オペレーティングシステム

第5回

プロセスの相互排除

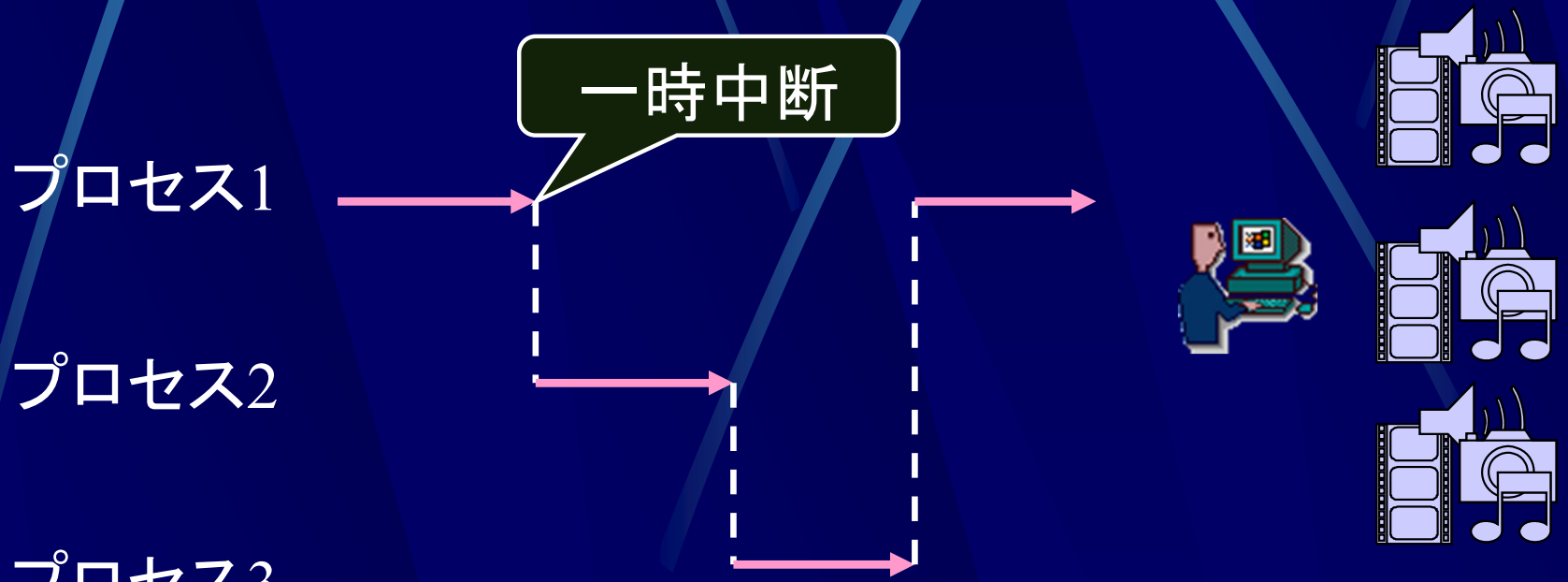
http://www.info.kindai.ac.jp/OS

E号館3階E-331 内線5459

takasi-i@info.kindai.ac.jp

プロセスの並行処理

- 並行処理(concurrent processing)
  - 複数のプロセスを(見かけ上)同時に実行



ユーザにとっては
3つのプロセスが
同時に実行されている

プロセスの並行処理

- プロセスの並行処理で起こること
  - プロセス協調
    - 仕事の分担や通信等，複数のプロセスが助け合う
  - プロセス競合
    - 複数のプロセスで有限のリソースを取り合う
    - プロセス間の調停，リソース割り当て制御が必要
  - プロセス干渉
    - 他プロセスの影響で異常が発生すること
    - 原因はプログラムのバグ

プロセスの並行処理

どんなプロセスでも並行処理できるのか？

プロセス1

```
x: = 0;
```

プロセス2

```
y: = 1;
```

プロセス3

```
print (x);
```

プロセス1とプロセス2は
並行処理可能
(どちらを先にしてもかまわない)

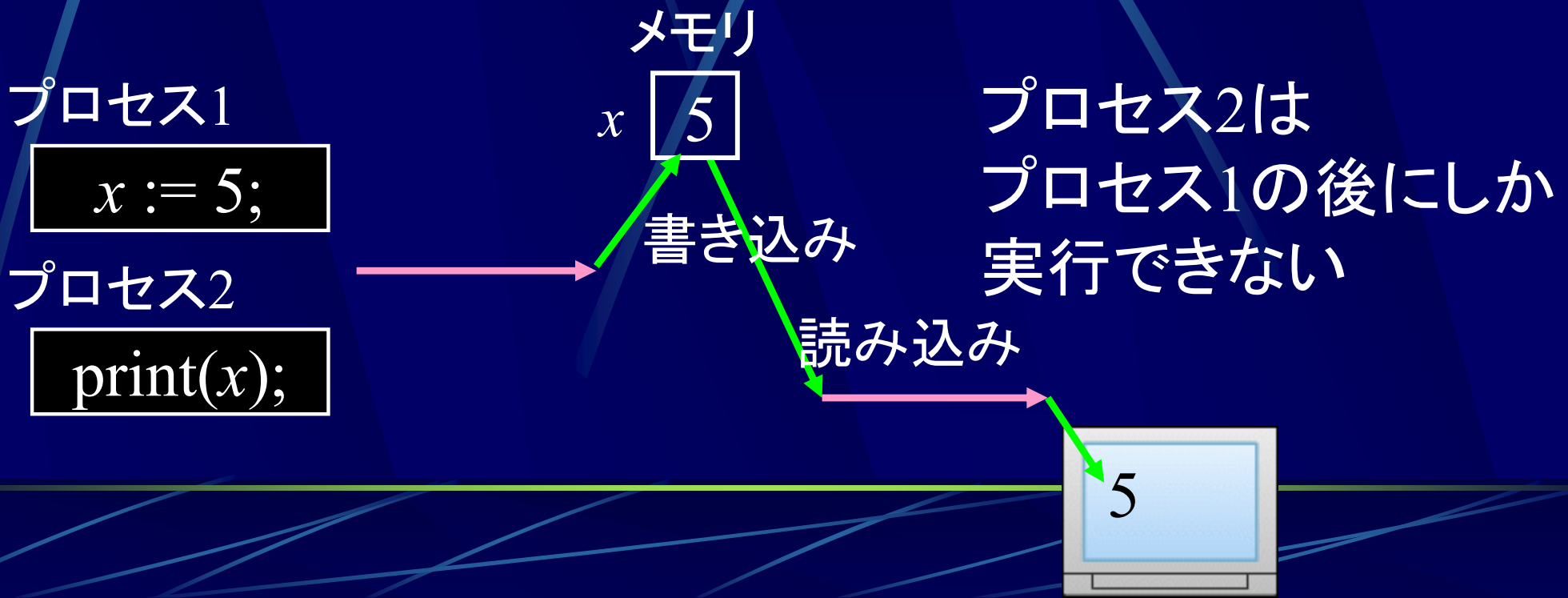
プロセス1とプロセス3は
プロセス1を先にしないといけない

並行プロセス, 逐次プロセス (concurrent process, sequential process)

- 並行プロセス(concurrent process)
  - 同時に実行可能なプロセス群
  - プロセスの実行順序に依存しない
- 逐次プロセス(sequential process)
  - 同時に実行することが不可能なプロセス群
  - プロセス間に依存関係がある

逐次プロセス
(sequential process)

- 逐次プロセス(sequential process)
  - 同時に実行することが不可能なプロセス群
  - プロセス間に依存関係がある



並行プロセス
(concurrent process)

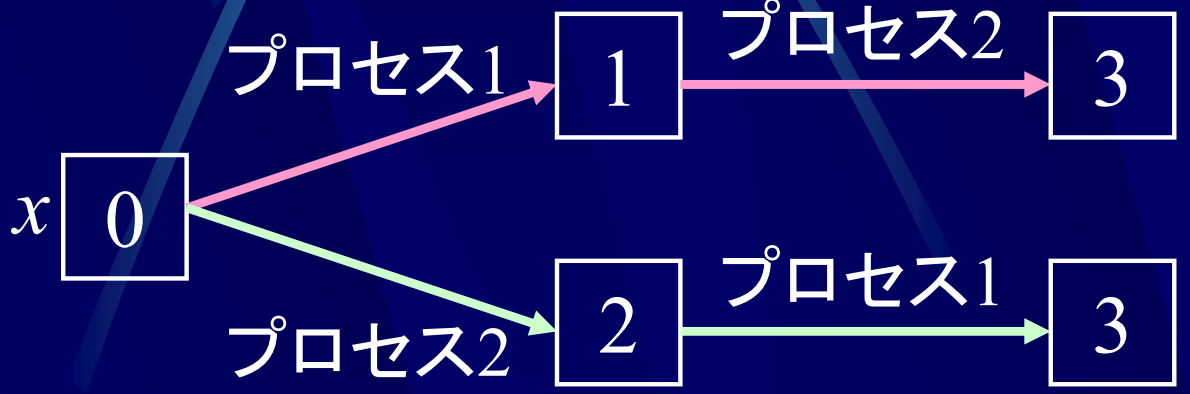
- 並行プロセス(concurrent process)
  - 同時に実行可能なプロセス群
  - プロセスの実行順序に依存しない

プロセス1

$x := x + 1;$

プロセス2

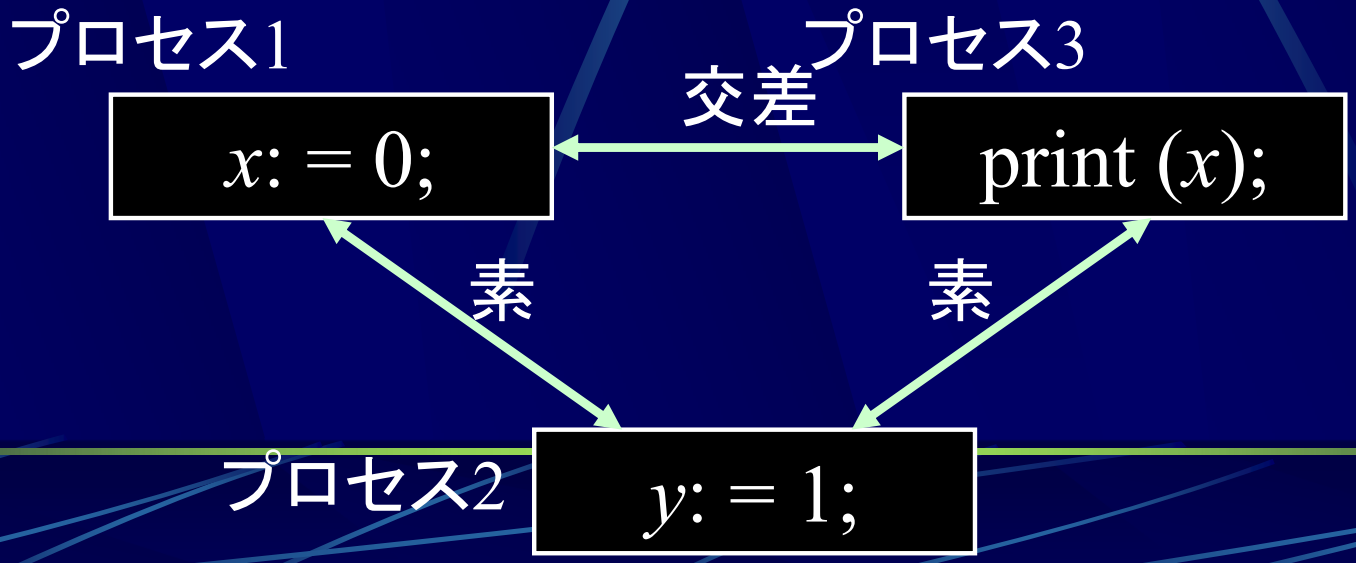
$x := x + 2;$



どちらを先に実行しても結果は同じ

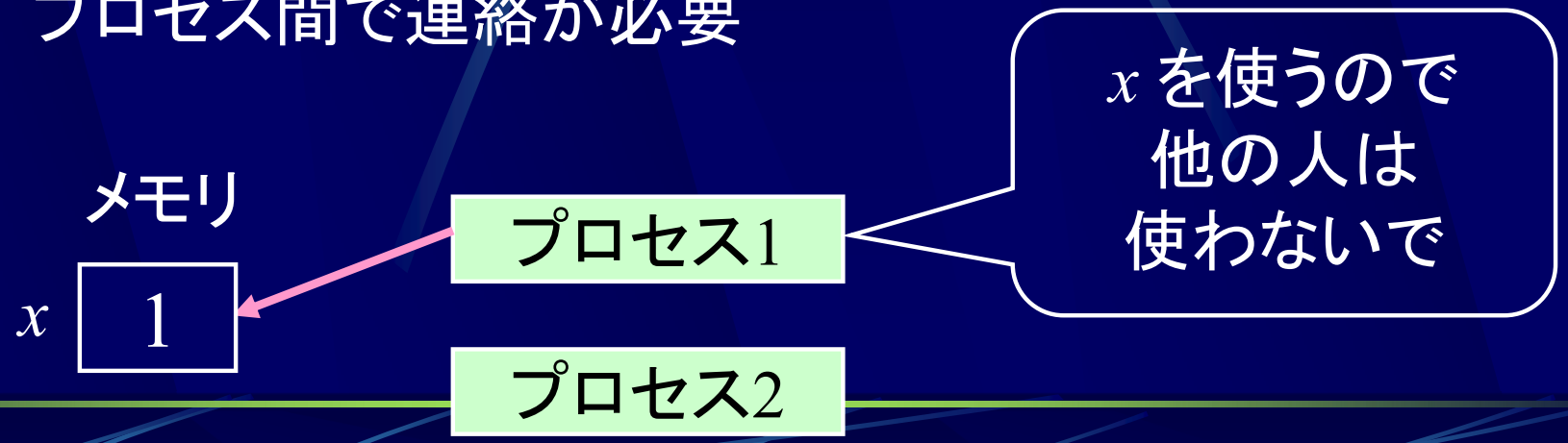
素, 交差
(disjoint, overlapping)

- 素(disjoint)
  - 共通に操作するデータが存在しないプロセス
- 交差(overlapping)
  - 共通のデータを操作しているプロセス



協同型逐次プロセス
(cooperating sequential process)

- 協同型逐次プロセス
  (cooperating sequential process)
  - データ,プログラム等の資源を共有する(交差している)並行プロセス群
    - 高々1つのプロセスが資源を占有するようにプロセス間で連絡が必要

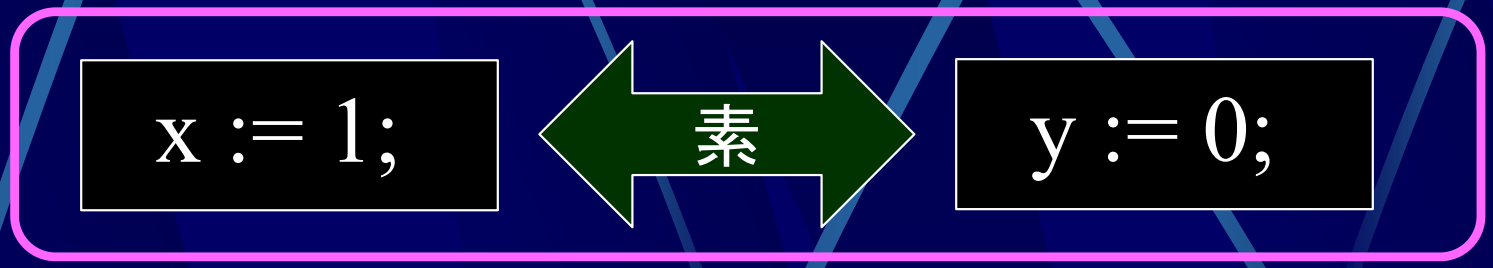


プロセス群の種類

プロセス群の種類		データ共有	実行順序
並行プロセス	互いに素な並行プロセス	素	同時に実行可能
並行プロセス	共同型逐次プロセス	交差	同時に実行可能だが排他制御が必要
逐次プロセス		交差	実行順序に依存関係

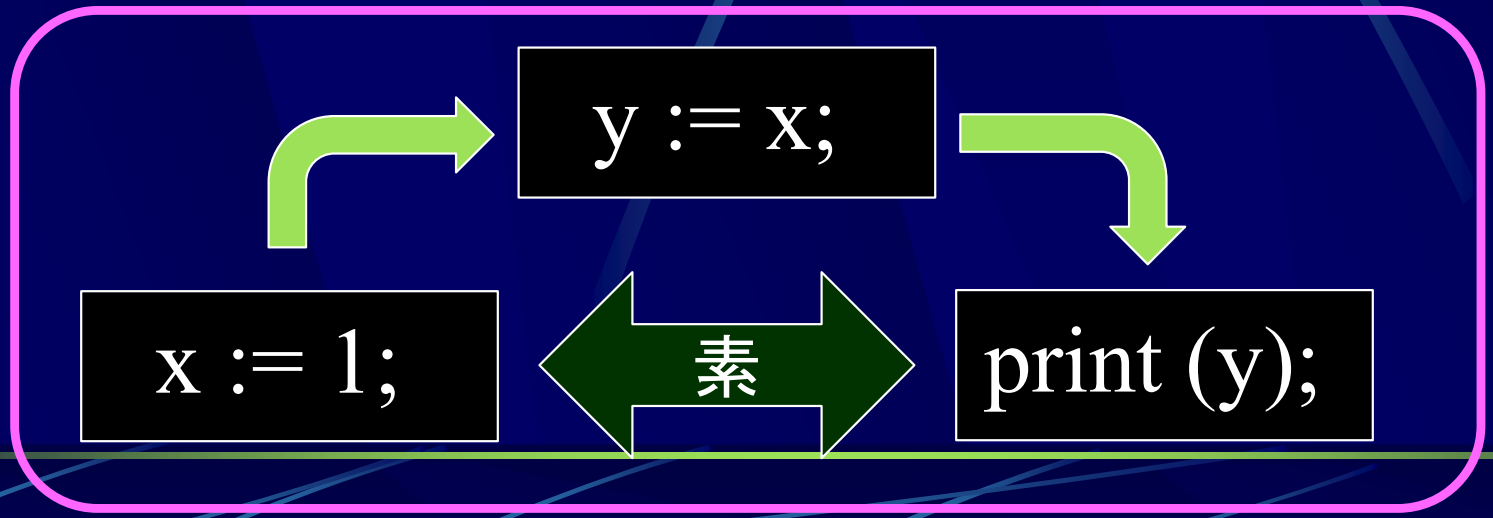
プロセス群の種類

プロセスが2個なら、互いに素なプロセスは並行プロセス



並行
プロセス

プロセスが3個以上では、互いに素でも逐次プロセスになる場合も



逐次
プロセス

共同型逐次プロセス

この2つはどちらを先に実行しても結果は同じ

プロセス1

$x := x + 1;$

プロセス2

$x := x + 2;$

しかしアセンブラレベルで見ると…?

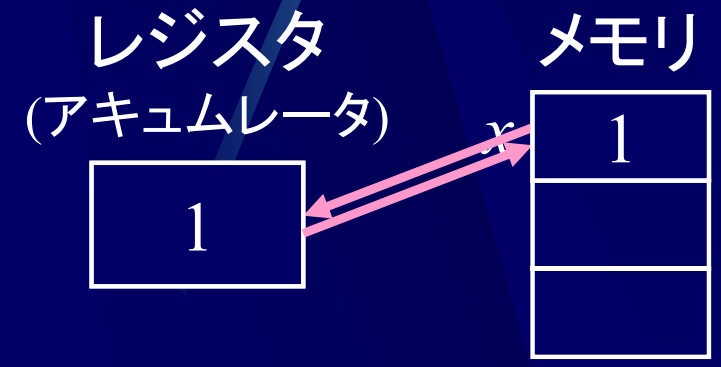
プロセス1	プロセス2
1.1 LOAD x	2.1 LOAD x
1.2 ADD 1	2.2 ADD 2
1.3 STORE x	2.3 STORE x

プロセス1,2は複数の命令から成っている

共同型逐次プロセス

プロセス1

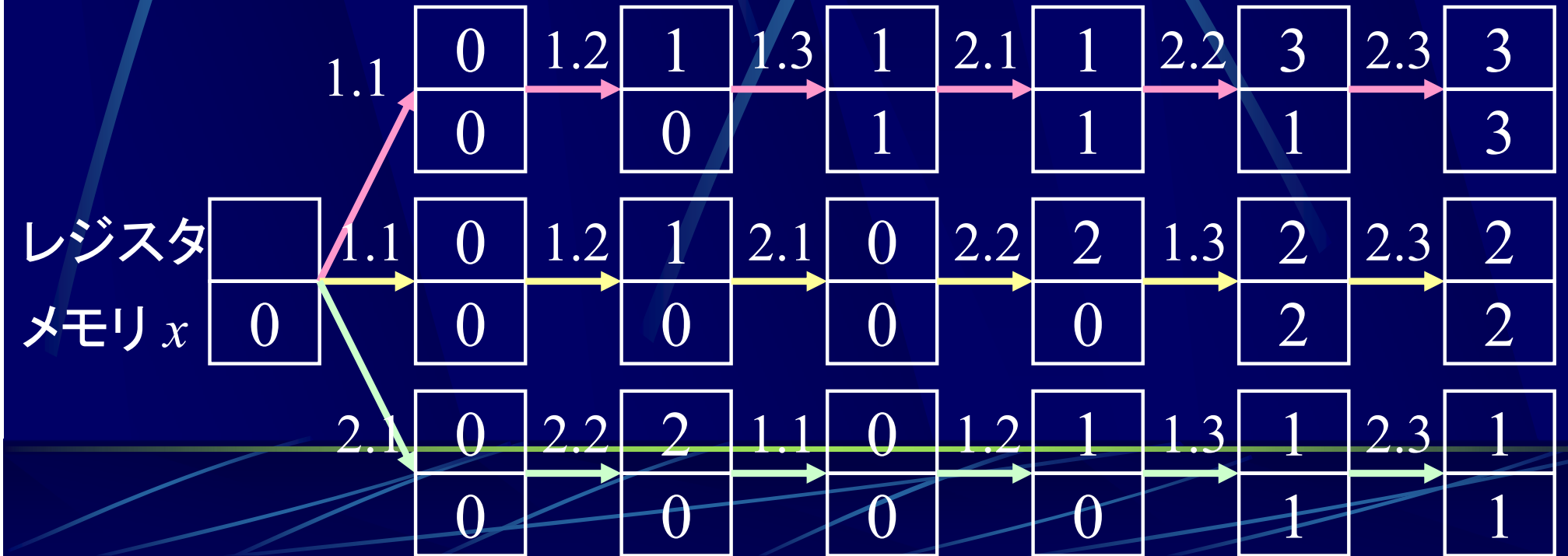
1.1 LOAD x	レジスタに x の値を読み込む
1.2 ADD 1	レジスタの値に 1 を足す
1.3 STORE x	レジスタの値を x に書き込む



共同型逐次プロセス

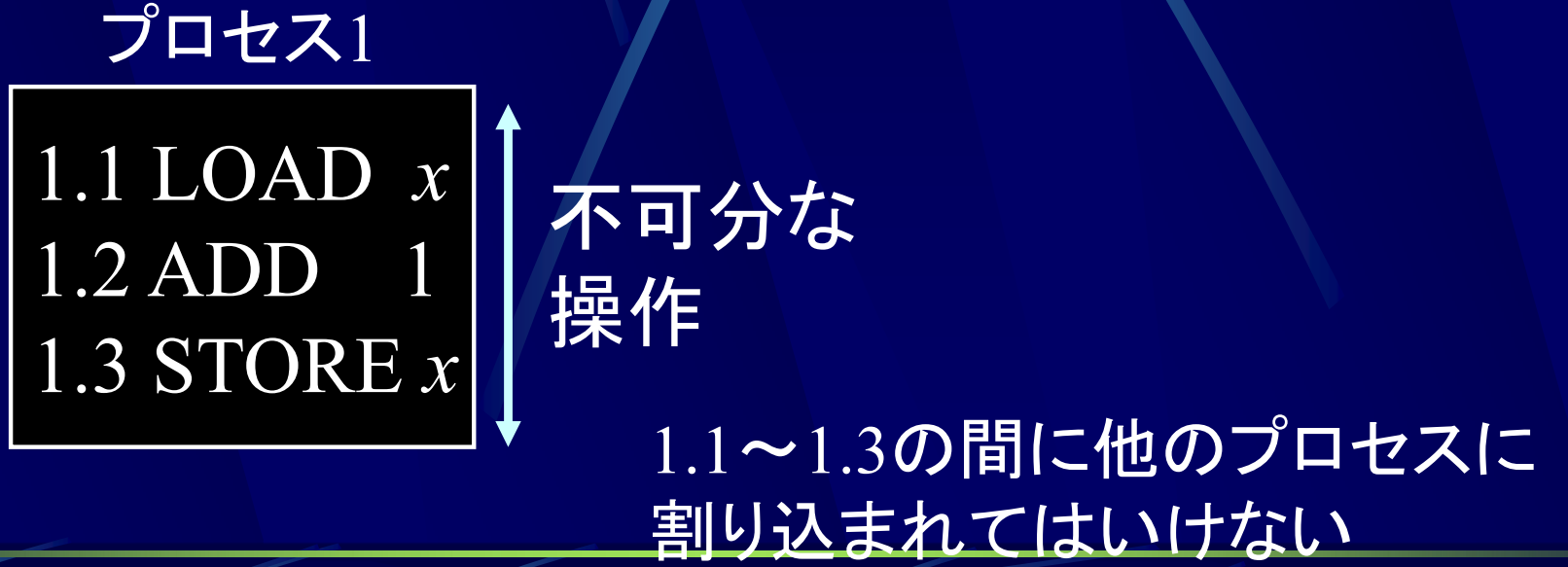
プロセス1	プロセス2
1.1 LOAD x	2.1 LOAD x
1.2 ADD 1	2.2 ADD 2
1.3 STORE x	2.3 STORE x

実行する順序により
結果が変わってしまう



不可分(indivisible)な操作

- 不可分(indivisible)な操作
  - 途中で他のプロセスに割り込まれること無しに実行される必要のある操作



共有資源, 逐次的資源
(shared resource, sequential resource)

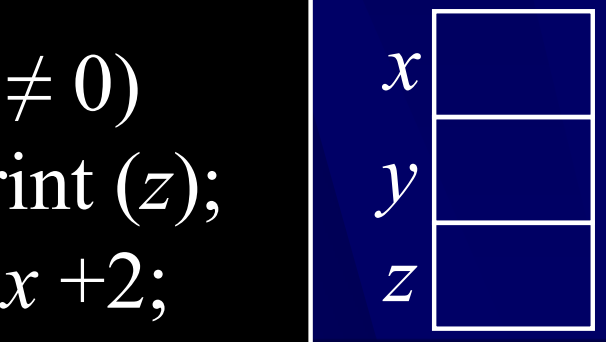
- 共有資源(shared resource)
  - プロセス間で共有されるプログラム, データ
- 逐次的資源(sequential resource)
  - 同時に1つのプロセスしか使用できない資源

プロセス1

```
y := input();
y := y +1;
x := x +1;
```

プロセス2

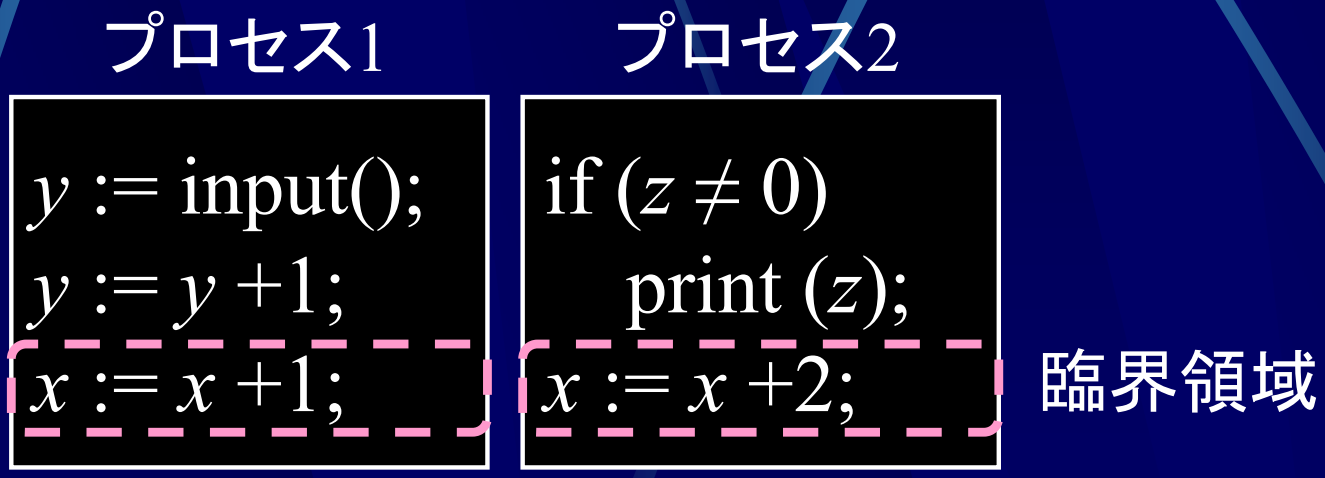
```
if (z ≠ 0)
    print (z);
x := x +2;
```



x は共有資源
かつ逐次的資源

臨界領域
(critical section, critical region)

- 臨界領域(critical section, critical region)
  - 逐次的資源を使用しているプロセスの部分



臨界領域に入るときは
他のプロセスが逐次的資源を使わないように
資源を占有する必要がある

臨界領域

WC = 臨界領域

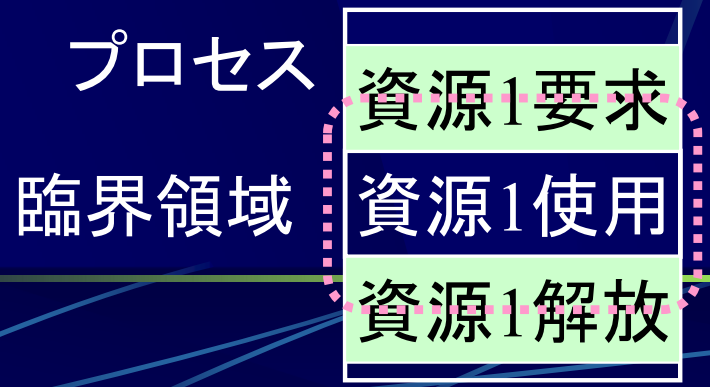
- 同時に入れるのは1人だけ
  - 2人以上が同時に入ってはいけない
- 待っていれば必ず入れる
  - 入りたい人が永久に待たされてはいけない

相互排除, 排他制御
(mutual exclusion, exclusive control)

- 相互排除(mutual exclusion),
  排他制御(exclusive control)
  - ある資源を高々1つのプロセスが占有するようにする
  - あるプロセスが資源を使用しているときは、他のプロセスは資源が解放されるまで待つ

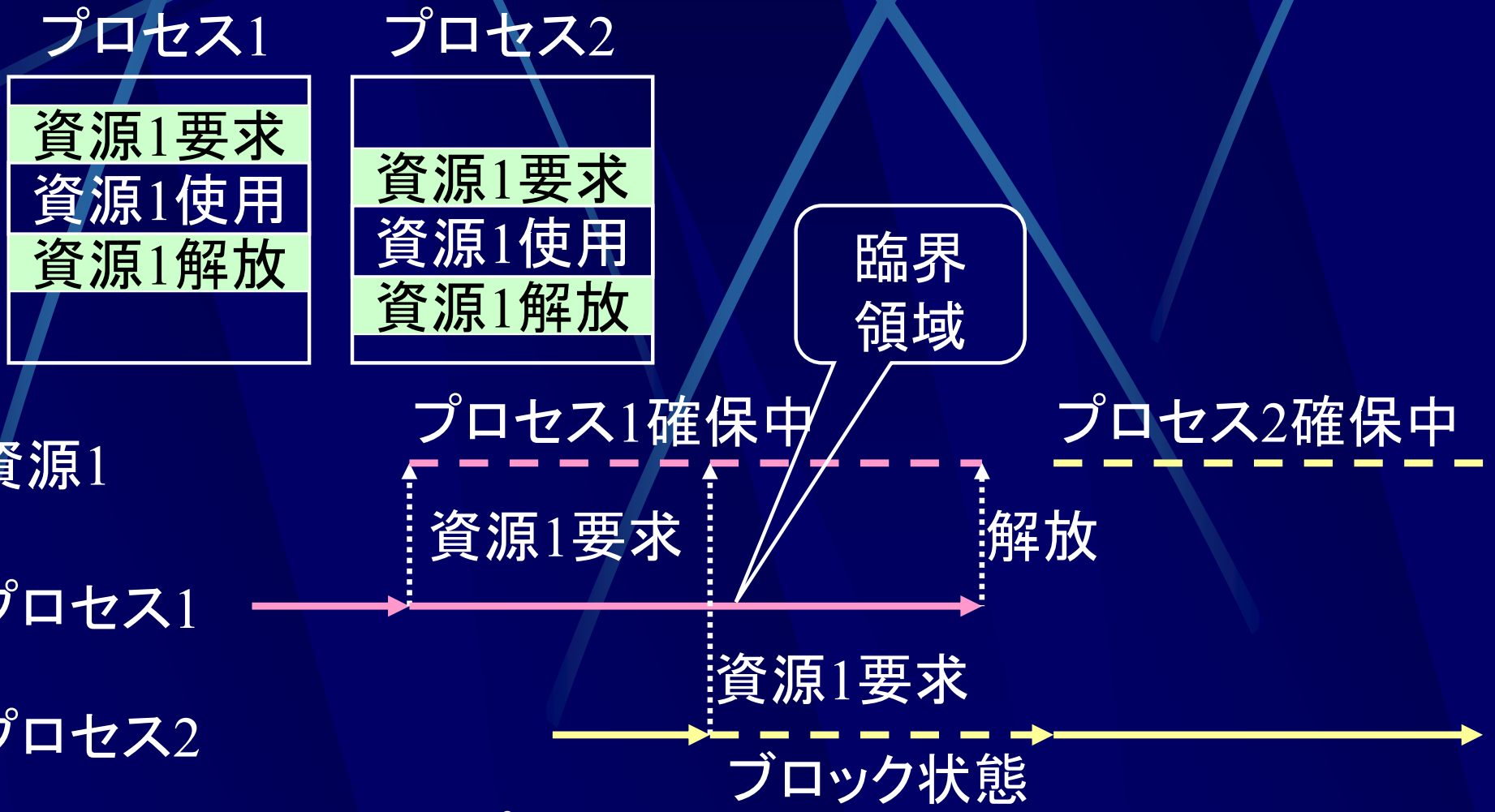
資源の要求，解放
(lock, unlock)

- 資源の要求(lock)
  - 資源を他のプロセスが使えないようにする
  - すでに他のプロセスが使っている場合はブロック状態に
- 資源の解放(unlock)
  - 資源を他のプロセスが使えるようにする



資源を使う前に資源を要求し、
使い終わったら資源を解放する

相互排除



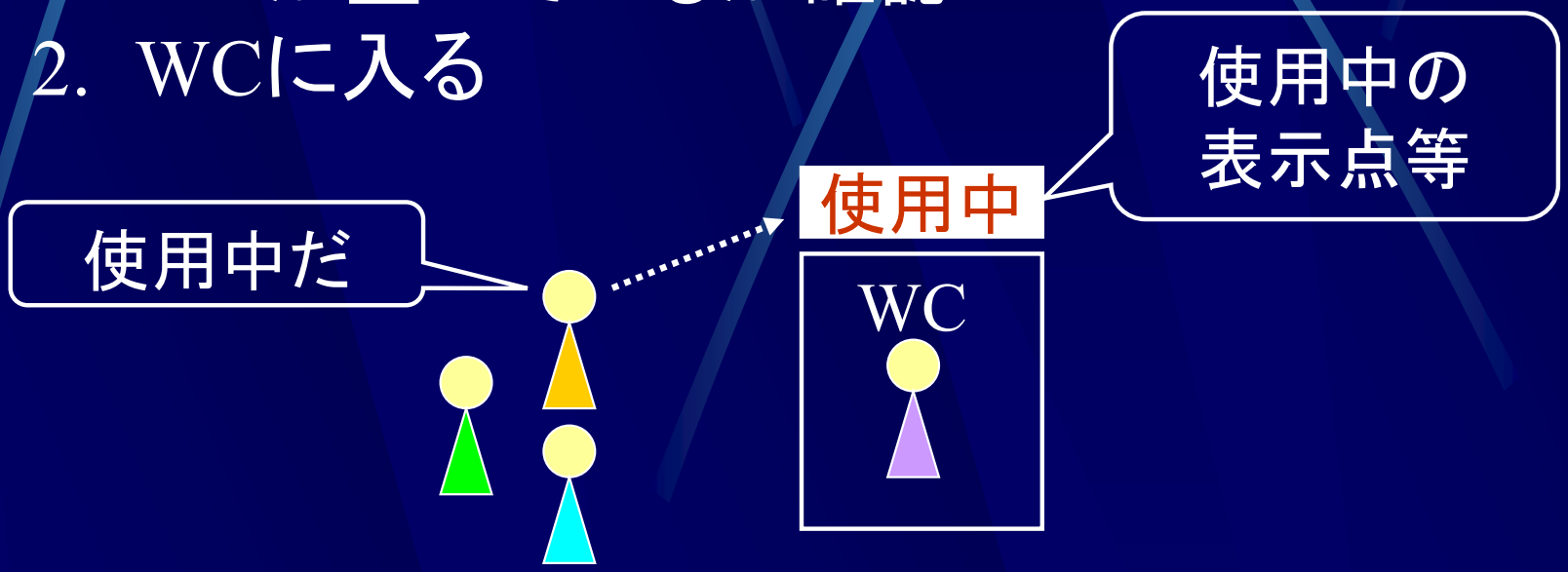
必要な資源が他のプロセスに
使用されているとブロック状態になる

相互排除

WC＝臨界領域

- 同時に入れるのは1人だけ
- 待っていれば必ず入れる

1. WCが空いているか確認
2. WCに入る



相互排除

WC＝臨界領域

- 同時に入れるのは1人だけ
- 待っていれば必ず入れる

1. WCが空いているか確認
2. WCに入る

相互排除の仮定

- 以降は、各プロセスは臨界領域(CS)と非臨界領域(NCS)を繰り返すとする

```
while (true) {
   CS();     /* 臨界領域 */
   NCS(); /* 非臨界領域 */
}
```

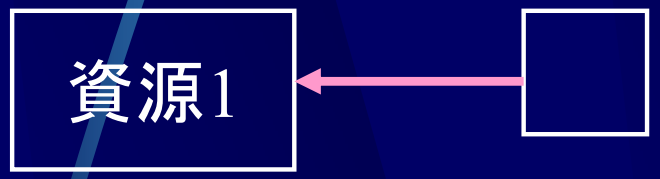
CS, NCS の実行内容は毎回異なっていてもいい

プロセスの停止は実行時間無限大の NCS と考える

フラグによる相互排除

相互排除するためには資源を他のプロセスが
使っていないかチェックが必要

資源1使用フラグ



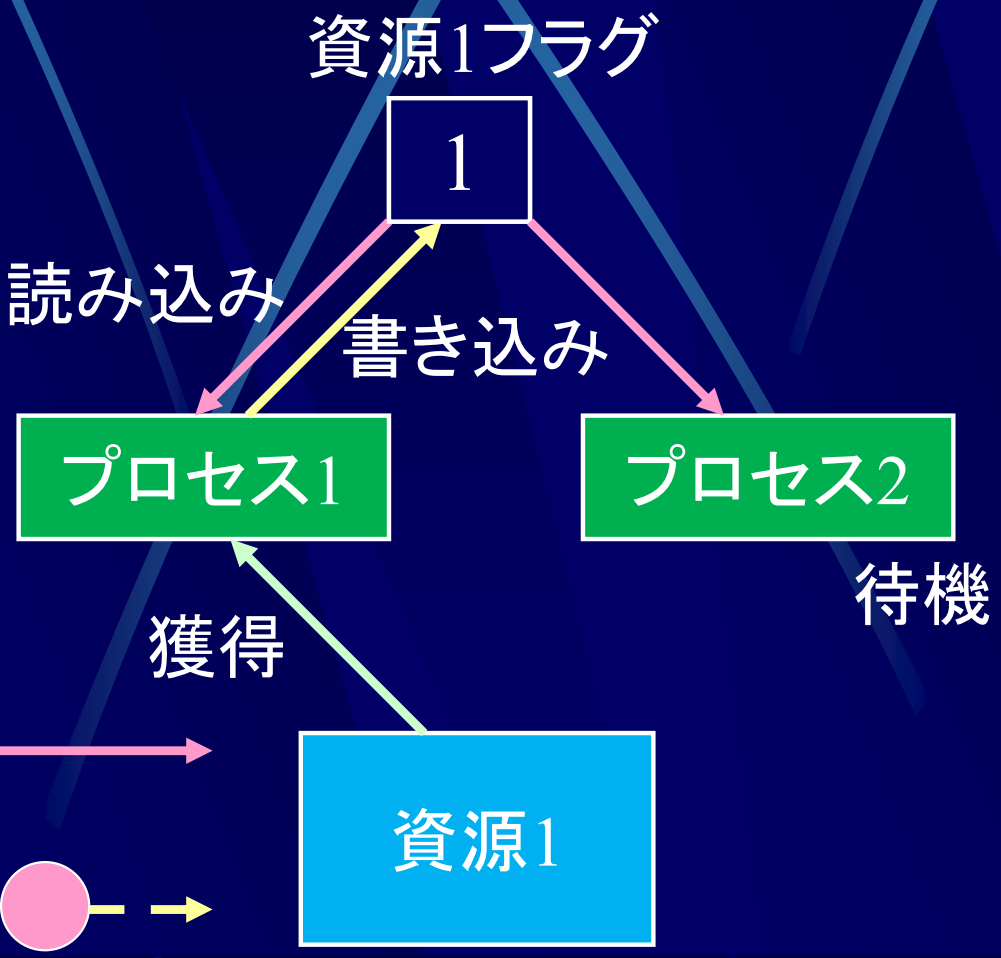
0：資源1が使用されていない
1：資源1が使用されている

```
while (flag = 1) /* フラグの値をチェック */
    wait();      /* フラグが 0 になるまで待つ */
flag := 1;       /* フラグを 1 にセット */
CS();            /* 資源を使用する臨界領域 */
flag := 0;       /* フラグを 0 にリセット */
```

これで OK ?

フラグによる相互排除

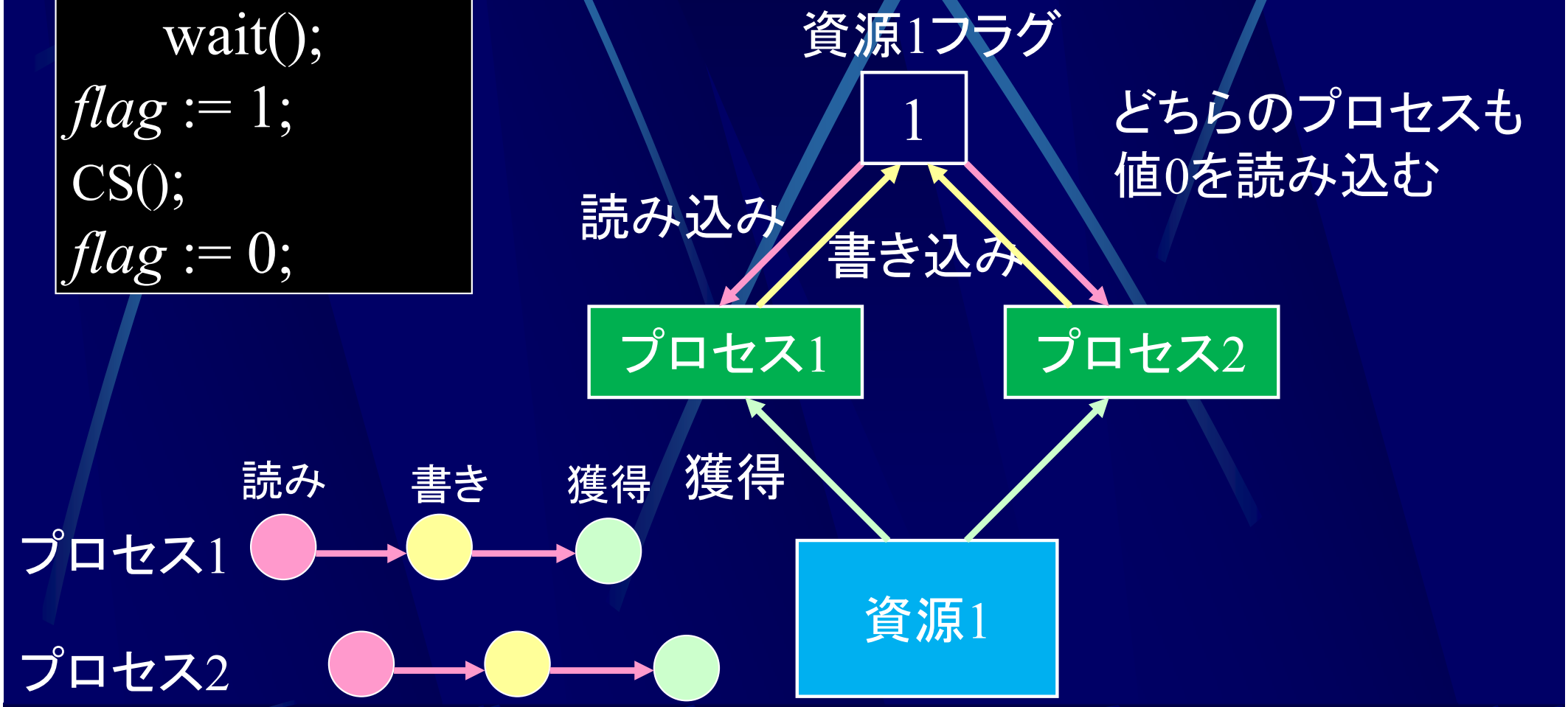
```
while (flag = 1)
    wait();
flag := 1;
CS();
flag := 0;
```



これでうまくいきそうだが…

フラグによる相互排除

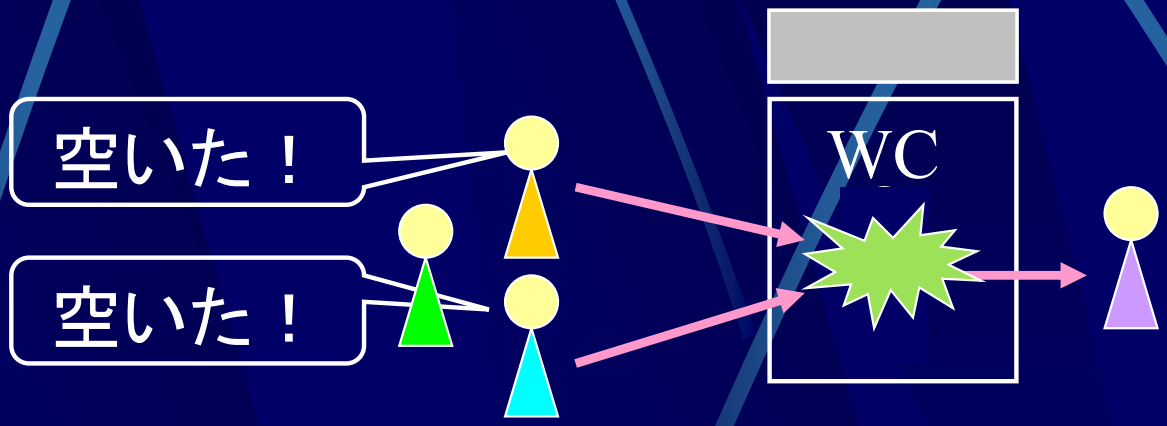
```
while (flag = 1)
    wait();
flag := 1;
CS();
flag := 0;
```



2つのプロセスが同時に資源を得てしまう

フラグによる相互排除

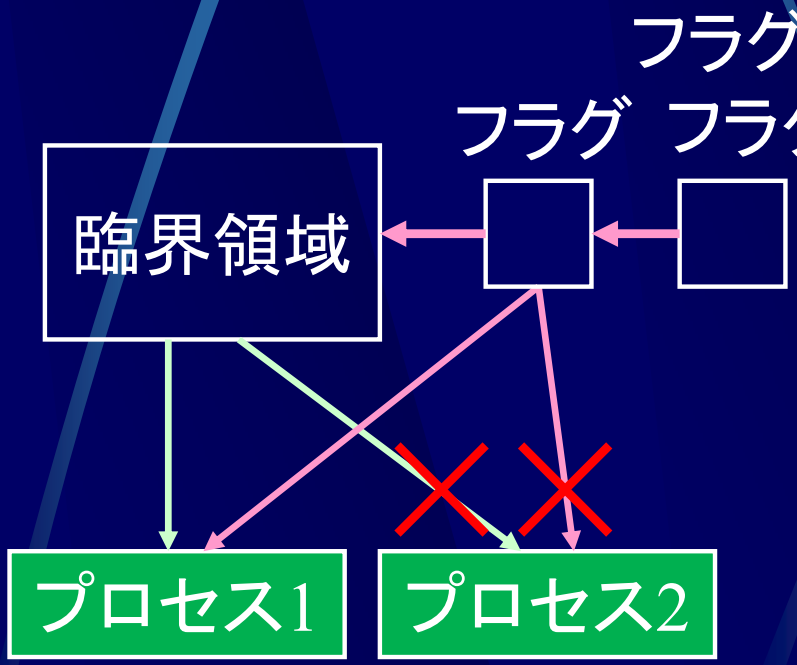
1. WCが空いているか確認
2. WCに入る



複数の人がほぼ同時に 1. をすると
WCに複数人が入ってしまう

フラグによる相互排除

臨界領域：同時には1台のプロセスしか入ってはいけない領域



フラグの読み込みと書き込みの間に
他のプロセスに割り込まれてはいけない

⇩

フラグ操作自身も臨界領域

フラグによる相互排除は
本質的に不可能

相互排除

- ソフトウェアによる相互排除
  - 相互排除アルゴリズムを使用
- ハードウェアによる相互排除
  - 機械語命令 Test and Set を使用
- 割込み禁止による相互排除
  - 割込み禁止命令を使用

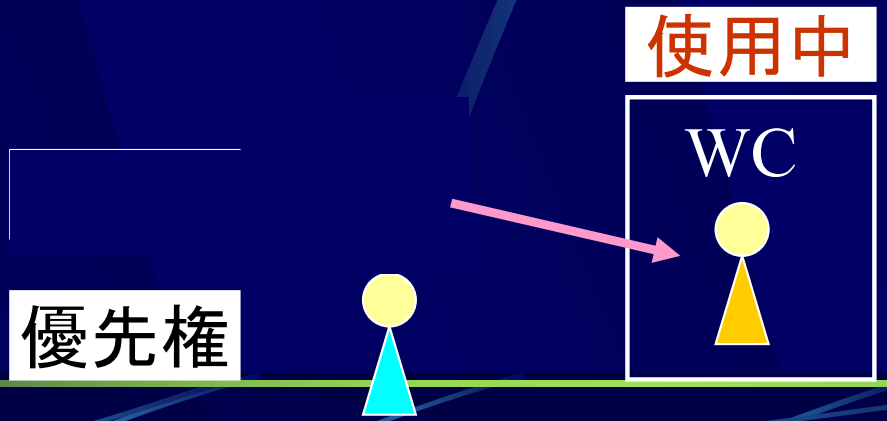
相互排除アルゴリズム

- 2プロセス間の相互排除
  - 交互実行アルゴリズム
  - Dekker のアルゴリズム
  - Peterson のアルゴリズム
- *N*プロセス間の相互排除
  - Dijkstra のアルゴリズム
  - Kuuth のアルゴリズム
  - Eisenberg と McGuire のアルゴリズム
  - Lamport のアルゴリズム

相互排除アルゴリズム
交互実行アルゴリズム

2人のうちどちらか片方が優先権を持つ

1. WCが空いているか確認
2. 優先権を持たない人は待つ
3. WCに入る
4. 相手に優先権を渡す



相互排除アルゴリズム
交互実行アルゴリズム

広域変数と初期値

```
int turn := 0;          /* 次に臨界領域に入るプロセス番号 */
```

プロセス0の臨界領域処理

```
while (turn = 1) wait(); /* プロセス1が臨界領域から出るまで待つ */
CS0();                   /* プロセス0の臨界領域 */
turn := 1;               /* 次はプロセス1が臨界領域に入る番 */
NCS0();                  /* プロセス0の非臨界領域 */
```

プロセス1の臨界領域処理

```
while (turn = 0) wait(); /* プロセス0が臨界領域から出るまで待つ */
CS1();                   /* プロセス1の臨界領域 */
turn := 0;               /* 次はプロセス0が臨界領域に入る番 */
NCS1();                  /* プロセス1の非臨界領域 */
```

相互排除アルゴリズム
交互実行アルゴリズム

プロセス0

```
while (turn = 1) wait();
CS0();
turn := 1;
NCS0();
```

プロセス1

```
while (turn = 0) wait();
CS1();
turn := 0;
NCS1();
```

CS*n* :　プロセス*n* の臨界領域
NCS*n* : プロセス*n* の非臨界領域

turn の値が0ならばプロセス0が、
1ならばプロセス1が臨界領域に入れる

プロセス0とプロセス1は交互に臨界領域を実行

欠点：片方が臨界領域を使わないときでも交互に実行
片方のプロセスがフリーズするともう片方も止まる

相互排除アルゴリズム
Dekker のアルゴリズム

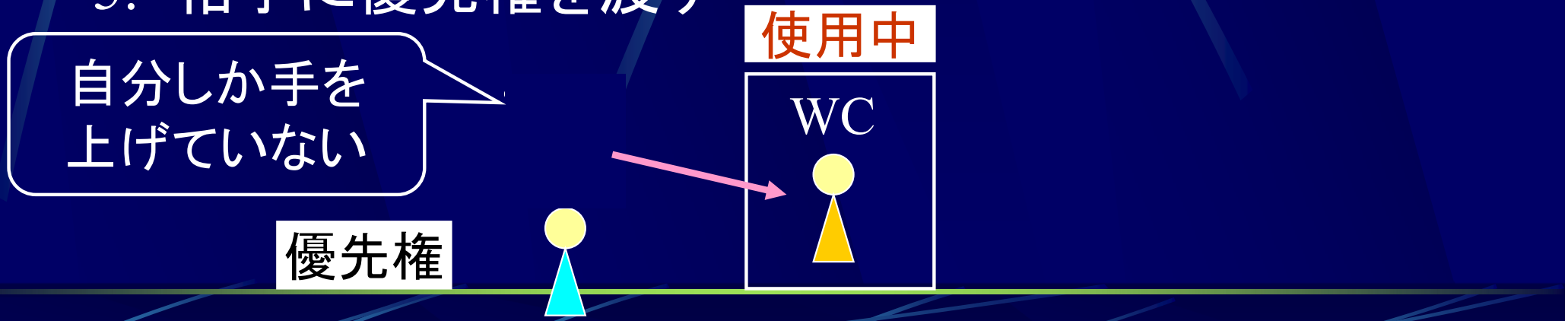
2人のうちどちらか片方が優先権を持つ

1. WCが空いているか確認
2. 手を上げる
3. 2人とも手を上げていた場合、優先権を持たない人は待つ
4. WCに入る
5. 相手に優先権を渡す

相互排除アルゴリズム
Dekker のアルゴリズム

2人のうちどちらか片方が優先権を持つ

1. WCが空いているか確認
2. 手を上げる
3. 2人とも手を上げていた場合、優先権を持たない人は待つ
4. WCに入る
5. 相手に優先権を渡す



相互排除アルゴリズム
Dekker のアルゴリズム

広域変数と初期値

```
int turn := 0;
boolean flag0:=false,
        flag1:=false;
```

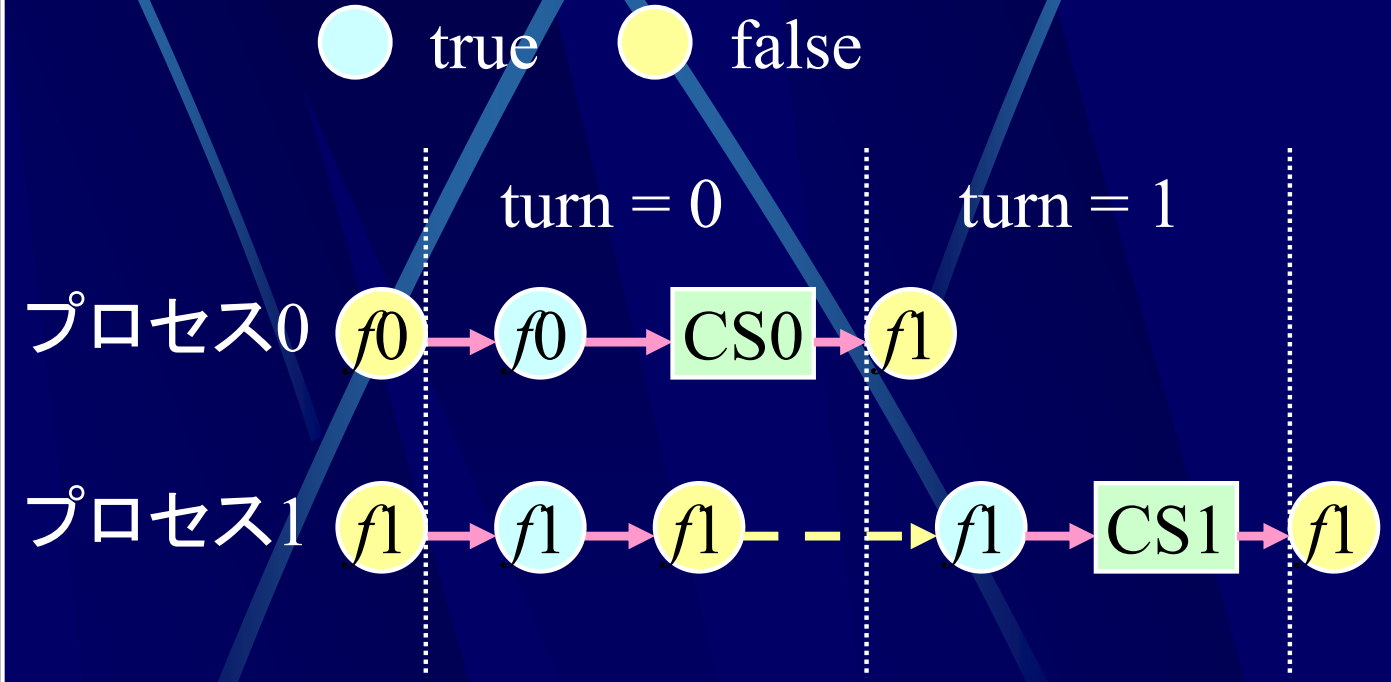
プロセス0の相互排除処理

```
flag0 := true;          /* 臨界領域に入ると宣言 */
while (flag1) {         /* P1が臨界領域か? */
  if (turn = 1) {       /* P1の番か? */
    flag0 := false;     /* 宣言を一旦取り下げ */
    while (turn = 1) wait(); /* P1が出るまで待つ */
    flag0 := true;      /* 再度宣言 */
  }
}
CS0();                  /* P0の臨界領域 */
turn := 1;              /* 次はP1が臨界領域に入る番 */
flag0 := false;         /* 宣言を取り下げ */
NCS0();                 /* P0の非臨界領域 */
```

相互排除アルゴリズム
Dekker のアルゴリズム

プロセス0

```
flag0 := true;
while (flag1) {
   if (turn = 1) {
      flag0 := false;
      while (turn = 1)
         wait();
      flag0 := true;
   }
}
CS0();
turn := 1;
flag0 := false;
NCS0();
```

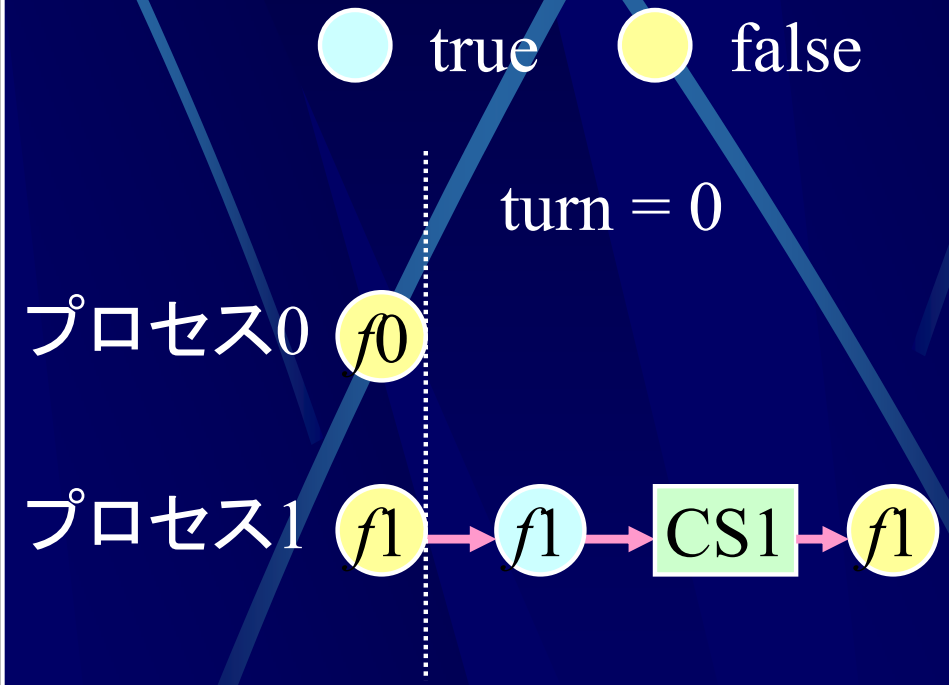


同時にはどちらか
片方のプロセスのみが
臨界領域に入れる

相互排除アルゴリズム
Dekker のアルゴリズム

プロセス0

```
flag0 := true;
while (flag1) {
   if (turn = 1) {
      flag0 := false;
      while (turn = 1)
         wait();
      flag0 := true;
   }
}
CS0();
turn := 1;
flag0 := false;
NCS0();
```

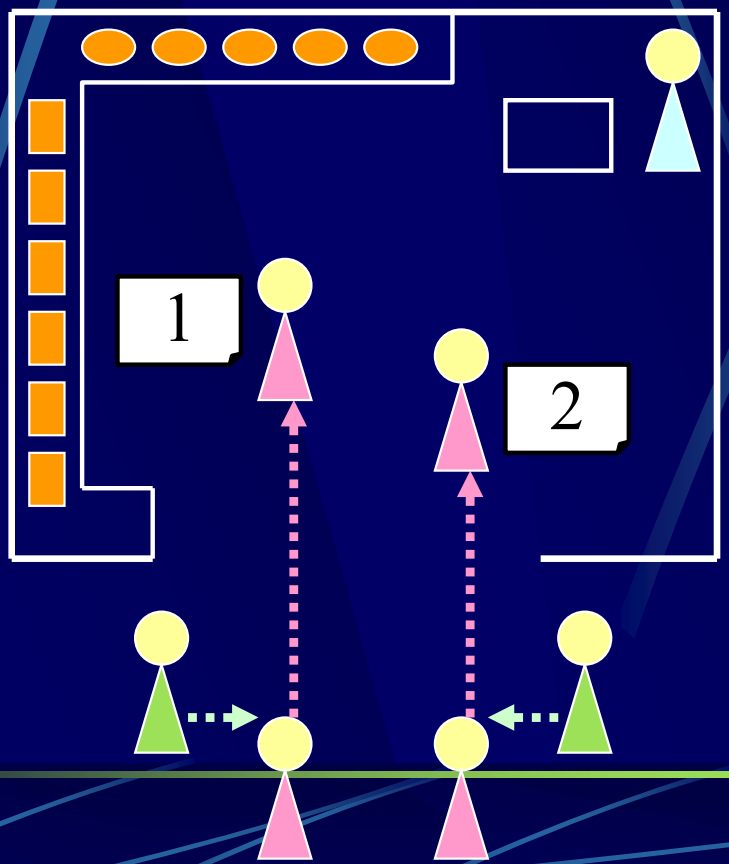


相手の番であっても、
相手が臨界領域に入ってなければ
自分が臨界領域に入れる

相手が(CSの外で)フリーズしても自分は動ける

相互排除アルゴリズム
Lamport のアルゴリズム

通称“パン屋のアルゴリズム”(bakery algorithm)



パン屋の入り口で整理券配布
番号の小さい人からレジで買える

レジ担当店員 (=臨界領域)

整理券配布店員

客 (=プロセス)

相互排除アルゴリズム
Lamport のアルゴリズム

広域変数と初期値

```
int pri[N]
 := {0, 0, …, 0};
boolean enter[N]
 := {false, false,
       …, false};
```

プロセスiの相互排除処理

```
enter[i] := true;       /* 優先順位取得開始 */
pri[i] := 1 + max {pri[0], pri[1], ..., pri[N-1]};
                        /* 優先順位を得る */
enter[i] := false;      /* 優先順位取得完了 */
for (int j := 1; j ≦ N; j++) {
   while (enter[j]) wait(); /* Pjが順位を得るまで待つ */
   while ((pri[j] ≠ 0)
            and ((pri[j], j) < (pri[i], i)))
        wait();         /* 優先順位の高いプロセスを待つ */
}
CSi();                  /* Piの臨界領域 */
pri[i] := 0;            /* 優先順位をリセット */
NCSi();                 /* Piの非臨界領域 */
```

相互排除アルゴリズム
Lamport のアルゴリズム

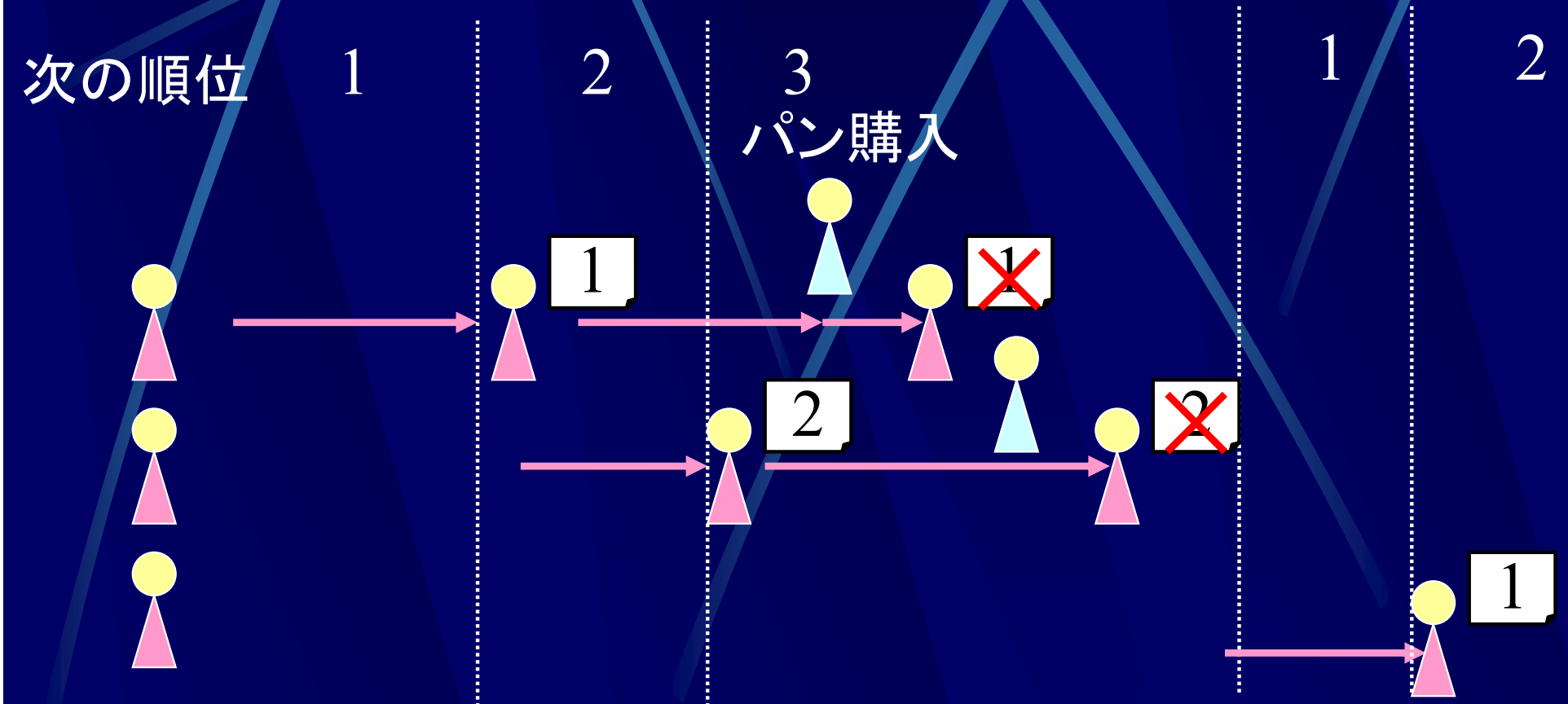
```
int pri[N] := {0, 0, …, 0};
```

```
enter[i] := true;       /* 優先順位取得開始 */
pri[i] := 1 + max {pri[0], pri[1], ..., pri[N-1]};
                        /* 優先順位を得る */
enter[i] := false;      /* 優先順位取得完了 */
```

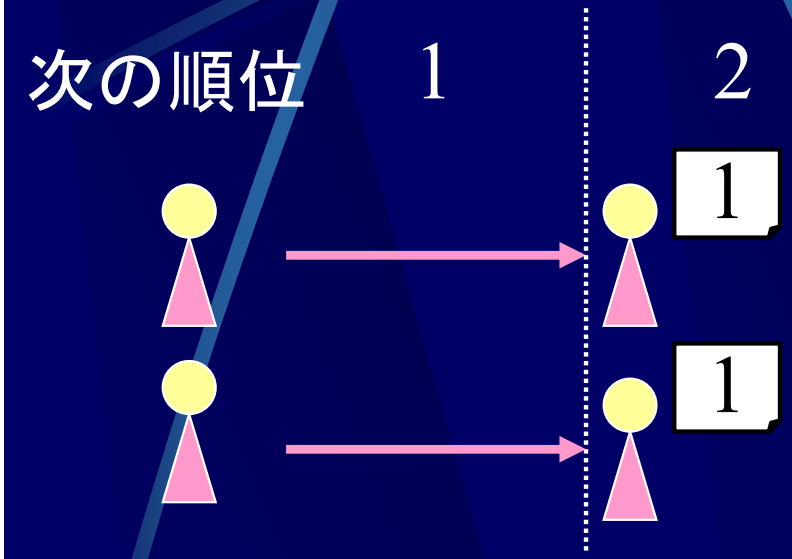
得られる優先順位は
(それまでに他のプロセスが得た順位)+1

0以外で最も小さい値が優先される

相互排除アルゴリズム
Lamport のアルゴリズム



相互排除アルゴリズム
Lamport のアルゴリズム



ほぼ同時に入店すると
同じ値の整理券が配られる

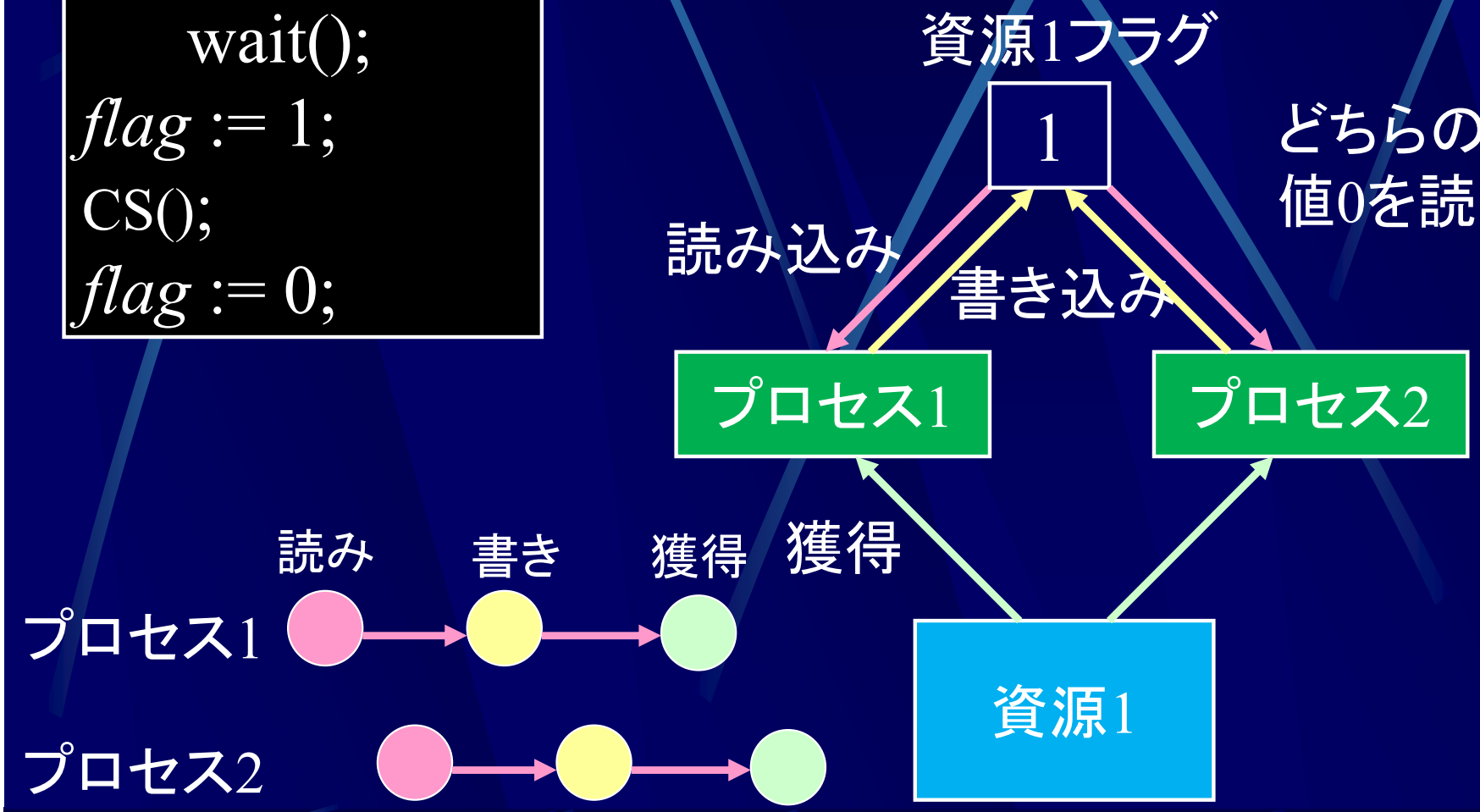
整理券の番号が同じときは
プロセス番号の小さい方優先

```
while ((pri[j] ≠ 0)
        and ((pri[j], j) < (pri[i], i)))
    wait();          /* 優先順位の高いプロセスを待つ */
```

まず優先順位で比較, 順位が同じならプロセス番号で比較

フラグによる相互排除(再掲)

```
while (flag = 1)
    wait();
flag := 1;
CS();
flag := 0;
```



2つのプロセスが同時に資源を得てしまう

フラグによる相互排除の問題点

フラグに対する機械語命令

READ : フラグの値を読む
WRITE : フラグに値を書き込む

この2命令しかないと READ と WRITE の間に
他のプロセスに割り込まれてしまう

ハードウェアに新たな機械語命令を加える

TEST&SET : フラグの値を読み
同時にフラグを1にする

TEST&SET 命令による相互排除

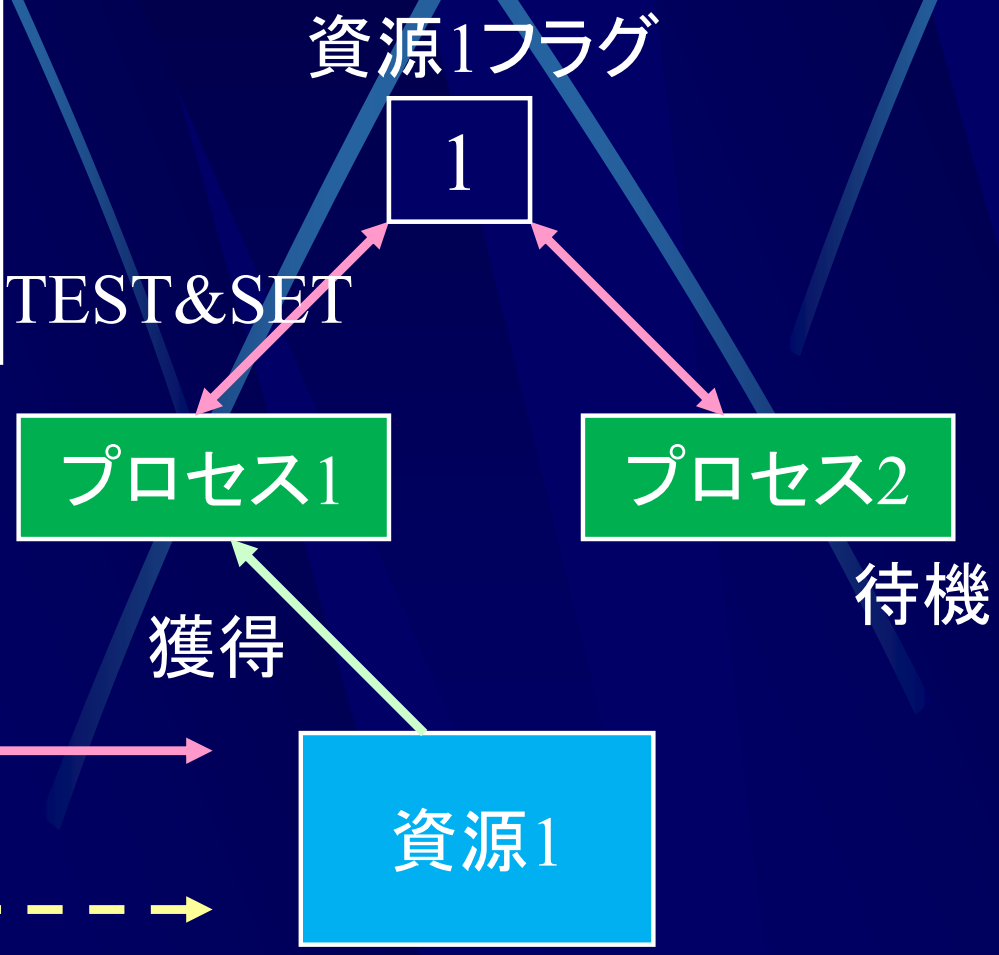
x := ts (*flag*);

x に *flag* の値を読み込み同時に *flag* の値を 1 にする

```
while (ts (flag) = 1) /* フラグをチェックし同時に 1 にセット */
    wait();           /* フラグが 0 になるまで待つ */
CS();                 /* 資源を使用する臨界領域 */
flag := 0;            /* フラグを 0 にリセット */
```

TEST&SET 命令による相互排除

```
while (ts (flag) = 1)
      wait();
CS();
flag := 0;
```



割込み禁止による相互排除

フラグによる相互排除の問題点
READ と WRITE の間に他のプロセスに割り込まれる

割込みを禁止にすればよい

```
割込み禁止;
CS();
割込み禁止解除;
NCS();
```

この間他のプロセスは実行されない

ただし、割込み禁止を多用するとシステムの効率が落ちる

相互排除

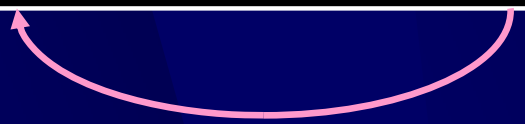
- ソフトウェアによる相互排除
  - 相互排除アルゴリズムを使用
    - 理論的には意味があるが実用的ではない
- ハードウェアによる相互排除
  - 機械語命令 Test and Set を使用
    - ハード的に機能を付ける必要あり
- 割込み禁止による相互排除
  - 割込み禁止命令を使用
    - 割込み禁止の多用はシステムの効率を下げる

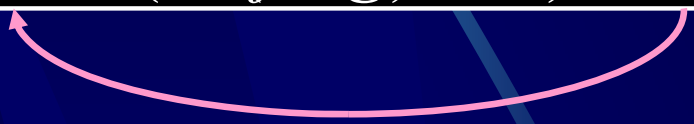
繁忙待機(busy-wait)

- 繁忙待機
  - プロセスがフラグ確認のためにループ

```
while (turn = 1) wait();
```

```
while (ts (flag) = 1) wait();
```

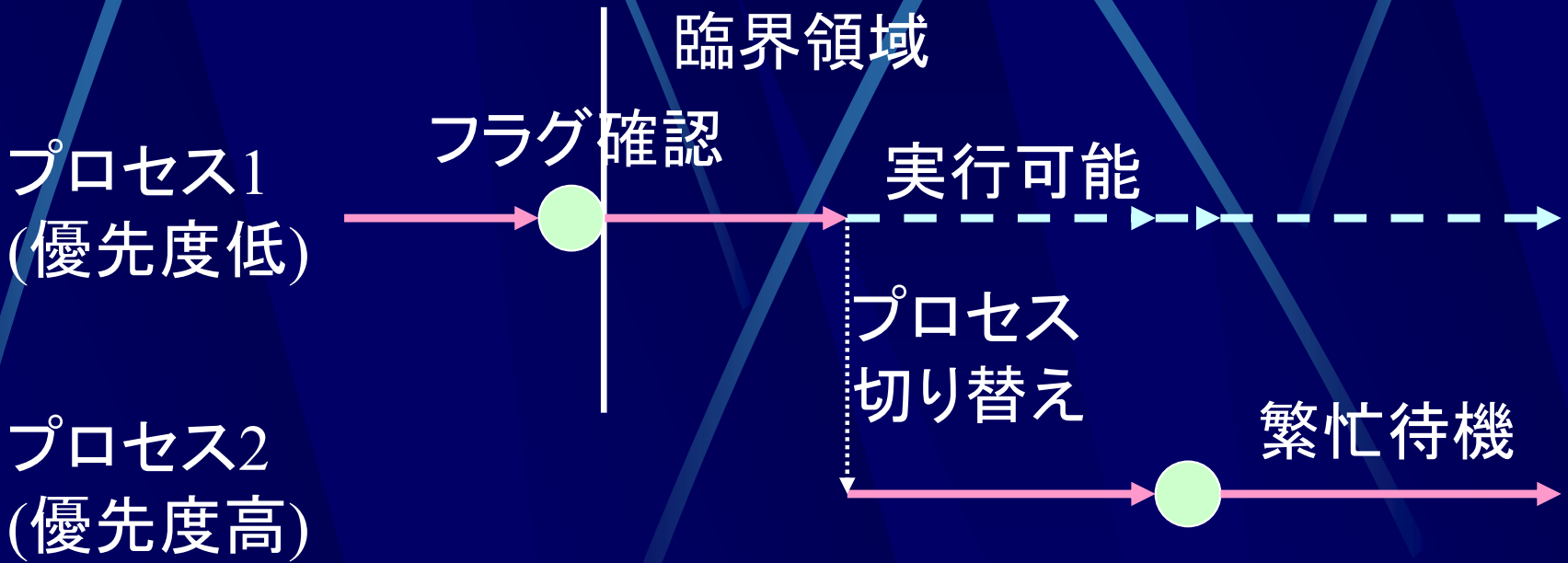




ループしている間、プロセスは生産的な
作業すること無しにCPUを浪費

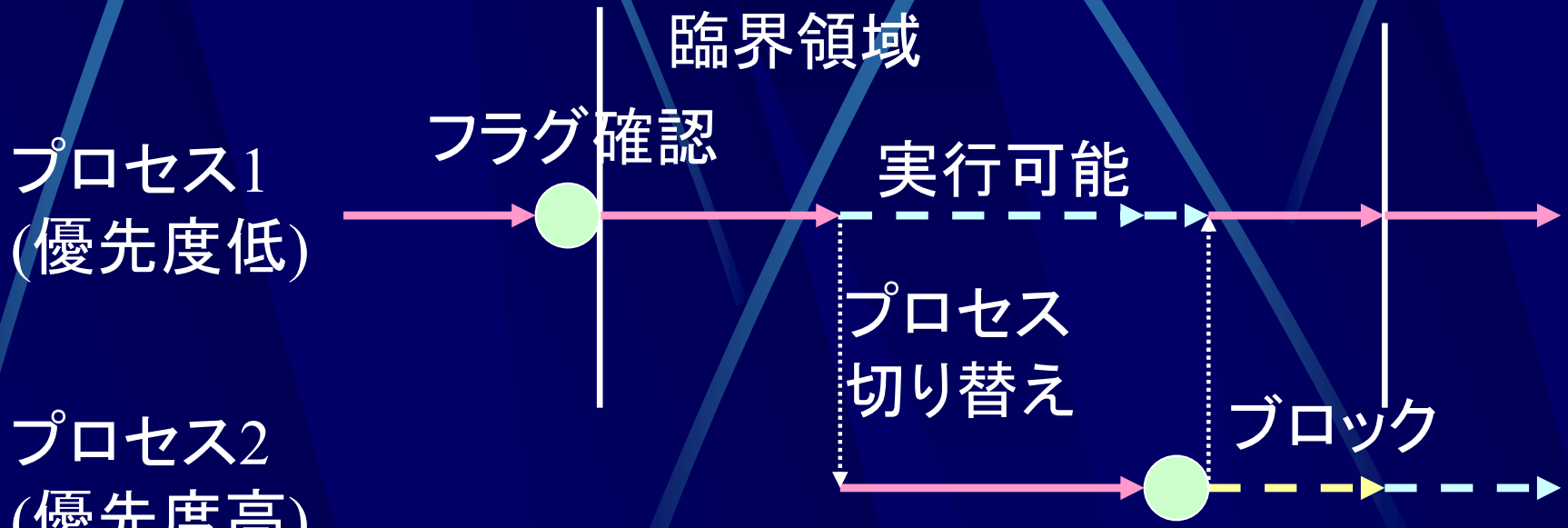
繁忙待機

優先度逆転問題



繁忙待機

ループ中はプロセスをブロック状態にする



参考：相互排除プログラム(java)

- MutualExclusion.java
  - 4種類の相互排除のアルゴリズムを繰り返す
- Mutex.java
  - スレッドを2つ(4つ)生成, 実行する
    - デフォルトでは交互実行アルゴリズム
    - -d を付けて実行すると Dekker のアルゴルズム
    - -p を付けて実行すると Peterson のアルゴリズム
    - -l を付けて実行すると Lamport のアルゴリズム

http://www.info.kindai.ac.jp/OS
からダウンロードし、各自実行してみること

参考：相互排除プログラム(java)

実行例

```
$ javac Mutex.java
$ java Mutex
 交互実行アルゴリズム開始
 スレッド0 : CS begin  (0秒経過)
 スレッド0 : CS end    (2秒経過)
                    スレッド1 : CS begin  (2秒経過)
                    スレッド1 : CS end    (3秒経過)
 スレッド0 : CS begin  (4秒経過)
 スレッド0 : CS end    (6秒経過)
                    スレッド1 : CS begin  (10秒経過)
```

参考：相互排除プログラム(java)

実行例

```
$ javac Mutex.java
$ java Mutex -d
Dekker のアルゴリズム開始
スレッド0 : CS begin  (0秒経過)
スレッド0 : CS end    (2秒経過)
                スレッド1 : CS begin  (2秒経過)
                スレッド1 : CS end    (3秒経過)
スレッド0 : CS begin  (4秒経過)
スレッド0 : CS end    (6秒経過)
スレッド0 : CS begin  (8秒経過)
```