オペレーティングシステム

第6回

プロセス間通信

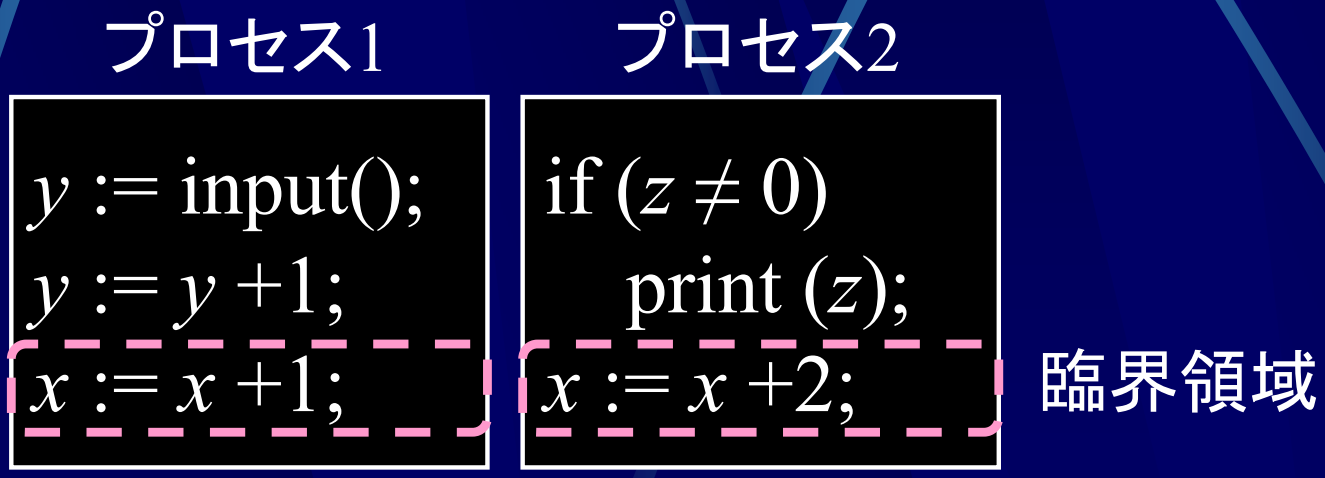
http://www.info.kindai.ac.jp/OS

E号館3階E-331 内線5459

takasi-i@info.kindai.ac.jp

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

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

相互排除

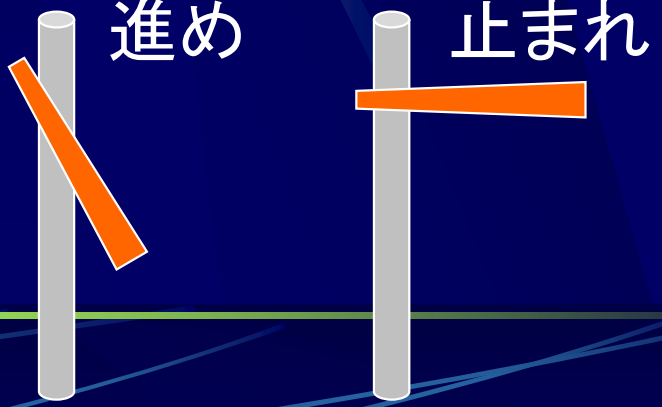
- ソフトウェアによる相互排除
  - 相互排除アルゴリズムを使用
- ハードウェアによる相互排除
  - 機械語命令 Test and Set を使用
- 割込み禁止による相互排除
  - 割込み禁止命令を使用
- セマフォによる相互排除
- モニタによる相互排除

セマフォ(semaphore)

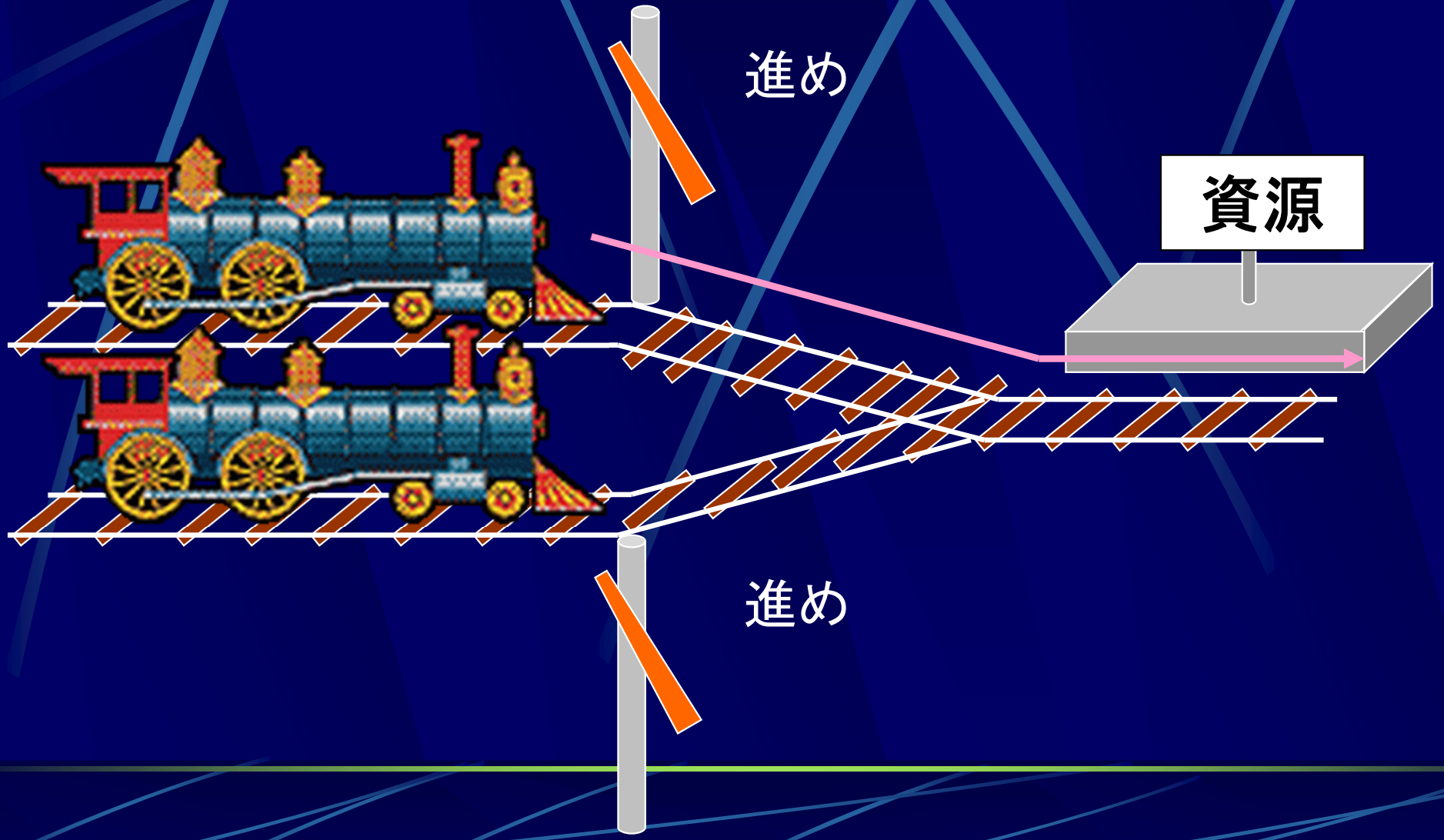
- セマフォ(semaphore)
  - プロセス間同期機構
  - Dijkstra が提案

セマフォ

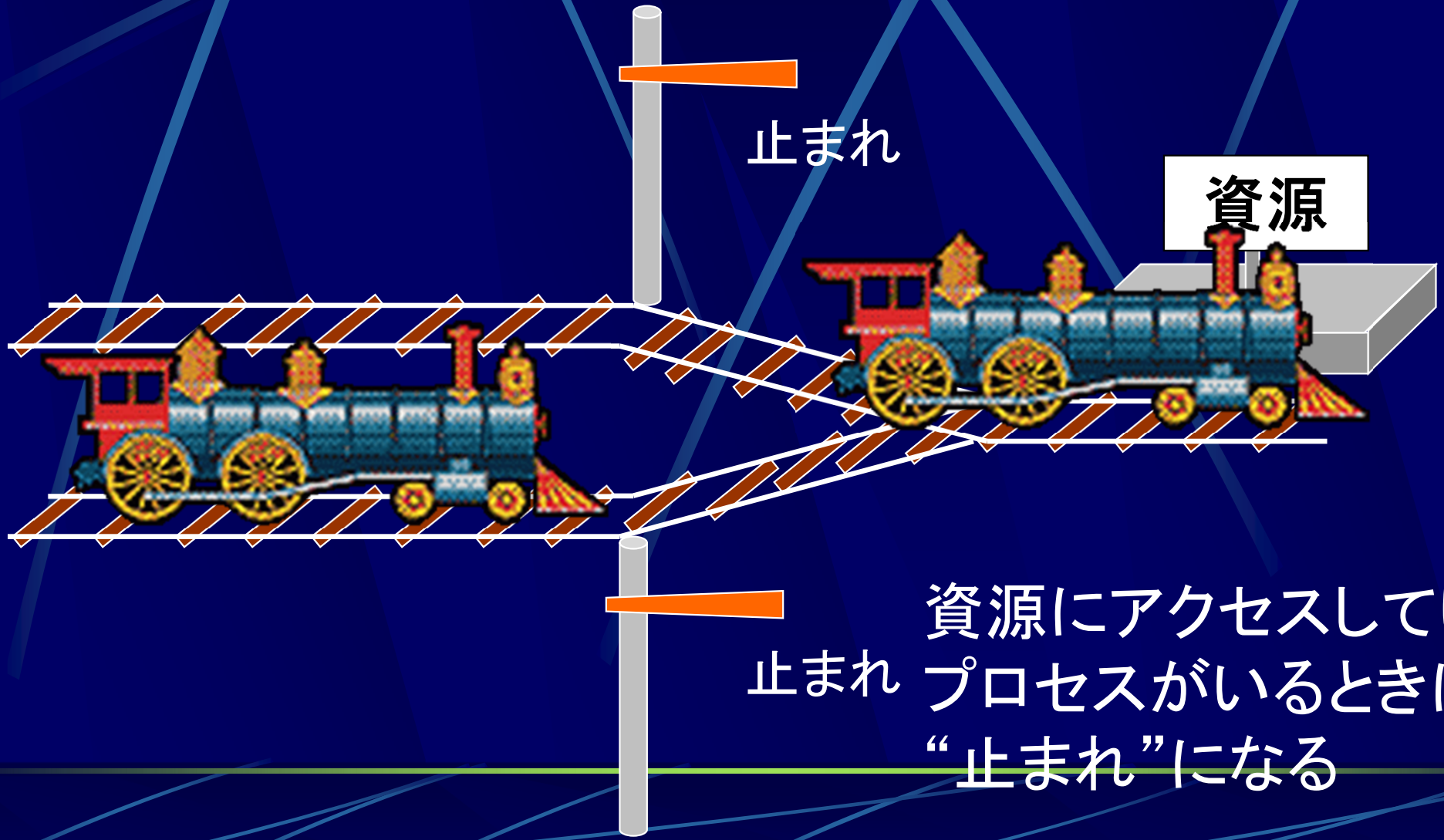
セマフォ = 腕木式信号機
“進め”(Passeren)と“止まれ”(Verhoog)の
2つの信号を出す



セマフォ



セマフォ



資源にアクセスしている
プロセスがいるときは
“止まれ”になる

セマフォ

- wait 命令 (P 命令, acquire 命令)
  - 資源を要求, 許可されない場合はブロック状態へ移行し待ちキューに加える
- signal命令 (V 命令, release 命令)
  - 資源を解放, 待ちキュー内のプロセスの1つを実行可能状態へ
- セマフォ変数
  - 空いている資源の数を示す
  - 1以上のとき wait 命令で資源を確保できる

セマフォ

セマフォ変数
(空いている資源の数)

2

プロセス1

プロセス2

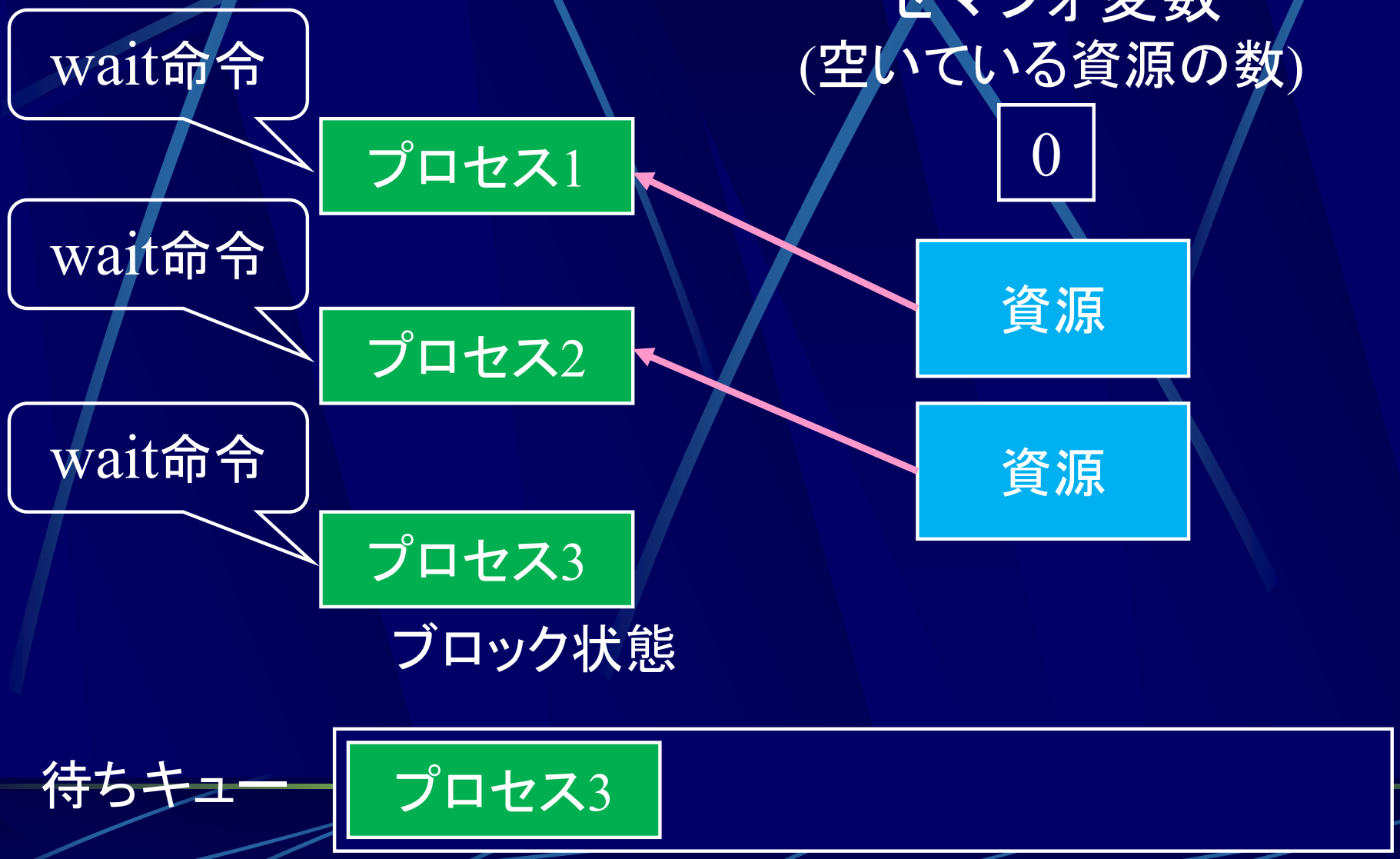
プロセス3

資源

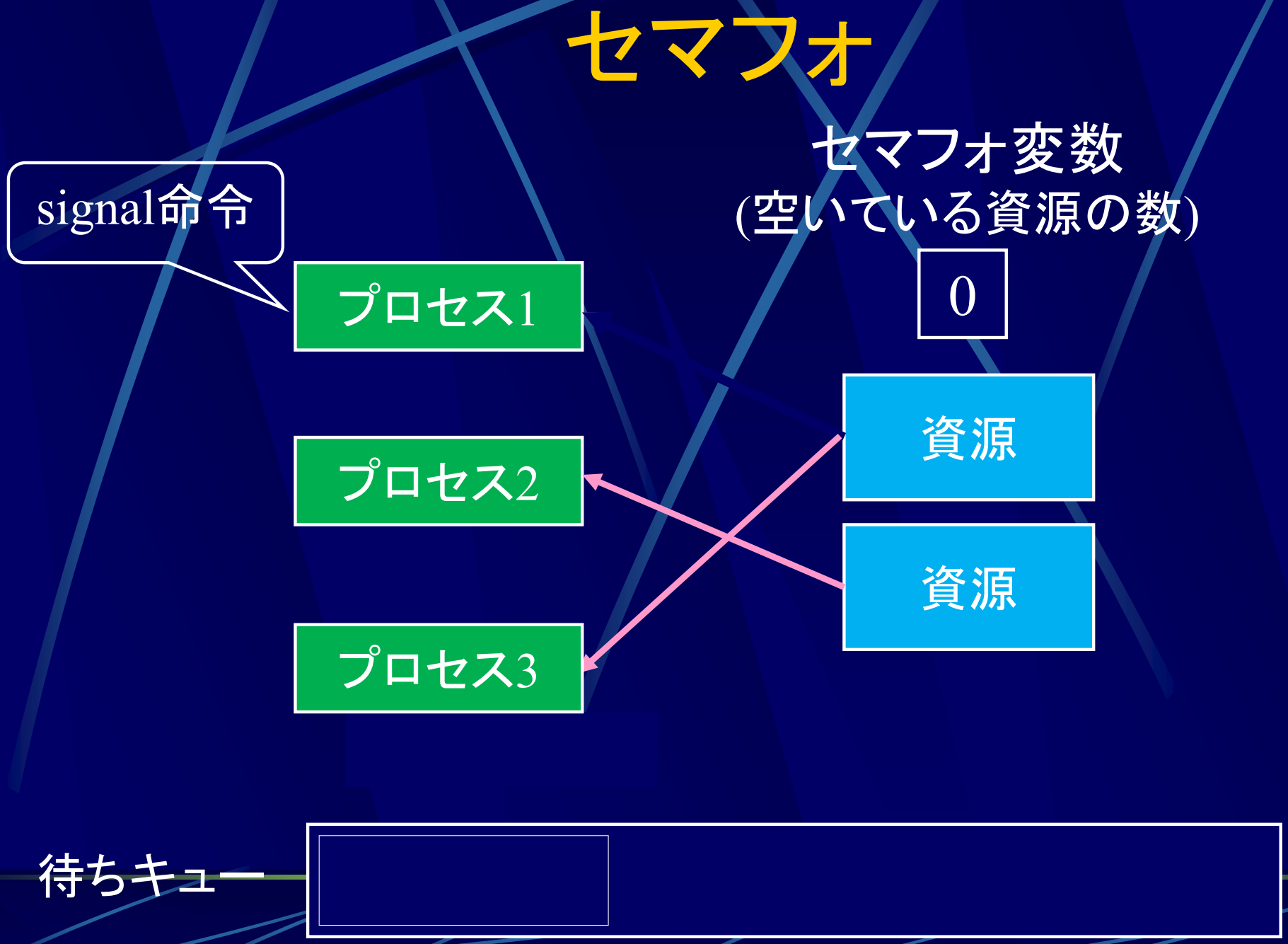
資源

待ちキュー

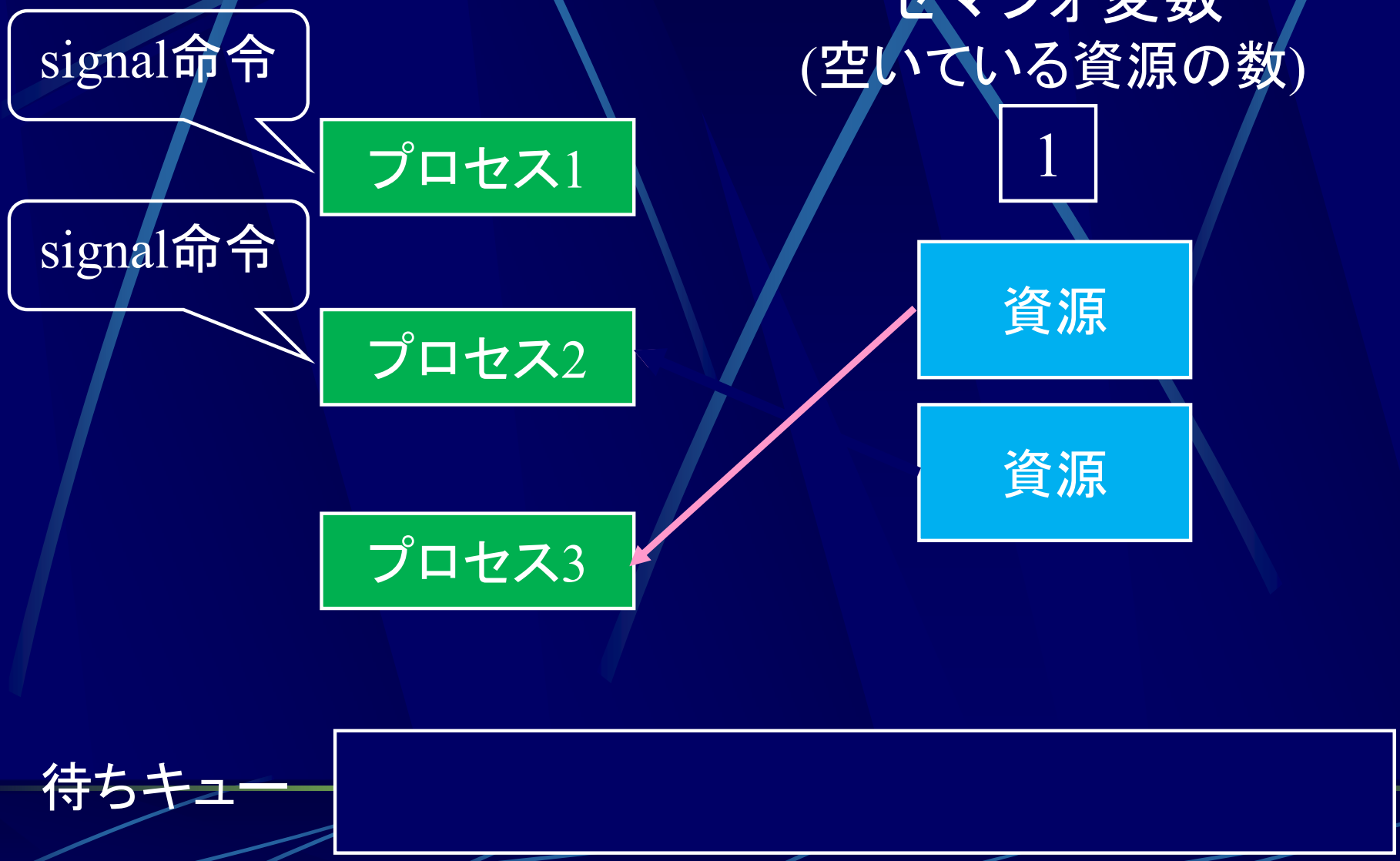
セマフォ


セマフォ変数
(空いている資源の数)
0
wait命令
wait命令
wait命令
プロセス1
プロセス2
プロセス3
資源
資源
ブロック状態
待ちキュー
プロセス3

セマフォ



セマフォ


signal命令
signal命令
プロセス1
プロセス2
プロセス3
セマフォ変数
(空いている資源の数)
1
資源
資源
待ちキュー

セマフォ

wait命令

資源を要求、
許可されない場合は
待ちキューに加える

```
if ( s > 0 ) {
    s := s - 1;
} else {
   ブロック状態に移行し
   待ちキューQsに加える;
}
```

signal命令

資源を解放、
待ちキュー内のプロセスの1つを
実行可能状態へ

```
if ( |Qs| ≧1 ) {
   Qs内のプロセスを
   1つ実行可能状態へ;
} else {
   s := s + 1;
}
```

wait 命令, signal 命令は不可分な(割込みされない)操作

test&set 命令で実現

セマフォを用いた相互排除

```
semaphore s := 1;   /* 資源数 1 */
```

```
wait(s);
CS();        /* 臨界領域 */
signal(s);
NCS();       /* 非臨界領域 */
```

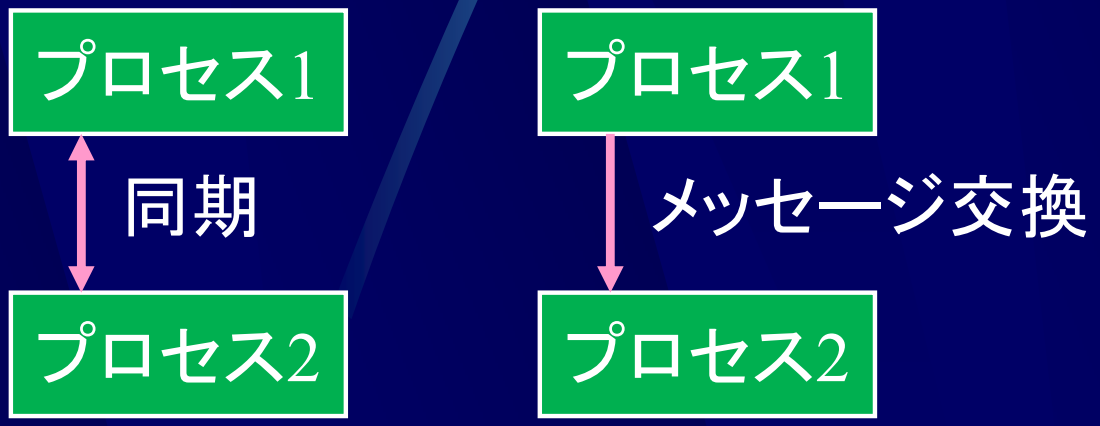
臨界領域の前後を wait 命令と signal 命令で挟む

強いセマフォ，弱いセマフォ

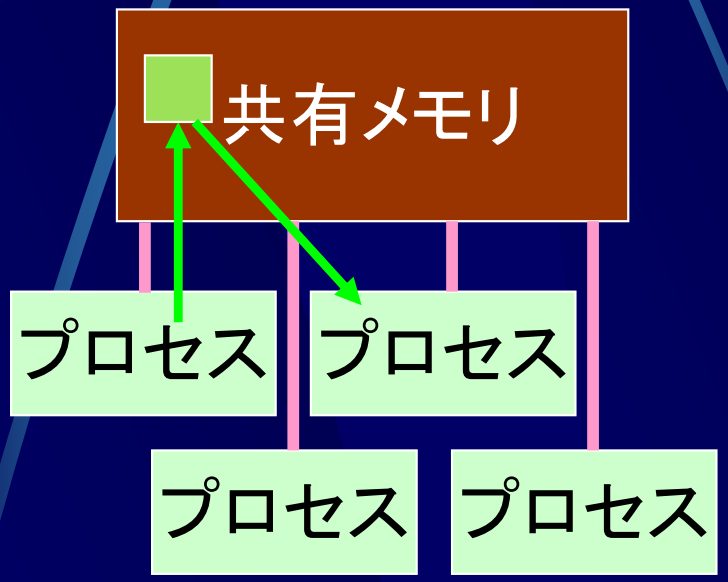
- 強いセマフォ
  - wait 命令によりブロック状態となるプロセスは FIFO 方式で実行可能状態に復帰する
- 弱いセマフォ
  - wait 命令によるブロック状態となるプロセスの復帰順序は決められていない

プロセス間通信
(Inter-Process Communication)

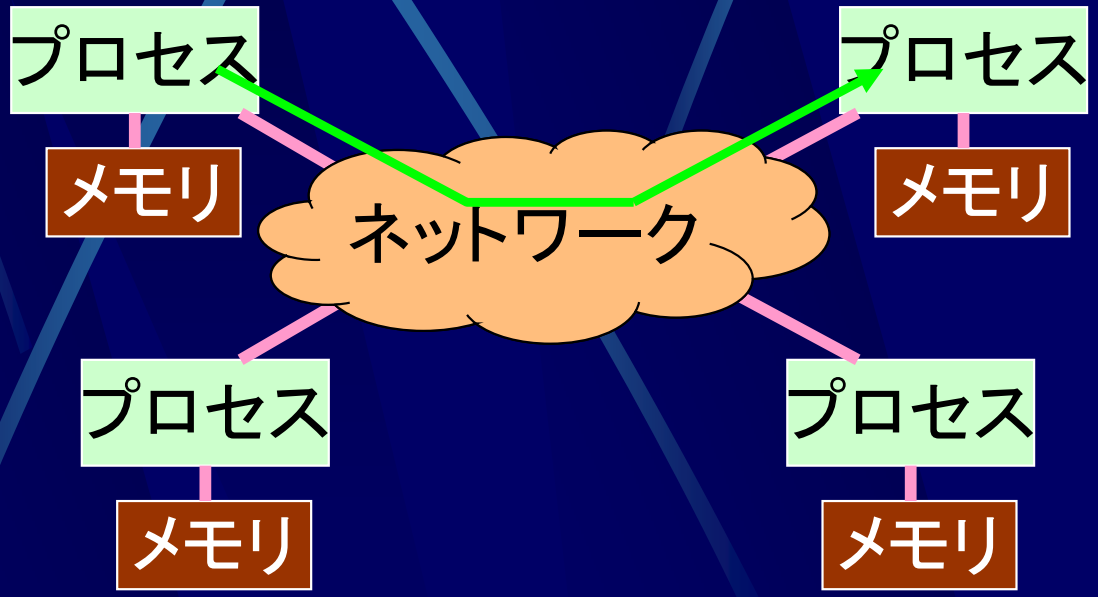
- プロセス間通信(Inter-Process Communication)
  - プロセス間で同期を取る
  - プロセス間でメッセージを交換する



通信方式とメモリ型

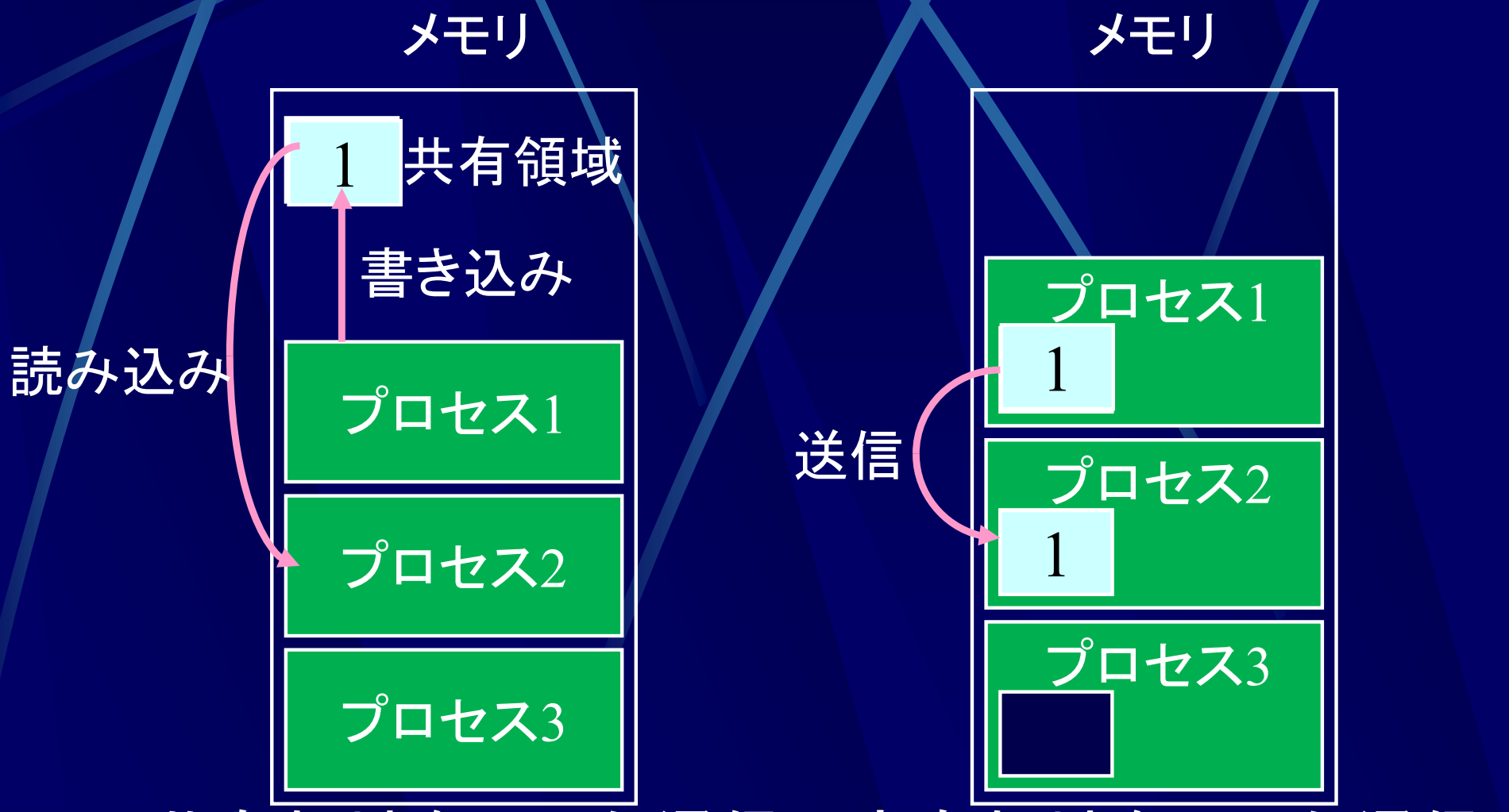


共有メモリ型



分散メモリ型

共有メモリ型プロセス間通信



共有領域を用いた通信　占有領域を用いた通信

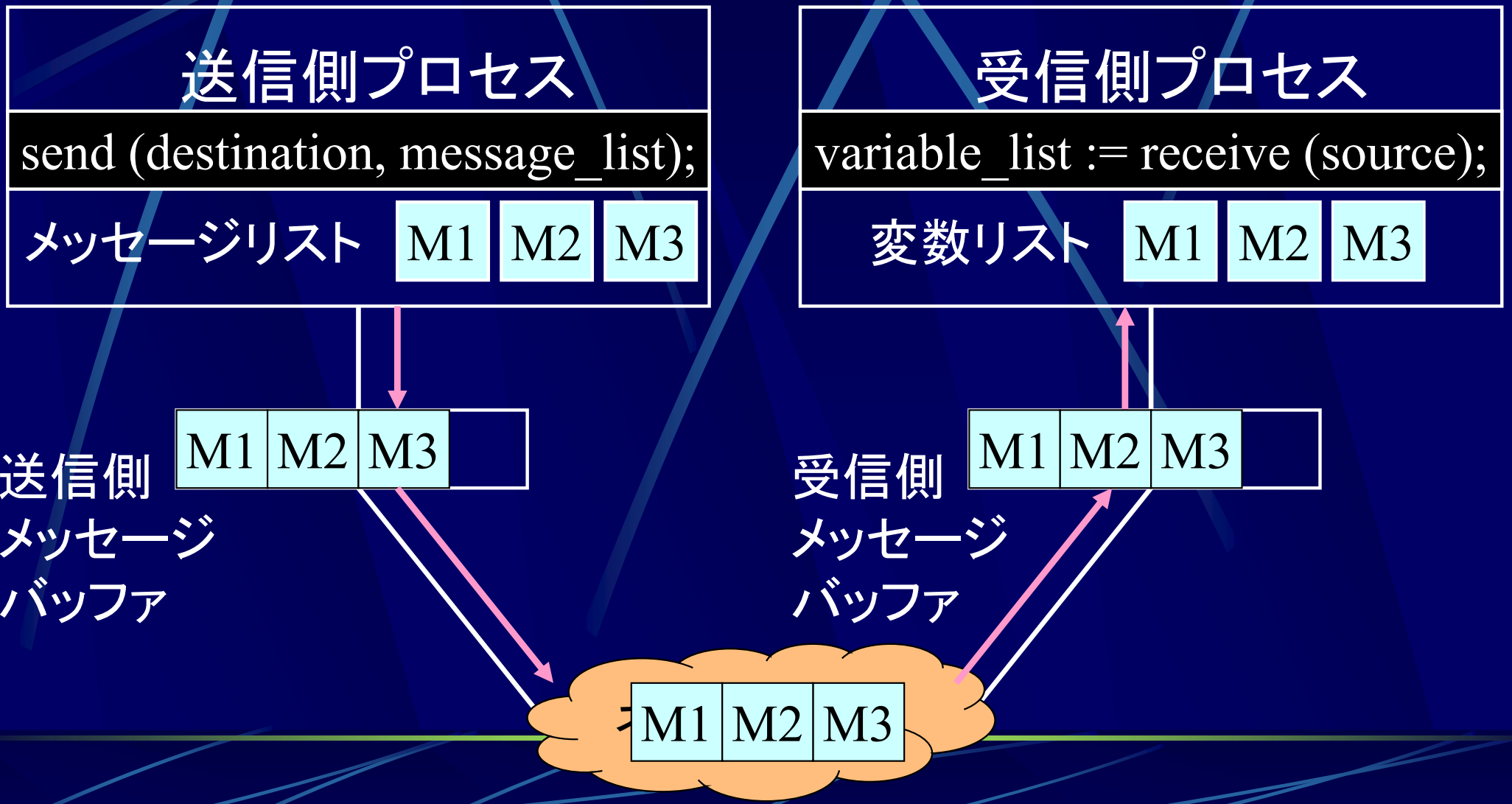
分散型プロセス間通信
送信, 受信(send, receive)

- 送信(send)
  - 指定した宛先へメッセージを送る
- 受信(receive)
  - 指定した送信元からのメッセージを受け取る

```
send (destination, message_list);
```

```
variable_list := receive (source);
```

分散型プロセス間通信
送信，受信

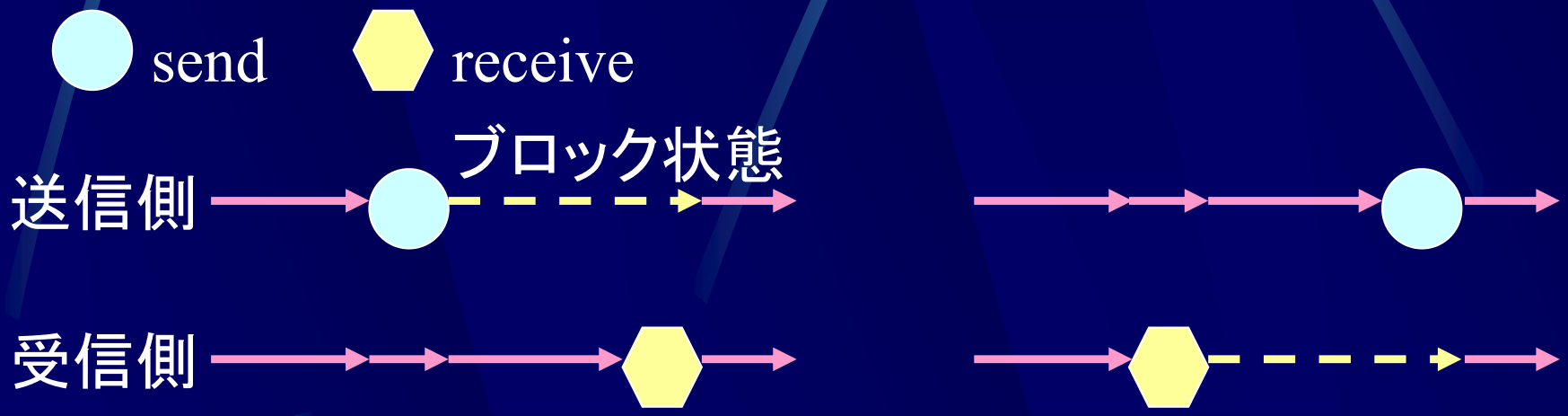


共有メモリ型・分散メモリ型の長所と短所

	長所	短所
共有メモリ型	• メモリへの読み書きだけで通信可能 • プロセス間の同期が容易	• プロセスの独立性が低い • 高度なメモリ管理が必要
分散メモリ型	• プロセスの独立性が高い • 実現は容易	• 通信のオーバヘッド • プロセス間の同期が困難

同期通信, ブロッキング型 (synchronous communication, blocking)

- 同期通信(synchronous communication), ブロッキング型(blocking)
  - 送信側は受信側が受信するまで待つ



同期通信

非同期通信, ノンブロッキング型
(asynchronous communication, nonblocking)

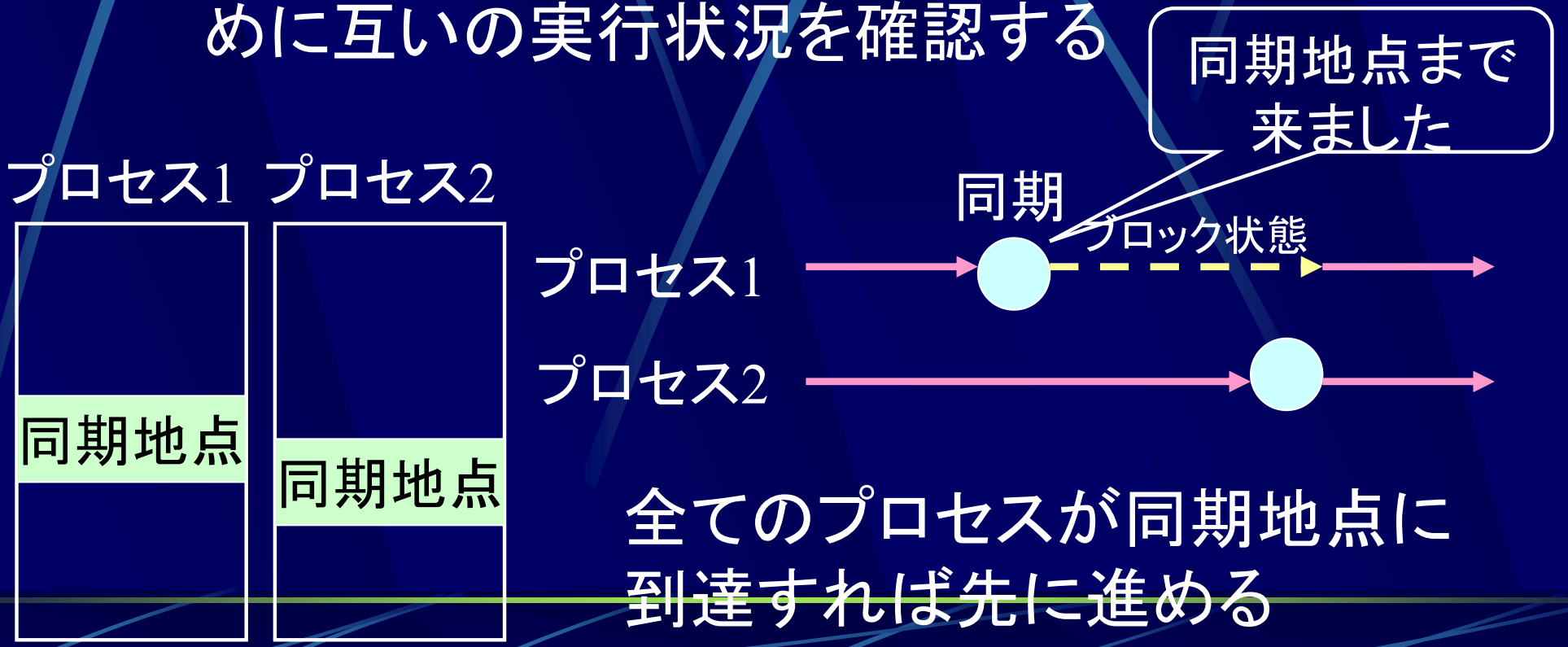
- 非同期通信(asynchronous communication), ノンブロッキング型(nonblocking)
  - 送信側は受信側の受信を待たない

同期通信・非同期通信の長所と短所

	長所	短所
同期通信	• 相手に届いたことが確認できる	• 相手の受信完了まで待つ必要がある • 相手が止まると自分も動けない ⇒ 一定時間返事が無ければ通信を打ち切る
非同期通信	• 相手の受信を待たなくていい • 相手が止まっても自分は動ける	• 相手に届いたかどうかわからない ⇒ 受信完了の通知が必要

プロセスの同期 (synchronization)

- プロセスの同期(synchronization)
  - 協同して動くプロセス群が足並みを揃えるために互いの実行状況を確認する

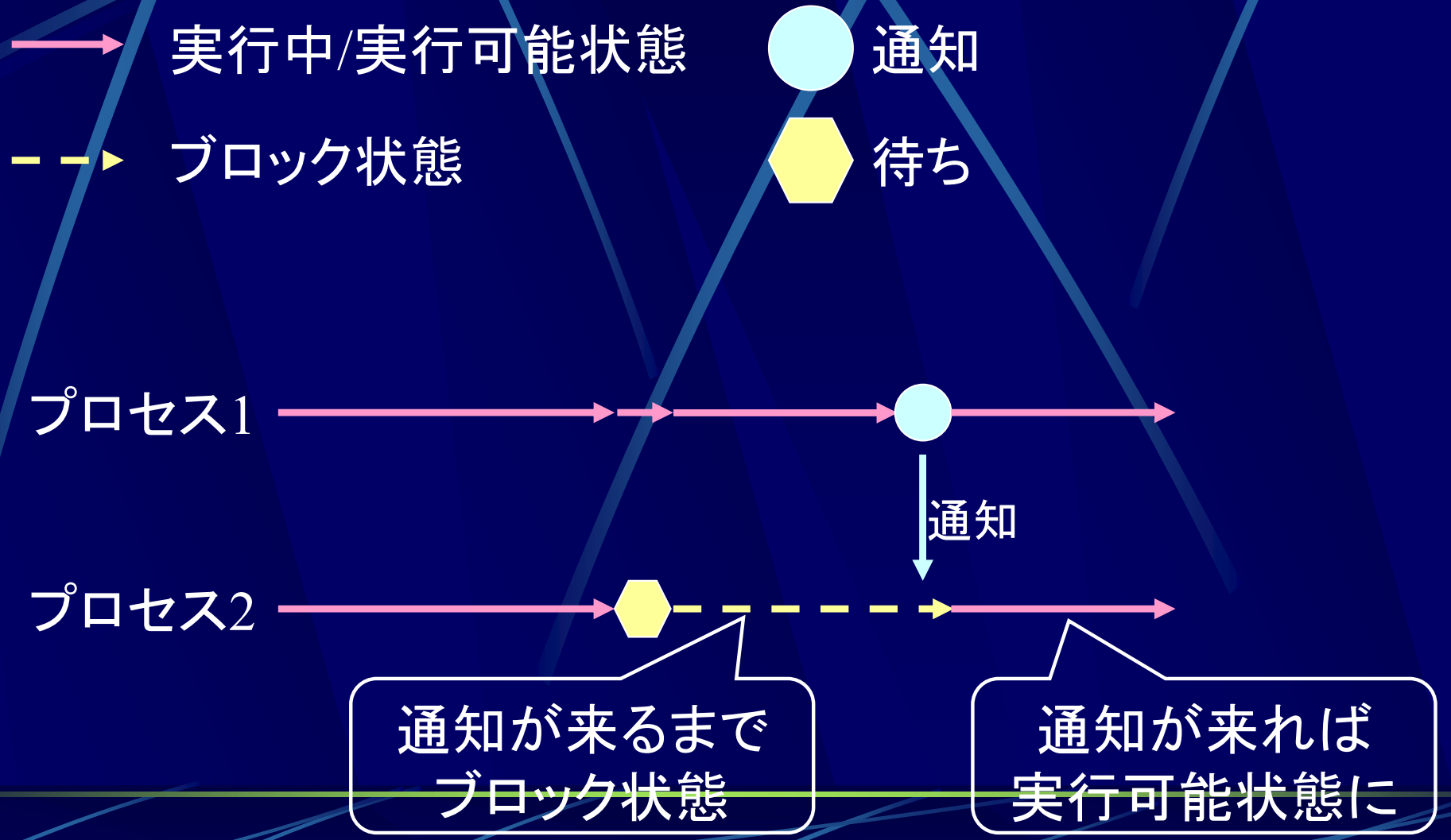


全てのプロセスが同期地点に到達すれば先に進める

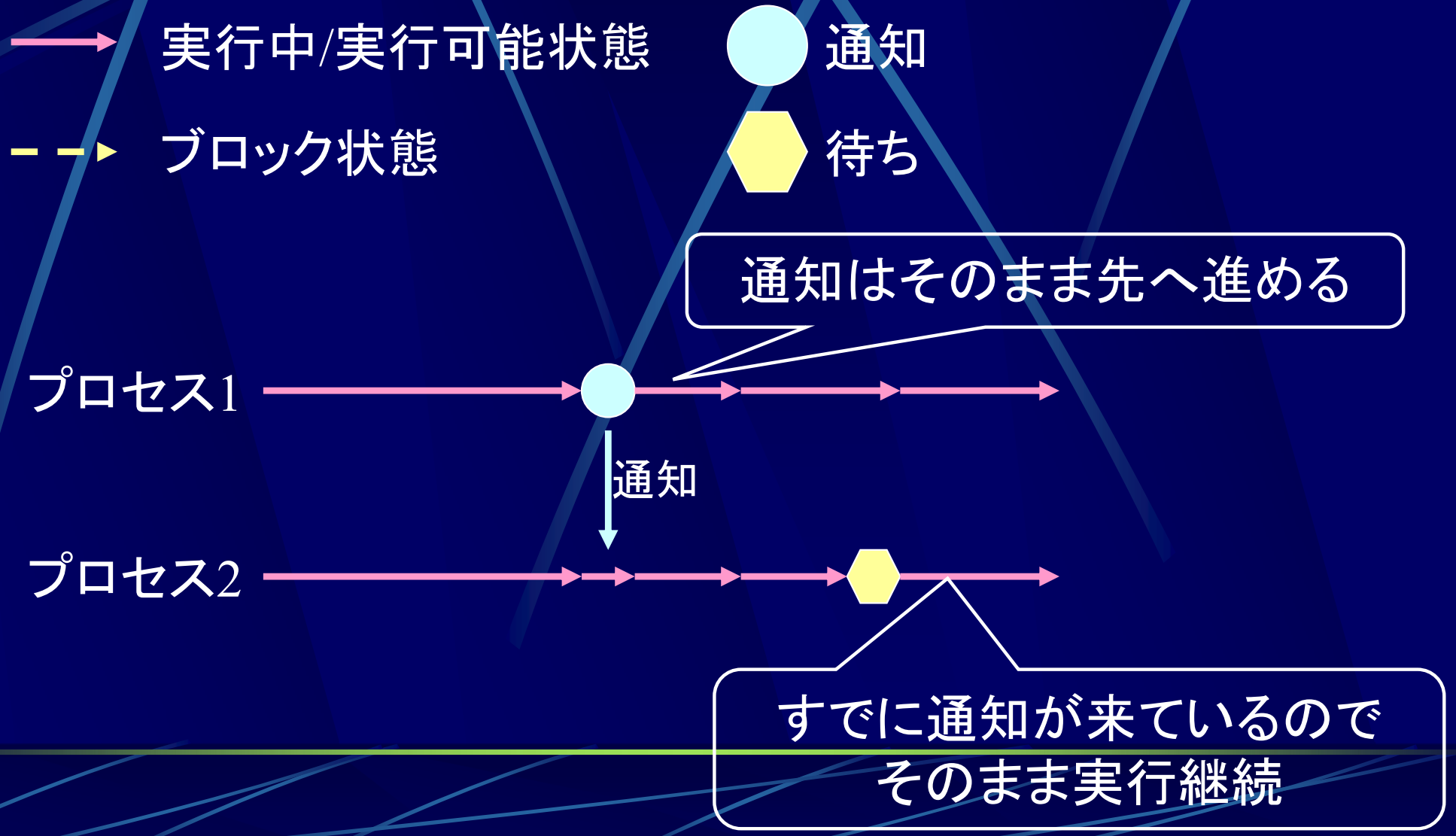
プロセスの同期
事象の連絡

- 通知(notify)
  - 同期を取る相手のプロセスに同期地点まで来たことを知らせる
- 待ち(wait)
  - 同期を取る相手のプロセスが同期地点に来るまで待つ

プロセスの同期
事象の連絡

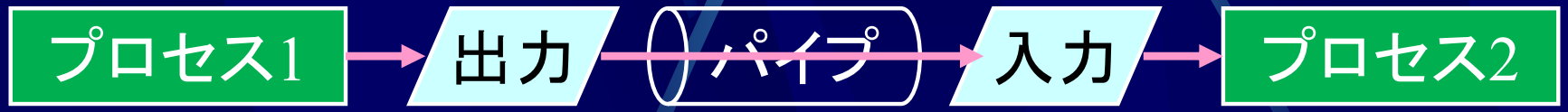


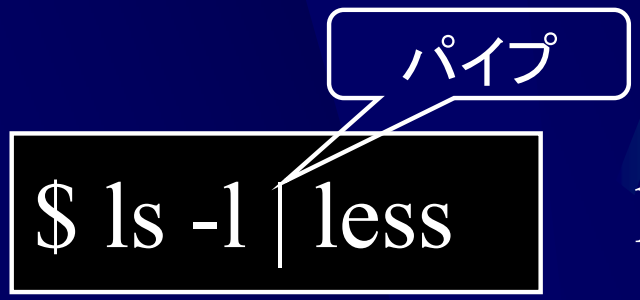
プロセスの同期
事象の連絡



プロセスの同期
パイプ処理

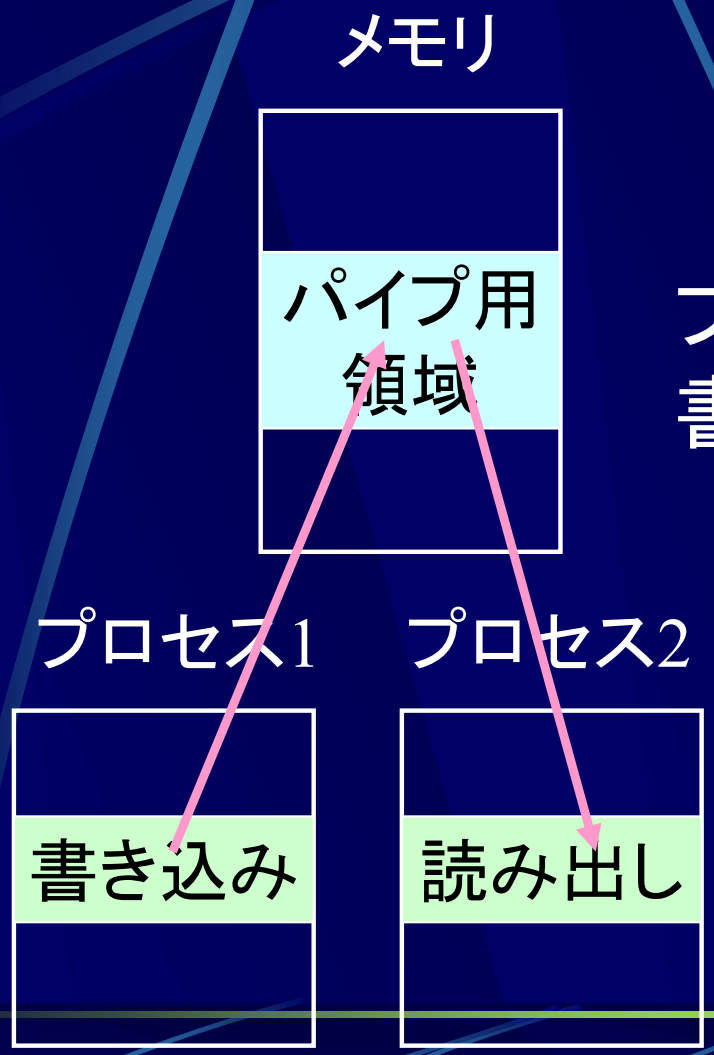
- パイプ(pipe)
  - システムコールで作られるプロセス間連絡路
    (実際はメモリで実現される)





ls -l の出力が less の入力となる

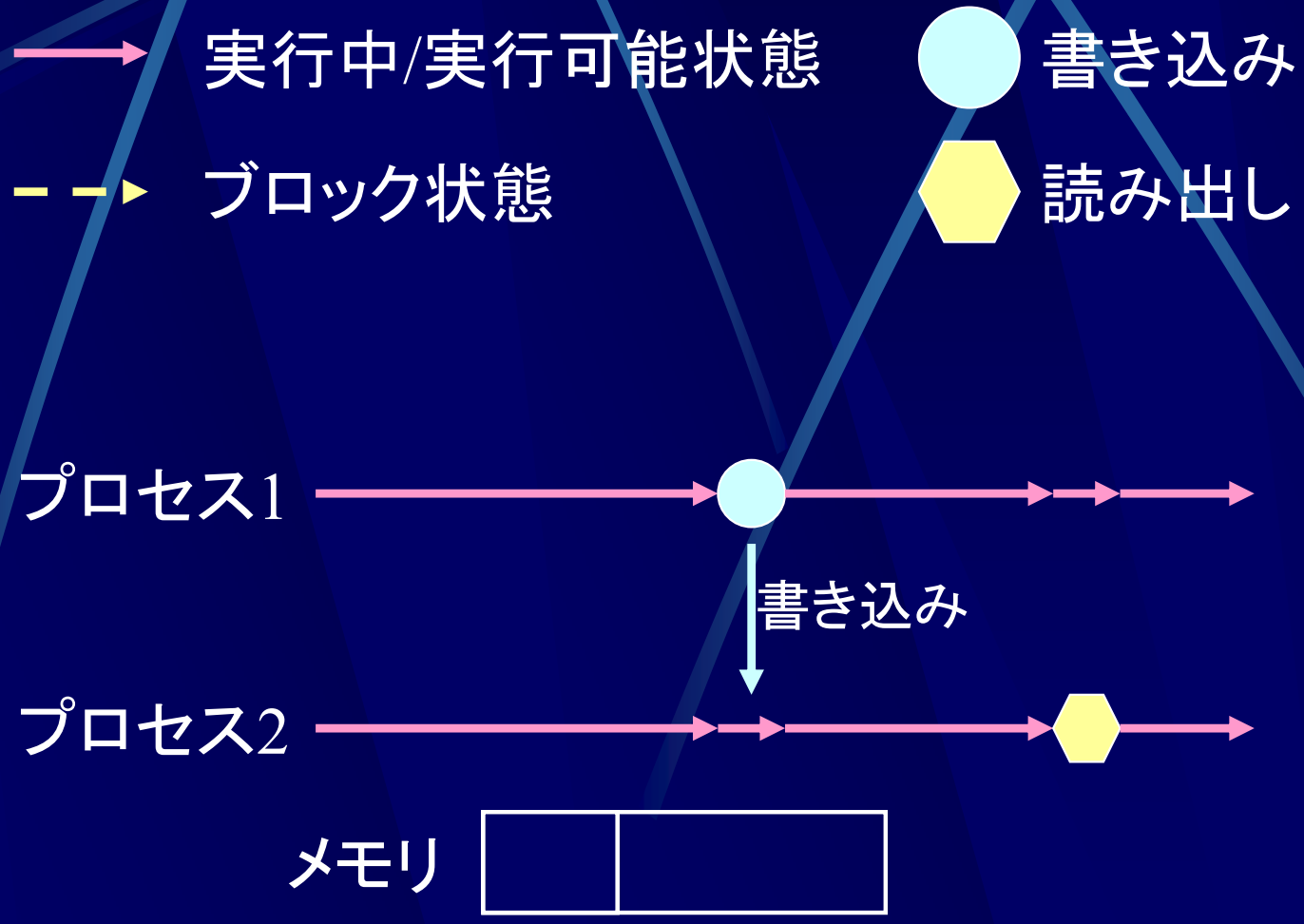
プロセスの同期
パイプ処理



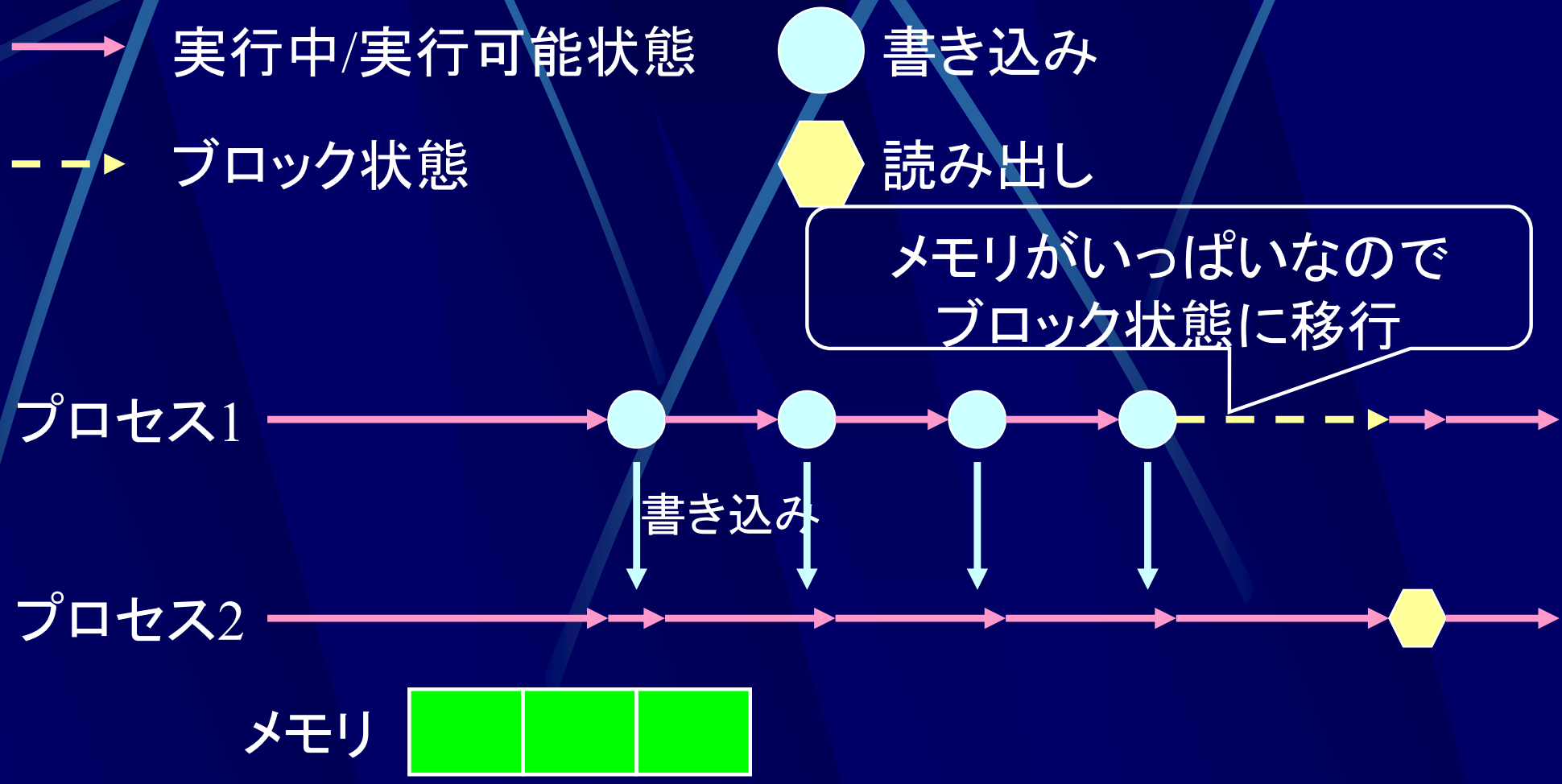
プロセス1がメモリのパイプ用領域に
書き込み, プロセス2がそれを読み出す

書き込みと読み込みを順次行うことで
大きなデータでも受け渡し可能

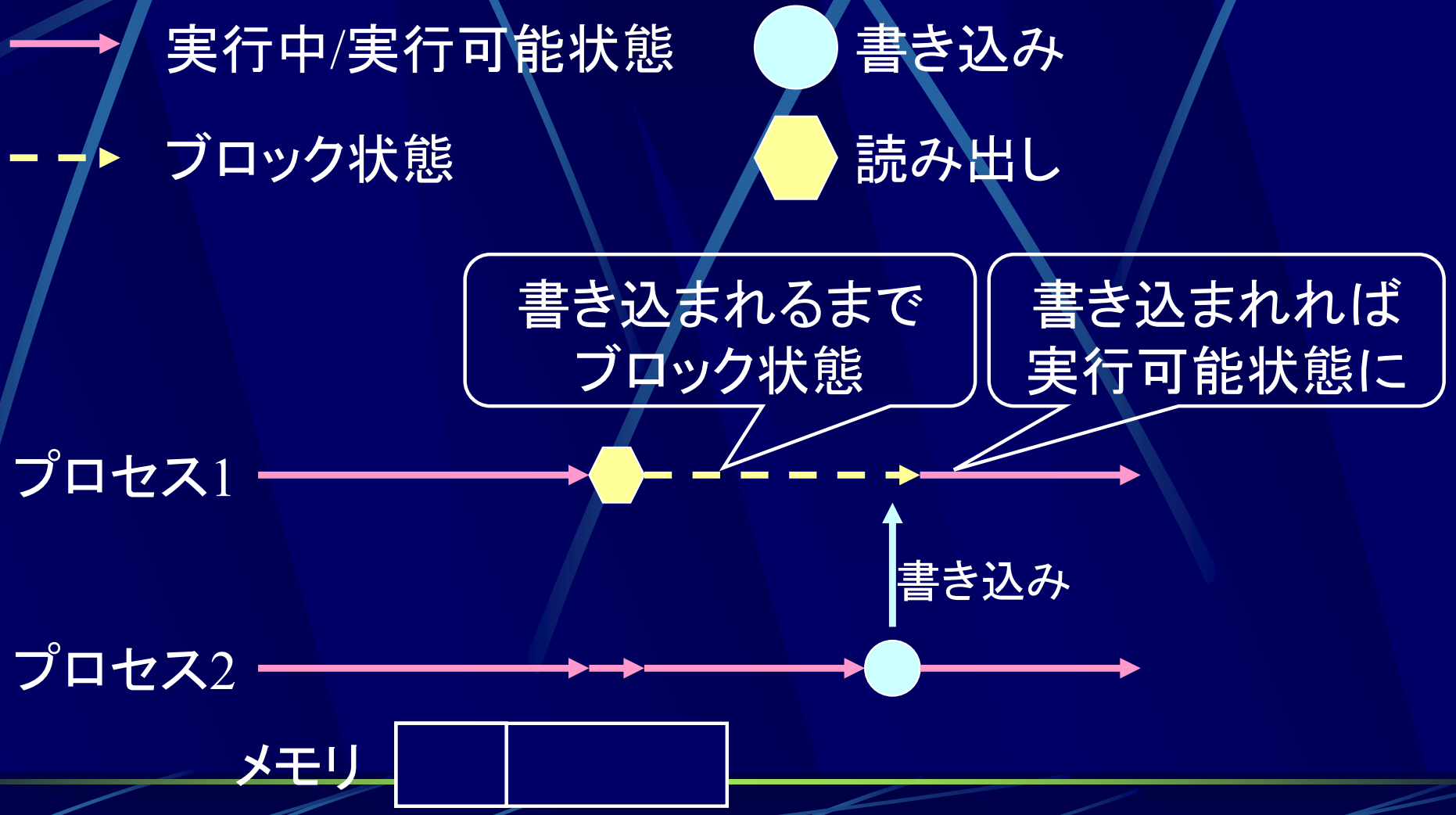
プロセスの同期
パイプ処理



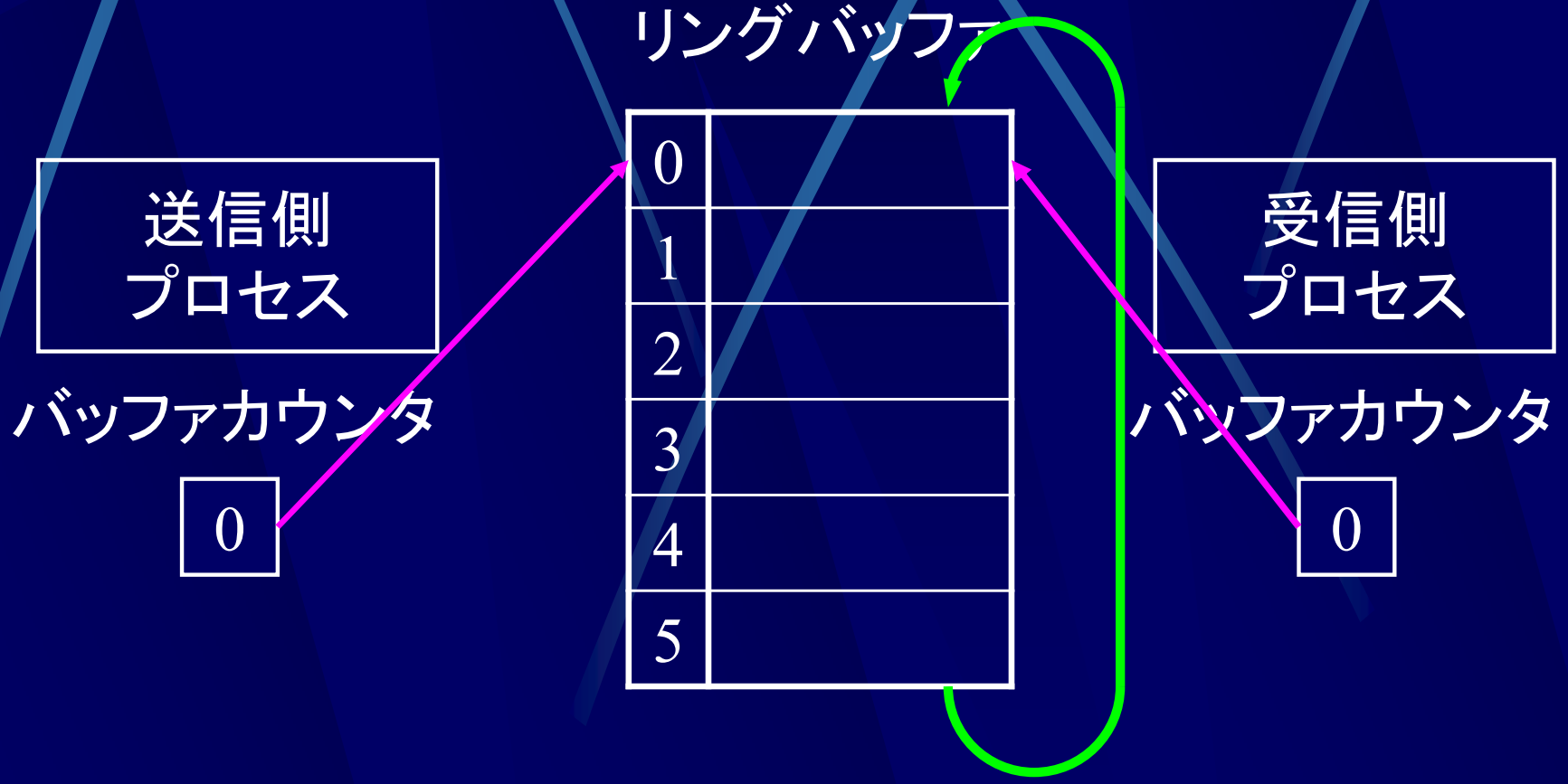
プロセスの同期
パイプ処理



プロセスの同期
パイプ処理

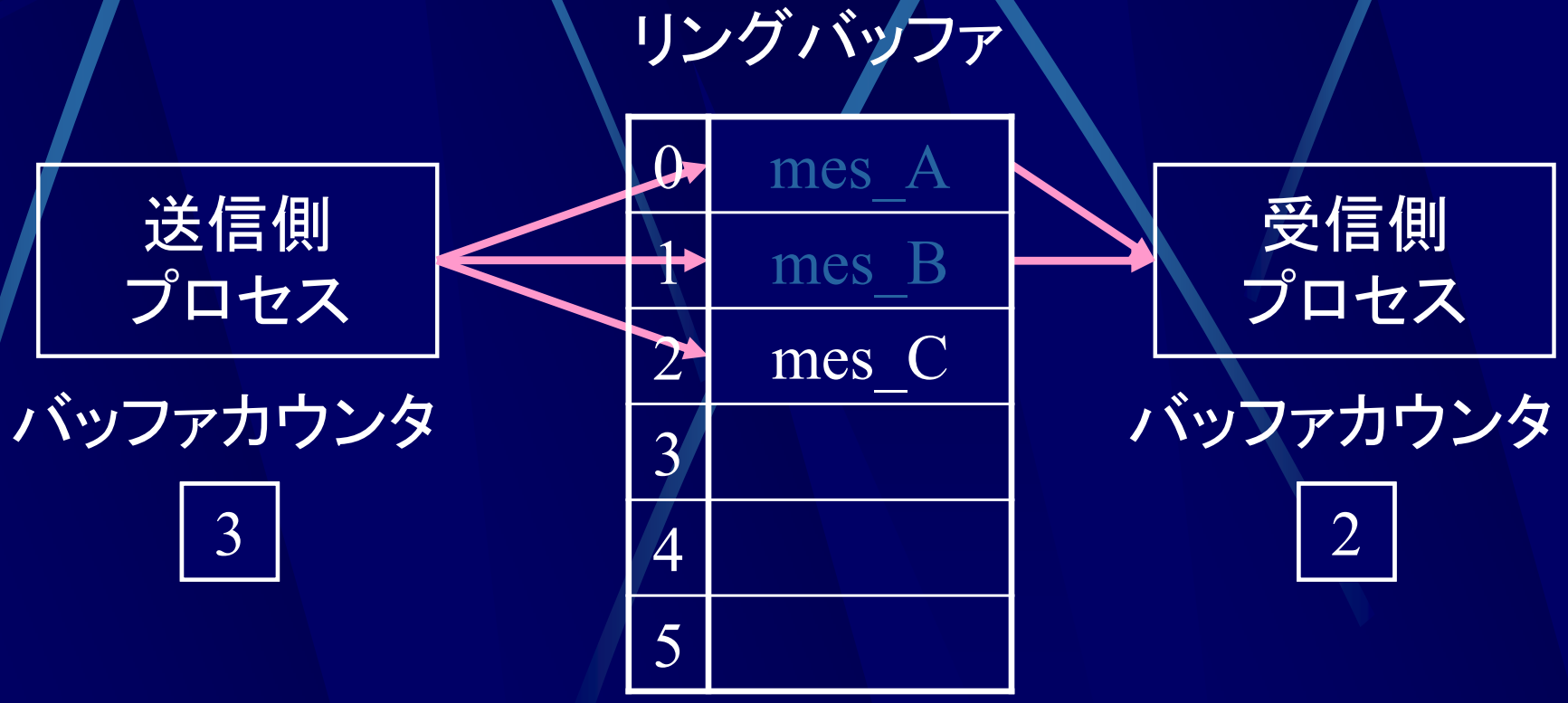


リングバッファ式パイプ処理



一番上と一番下は繋がっている

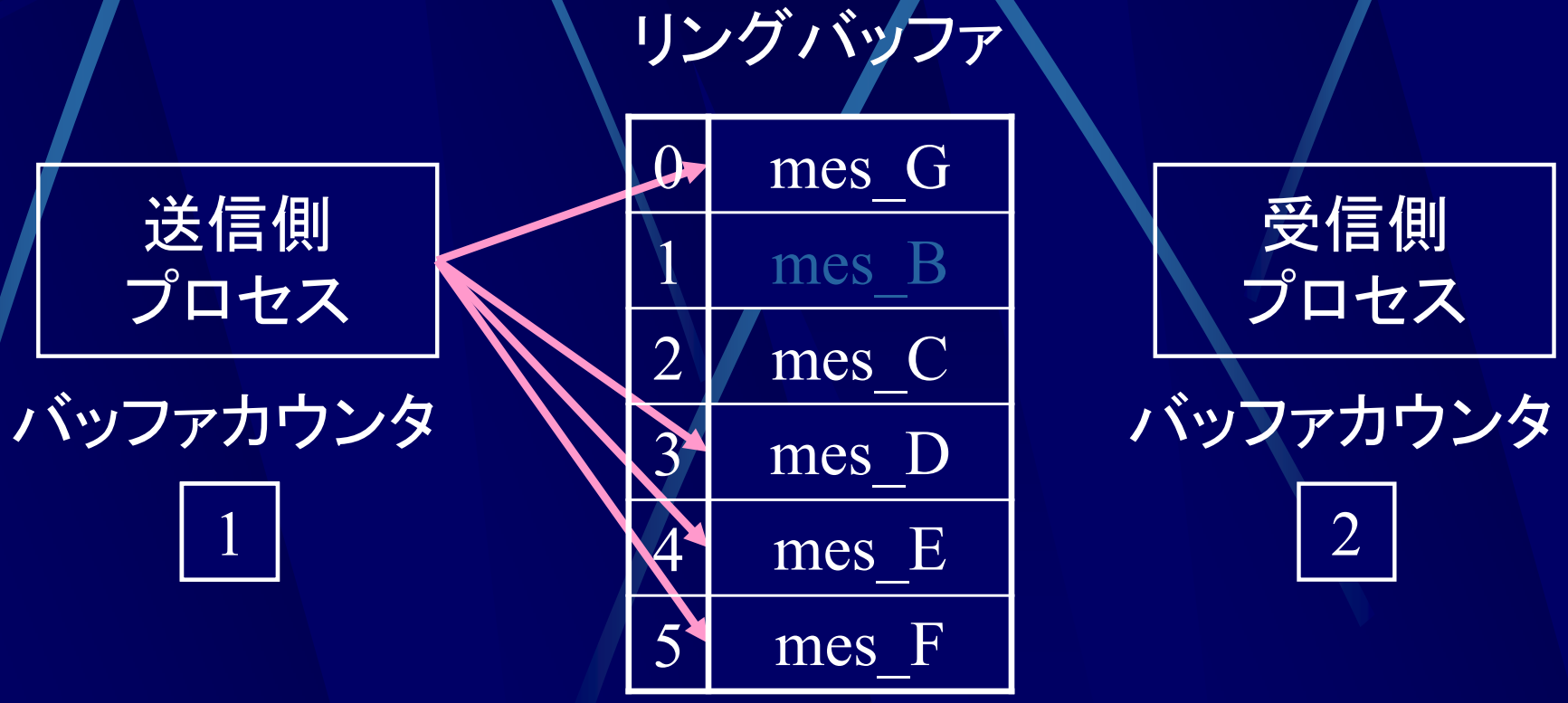
リングバッファ式パイプ処理



バッファに書き込む度に
カウンタの値を増やす

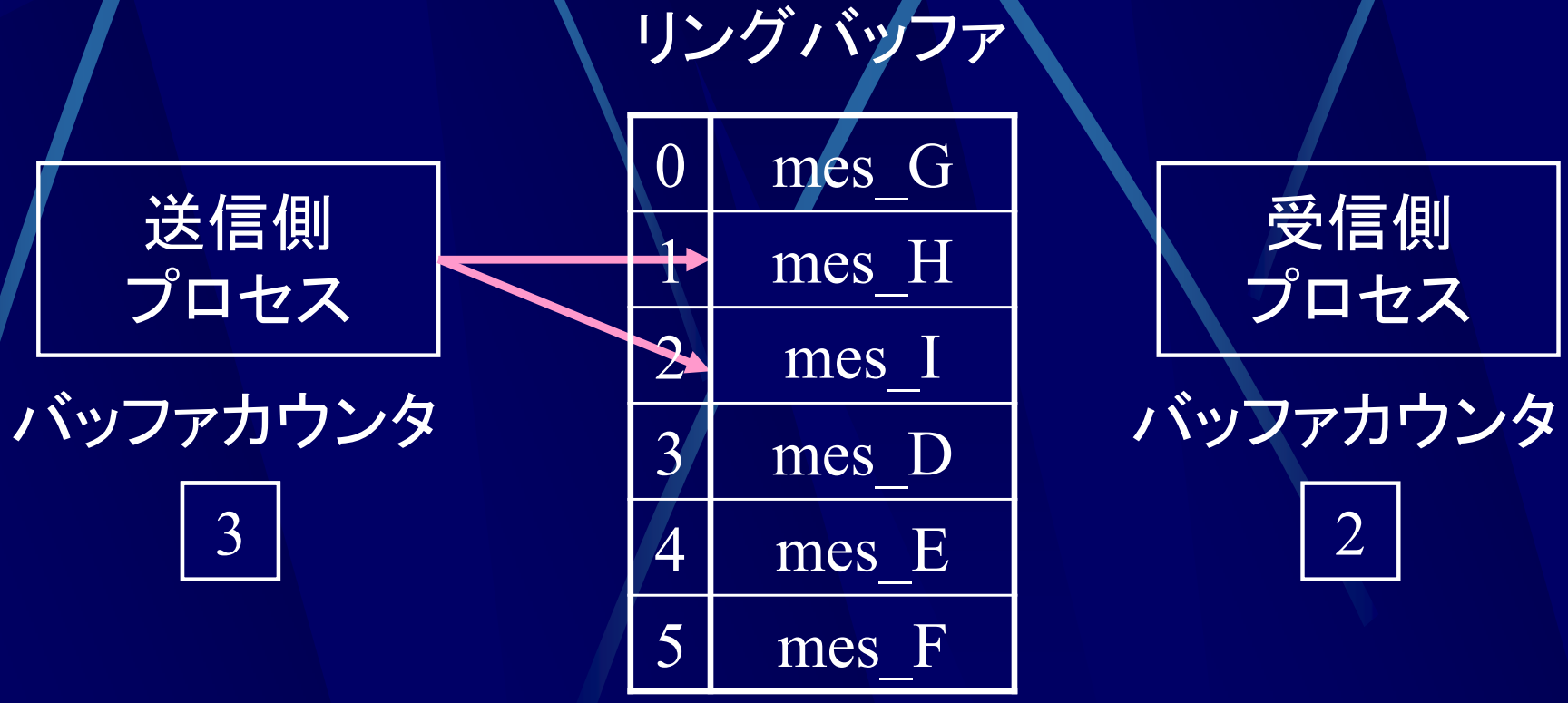
バッファから読み出す度に
カウンタの値を増やす

リングバッファ式パイプ処理



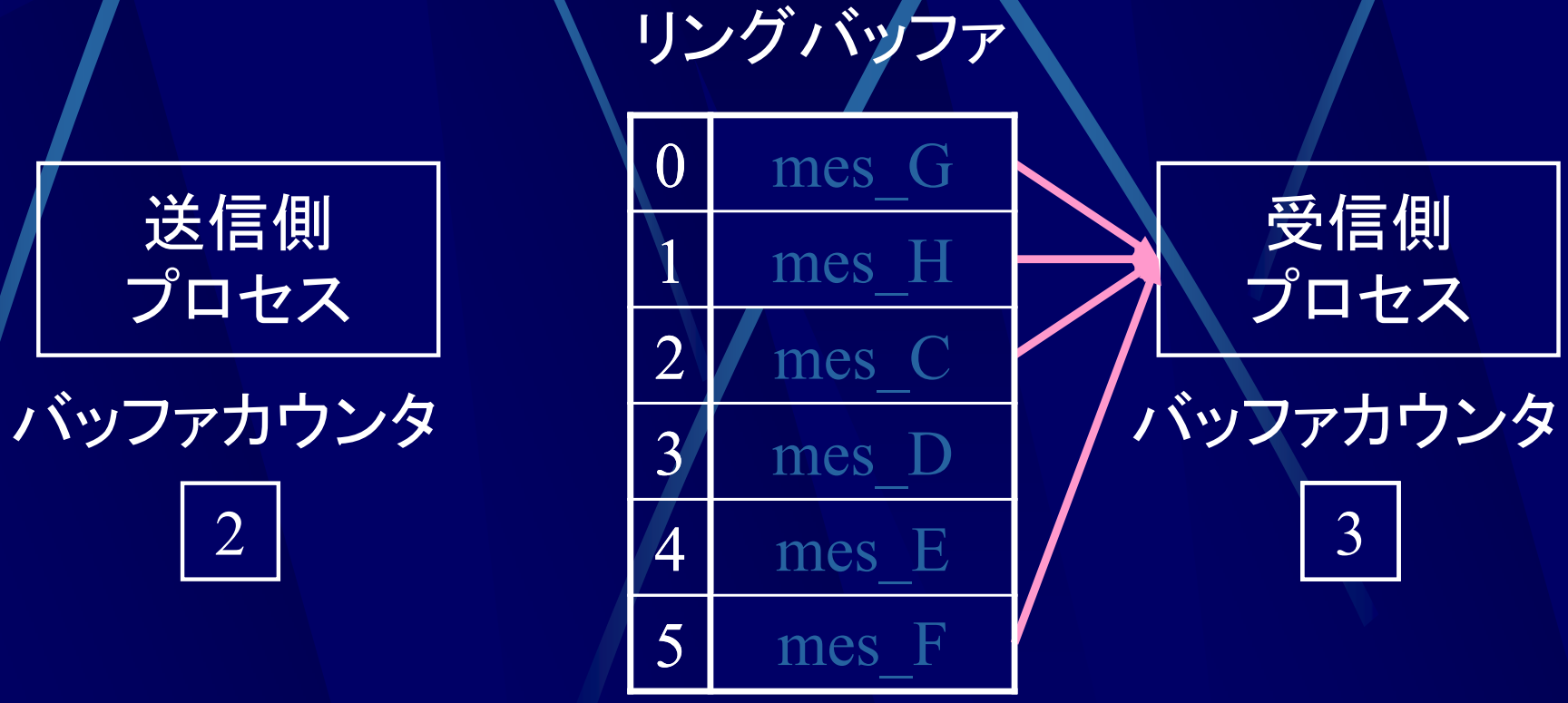
バッファの末尾まで行くと先頭に戻る

リングバッファ式パイプ処理



バッファがいっぱいのときに送信すると
受信側が未読のメッセージを上書きしてしまう

リングバッファ式パイプ処理



バッファが空のときに受信すると
既読のメッセージを再読み出ししてしまう

セマフォを用いたパイプ処理

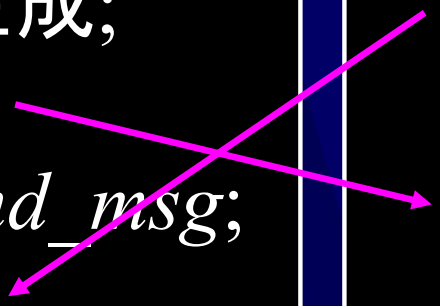
```
semaphore s := N;      /* 空きバッファ数 */
semaphore m := 0;      /* メッセージ数 */
Message Buffer[N]; /* メッセージバッファ */
```

送信側

```
int i := 0;
while (true){
    send_msg の生成;
    wait (s);
    Buffer[i] := send_msg;
    signal (m);
    i := (i + 1) mod N;
}
```

受信側

```
int j := 0;
while (true){
    wait (m);
    recv_msg := Buffer[j];
    signal (s);
    j := (j + 1) mod N;
    recv_mes の処理;
}
```



セマフォを用いたパイプ処理

空きバッファ数　メッセージ数

s 6　　m 0

リングバッファ

0	
1	
2	
3	
4	
5	

送信側プロセス

バッファカウンタ

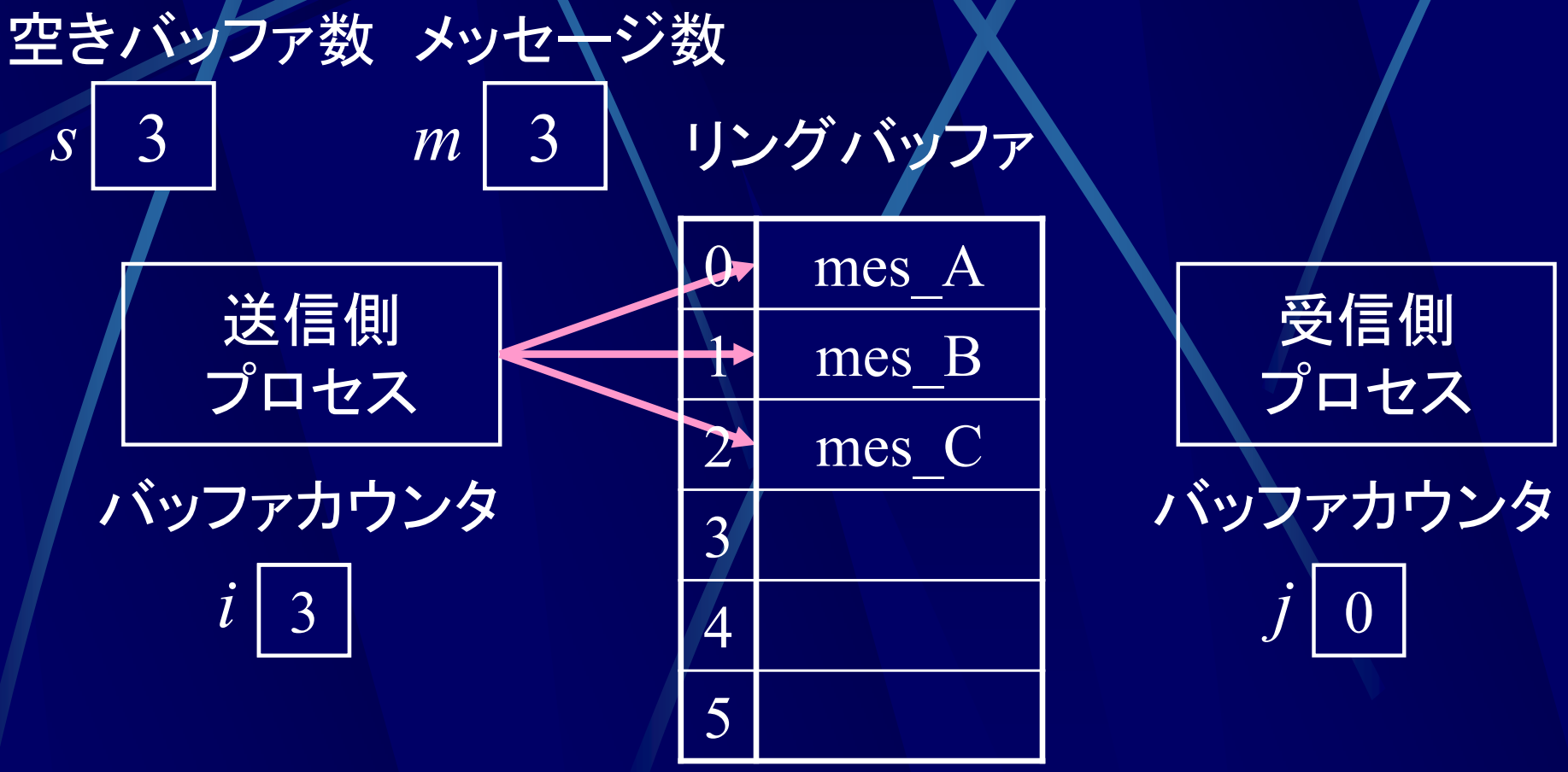
i 0

受信側プロセス

バッファカウンタ

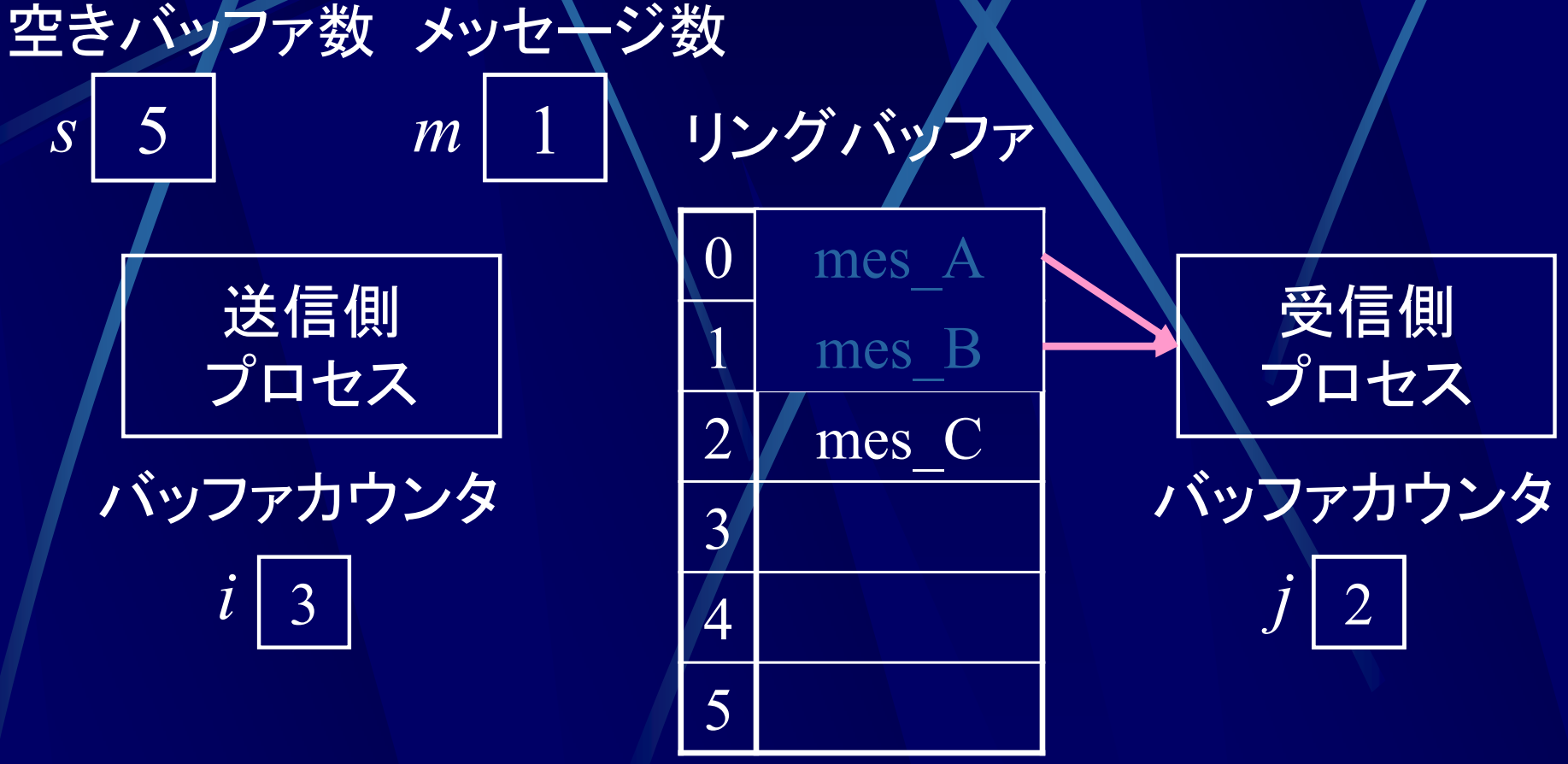
j 0

セマフォを用いたパイプ処理



送信前に s を減らし
送信後に m を増やす

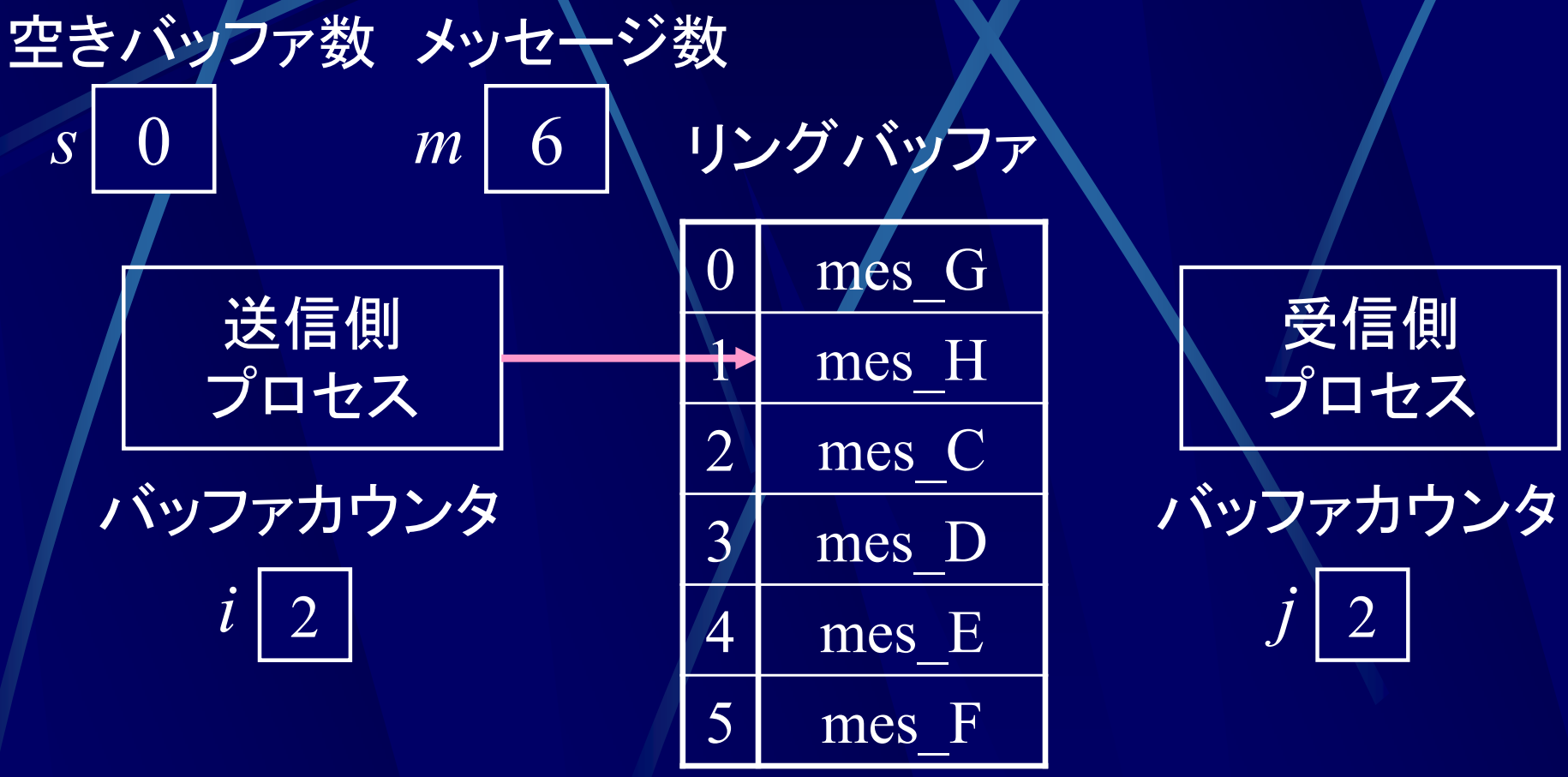
セマフォを用いたパイプ処理



送信前に s を減らし
送信後に m を増やす

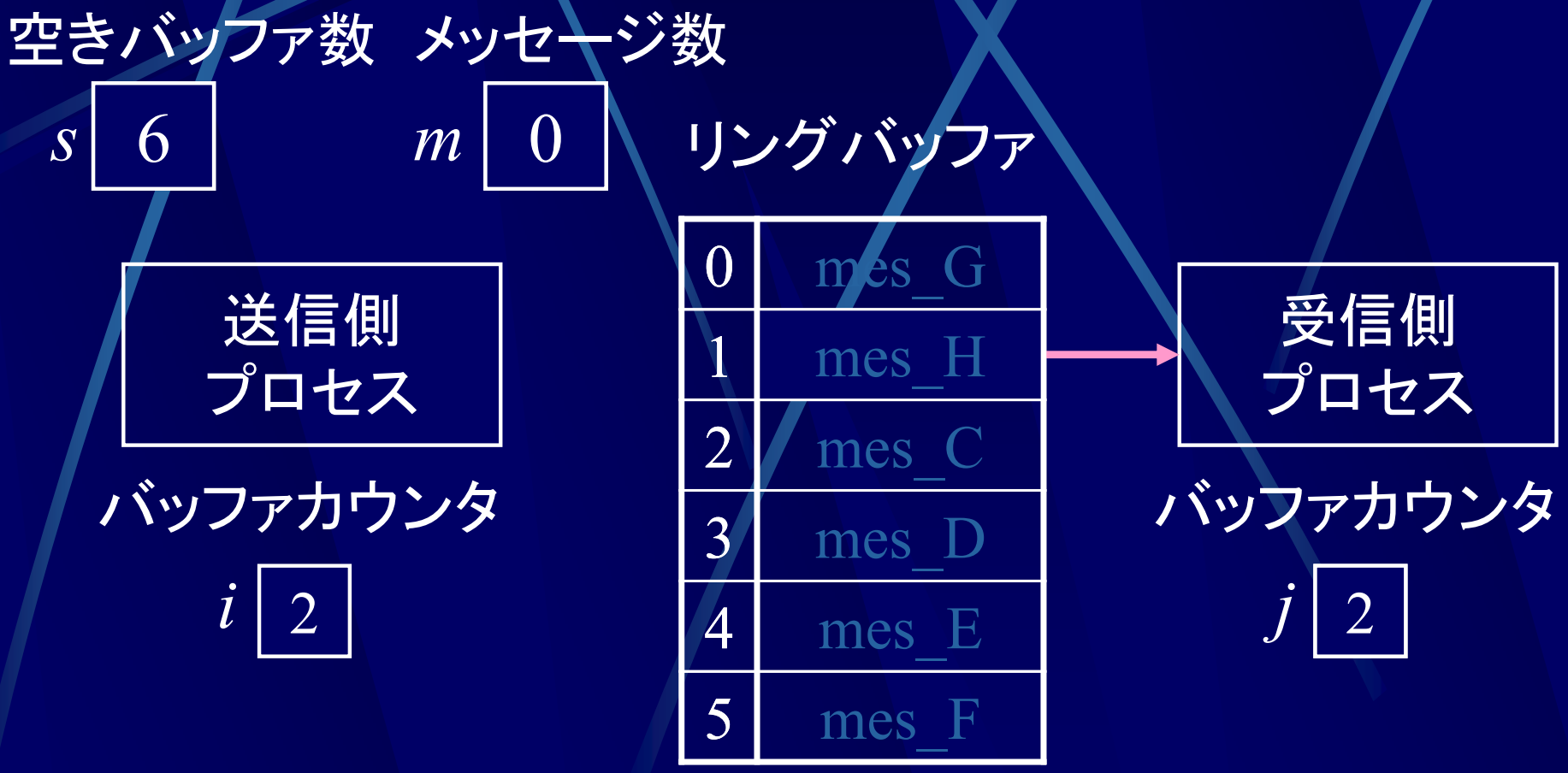
受信前に m を減らし
受信後に s を増やす

セマフォを用いたパイプ処理



s が 0 なので送信は拒否される

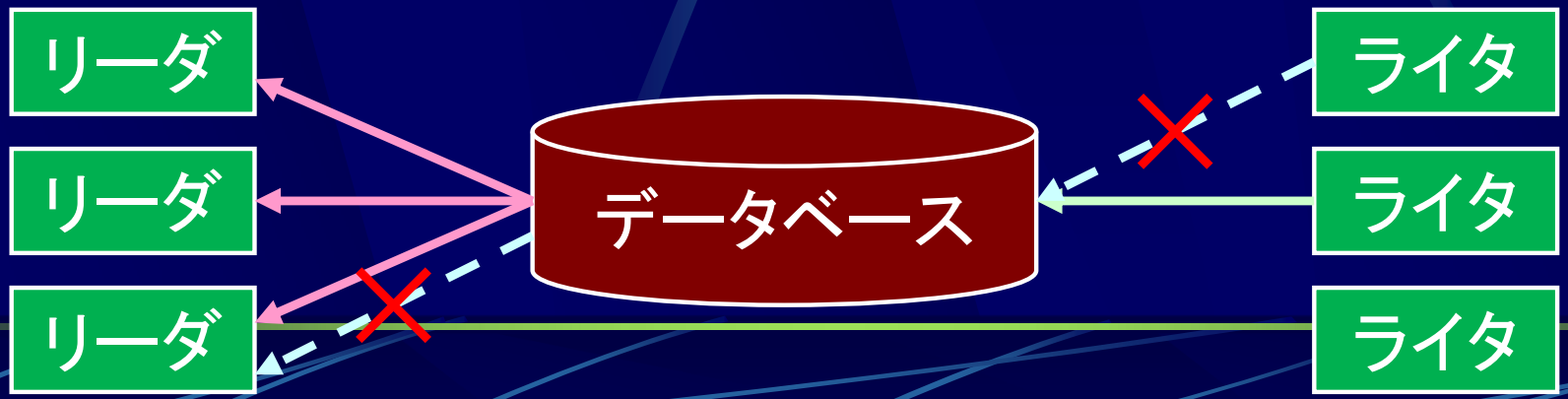
セマフォを用いたパイプ処理



m が 0 なので受信は拒否される

リーダライタ問題

- データベースに対する操作
  - 複数のリーダ(読み手)とライタ(書き手)がいる
  - 読み: 複数のリーダが同時に読み出し可能
  - 書き: ライタの書き込み中は他のリーダの読み出し, 他のライタの書き込みは不可



リーダライタ問題

```
semaphore w := 1;
semaphore m := 1;
int r := 0;
```

ライタ

```
while (true){
    write_data の生成;
    wait (w);
    write (write_data);
    signal (w);
}
```

リーダ

```
while (true){
    wait (m);
    if (r = 0) wait (w);
    ++r;
    signal (m);
    read_data := read();
    wait (m);
    --r;
    if (r = 0) signal (w)
    signal (m);
    read_data の処理;
}
```

セマフォの問題点

- セマフォの問題点
  - wait 命令と signal 命令の順序を間違え易い

```
signal(s);
CS();
wait(s);
NCS();
```

CSが保護されない

```
wait(s);
CS();
wait(s);
NCS();
```

デッドロックとなる

セマフォの問題点

- セマフォの問題点
  - wait 命令と signal 命令の順序を間違え易い
  - 条件文などで wait, signal 命令を飛び越してしまう
  - wait, signal 命令がプログラム内に分散しデバグが困難になる
  - wait 命令時にセマフォの値が 0 だとデッドロック
  - 複数のセマフォのうち 1 つを持つことができない
  - セマフォを必要としないプロセスにもセマフォが見えてしまう

モニタ(monitor)

- モニタ(monitor)
  - B.Hansen と Hoare が提案
  - 共同資源とそれに対する操作の一体構造
  - オブジェクト指向
  - Java の同期機構にも採用

モニタの概念

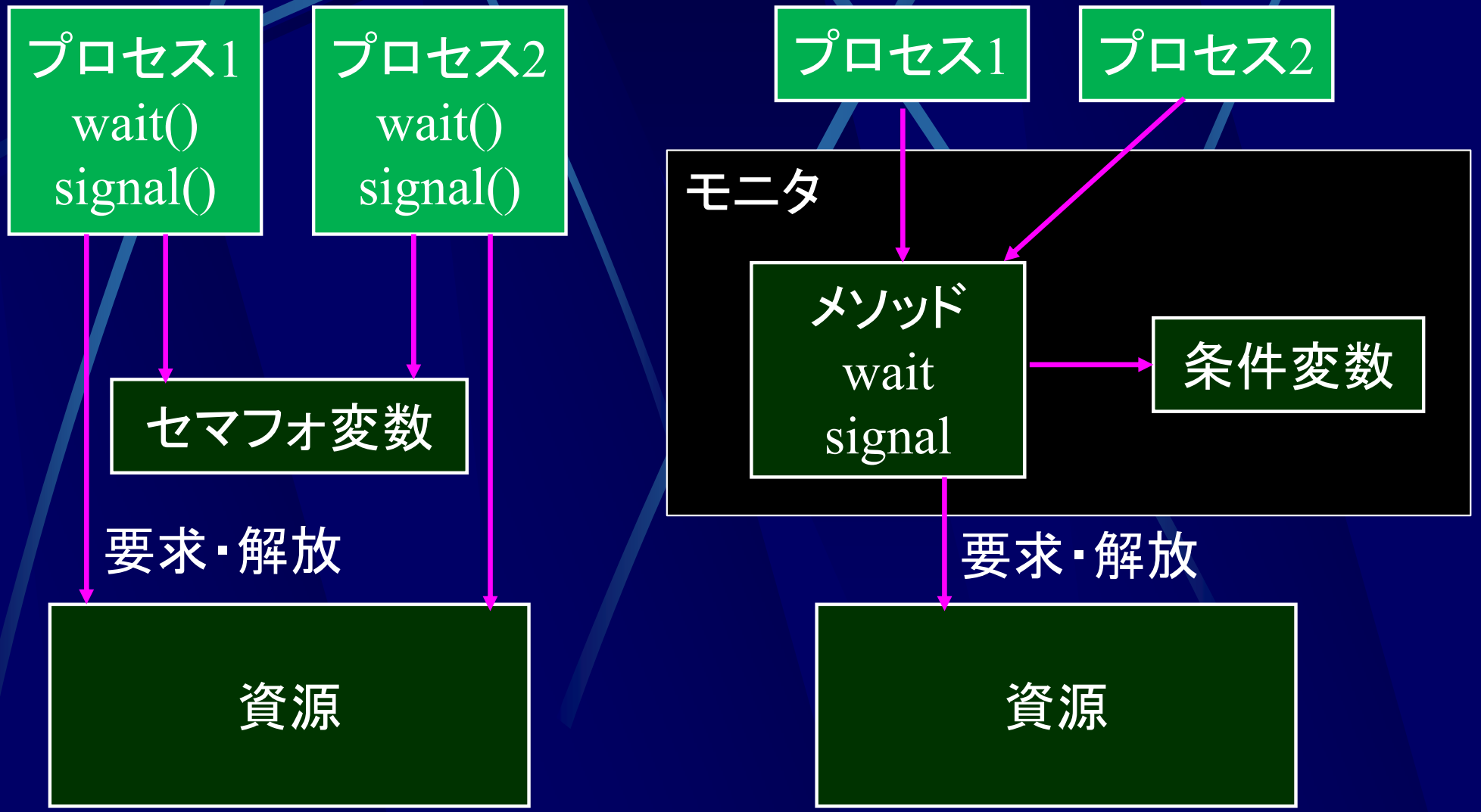
- 局所変数はモニタ内部からのみアクセス可能
- モニタへのアクセスはメソッド呼び出しのみ可能
- wait, signal 命令はモニタ内部でのみ使用可能
  (外部からの使用はメソッド呼び出しのみ)
- 同時にモニタを操作できるのは1プロセスのみ

  ⇒自動的に排他制御が行われる

セマフォのように wait, signal 命令が
プログラム中に散らばることが無い

資源へのアクセスがモニタ内部で完結

セマフォとモニタ


プロセス1
wait()
signal()
プロセス2
wait()
signal()
セマフォ変数
要求・解放
資源
プロセス1
プロセス2
モニタ
メソッド
wait
signal
条件変数
要求・解放
資源
セマフォ
モニタ

条件変数

- 条件変数
  - ある条件が成立するまでプロセスを待機させる
- 条件変数の操作
  - wait
    - 待ち状態に
  - signal
    - 待ち状態にあるプロセスのうち1つを実行可能に
  - queue
    - 待ち状態のプロセスの有無を返す

モニタを用いた相互排除

変数は全て private

```
monitor {
   private condition resource ;        /* 条件変数 */
   private int s :=1;                  /* 空き資源の数 */

   public void csBegin() {             /* 臨界領域開始処理 */
      if (s = 0) resource.wait ;       /* 資源を要求 */
      --s;
   }

   public void csEnd() {               /* 臨界領域終了処理 */
      ++s;
      resource.signal;                 /* 資源解放 */
   }
}
```

wait, signal は monitor 内部でのみ使用可能

モニタを用いた相互排除

セマフォ場合

```
wait (s);                  /* 資源を要求 */
CS_i();                    /* 臨界領域 */
signal (s);                /* 資源解放 */
NCS_i();                   /* 非臨界領域 */
```

モニタの場合

```
monitor.csBegin(); /* 資源を要求 */
CS_i();            /* 臨界領域 */
monitor.csEnd();   /* 資源解放 */
NCS_i();           /* 非臨界領域 */
```

セマフォ変数や
wait, signal 命令を
扱わなくていい

モニタを用いたパイプ処理

(変数宣言部)

```
monitor {
   private condition empty, full ;          /* 条件変数 */
   private int i ;          /* バッファへの次の書き込み位置 */
   private int j ;          /* バッファからの次の読み出し位置 */
   private int m ;          /* バッファ中のメッセージ数 */
   private Message Buffer[N];               /* バッファ */
                                        (次へ続く)
```

変数は全て private 変数

(メソッド部) モニタを用いたパイプ処理

```
public void put (Message msg) { /* バッファへの書き込み */
    if (m ≧ N) full.wait;        /* バッファがいっぱいなら待つ */
    Buffer[i] := msg; i := (i +1) mod N; ++m;   /* 書き込み */
    empty.signal;
}

public Message get() {                  /* バッファからの読み出し */
    if (m = 0) empty.wait;                  /* バッファが空なら待つ */
    msg := Buffer[j]; j := (j +1) mod N; --m;    /* 読み出し */
    full.signal;
    return msg;
}
}
```

資源を使用するプロセスは
put(), get() を呼び出すだけでいい

モニタを用いたパイプ処理

送信側

```
while (true){
   send_msg の生成;
   monitor.put (send_msg);
}
```

受信側

```
while (true){
   recv_msg := monitor.get();
   recv_mes の処理;
}
```

セマフォ変数や
位置を示す変数を
扱わなくていい

セマフォとモニタ

- セマフォ
  - wait 命令でしか資源の有無がわからない
  - 資源が無かった場合は強制的に待ち状態
    - Java 等では try&wait 命令が実装
      - 資源を獲得できれば true を、できなければ false を返す
- モニタ
  - 資源の有無を queue 命令で調べられる

    ⇒資源が無い場合に待ちに入るか選択できる

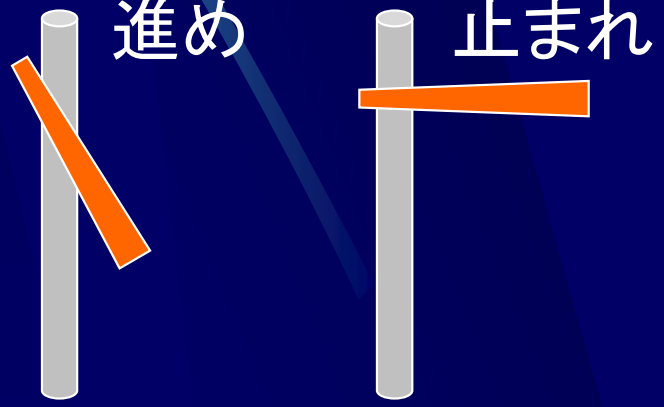
セマフォとモニタ

	セマフォ	モニタ
提案者	Dijkstra(1965)	B.Hansen(1973) Hoare(1974)
形態	手続き (サブルーチン)	構造化 (オブジェクト指向)
可読性/保守性	低	高
提供形態	ライブラリ等	言語仕様の一部
対応言語	多	少(Java)

コラム:P命令, V命令とは？

はっきりとした由来は不明
オランダ語の頭文字らしいが…

- P命令
  - Passren (進め)
  - Prolagen (probeer te verlagen, 減らす)
  - Proberen (テストする)
  - Passeer (通過)
- V命令
  - Verhoog (止まれ)
  - Verhogen (probeer te verhogen, 増やす)



Java でのセマフォ使用

- Semaphore クラスを使用
  - java.util.concurrent.Semaphore

コンストラクタ

```
public Semaphore (int permit)   /* permit : 資源数 */
```

wait 命令

```
public void acquire ()
        throws InterruptedExeption
```

signal 命令

```
public void release ()
```

```
import java.util.concurrent.Semaphore;
class SemaphoreSample {
        /* セマフォ変数の生成 */
        Semaphore s = new Semaphore (5); // 資源数5

        /* セマフォ変数への wait 命令 */
        try {
                s.acquire();
        } catch (InterruptedException e) {
                System.exit(1);
        }

        /* セマフォ変数への signal 命令 */
        s.release();
}
```

Java でのモニタ使用

- Synchronized 文を使用

```
Synchronized (Expression)
                    /* Expression : ロックオブジェクト */
        Block
            /* このブロックは1スレッドしか入れない */
```

```
class MonitorSample {
        /* モニタ変数の生成 */
        private m = 5; // 資源数5
        Object lock = new Object(); /* ロックオブジェクト */
        /* モニタ変数への wait メソッド */
        void wait () {
                while (true) {
                        Synchronized (lock) {
                                if (m>0) {
                                        --m;
                                        return;
                                }
                        }
                しばらく待つ;
                }
        }
}
```

この中へは1スレッドしか入れない

参考：セマフォを用いた相互排除プログラム(Java)

- SemaphoreMutex.java
  - セマフォを用いた相互排除アルゴリズム (スレッド数4, 資源数2)

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：セマフォを用いた相互排除プログラム(Java)

実行例

```
$ javac SemaphoreMutex.java
$ java SemaphoreMutex
 1 : CS begin
 0 :              CS begin
 1 :              CS end
 0 :                        CS begin
 1 : CS end
 0 :                                  CS begin
 0 :                        CS end
```

CSに入れるのは
同時には2つまで

セマフォ変数(空いている資源数)の値

参考：セマフォを用いたパイプ処理プログラム(Java)

- SemaphorePipe.java
  - セマフォを用いたパイプ処理アルゴリズム (スレッド数2, バッファサイズ4)

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：セマフォを用いたパイプ処理プログラム(Java)

実行例

```
$ javac SemaphorePipe.java
$ java SemaphorePipe
 b[0] ← ‘o’
                    b[0] → ‘o’
 b[1] ← ‘m’
 b[2] ← ‘s’
 b[3] ← ‘a’
                    b[1] → ‘m’
                    b[2] → ‘s’
```

読み/書きを行ったバッファの位置

参考：モニタを用いた相互排除プログラム(Java)

- MonitorMutex.java
  - モニタを用いた相互排除アルゴリズム (スレッド数4, 資源数2)

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：モニタを用いたパイプ処理プログラム(Java)

- MonitorPipe.java
  - モニタを用いたパイプ処理アルゴリズム
    (スレッド数2, バッファサイズ4)

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