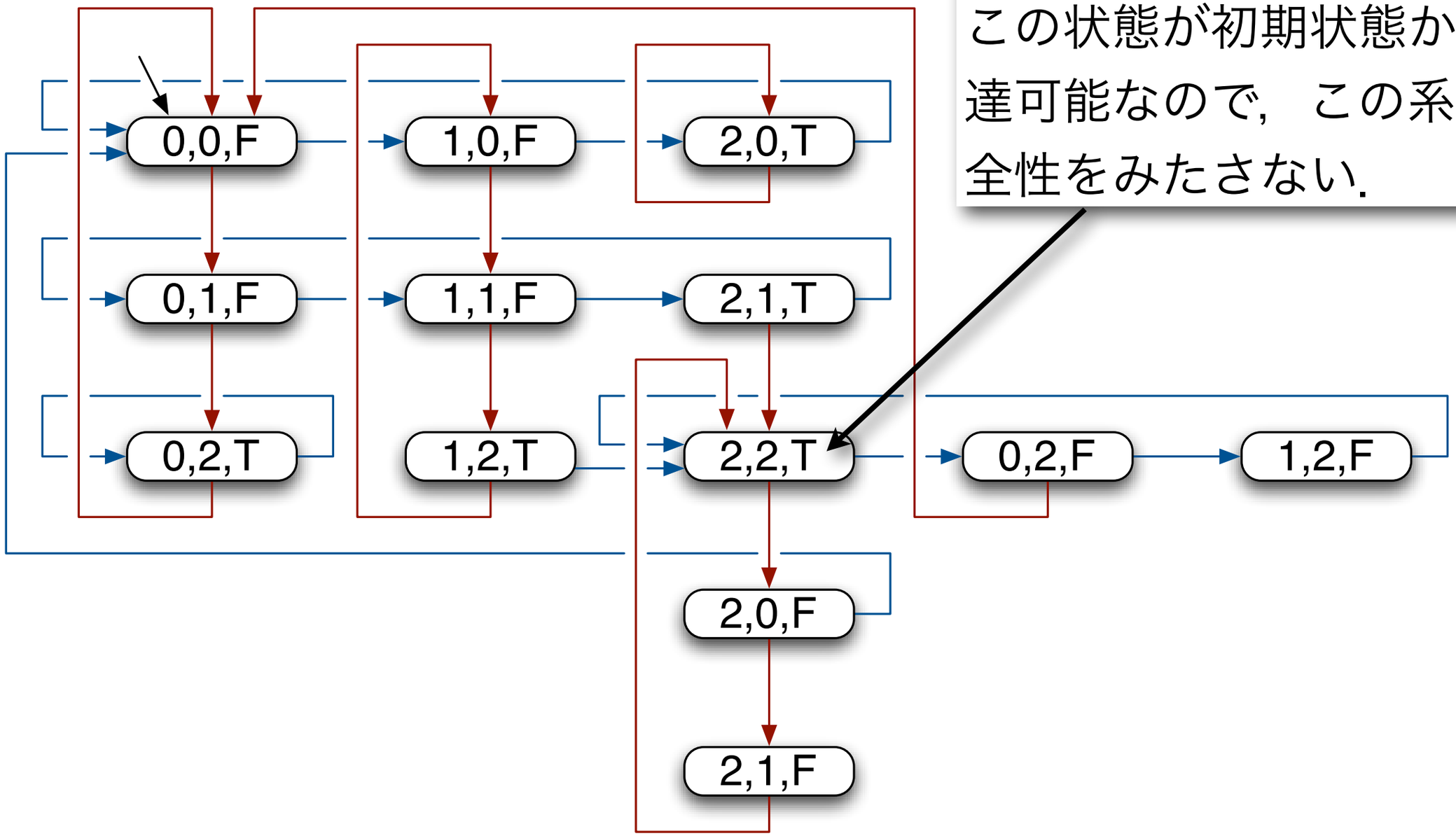

```
while (true) {
   q0: while (in_use);
   q1: in_use = true;
   q2: in_use = false;
}
```



アルゴリズムの正当性

- 安全性：¬(p2∧q2)
  - ２つのスレッドが同時にCSに入らないことは，スレッドの状態が同時にp2とq2にならないこととして定式化できる.
  - スレッド p, q の状態遷移図を合成した系を考え，初期状態から p2∧q2 をみたす状態に到達する経路がないことを示せばよい.
  - 合成した系の状態は，各スレッドのラベルと変数 in_use の３つ組として表現される.

合成した状態遷移図



２つのスレッドが同時にCSに入ったことを表している.
この状態が初期状態から到達可能なので, この系は安全性をみたさない.

アルゴリズム1

```
// shared variable
int turn = 0; // 0 or 1
```

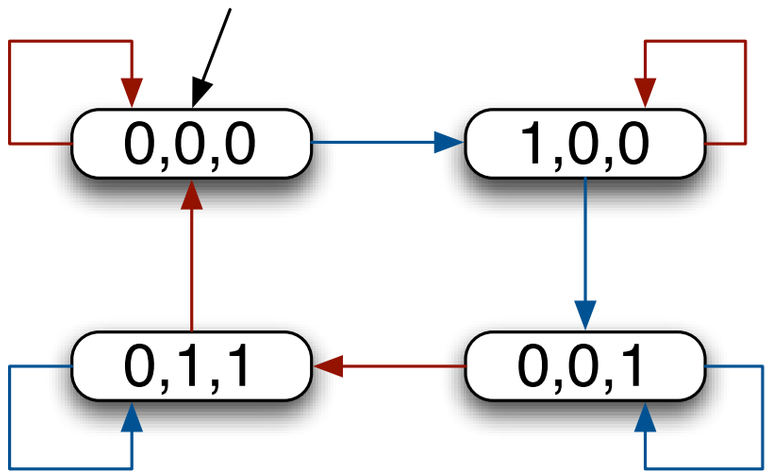
```
// thread p
while (true) {
  p0: NC
  p1: while (turn == 1);
  p2: CS
  p3: turn = 1;
}
```

```
// thread q
while (true) {
  q0: NC
  q1: while (turn == 0);
  q2: CS
  q3: turn = 0;
}
```

簡略化と状態遷移図の合成

```
while (true) {
   p0: while (turn == 1);
   p1: turn = 1;
}
```

```
while (true) {
   q0: while (turn == 0);
   q1: turn = 0;
}
```



初期状態から p1∧q1 をみたす状態には到達可能ではないので，安全性はみたされている．

アルゴリズム１の問題点

- スレッドが飢餓状態に陥ることがある.
  - 何らかの原因で片方のスレッドが停止した場合，など，２つあるスレッドの一つがいつまでもCSに入ろうとしない場合，もう片方はいつまでたっても自分の順番がこないことがある.

- 注：デッドロックにはならない.
  - ２つがCSに入ろうとすれば片方は必ず入れる.

アルゴリズム２

```
// shared variables
bool wantp = false, wantq = false;
```

```
// thread p
while (true) {
  p0: NC
  p1: while (wantq);
  p2: wantp = true;
  p3: CS
  p4: wantp = false;
}
```

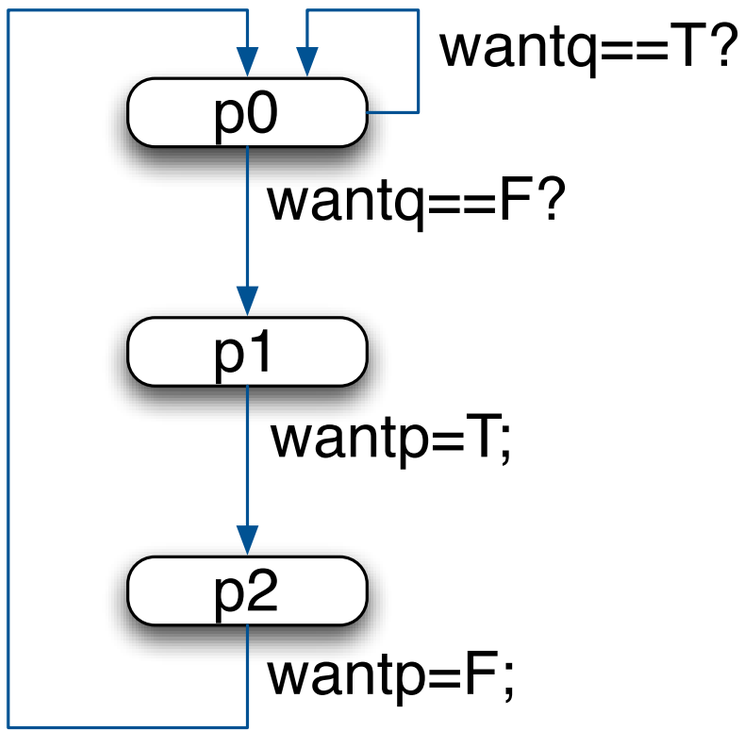
```
// thread q
while (true) {
  q0: NC
  q1: while (wantp);
  q2: wantq = true;
  q3: CS
  q4: wantq = false;
}
```

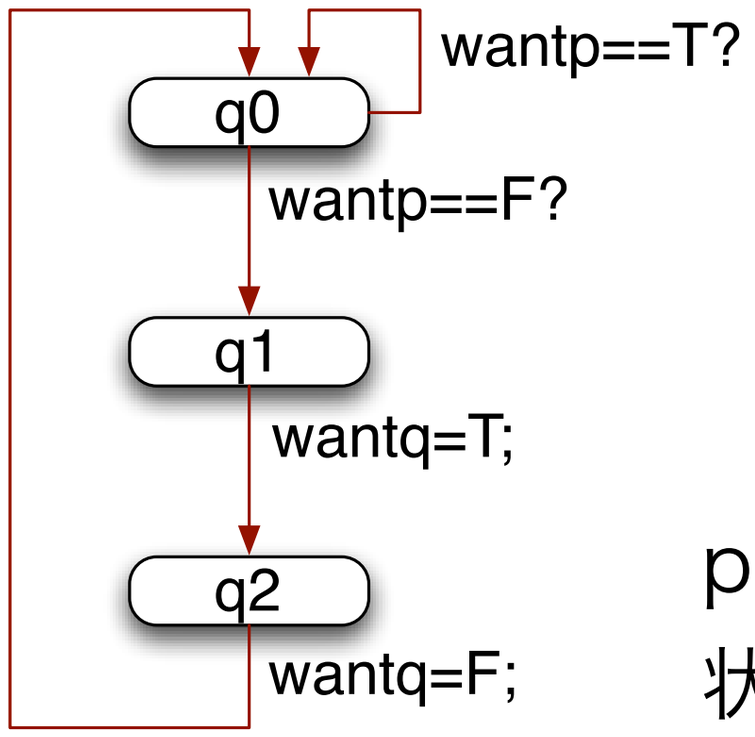
アルゴリズム2の安全性(1)

```
while (true) {
   p0: while (wantq);
   p1: wantp = true;
   p2: wantp = false;
}
```

スレッドpの簡略化

(qも同様)

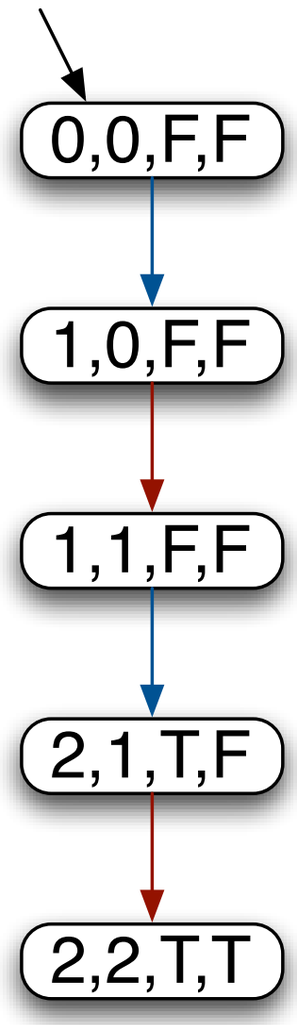




p, q それぞれの

状態遷移図

アルゴリズム2の安全性(2)



- アルゴリズム0,1の場合と同様に、両スレッドが同時にCSに入った状態(この場合はp2∧q2)に初期状態から到達可能でないことを示せばよい.
- しかしこのアルゴリズムでは，左のような実行系列が存在する．よって安全ではない.
  - ここで 2,1,T,F は各スレッドの状態が p2, q1 で，変数 wantp, wantq の値がそれぞれ true, false である状態

アルゴリズム3

```
// shared variables
bool wantp = false, wantq = false;
```

```
// thread p
while (true) {
  p0: NC
  p1: wantp = true;
  p2: while (wantq);
  p3: CS
  p4: wantp = false;
}
```

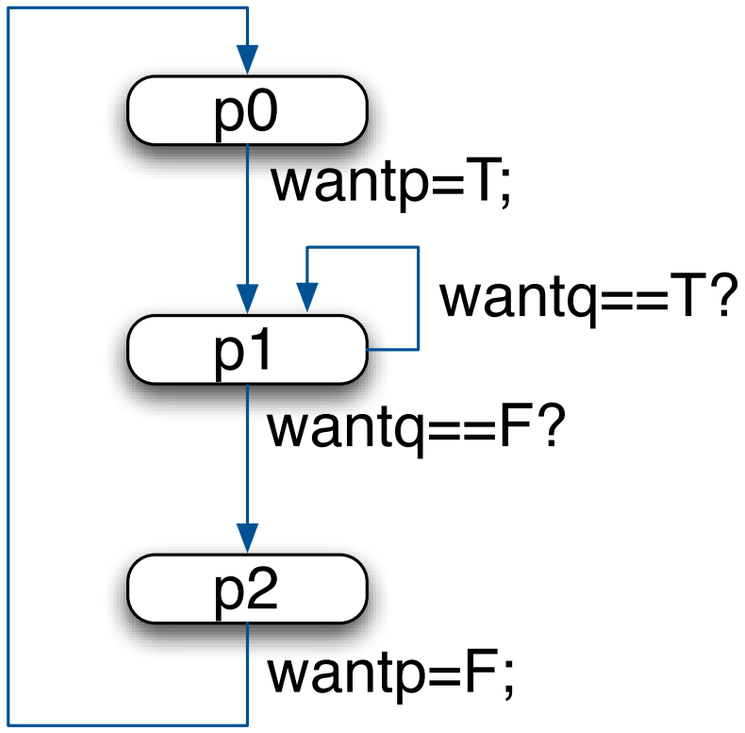
```
// thread q
while (true) {
  q0: NC
  q1: wantq = true;
  q2: while (wantp);
  q3: CS
  q4: wantq = false;
}
```

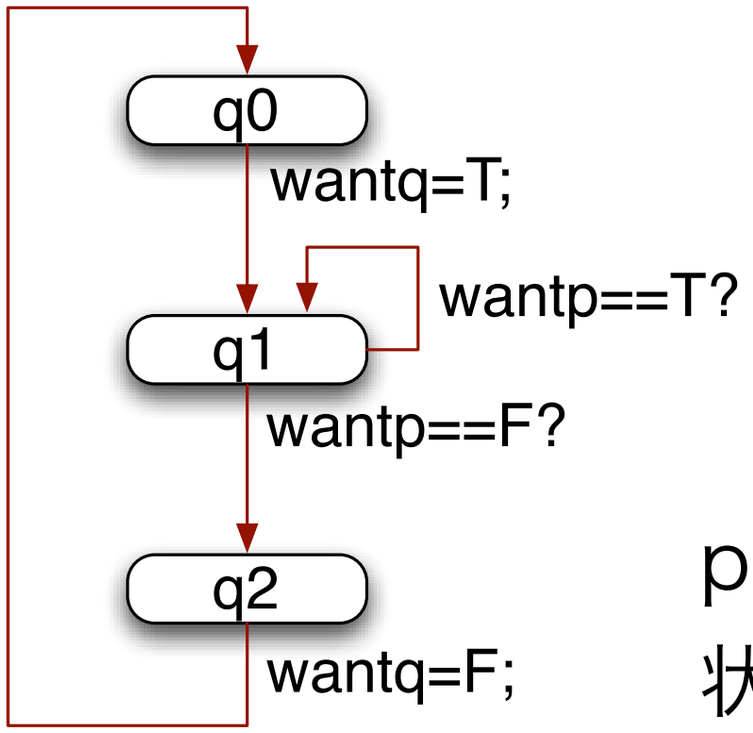
アルゴリズム２のp1(q1)とp2(q2)を交換したもの.

アルゴリズム3の安全性(1)

```
while (true) {
  p0: wantp = true;
  p1: while (wantq);
  p2: wantp = false;
}
```

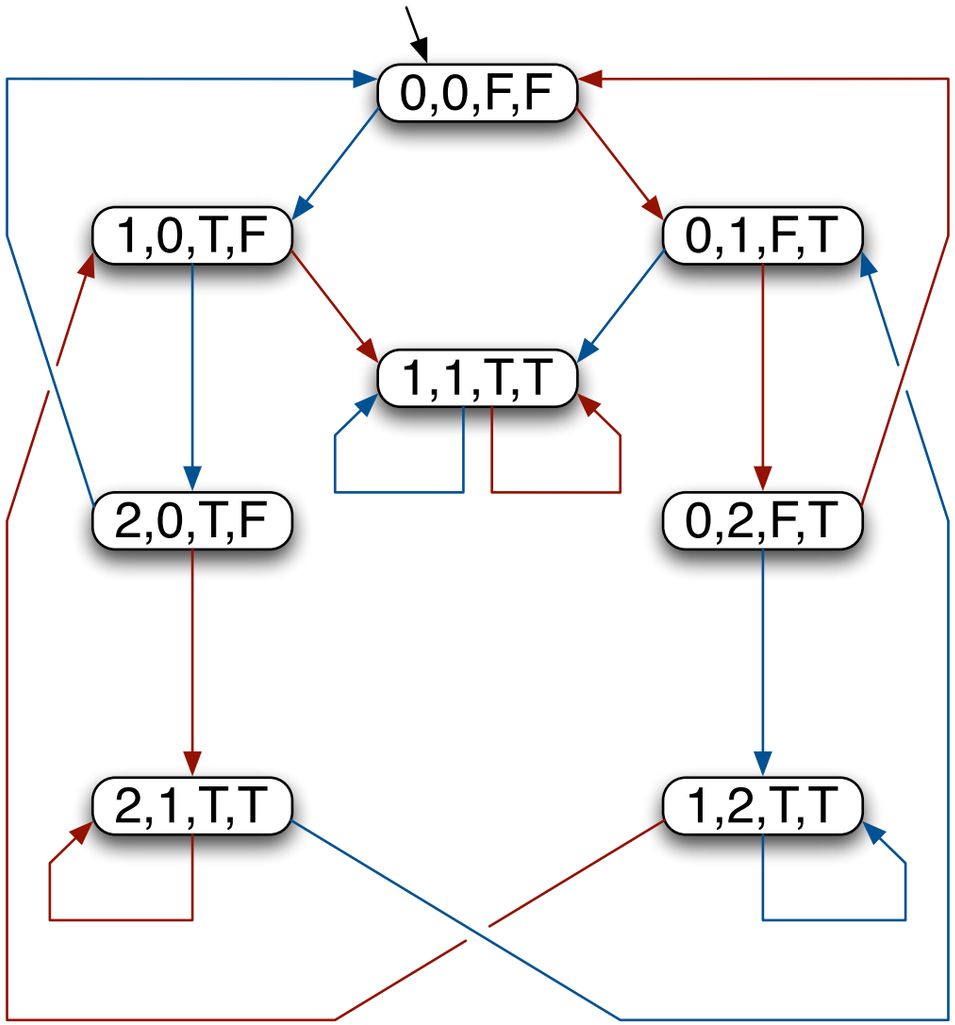
スレッドpの簡略化
(qも同様)





p, q それぞれの
状態遷移図

アルゴリズム3の安全性(2)



- 合成した系は左のようになる．この系では，初期状態から到達可能な状態の中に p2∧q2 をみたしているものはない．よって安全性をみたしている．

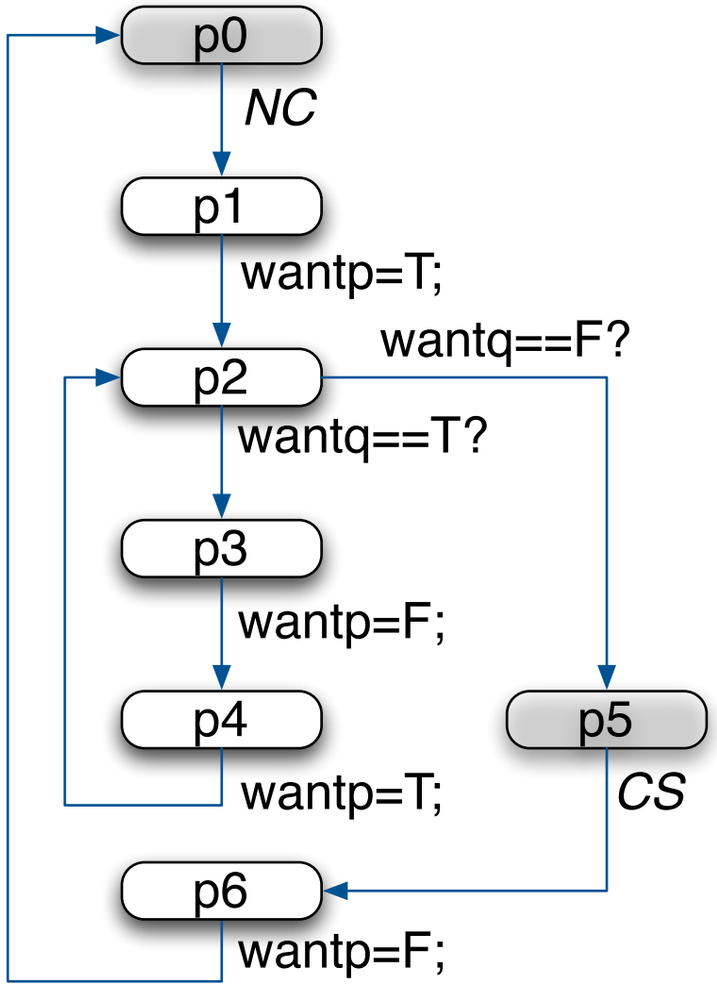
アルゴリズム3の活性

- アルゴリズム3はデッドロックを起こす可能性がある.
  - 理由：合成した系の状態 (1,1,T,T) からは他のどの状態にも遷移できない. この状態はCSに入る前であるため, いずれのスレッドもCSに入れないことになる.
    - これはライブブロック(live lock)とも呼ばれる.
  - 原因：いったん wantp (wantq) を真にした後, スレッドはCSを抜けるまでけっして偽にしない（≒CSに入るのに固執する）ため.

アルゴリズム４

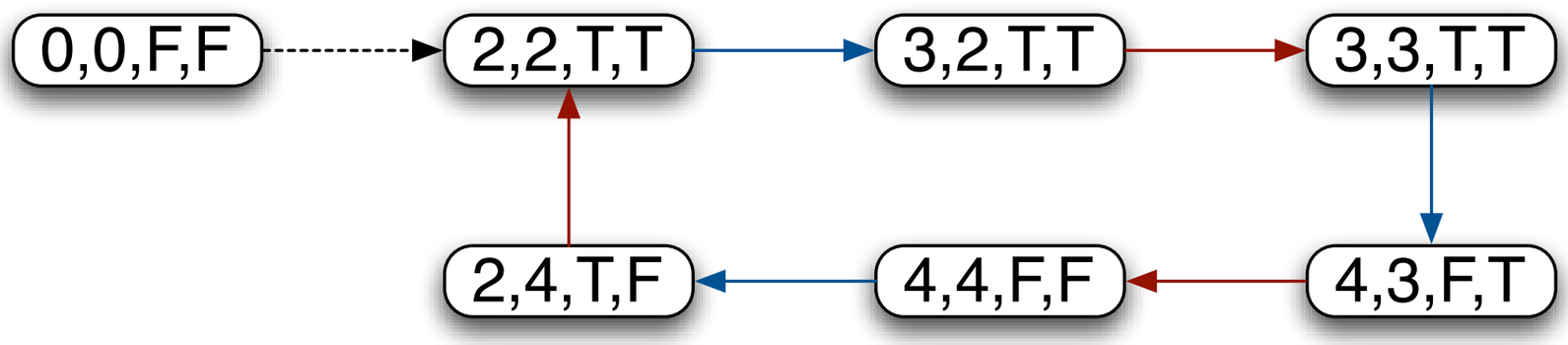
```
// shared variables
bool wantp = false, wantq = false;
```

```
// thread p
while (true) {
   p0: NC
   p1: wantp = true;
   p2: while (wantq) {
   p3:       wantp = false;
   p4:       wantp = true;
       }
   p5: CS
   p6: wantp = false;
}
```



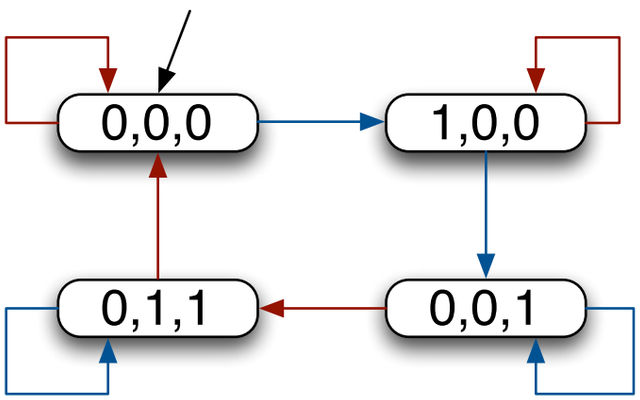
アルゴリズム4の安全性と活性

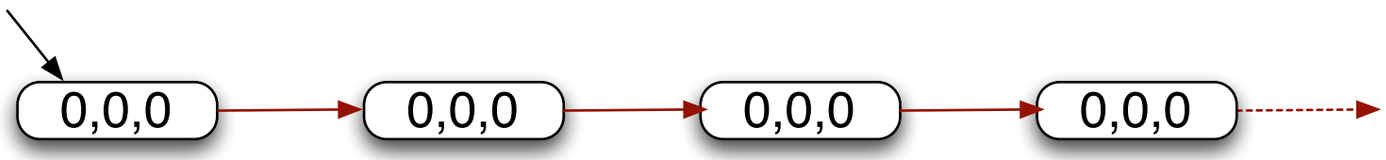
- 安全性
  - 初期状態から p5∧q5 をみたす状態には到達できない.
- 活性
  - 以下のような実行系列で「ループ」して，２つともCSに入れないことがある（飢餓）.



アルゴリズム4における「飢餓」

- アルゴリズム4のループは，命令のスケジュールの仕方によって抜け出すことができる．
  - アルゴリズム3のデッドロックの場合は，どうやっても抜け出すことはできない．
- 疑問：スケジューリングによって飢餓が起るのであれば，アルゴリズム1において下右のような実行系列による飢餓も起り得るのでは？





公平性(fairness)

- ２つ以上のスレッドが実行可能であるとき，どのスレッドもいつかは必ず実行されると仮定する．この仮定を公平性と呼ぶ.
  - 弱い公平性(weak fairness)
- 以後，スレッドの実行は公平であるとする.
  - 前ページのアルゴリズム1の実行系列は公平ではないため，このような実行は考えない.
  - 一方，アルゴリズム4の飢餓は公平性を満たしていても起こり得る.

アルゴリズム0～４のまとめ

アルゴリズム	相互排除	デッドロック	飢餓
0	×		
1	○		×
2	×		
3	○	×	
4	○	○	×

Dekkerのアルゴリズム

```
// shared variables
bool wantp = false, wantq = false;
int turn = 0;  // 0 or 1
```

```
// thread p
while (true) {
  p0: NC
  p1: wantp = true;
  p2: while (wantq) {
  p3:     if (turn == 1) {
  p4:         wantp = false;
  p5:         while (turn == 1);
  p6:         wantp = true;
          }
      }
  p7: CS
  p8: turn = 1;
  p9: wantp = false;
}
```

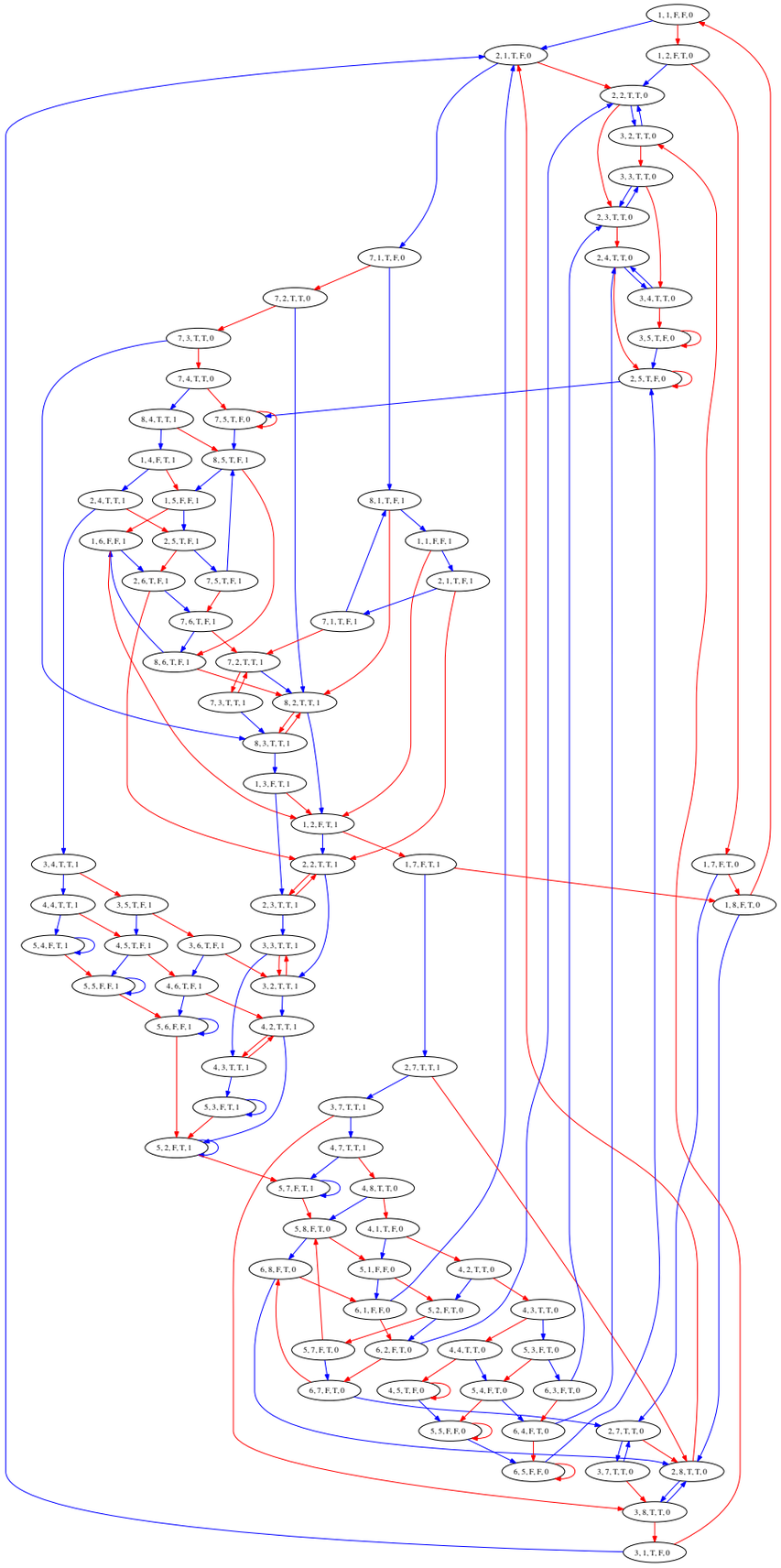
Petersonのアルゴリズム

```
// shared variables
bool wantp = false, wantq = false;
int last = 0;  // 0 or 1
```

```
// thread p
while (true) {
  p0: NC
  p1: wantp = true;
  p2: last = 0;
  p3: while (wantq && last == 0);
  p4: CS
  p5: wantp = false;
}
```

Dekker/Pertersonのアルゴリズム

- どちらのアルゴリズムも，安全性と活性をみたしている.
  - Petersonのアルゴリズムのp3における条件式では，2つの変数の読み出しがおこなわれているが，これらはアトミックでなくてもよい.



Dekkerのアルゴリズムの状態遷移図

アトミックな命令による相互排除

- いままでの仮定
  - 1個の変数に対する読み出しと書き込みのみがアトミックである.
- この仮定では，条件式で読み出した共有変数の値が次の文を実行する前に書き換えられてしまうおそれがあった.
- もし，複数の読み書きを組み合わせたものがアトミックに実行できるのであれば，相互排除はもっと容易に実現できるのではないか？

Test and Set

- 関数 test_and_set は，引数で参照される共有変数の読み出しと書き込みをアトミックにおこなう．

```
atomic bool test_and_set(bool *var) {
    bool tmp = *var;
    *var = true;
    return tmp;
}
```

test_and_setによる相互排除

```
// shared variables
bool in_use = false;
```

```
// thread p
while (true) {
  p0: NC
  p1: while (test_and_set(&in_use));
  p2: CS
  p3: in_use = false;
}
```

アルゴリズム0に似ているが，変数 in_use の参照とセットとアトミックに行っているために他のスレッドによって書き換えられず，相互排除が実現できる.

Compare and Swap (CAS)

```
atomic bool
compare_and_swap(bool *var, bool old, bool new) {
    bool tmp = *var;
    if (*var == old)
        *var = new;
    return tmp;
}
```

- いくつかのプロセッサで機械語命令として用意されており，マルチプロセッサ環境における相互排除に用いられている.
  - test_and_set(var)は compare_and_swap(var, false, true)で実現可能

Exchange

```
atomic bool exchange(bool *var, bool new) {
    bool tmp = *var;
    *var = new;
    return tmp;
}
```

- これも機械語命令として用意されていることが多い（例:x86のxchgl等）.
  - Xv6のカーネルで用いられている.
- test_and_set(var)はexchange(var, true) で実現できる.

__sync_lock_test_and_set

```
type __sync_lock_test_and_set(type *p, type v);
```

- GCCで利用可能なアトミック操作
  - 副作用：pが指す先にvを代入
  - 戻り値：pが指していた代入前の値
    - test_and_setという名前が付いているが内容はexchange

- type
  - 整数型(int, unsigned int, long, ...)

スピンロック(spin lock)

- いままで紹介したアルゴリズムでは，条件がみたされるまでループで待っている．このような待ち状態の実現方法をスピンロックと呼ぶ
  - 繁忙待機(busy waiting)と同じ概念
- 利用
  - いわゆるロックフリーアルゴリズムやユーザスレッドの実現
    - カーネルモードに移行しなくてよいため
  - カーネル内のロックに用いられることがある

スリープロック(sleep lock)

- 条件を満たすまでSLEEPING(WAITING)状態で待つ
  - スピンロックのように無駄にCPU時間を消費しないため，多くのOSで（ユーザプログラムの）プロセスやスレッドのロックに用いられる
  - カーネルによるサポートが必要

xv6：カーネルのロック機構

- struct spinlock
  - スピンロックのための構造体
- void acquire(struct spinlock *lk);
  - lkが指しているロックを確保する
- void release(struct spinlock *lk);
  - lkが指しているロックを解放する

xv6：spinlock構造体 (spinlock.h)

```
// Mutual exclusion lock.
struct spinlock {
    uint locked;       // Is the lock held?

    // For debugging:
    char *name;        // Name of lock.
    struct cpu *cpu;   // The cpu holding the lock.
};
```

- フィールドlockedへの不可分な操作によってロックを実現する
- 他のフィールドはデバッグ用

xv6: acquire

```
void acquire(struct spinlock *lk) {
    // 割り込みを禁止する
    push_off();
    // すでにロックを確保してるはずはない
    if (holding(lk))
        panic("acquire");
    // スピンロックによる待ち
    while (__sync_lock_test_and_set(&lk->locked, 1) != 0)
        ;
    // フェンス
    __sync_synchronize();
    // ロックを確保しているCPUコアを記録
    lk->cpu = mycpu();
}
```

xv6: release

```
void release(struct spinlock *lk) {
    // ロックを確保してるはず

    if (!holding(lk)) panic("release");
    // ロックを確保していたCPUコアの記録を削除

    lk->cpu = 0;
    // フェンス

    __sync_synchronize();
    // lk->locked = 0; をアトミックに行う

    __sync_lock_release(&lk->locked);
    // 割り込み禁止解除

    pop_off();
}
```

CPUの管理(1)

```
// CPUコア毎の状態
struct CPU {
    // このCPUが実行しているプロセス
    struct proc *proc;
    // カーネルスレッド（スケジューラ）のコンテキスト
    struct context scheduler;
    // 割り込み禁止の深さ
    int noff;
    // 割り込み禁止／許可フラグ
    int intena;
};
```

```
// 全CPUコアの状態
struct cpu cpus[NCPU];
```

CPUの管理(2)

```
// 実行中のCPU番号（レジスタtpに格納されている）
int cpuid() {
    int id = r_tp();
    return id;
}

// 実行中のCPU構造体へのポインタ
struct cpu *mycpu() {
    int id = cpuid();
    struct cpu *c = &cpus[id];
    return c;
}
```

割り込みの禁止と許可

- push_off() / pop_off()
  - 割り込みを禁止する
  - push_offを実行した回数だけ実行すると，1回目にpush_offを実行する直前の割り込み許可/禁止状態に戻る
- intr_off() / intr_on()
  - 実行したCPUコアの割り込みを禁止/許可する
- intr_get()
  - 実行中のCPUコアの割り込み許可/禁止状態

push_off

```
void push_off() {
    // CPUの割り込みフラグ
    int old = intr_get();
    // 割り込みを禁止する
    intr_off();
    // 1回目のpush_offの実行時点の割り込みフラグを記録
    if (mycpu()->noff == 0)
        mycpu()->intena = old;
    // CPU毎にpush_offを実行した回数をカウント
    mycpu()->noff += 1;
}
```

pop_off

```
void pop_off() {
    struct cpu *c = mycpu();
    // この時点では割り込み可能になっていないはず
    if (intr_get())
        panic("pop_off - interruptible");
    // CPU毎のpush_offのカウントを1減らす
    c->noff -= 1;
    // noffの値は0以上のはず
    if (c->noff < 0)
        panic("pop_off");
    // noffが0かつ1回目のpush_offの時点で割り込み可なら
    // 再度割り込みを許可する
    if (c->noff == 0 && c->intena)
        intr_on();
}
```

まとめ

- 同期と排他制御(1)
  - 同期，アトミック性，状態遷移図によるモデル化
  - 相互排除，デッドロック，飢餓，公平性
  - Dekkerのアルゴリズム, Petersonのアルゴリズム
  - TS, CAS, XCHG命令
  - スピンロック，スリープロック
  - xv6の同期機構