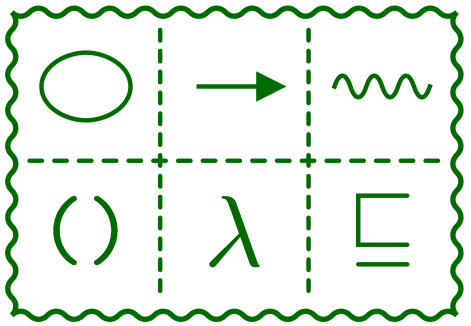
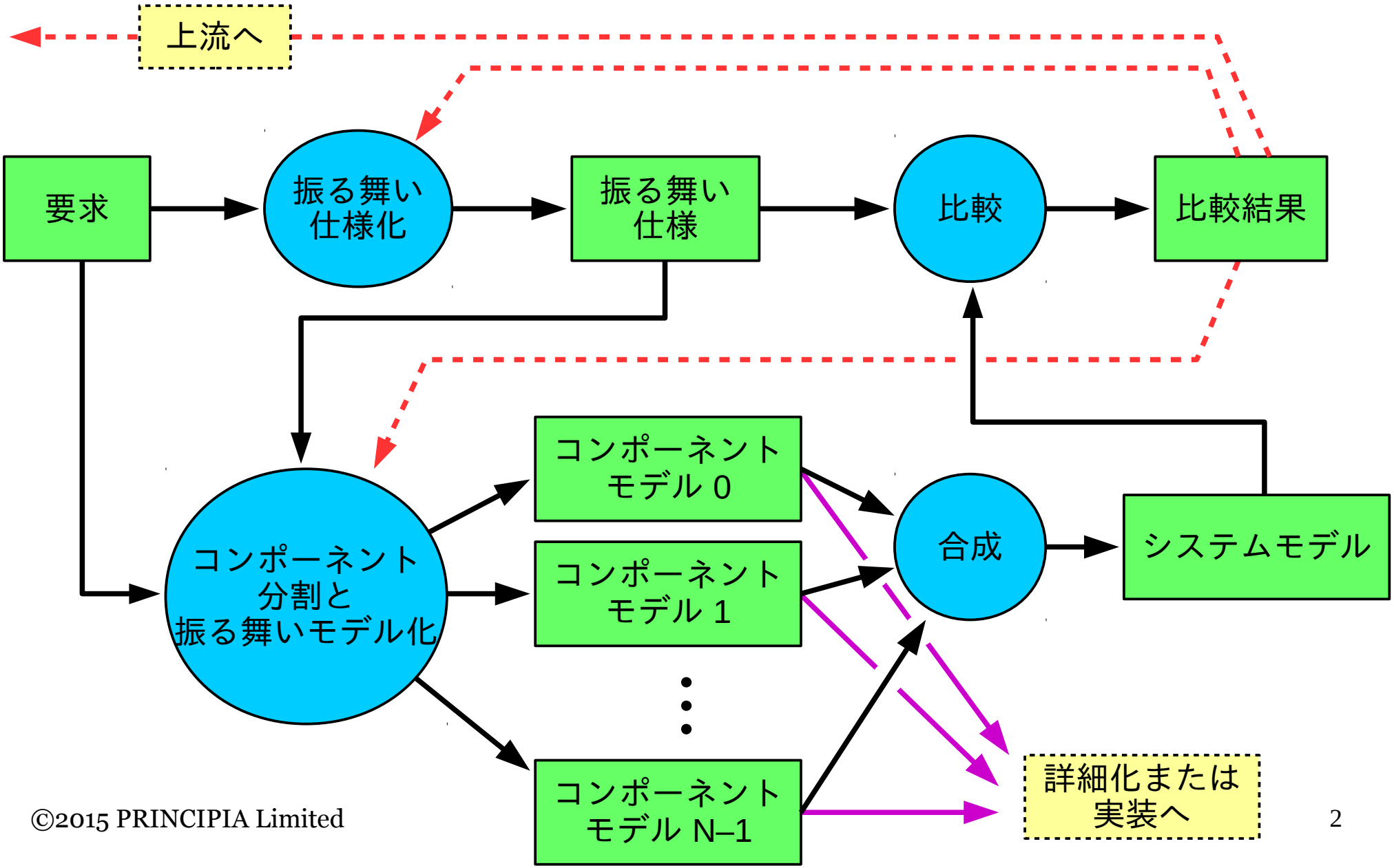


並行システムの検証と実装

第12章 並行システムの実装 1
同期機構による実装

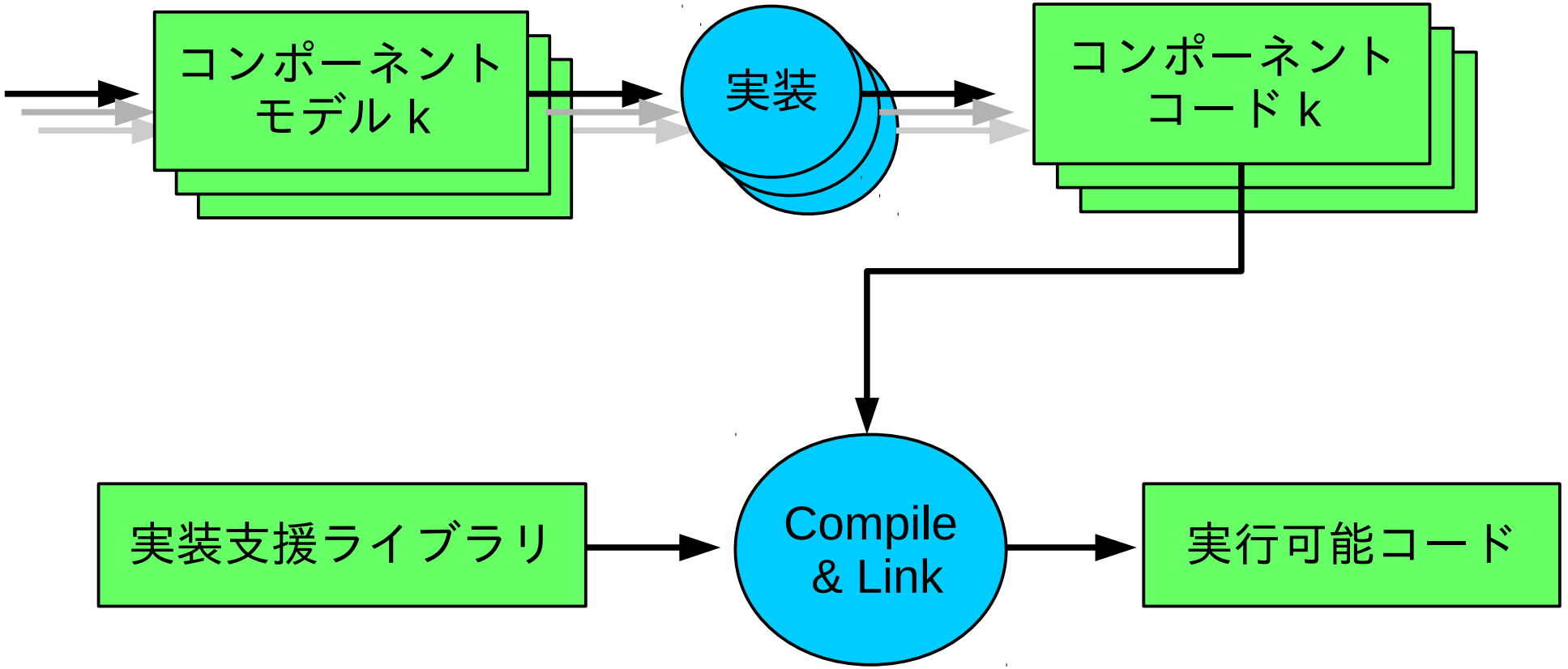
PRINCIPIA Limited
初谷 久史

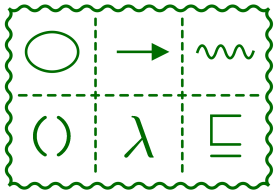
システムの設計（振る舞い側面）



モデルから実装へ

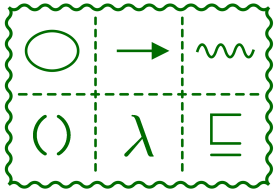
上流から





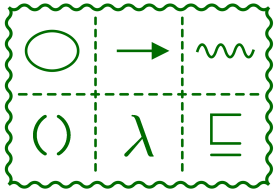
並行システムの実装

- CSP 実装支援ライブラリ MCCSP による実装
- 同期機構による実装
- 不可分操作によるロックフリーアルゴリズムの実装



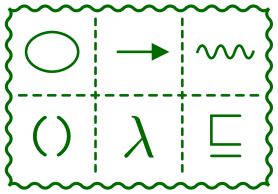
モデルから実装へ: 2つのケース

- システムを CSP の考え方（イベントによる同期型相互作用）で構築する
  - システムの振る舞いを CSP でモデル化し検査する
  - CSP の考え方をサポートするプログラミング言語やライブラリを使って実装する
- システムをオペレーティングシステムが提供する同期機構を使って実装する
  - CSP でモデル化した同期機構を部品としてシステムのモデルを作成し検査する
  - 同期機構を使って実装する



2つのケースの比較

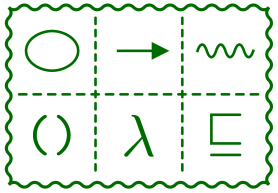
	利点	欠点
CSP	考え方がシンプルで，モデル化しやすく，検査において理論とツールの支援を受けやすい	オペレーティングシステム，プログラミング言語，ライブラリ等の支援が少ない
同期機構	現在の計算機アーキテクチャに適合しており性能が出しやすい	モデル化が難しく，検査すべき性質も表現しにくい．モデルの規模が大きくなりがちでツールの能力を超えやすい



POSIX.1 (IEEE Std 1003.1)

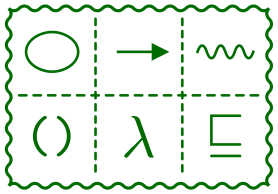
"*POSIX.1-2008 defines a standard operating system interface and environment, including a command interpreter (or "shell"), and common utility programs to support applications portability at the source code level. It is intended to be used by both application developers and system implementors.*"

http://pubs.opengroup.org/onlinepubs/9699919799/



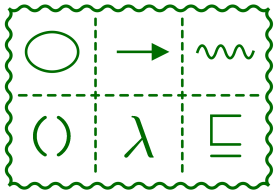
POSIX.1 スレッド関連

- スレッドライブラリ pthread.h
  - スレッドの作成と終了
  - ミューテックス
  - 条件変数
- 共有メモリ sys/mman.h
- セマフォ semaphore.h



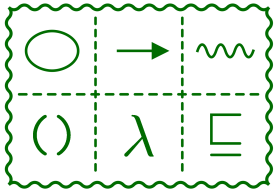
同期機構の比較

	POSIX	Win32	µITRON
Thread	pthread_create pthread_join pthread_exit	CreateThread _beginthreadex	cre_tsk
Mutex	pthread_mutex_lock pthread_mutex_unlock	WaitForSingleObject ReleaseMutex EnterCriticalSection LeaveCriticalSection	loc_mtx unl_mtx
Condition Variable	pthread_cond_wait pthread_cond_signal pthread_cond_broadcast	SleepConditionVariableCS WakeConditionVariable WakeAllConditionVariable	
Shared Memory	shm_open shm_unlink mmap	CreateFileMapping MapViewOfFile	
Semaphore	sem_wait sem_post	WaitForSingleObject ReleaseSemaphore	wai_sem sig_sem
Event Object Event Flag		WaitForSingleObject SetEvent ResetEvent	set_flg clr_flg wai_flg



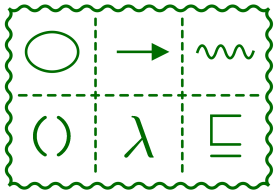
プロセス間での使用

	プロセス間で使用可能か?
Mutex	option pthread_mutexattr_setpshared PTHREAD_PROCESS_SHARED
Condition Variable	option pthread_condattr_setpshared PTHREAD_PROCESS_SHARED
Semaphore	YES
Shared Memory	YES



スレッドの作成と終了

- 作成
  - pthread_create
- 終了
  - return or pthread_exit
- 終了待ち
  - pthread_join



pthread_create

スレッドを作成する

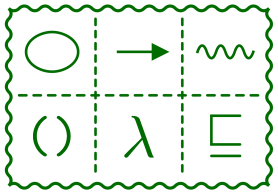
```
int pthread_create(pthread_t *thread,
     const pthread_attr_t *attr,
     void *(*start_routine)(void*), void *arg)
```

Parameters

thread	作成したスレッドの ID を格納する変数へのポインタ
attr	属性（既定値の場合は NULL を指定）
start_routine	スレッドの処理を記述した関数へのポインタ
arg	*start_routine* に渡す引数

Return Value

成功の場合は 0，失敗の場合はエラー番号



pthread_join

スレッドの終了を待つ

```
int pthread_join(pthread_t thread, void **value_ptr)
```

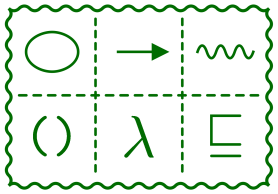
Parameters

thread　　スレッド ID

value_ptr　　スレッドの終了値を格納する変数へのポインタ

Return Value

成功の場合は 0，失敗の場合はエラー番号



pthread_exit

スレッドを終了する

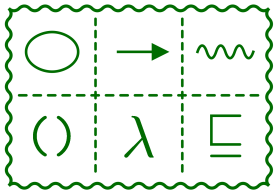
```
void pthread_exit(void *value_ptr)
```

Parameters

value_ptr　　スレッドの終了値

Return Value

なし

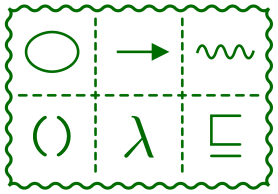


pthread_create 例

```
void *thread_entry(void *arg)
{
    printf("hello, thread %s\n", (char *)arg);
    return arg;
}
```

start_routine から return するとスレッドが終了する

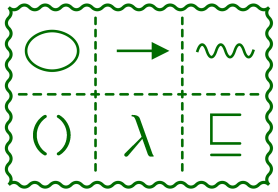
関数呼び出しの深いところで終了するには
pthread_exit を使う



pthread_create 例

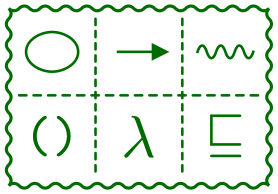
```
#include <stdio.h>
#include <pthread.h>
#include <assert.h>
```

```
int main()
{
    pthread_t th;
    void *arg, *retcode;
    int r;

    arg = "foo";
    r = pthread_create(&th, NULL, &thread_entry, arg);
    assert(r == 0);
    r = pthread_join(th, &retcode);
    assert(r == 0);
    assert(retcode == arg);
    return 0;
}
```



ミューテックス

- 作成と破壊
  - pthread_mutex_init
  - pthread_mutex_destroy
- ロックとアンロック
  - pthread_mutex_lock
  - pthread_mutex_unlock



pthread_mutex_init

ミューテックスを初期化する

```
int pthread_mutex_init(pthread_mutex_t *mutex,
    const pthread_mutexattr_t *attr)
```

Parameters

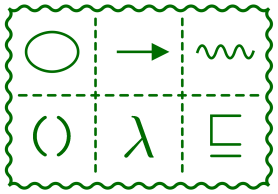
mutex　　ミューテックスへのポインタ
attr　　属性（既定値の場合は NULL を指定）

Return Value

成功の場合は 0，失敗の場合はエラー番号

Remarks

属性が既定値の場合は mutex = PTHREAD_MUTEX_INITIALIZER で初期化できる



pthread_mutex_destroy

ミューテックスを破壊する

```
int pthread_mutex_destroy(pthread_mutex_t *mutex)
```

Parameters

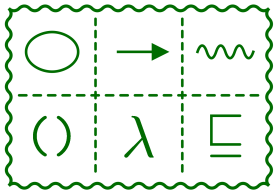
mutex　　ミューテックスへのポインタ

Return Value

成功の場合は 0，失敗の場合はエラー番号

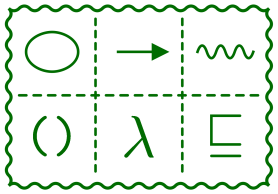
Remarks

ロックされているミューテックスに対して呼び出した場合の挙動は未定義



ミューテックスの例

```
pthread_mutex_t mutex;

r = pthread_mutex_init(&mutex, NULL);
assert(r == 0);

/* use mutex */

r = pthread_mutex_destroy(&mutex);
assert(r == 0);
```



pthread_mutex_lock / unlock

ミューテックスをロック・アンロックする

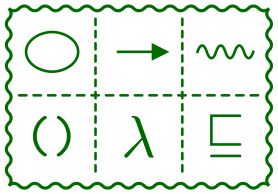
```
int pthread_mutex_lock(pthread_mutex_t *mutex)
int pthread_mutex_unlock(pthread_mutex_t *mutex)
```

Parameters

mutex　　ミューテックスへのポインタ

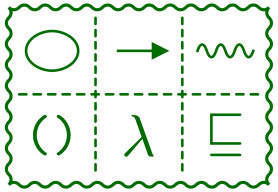
Return Value

成功の場合は 0，失敗の場合はエラー番号



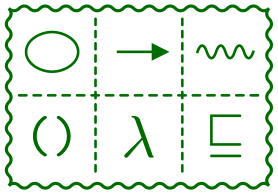
ミューテックスの例

```
r = pthread_mutex_lock(&mutex);
assert(r == 0);

/* critical section */

r = pthread_mutex_unlock(&mutex);
assert(r == 0);
```



条件変数

- 作成と破壊
  - pthread_cond_init
  - pthread_cond_destroy
- 操作
  - pthread_cond_wait
  - pthread_cond_signal
  - pthread_cond_broadcast



pthread_cond_init

条件変数を初期化する

```
int pthread_cond_init(pthread_cond_t *cond,
     const pthread_condattr_t *attr)
```

Parameters

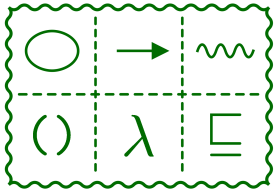
cond	条件変数へのポインタ
attr	属性（既定値の場合は NULL を指定）

Return Value

成功の場合は 0，失敗の場合はエラー番号

Remarks

属性が既定値の場合は cond = PTHREAD_COND_INITIALIZER で初期化できる



pthread_cond_destroy

条件変数を破壊する

```
int pthread_cond_destroy(pthread_cond_t *cond)
```

Parameters

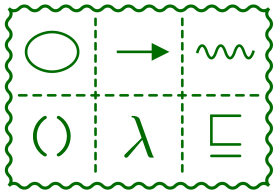
cond　　条件変数へのポインタ

Return Value

成功の場合は 0，失敗の場合はエラー番号

Remarks

待ちスレッドがある条件変数に対して呼び出した場合の挙動は未定義



pthread_cond_wait

スレッドを待ち状態にする

```
int pthread_cond_wait(pthread_cond_t *cond,
                      pthread_mutex_t *mutex)
```

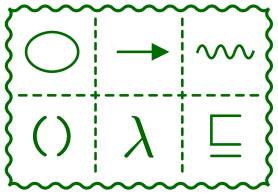
Parameters

cond　　条件変数へのポインタ

mutex　　ミューテックスへのポインタ

Return Value

成功の場合は 0，失敗の場合はエラー番号



pthread_cond_signal

条件変数を待っているスレッドを1つ解放する

```
int pthread_cond_signal(pthread_cond_t *cond)
```

Parameters

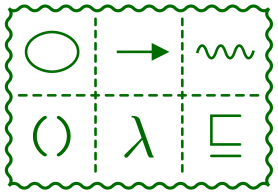
cond　　条件変数へのポインタ

Return Value

成功の場合は 0，失敗の場合はエラー番号

Remarks

待ちスレッドがない場合は何の効果もない
解放されるスレッドの選択はスケジューリングポリシーによる



pthread_cond_broadcast

条件変数を待っているスレッドをすべて解放する

```
int pthread_cond_broadcast(pthread_cond_t *cond)
```

Parameters

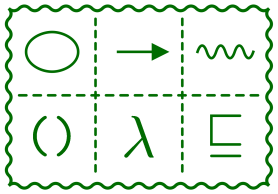
cond　　条件変数へのポインタ

Return Value

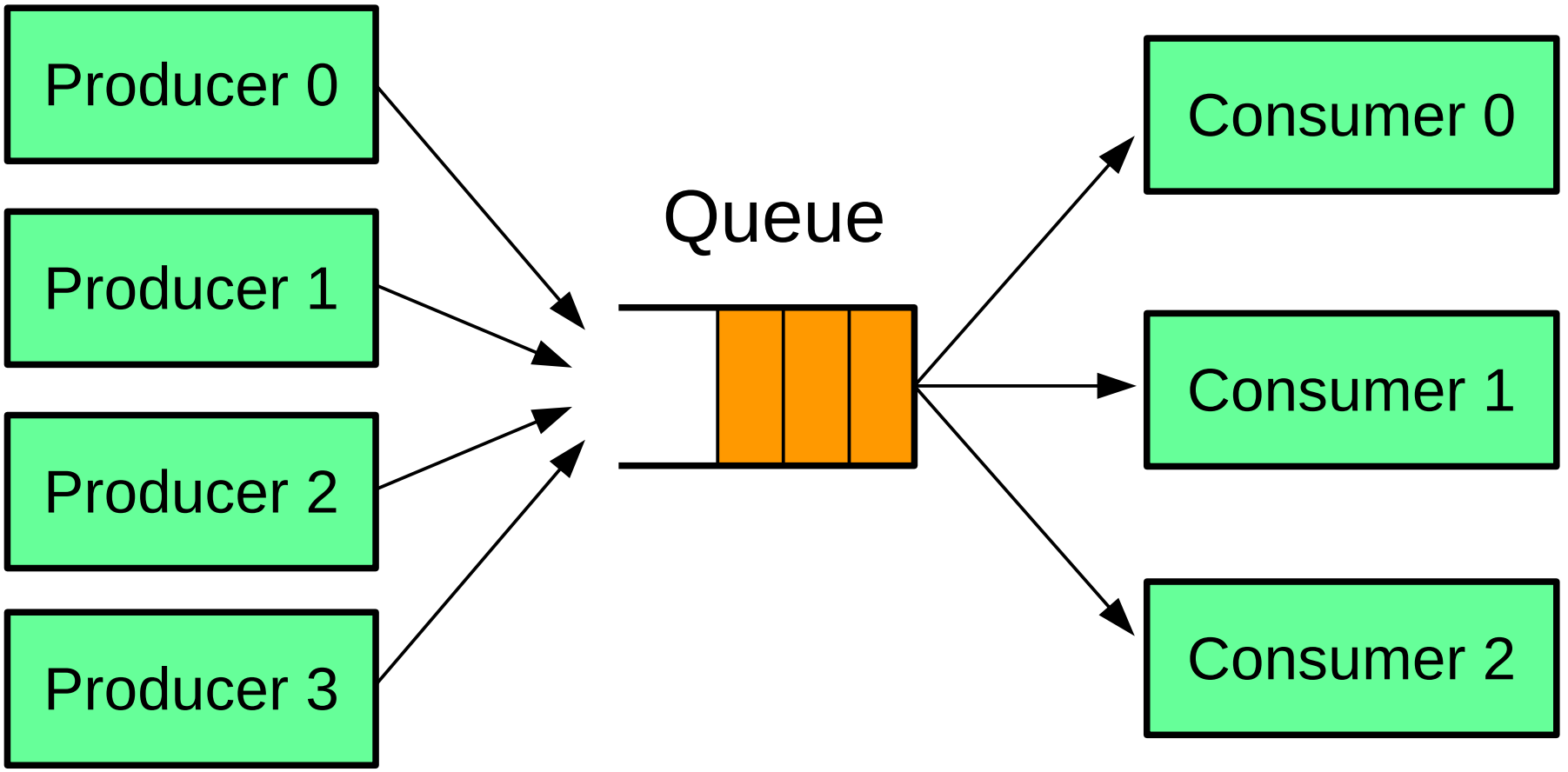
成功の場合は 0，失敗の場合はエラー番号

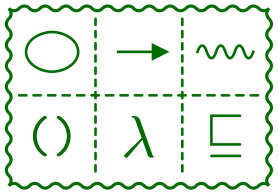
Remarks

待ちスレッドがない場合は何の効果もない
解放されるスレッドの選択はスケジューリングポリシーによる



生産者・消費者問題





生産者・消費者問題の仕様

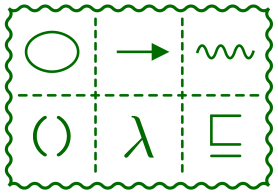
```
(define NP 2)
(define NC 2)
(define L 2)
(define M 2)
```

NP: 生産者数
NC: 消費者数
L: バッファの大きさ
M: データの種類

```
(define IM (interval 0 M))
(define DM (map list IM))

(define-channel in (x) DM)
(define-channel out (x) DM)

(define-channel enq (x) DM)
(define-channel deq (x) DM)
```

データの生産・消費の代わりにチャネルの入出力を使う

仕様記述のための内部チャネル



生産者・消費者問題の仕様

生産者

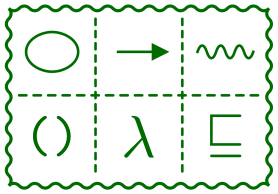
```
(define-process IN (? in (x) (! enq (x) IN)))
```

消費者

```
(define-process OUT (? deq (x) (! out (x) OUT)))
```

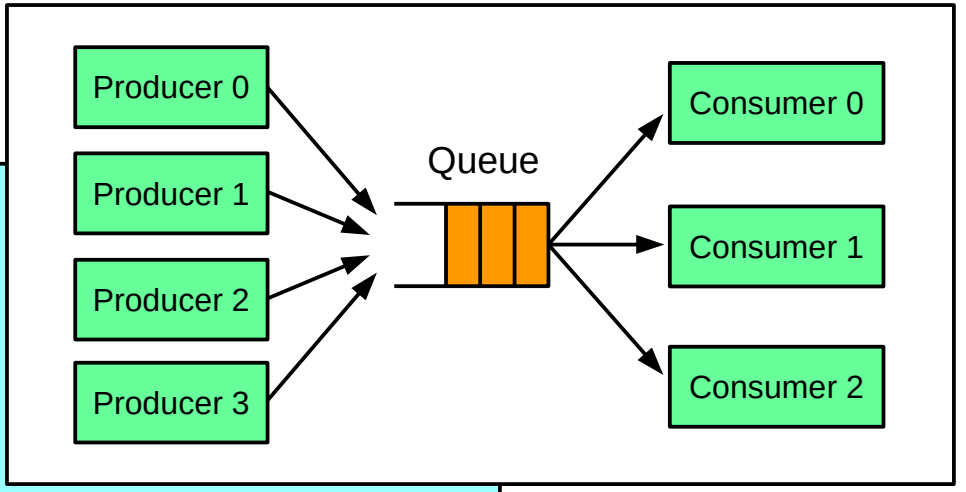
Queue

```
(define-process (QUE xs)
  (alt
    (if (< (length xs) L)
        (? enq (x) (QUE (append xs (list x))))
        STOP)
    (if (null? xs)
        STOP
        (! deq ((car xs)) (QUE (cdr xs))))))
```

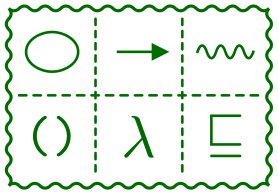


生産者・消費者問題の仕様

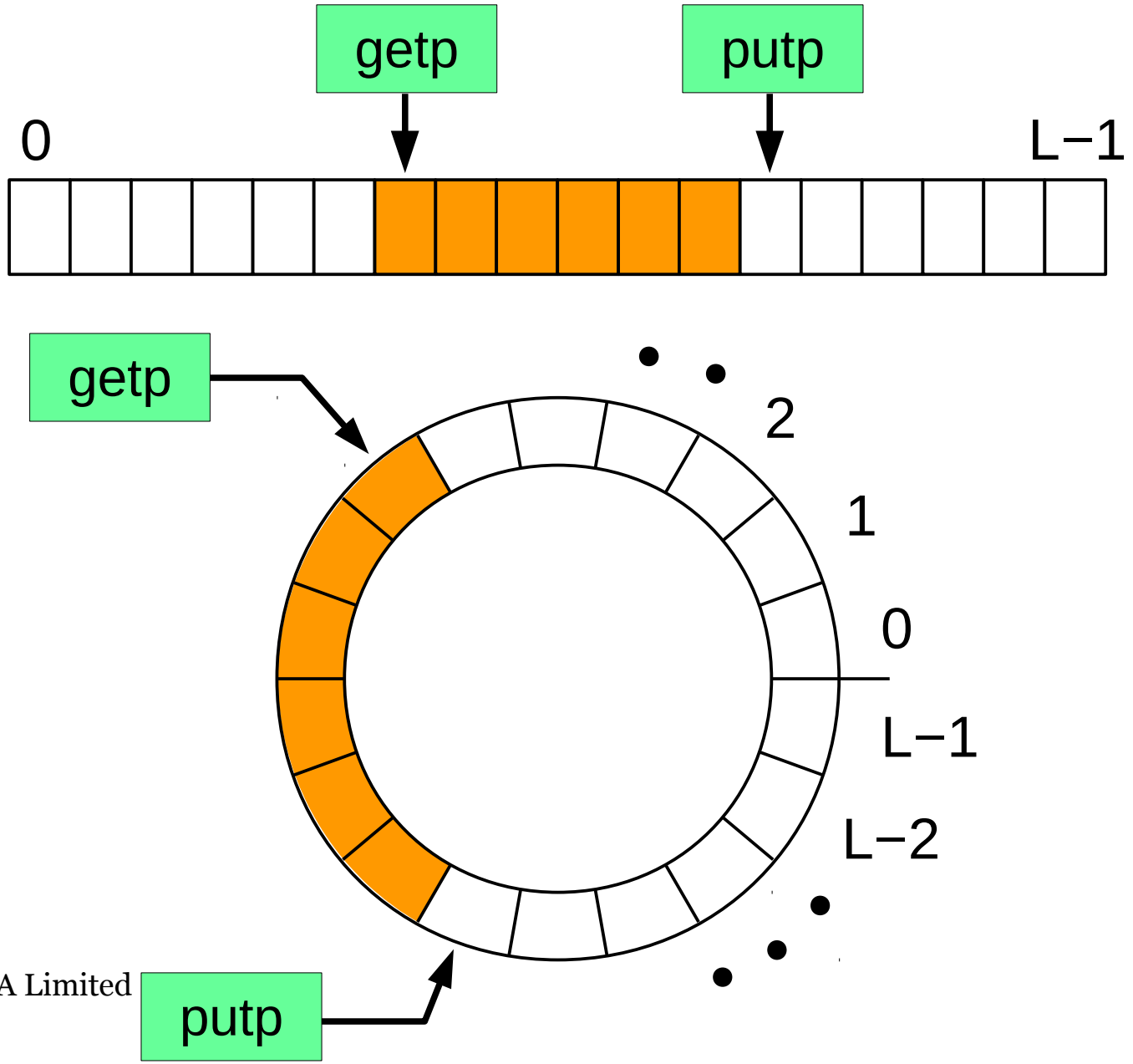
仕様

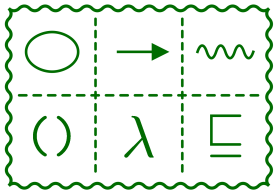



```
(define-process SPEC
  (hpar (list enq deq)
    (par '()
      (if (= NP 1)
          IN
          (xpar k (interval 0 NP) '() IN))
      (if (= NC 1)
          OUT
          (xpar k (interval 0 NC) '() OUT)))
    (QUE '())))
```

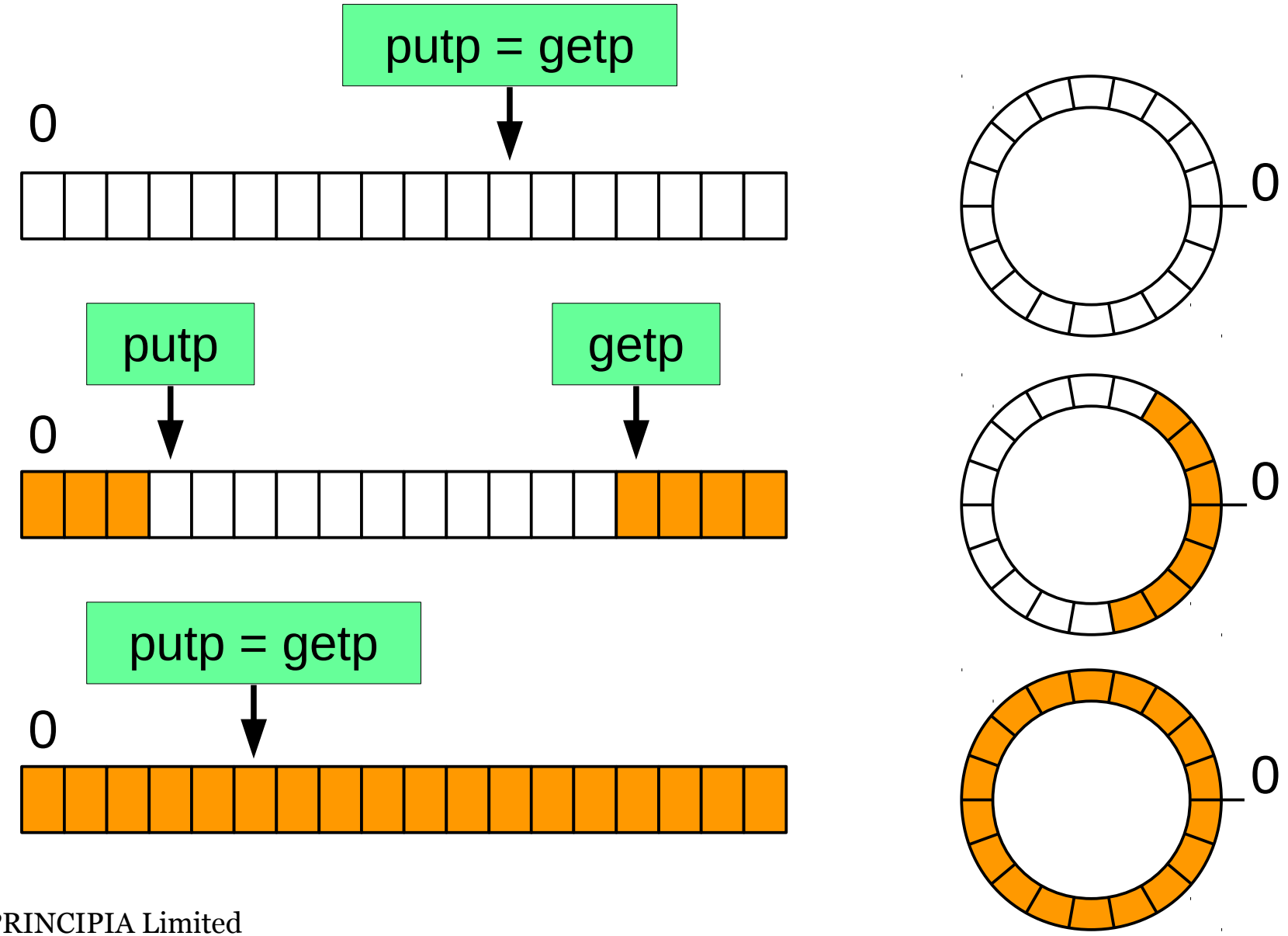


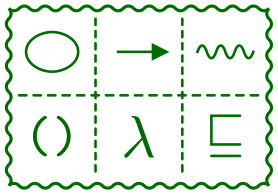
キューの実装: リングバッファ



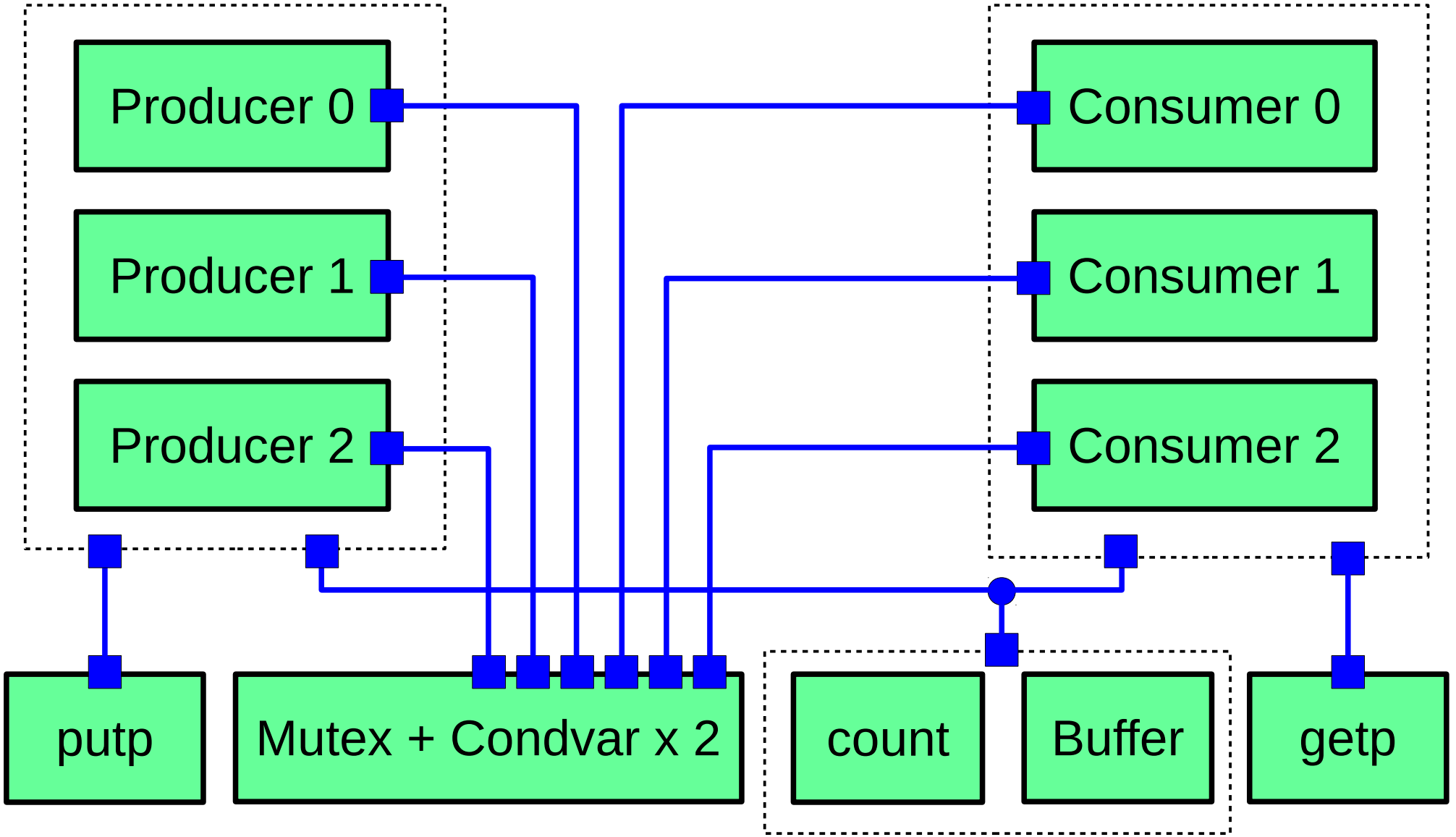


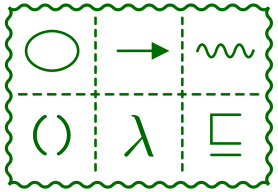
リングバッファの状態分類





生産者・消費者問題: 構造図





イベント定義: 定義域

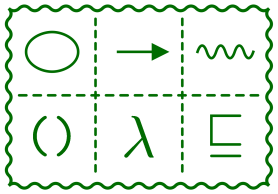
```
(define N (+ NP NC))
(define I (interval 0 N))
(define D (map list I))

(define IL (interval 0 L))
(define DL (map list IL))

(define ILx (interval 0 (+ L 1)))
(define DLx (map list ILx))

(define DLM (combinations (list IL IM)))
```

ミューテックスと条件変数の
システムコールチャネル定義域

putp, getp の変域

count の変域

共有メモリ上にあるリングバッファ用配列の
読み書きチャネルの定義域



イベント定義

Buffer

```
(define-channel buf.rd (k x) DLM)
(define-channel buf.wr (k x) DLM)
```

count

```
(define-channel count.rd (x) DLx)
(define-channel count.wr (x) DLx)
```

putp

```
(define-channel putp.rd (x) DL)
(define-channel putp.wr (x) DL)
```

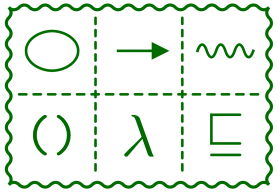
getp

```
(define-channel getp.rd (x) DL)
(define-channel getp.wr (x) DL)
```

mutex + condvar x 2

```
(define-channel ret (k) D)
(define-channel lock (k) D)
(define-event unlock)
(define-channel wait0 (k) D)
(define-channel wait1 (k) D)
(define-event signal0)
(define-event signal1)
(define-event broadcast0)
(define-event broadcast1)
```

lock, wait はシステムコール型
unlock, signal, broadcast は同期型



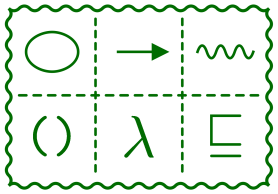
補助関数定義

```
(define (update xs k x)
  (if (= k 0)
      (cons x (cdr xs))
      (cons (car xs)
            (update (cdr xs) (- k 1) x))))
```

```
(update '(0 1 2 3) 2 'X)
=> (0 1 X 3)
```

```
(define (insert x xs)
  (if (null? xs)
      (list x)
      (if (< x (car xs))
          (cons x xs)
          (cons (car xs)
                (insert x (cdr xs))))))
```

```
(insert 5 '(0 2 4 6))
=> (0 2 4 5 6)
```



プロセス定義: 共有メモリ

```
(define-process (SV m rd wr)
  (alt
    (! rd (m) (SV m rd wr))
    (? wr (x) (SV x rd wr))))
```

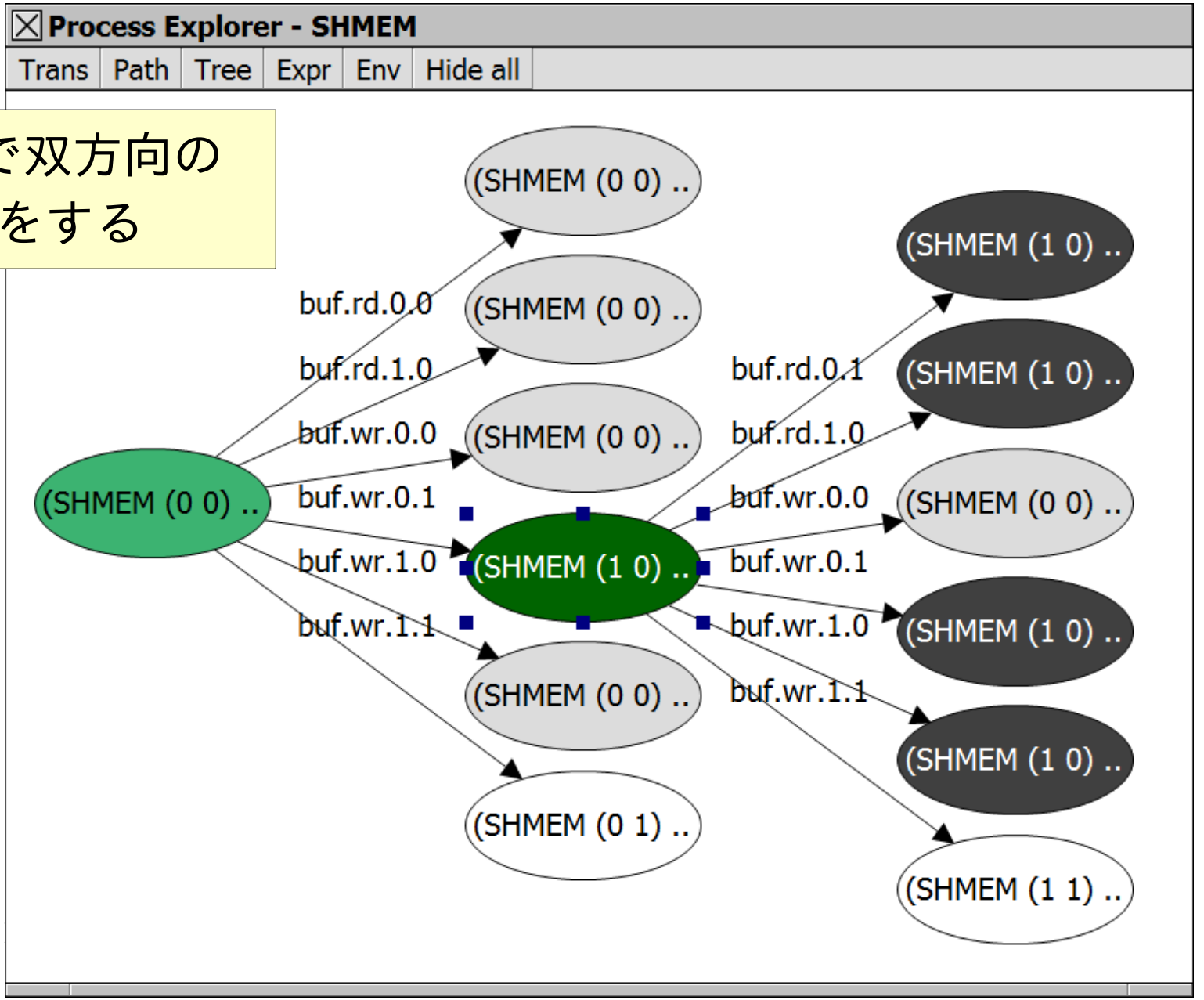
count
putp
getp

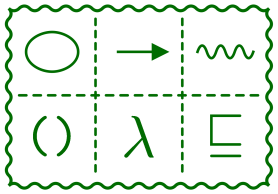
```
(define-process (SHMEM xs rd wr)
  (alt
    (? rd (k x) (equal? (list-ref xs k) x)
      (SHMEM xs rd wr))
    (? wr (k x)
      (SHMEM (update xs k x) rd wr))))
```

Buffer

配列 SHMEM の計算木

1回の通信で双方向のデータ交換をする



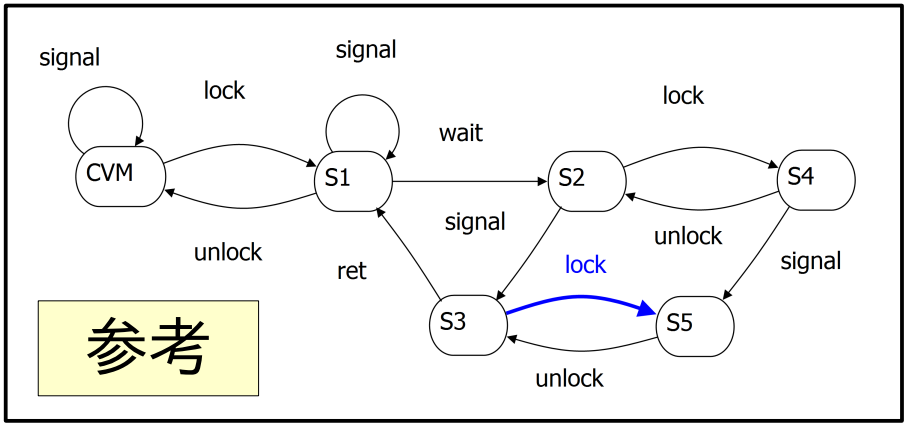


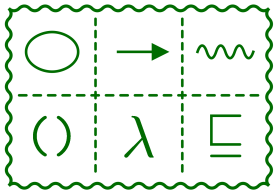
ミューテックス＋条件変数

```
(define-process (CVML m ms cs)
  (alt
    (? lock (k)
       (and (not (eq? j m))
            (not (member k ms))
            (not (member k cs)))
       (CVML m (insert k ms) cs))
    (! unlock
       (if m
           (CVML #f ms cs)
           STOP))
    (? wait (k) (eqv? k m)
       (CVML #f ms (insert m cs)))
    (! signal
       (if (null? cs)
           (CVML m ms cs)
           (xndc k cs
             (CVML m (insert k ms) (remove k cs)))))
    (! broadcast
       (CVML m (list-sort < (append cs ms)) '()))
    (if (or m (null? ms))
        STOP
        (xndc k ms
          (! ret (k) (CVML k (remove k ms) cs))))))
```

条件変数1個の場合

lock プロセス非強解
待ちプロセス非強解

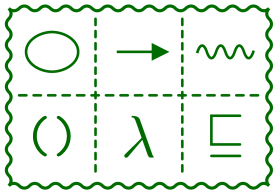




ミューテックス + 条件変数: プロセスパラメータ

```
(define-process (CVML m ms cs)
  (alt ...
```

m	ミューテックスをロックしているプロセスの ID ロックされていない場合は #f
ms	ミューテックスを待っているプロセスの ID リスト（昇順）
cs	条件変数を待っているプロセスの ID リスト（昇順）

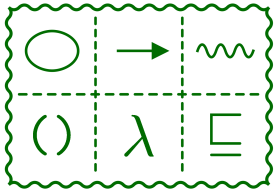


ミューテックス + 条件変数: lock

```
(define-process (CVML m ms cs)
  (alt
    (? lock (k)
       (and (not (eq? k m))
            (not (member k ms))
            (not (member k cs)))
       (CVML m (insert k ms) cs))
    ...
```

誤り検出 + モデル有限化のためのガード

ミューテックス待ちリストに加える

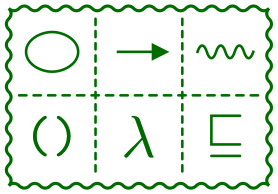


ミューテックス + 条件変数: unlock

```
(define-process (CVML m ms cs)
  (alt
    ...
    (! unlock
       (if m
           (CVML #f ms cs)
           STOP))
    ...
```

誤り検出のためのガード

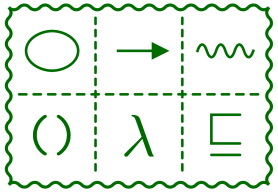
ミューテックスをアンロックする



ミューテックス + 条件変数: wait

```
(define-process (CVML m ms cs)
  (alt
    ...
    (? wait (k) (eqv? k m)
       (CVML #f ms (insert m cs)))
    ...
```

ミューテックスをアンロックし，かつ同時に（不可分に）
プロセスを条件変数の待ちリストに加える

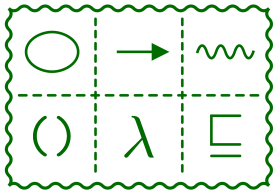


ミューテックス + 条件変数: signal

```
(define-process (CVML m ms cs)
  (alt
    ...
    (! signal
       (if (null? cs)
           (CVML m ms cs)
           (xndc k cs
             (CVML m (insert k ms) (remove k cs)))))
    ...
```

条件変数を待っているプロセスがなければ何もしない

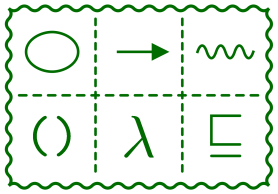
条件変数を待っているプロセスから非決定的に1つ選択し
ミューテックスの待ちリストへ移す



ミューテックス + 条件変数: broadcast

```
(define-process (CVML m ms cs)
  (alt
    ...
    (! broadcast
       (CVML m (list-sort < (append cs ms)) '()))
    ...
```

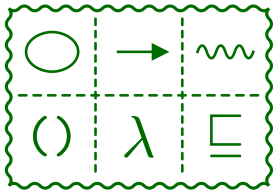
条件変数を待っているすべてのプロセスを
ミューテックスの待ちリストへ移す



ミューテックス＋条件変数: リターン（ロック獲得）

```
(define-process (CVML m ms cs)
  (alt
    ...
    (if (or m (null? ms))
        STOP
        (xndc k ms
          (! ret (k) (CVML k (remove k ms) cs))))))
```

ミューテックスが非ロック状態で，かつミューテックスを待っているプロセスがある場合は，非決定的に1つを選択しロックを与える（リターンする）



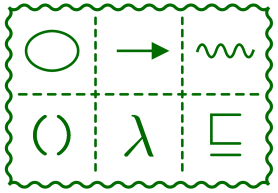
ミューテックス + 条件変数 x 2

```
(define-process CVM
  (CVML #f '() '() '()))

(define-process (CVML m ms cs0 cs1)
  (alt
    (? lock (k)
       (and (not (eq? k m))
            (not (member k ms))
            (not (member k cs0))
            (not (member k cs1)))
       (CVML m (insert k ms) cs0 cs1))
    (! unlock
       (if m
           (CVML #f ms cs0 cs1)
           STOP))
    (? wait0 (k) (eqv? k m)
       (CVML #f ms (insert m cs0) cs1))
    (? wait1 (k) (eqv? k m)
       (CVML #f ms cs0 (insert m cs1)))
    (! signal0
       (if (null? cs0)
           (CVML m ms cs0 cs1)
           (xndc k cs0
             (CVML m (insert k ms) (remove k cs0) cs1))))
    (! signal1
       (if (null? cs1)
           (CVML m ms cs0 cs1)
           (xndc k cs1
             (CVML m (insert k ms) cs0 (remove k cs1)))))
    (! broadcast0
       (CVML m (list-sort < (append cs0 ms)) '() cs1))
    (! broadcast1
       (CVML m (list-sort < (append cs1 ms)) cs0 '()))
    (if (or m (null? ms))
        STOP
        (xndc k ms
          (! ret (k) (CVML k (remove k ms) cs0 cs1))))))
```

(define-process (CVML m ms **cs0 cs1**)

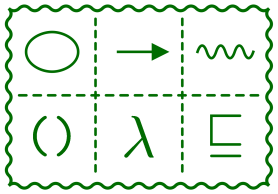
条件変数待ちリストを2つ用意

wait, signal, broadcast をそれぞれ2セット用意



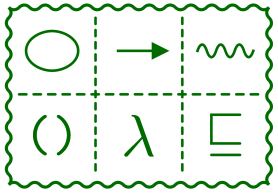
プロセス定義: 生産者

```
(define-process (P k)
  (? in (x)
     (! lock (k)
        (! ret (k) (P-LOOP k x)))))

(define-process (P-LOOP k x)
  (? count.rd (c)
     (if (= c L)
         (! wait0 (k)
            (! ret (k) (P-LOOP k x)))
         (? putp.rd (i)
            (! buf.wr (i x)
               (! putp.wr ((mod (+ i 1) L))
                  (! count.wr ((+ c 1))
                     (! signal1
                        (! unlock (P k)))))))))))
```

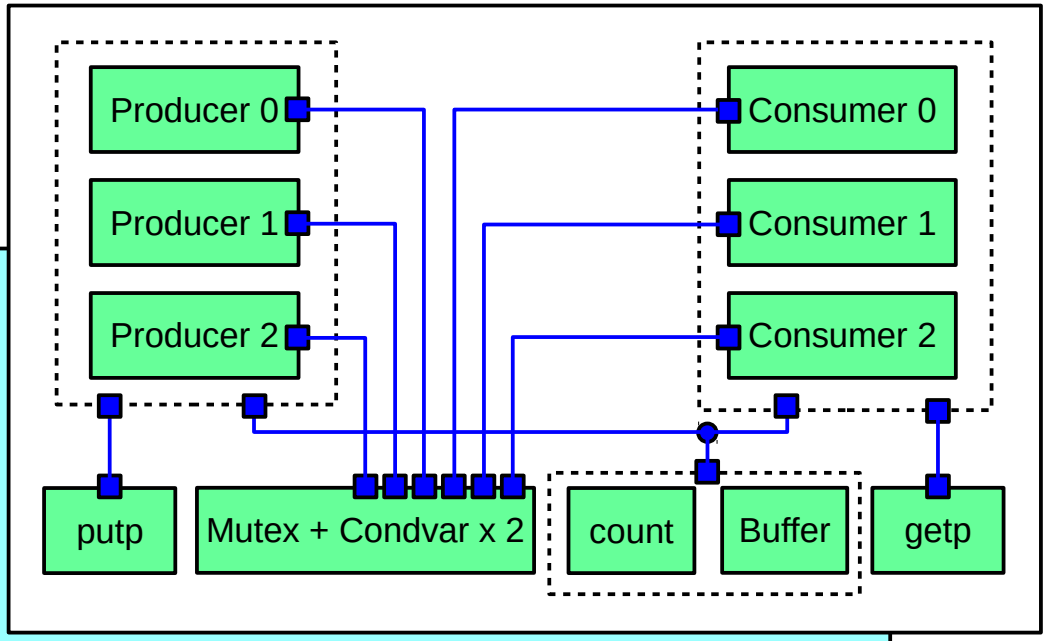


プロセス定義: 消費者

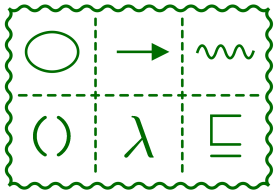
```
(define-process (Q k)
  (! lock (k)
     (! ret (k) (Q-LOOP k))))

(define-process (Q-LOOP k)
  (? count.rd (c)
     (if (= c 0)
         (! wait1 (k)
            (! ret (k) (Q-LOOP k)))
         (? getp.rd (i)
            (? buf.rd (ii x) (= i ii)
               (! getp.wr ((mod (+ i 1) L))
                  (! count.wr ((- c 1))
                     (! signal0
                        (! unlock
                           (! out (x) (Q k))))))))))
```



並行合成

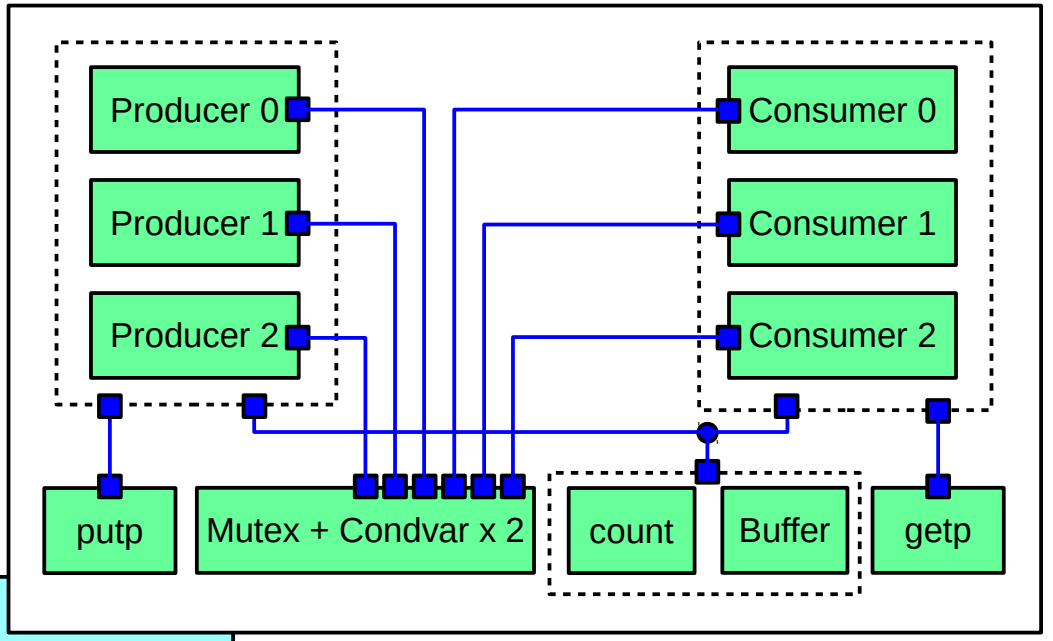



```
(define-process SYS
  (par X
    (par '()
      (if (= NP 1)
          (P 0)
          (xpar k (interval 0 NP) '() (P k)))
      (if (= NC 1)
          (Q NP)
          (xpar k (interval NP (+ NP NC)) '() (Q k))))
    (par '()
      (SV 0 putp.rd putp.wr)
      (SV 0 getp.rd getp.wr)
      (SV 0 count.rd count.wr)
      (SHMEM (map (lambda (i) 0) IL) buf.rd buf.wr)
      CVM)))
```

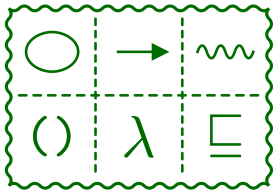


隠蔽

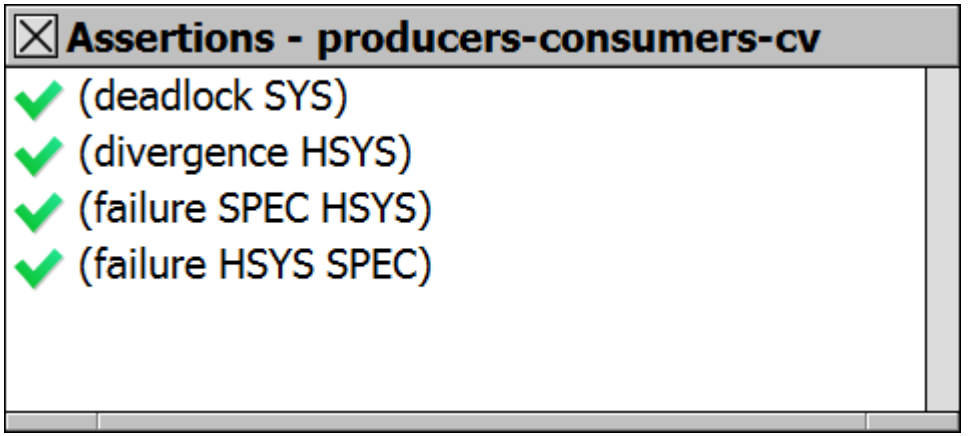
```
(define-process HSYS
  (hide X SYS))
```



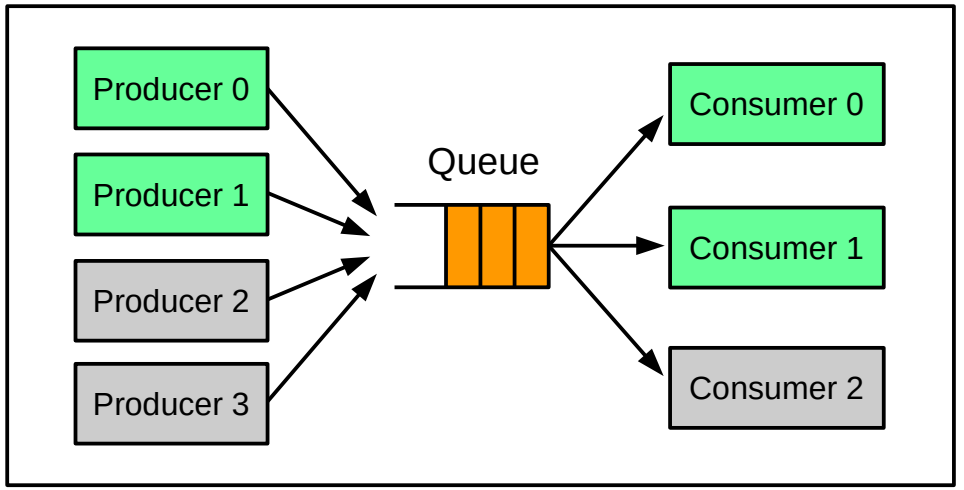

```
(define X
  (list ret lock unlock
        wait0 signal0 broadcast0
        wait1 signal1 broadcast1
        count.rd count.wr
        getp.rd getp.wr
        putp.rd putp.wr
        buf.rd buf.wr))
```

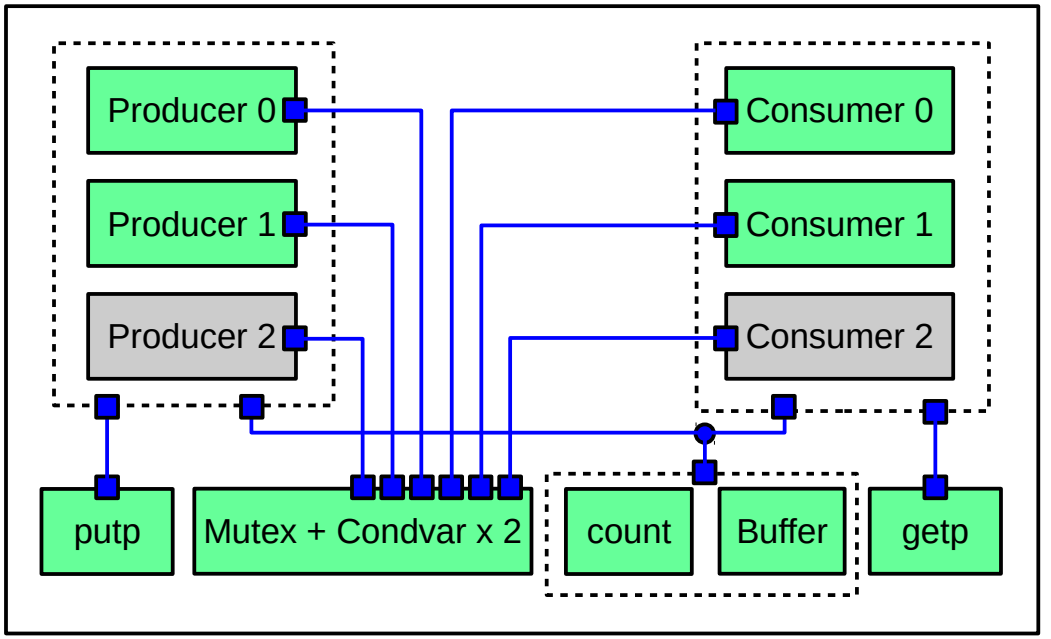


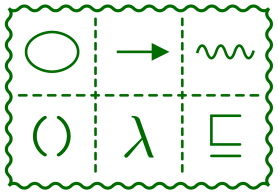
検査



$$SPEC =_F HSYS$$







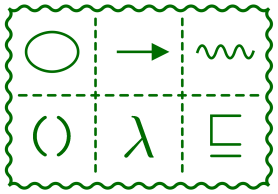
生産者・消費者問題の実装

producers-consumers-cv.c

```
#define NP 1
#define NC 6
#define L 3
```

NP:　生産者数
NC:　消費者数
L:　バッファの大きさ

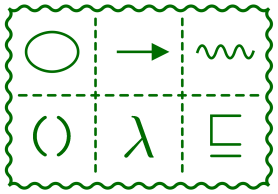
```
pthread_mutex_t mutex;
pthread_cond_t cond_empty;
pthread_cond_t cond_full;
volatile int buf[L];
volatile int count, getp, putp;
```



生産者

```
void put(int x)
{
    pthread_mutex_lock(&mutex);
    while (count == L) {
        pthread_cond_wait(&cond_full, &mutex);
    }
    buf[putp++] = x;
    if (putp == L) putp = 0;
    count++;
    pthread_cond_signal(&cond_empty);
    pthread_mutex_unlock(&mutex);
}
```

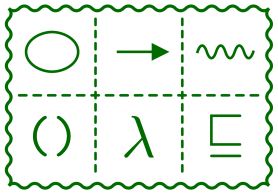
```
(define-process (P k)
  (? in (x)
     (! lock (k)
        (! ret (k)
          (P-LOOP k x)))))

(define-process (P-LOOP k x)
  (? count.rd (c)
     (if (= c L)
         (! wait0 (k)
            (! ret (k) (P-LOOP k x)))
         (? putp.rd (i)
            (! buf.wr (i x)
               (! putp.wr ((mod (+ i 1) L))
                  (! count.wr ((+ c 1))
                     (! signal1
                        (! unlock (P k)))))))))
```



消費者

```
(define-process (Q k)
  (! lock (k)
     (! ret (k)
       (Q-LOOP k))))

(define-process (Q-LOOP k)
  (? count.rd (c)
     (if (= c 0)
         (! wait1 (k)
            (! ret (k)
              (Q-LOOP k)))
         (? getp.rd (i)
            (? buf.rd (ii x) (= i ii)
               (! getp.wr ((mod (+ i 1) L))
                  (! count.wr ((- c 1))
                     (! signal0
                        (! unlock
                           (! out (x) (Q k)))))))))))
```

```
int get(void)
{
    int x;
    pthread_mutex_lock(&mutex);
    while (count == 0) {
        pthread_cond_wait(&cond_empty, &mutex);
    }
    x = buf[getp++];
    if (getp == L) getp = 0;
    count--;
    pthread_cond_signal(&cond_full);
    pthread_mutex_unlock(&mutex);
    return x;
}
```

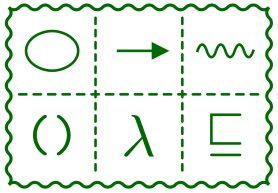


生産者ループ

```
void *producer(void *arg)
{
    int k = (intptr_t)arg;
    int i;
    for (i = 0; i < M; ++i) {
        put(k * M + i);
    }
    return NULL;
}
```

k はプロセス ID（0〜NP−1)

生産者間で重複しない
M 個の整数を生産



消費者ループ

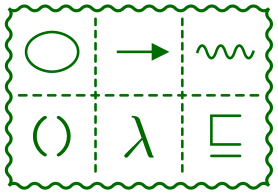
```
void *consumer(void *arg)
{
    int k = (intptr_t)arg;
    int x, n, i;
    n = NP * M / NC;
    n = (k > 0) ? n : NP * M - (NC - 1) * n;
    for (i = 0; i < n; ++i) {
        x = get();
    }
    return NULL;
}
```

k はプロセス ID（0～NC−1)

データは NP * M 個生産される

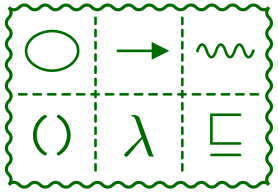
プロセス ID が 1～NC−1 である各プロセスはそのうちの 1/NC のデータを消費する（端数切捨て）

プロセス ID 0 のプロセスは残りの端数を含めて消費する



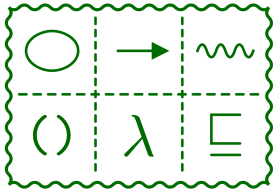
main

```
int main() {
    pthread_t thp[NP], thc[NC];
    int i;
    pthread_mutex_init(&mutex, NULL);
    pthread_cond_init(&cond_empty, NULL);
    pthread_cond_init(&cond_full, NULL);
    for (i = 0; i < NP; ++i)
        pthread_create(&thp[i], NULL, &producer, (void *)i);
    for (i = 0; i < NC; ++i)
        pthread_create(&thc[i], NULL, &consumer, (void *)i);
    ...
```



main

```
    ...
    for (i = 0; i < NP; ++i)
        pthread_join(pth[i], &retcode);
    for (i = 0; i < NC; ++i)
        pthread_join(cth[i], &retcode);
    pthread_mutex_destroy(&mutex);
    pthread_cond_destroy(&cond_empty);
    pthread_cond_destroy(&cond_full);
    return 0;
}
```



main thread による監視例

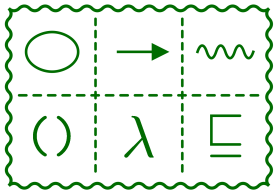
各消費数をカウント

```
volatile int cc[NC];
```

```
void *consumer(void *arg) {
    int k = (intptr_t)arg;
    ...
    for (i = 0; i < n; ++i) {
        x = get();
        cc[k]++;
    }
    return NULL;
}
```

main thread で1秒ごとに表示

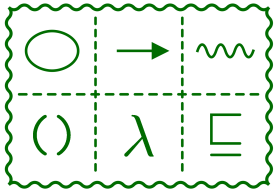
```
while (1) {
    for (i = 0; i < NC; ++i)
        printf("%8u ", cc[i]);
    printf("\n");
    sleep(1);
}
```



実行例

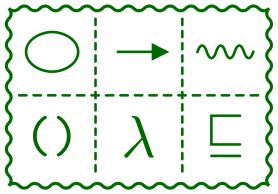
~/projects/topse/trunk/vic/lectures/programs

```
$ ./producers-consumers-cv
        13          6          5          3
     45571      45559      45559      45557
     91370      91358      91358      91356
    137522     137510     137509     137508
    183330     183319     183318     183317
    229964     229953     229952     229951
    275534     275523     275523     275521
    323110     323099     323099     323097
    370815     370804     370803     370802
    417088     417077     417077     417075
    464495     464485     464484     464482
    512079     512069     512068     512066
    559562     559552     559551     559550
    607271     607261     607260     607259
    654033     654023     654022     654021
    700902     700892     700890     700889
```



問題

- 問題
  - バッファの更新・読出しをロックの外で行ったらどうなるか
  - 条件変数の待ち状態から抜けたあとはなぜループするのか
  - 条件変数を1つにしたらどうなるか
- モデルで分析を行い，不具合がある場合はその理由あるいは発生シーケンスを説明し，実装で不具合を再現させてください

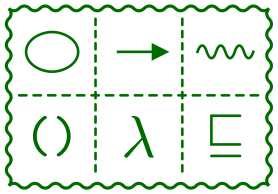


バッファ更新をロックの外へ出す

```
void put(int x)
{
    pthread_mutex_lock(&mutex);
    while (count == L) {
        pthread_cond_wait(&cond_full, &mutex);
    }
    buf[putp++] = x;
    if (putp == L) putp = 0;
    count++;
    pthread_cond_signal(&cond_empty);
    pthread_mutex_unlock(&mutex);
}
```

生産者の場合

```
void put(int x)
{
    int i;
    pthread_mutex_lock(&mutex);
    while (count == L) {
        pthread_cond_wait(&cond_full, &mutex);
    }
    i = putp++;
    if (putp == L) putp = 0;
    count++;
    pthread_cond_signal(&cond_empty);
    pthread_mutex_unlock(&mutex);
    buf[i] = x;
}
```

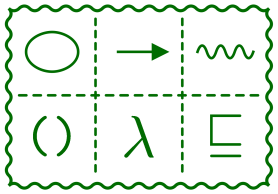


条件変数待ちでループしない

```
void put(int x)
{
    pthread_mutex_lock(&mutex);
    while (count == L) {
        pthread_cond_wait(&cond_full, &mutex);
    }
    buf[putp++] = x;
    if (putp == L) putp = 0;
    count++;
    pthread_cond_signal(&cond_empty);
    pthread_mutex_unlock(&mutex);
}
```

生産者の場合

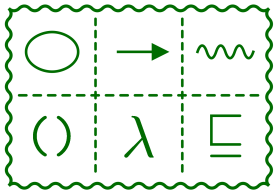
```
void put(int x)
{
    pthread_mutex_lock(&mutex);
    if (count == L) {
        pthread_cond_wait(&cond_full, &mutex);
    }
    buf[putp++] = x;
    if (putp == L) putp = 0;
    count++;
    pthread_cond_signal(&cond_empty);
    pthread_mutex_unlock(&mutex);
}
```



条件変数待ちでループしない

```
void put(int x)
{
    pthread_mutex_lock(&mutex);
    while (count == L) {
        pthread_cond_wait(&cond_full, &mutex);
    }
    buf[putp++] = x;
    if (putp == L) putp = 0;
    count++;
    pthread_cond_signal(&cond_empty);
    pthread_mutex_unlock(&mutex);
}
```

```
void put(int x)
{
    pthread_mutex_lock(&mutex);
    while (count == L) {
        pthread_cond_wait(&cond_full, &mutex);
        assert(count < L);
    }
    buf[putp++] = x;
    if (putp == L) putp = 0;
    count++;
    pthread_cond_signal(&cond_empty);
    pthread_mutex_unlock(&mutex);
}
```



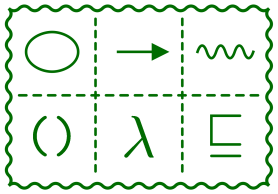
条件変数を1つにする

生産者

```
void put(int x)
{
    pthread_mutex_lock(&mutex);
    while (count == L) {
        pthread_cond_wait(&cond, &mutex);
    }
    buf[putp++] = x;
    if (putp == L) putp = 0;
    count++;
    pthread_cond_signal(&cond);
    pthread_mutex_unlock(&mutex);
}
```

消費者

```
int get(void)
{
    int x;
    pthread_mutex_lock(&mutex);
    while (count == 0) {
        pthread_cond_wait(&cond, &mutex);
    }
    x = buf[getp++];
    if (getp == L) getp = 0;
    count--;
    pthread_cond_signal(&cond);
    pthread_mutex_unlock(&mutex);
    return x;
}
```

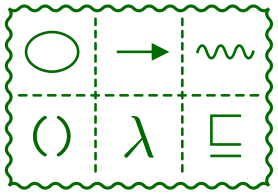


ロック外でのバッファ更新

```
(define-process (P-LOOP k x)
  (? count.rd (c)
     (if (= c L)
         (! wait0 (k)
            (! ret (k)
               (P-LOOP k x)))
         (? putp.rd (i)
            (! putp.wr ((mod (+ i 1) L))
               (! count.wr ((+ c 1))
                  (! signal1
                     (! unlock
                        (! buf.wr (i x) (P k)))))))))))
```

生産者

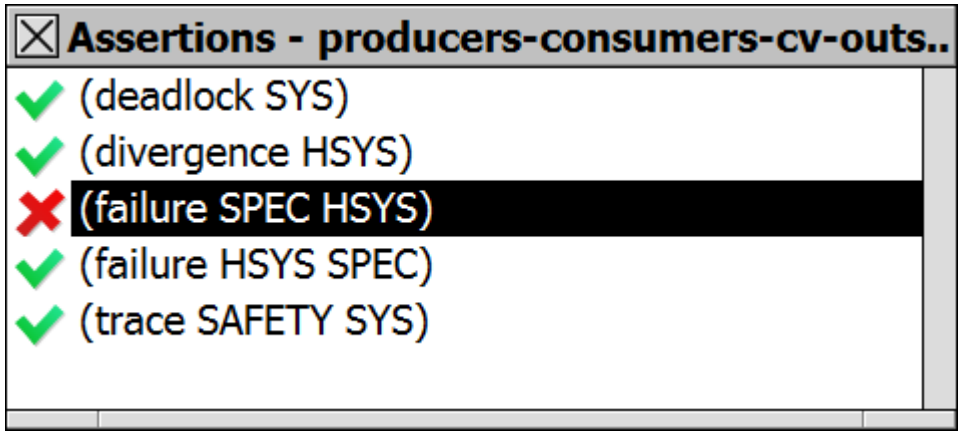
バッファ更新を unlock の後で行うように修正

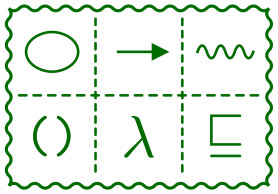


検査

```
(define NP 1)
(define NC 1)
(define L 1)
(define M 2)
```

NP:　生産者数
NC:　消費者数
L:　バッファの大きさ
M:　データの種類





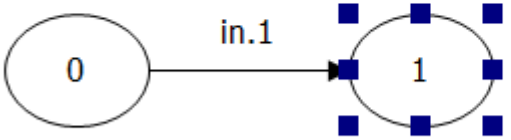
分析

Transitions

event	target
in.0	10
in.1	13
out.1	3

Process Explorer - SPEC

Trans | Path | Accept | Compo | Tree | Expr | Env | Hide



Path

	event	source
0	in.1	0

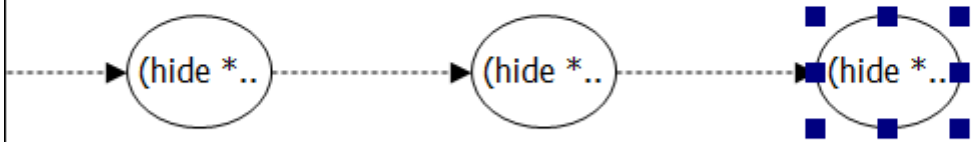
トレース違反

Transitions

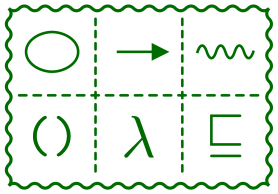
event	target
tau	(hide * (par * (par
out.0	(hide * (par * (par

Process Explorer - HSYS

Trans | Path | Tree | Expr | Env | Hide all



隠蔽をはがす



分析

Path

	event	s
0	in.1	
1	lock.0	
2	ret.0	
3	count.rd.0	
4	putp.rd.0	
5	putp.wr.0	
6	count.wr.1	
7	signal1	
8	lock.1	
9	unlock	
10	ret.1	
11	count.rd.1	
12	getp.rd.0	
13	buf.rd.0.0	
14	getp.wr.0	
15	count.wr.0	
16	signal0	
17	unlock	

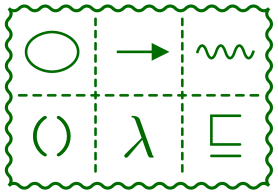
生産者が count と putp 更新
バッファへの書き込みはまだ行っていない

消費者がロックを要求

生産者がアンロック

消費者がバッファからデータを引き取る

消費者はバッファの初期値を読んでいる

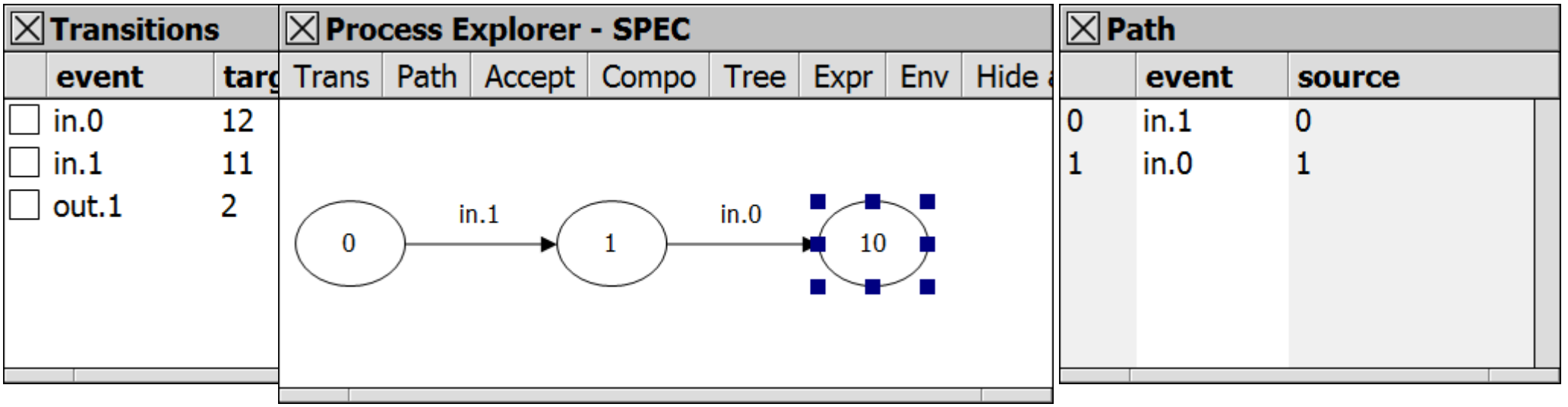


ロック外でのバッファ参照

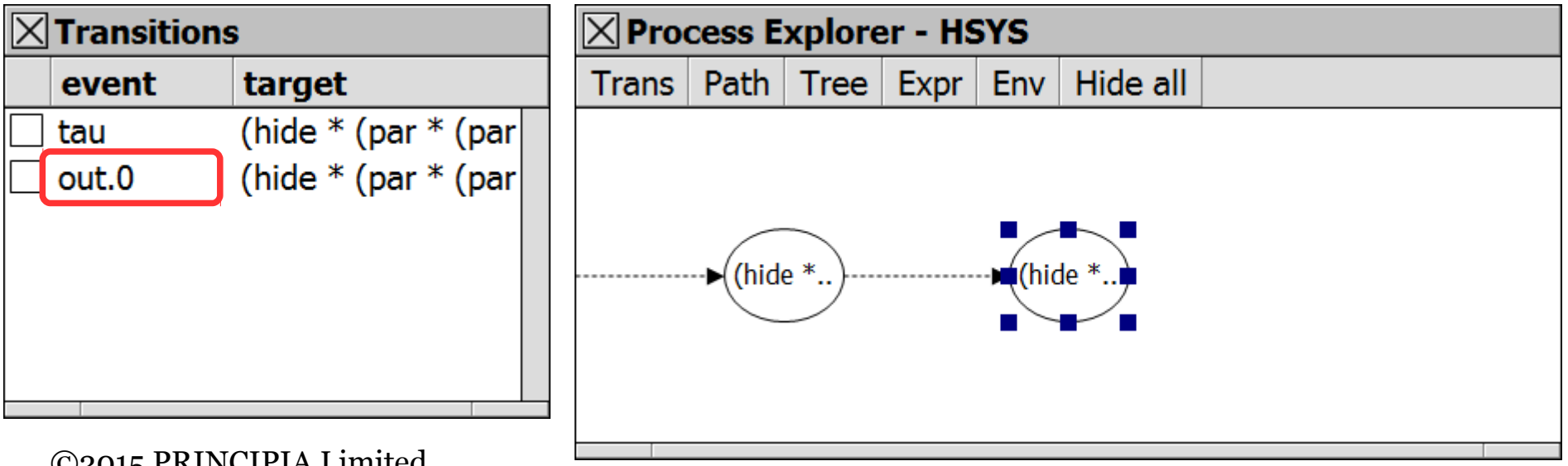
```
(define-process (Q-LOOP k)
  (? count.rd (c)
     (if (= c 0)
         (! wait1 (k)
            (! ret (k) (Q-LOOP k)))
         (? getp.rd (i)
            (! getp.wr ((mod (+ i 1) L))
               (! count.wr ((- c 1))
                  (! signal0
                     (! unlock
                        (? buf.rd (ii x) (= i ii)
                           (! out (x) (Q k)))))))))))
```

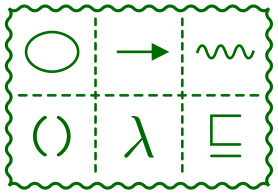
消費者

分析



トレース違反





Path

	event	s
0	in.1	
1	lock.0	
2	ret.0	
3	count.rd.0	
4	putp.rd.0	
5	buf.wr.0.1	
6	putp.wr.0	
7	count.wr.1	
8	signal1	
9	lock.1	
10	unlock	
11	ret.1	
12	count.rd.1	
13	getp.rd.0	
14	getp.wr.0	
15	count.wr.0	
16	signal0	
17	unlock	
18	in.0	
19	lock.0	
20	ret.0	
21	count.rd.0	
22	putp.rd.0	
23	buf.wr.0.0	
24	buf.rd.0.0	

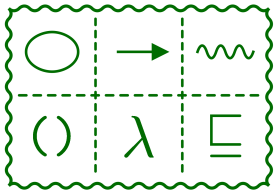
分析

生産者がデータ 1 でバッファ更新

消費者が count, getp 更新
バッファからの読出しは
まだ行っていない

生産者がデータ 0 でバッファ更新

引き取る前に上書きされている



実装での再現

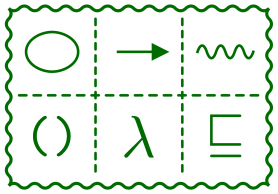
データの総和を求める

```
volatile long cc[NC];

void *consumer(void *arg) {
    int k = (intptr_t)arg;
    int n, i;
    long s = 0;
    n = NP * M / NC;
    n = (k > 0) ? n : NP * M - (NC -
    for (i = 0; i < n; ++i) {
        s += get();
    }
    cc[k] = s;
    return NULL;
}
```

main （スレッド終了後）

```
s = 0;
for (i = 0; i < NC; ++i)
  s += cc[i];
printf("%ld\n", s);
```

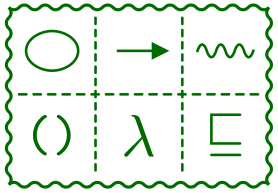
プログラムが正しければ
0 ～ NP * M–1 の総和
NP * M * (NP * M – 1) / 2
に一致する



再現の頻度を高めるポイント

```
void put(int x) {
    int i;
     pthread_mutex_lock(&mutex);
     while (count == L) {
         pthread_cond_wait(&cond_full, &mutex);
     }
     i = putp++;
     if (putp == L) putp = 0;
     count++;
     pthread_cond_signal(&cond_empty);
     pthread_mutex_unlock(&mutex);
     printf("\n");
     buf[i] = x;
}
```

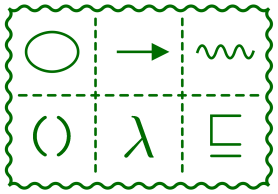
unlock とバッファ更新の間に
時間のかかる処理を入れると発生する



条件変数待ちループなし

```
(define-process (P-LOOP k x)
  (? count.rd (c)
     (if (= c L)
         (! wait0 (k)
            (! ret (k)
               (? count.rd (c)
                  (P-WRITE k x c))))
         (P-WRITE k x c))))

(define-process (P-WRITE k x c)
  (? putp.rd (i)
     (! buf.wr (i x)
        (! putp.wr ((mod (+ i 1) L))
           (! count.wr ((+ c 1))
              (! signal1
                 (! unlock (P k))))))))
```

起こされたら count を再読み込みして
そのまま書き込みへ行く



検査

```
(define NP 1)
(define NC 1)
(define L 1)
(define M 2)
```

Assertions - producers-cons..
- (divergence HSYS)
- (failure SPEC HSYS)
- (failure HSYS SPEC)
- (deadlock SYS)
- (trace SAFETY SYS)

NP: 生産者数
NC: 消費者数
L: バッファの大きさ
M: データの種類

```
(define NP 1)
(define NC 2)
(define L 1)
(define M 2)
```

Assertions - producers-cons..
- (divergence HSYS)
- (failure SPEC HSYS)
- (failure HSYS SPEC)
- (deadlock SYS)
- (trace SAFETY SYS)

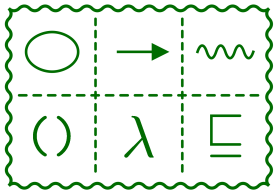
バッファサイズ L = 1 で count に 2 を代入しようとしている

```
(define NP 2)
(define NC 1)
(define L 1)
(define M 2)
```

Backtrace

procedure	variable	value
0. error	who	count.wr
1. (lambda (arg-ids1	msg	"channel argument list is out of t
2. (lambda i.args (m	irritants	(#<channel:count.wr:27> (2))
3. (lambda i.args (m	[0]	#<record:&error>
4. %rmap/1	[1]	#<record:&who>
5. map	[2]	#<record:&message>

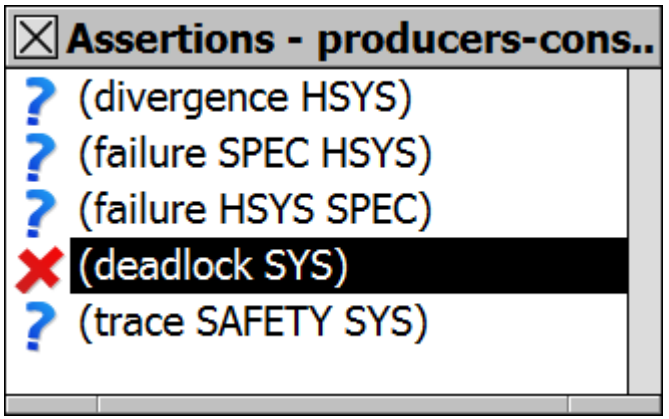
"channel argument list is out of the domain."



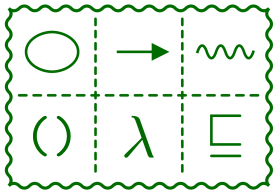
条件変数待ちループなし

```
(define-process (P-LOOP k x)
  (? count.rd (c)
     (if (= c L)
         (! wait0 (k)
            (! ret (k)
               (? count.rd (c)
                  (if (= c L)
                     STOP
                     (P-WRITE k x c)))))
         (P-WRITE k x c))))
```

起こされた直後に count = L ならばデッドロックさせる



隠蔽していないシステムプロセス SYS に対して検査を行うと分析が容易になる



分析

Path

	event
0	in.1
1	lock.0
2	ret.0
3	count.rd.0
4	putp.rd.0
5	buf.wr.0.1
6	putp.wr.0
7	count.wr.1
8	signal1
9	in.0
10	unlock
11	lock.1
12	ret.1
13	count.rd.1
14	wait0.1
15	lock.2
16	ret.2
17	count.rd.1
18	getp.rd.0
19	buf.rd.0.1
20	getp.wr.0
21	count.wr.0
22	signal0

Path

	event
23	in.1
24	unlock
25	lock.0
26	tau
27	ret.0
28	count.rd.0
29	putp.rd.0
30	buf.wr.0.1
31	putp.wr.0
32	count.wr.1
33	signal1
34	unlock
35	ret.1
36	out.1
37	lock.2
38	in.0
39	lock.0
40	count.rd.1

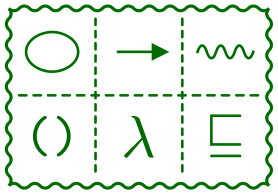
生産者 0 がデータ格納

生産者 1 はバッファフルで wait

消費者がデータを引き取り生産者 1 を起こす

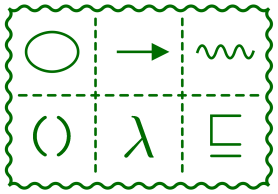
生産者 0 がデータ格納 count = 1（フル）

生産者 1 が起きて count = 1 を読む



実装での再現

```
void put(int x)
{
    pthread_mutex_lock(&mutex);
    while (count == L) {
        pthread_cond_wait(&cond_full, &mutex);
        assert(count < L);
    }
    buf[putp++] = x;
    if (putp == L) putp = 0;
    count++;
    pthread_cond_signal(&cond_empty);
    pthread_mutex_unlock(&mutex);
}
```



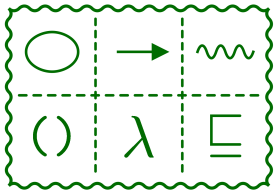
再現の頻度を高めるポイント

条件変数を待っているプロセスの中からどれを起こすかというスケジューリングの問題なので，外部から制御はできない

バッファサイズを小さくし，プロセスを増やすことで待ち状態を起きやすくすると発生しやすい

```
(define NP 10)
(define NC 10)
(define L 1)
(define M 2)
```

NP:	生産者数
NC:	消費者数
L:	バッファの大きさ
M:	データの種類



条件変数を1つにする

条件変数 0 のみ使用

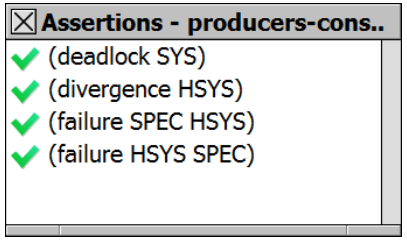
```
(define-process (P-LOOP k x)
  (alt
    (! full
       (! wait0 (k)
          (! ret (k) (P-LOOP k x))))
    (! inc
       (? putp.rd (i)
          (! buf.wr (i x)
             (! signal0
                (! unlock (P k)))))))))
```

```
(define-process (Q-LOOP k)
  (alt
    (! empty
       (! wait0 (k)
          (! ret (k) (Q-LOOP k))))
    (! dec
       (? getp.rd (i)
          (? buf.rd (ii x) (= i ii)
             (! signal0
                (! unlock
                   (! out (x) (Q k)))))))))
```

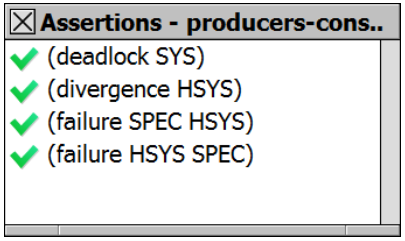
検査

NP	生産者数
NC	消費者数
L	バッファの大きさ
M = 2	データの種類

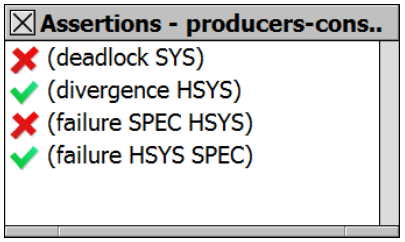
```
(define NP 1)
(define NC 1)
(define L 1)
```



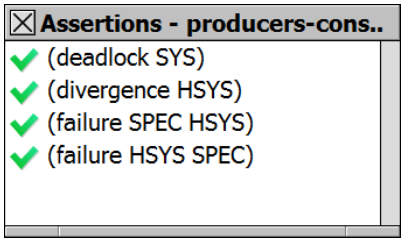

```
(define NP 1)
(define NC 1)
(define L 2)
```



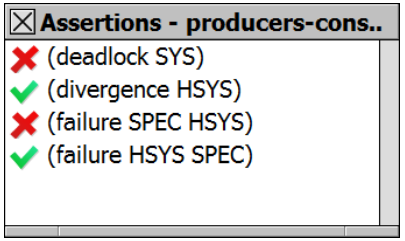

```
(define NP 2)
(define NC 1)
(define L 1)
```



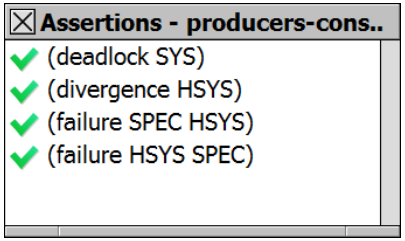

```
(define NP 2)
(define NC 1)
(define L 2)
```



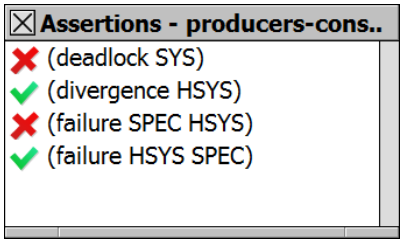

```
(define NP 1)
(define NC 2)
(define L 1)
```



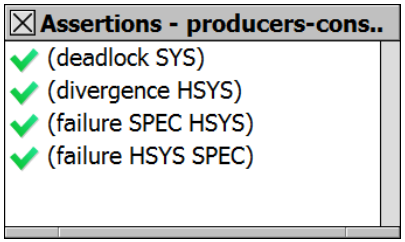

```
(define NP 1)
(define NC 2)
(define L 2)
```

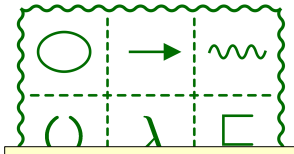



```
(define NP 2)
(define NC 2)
(define L 1)
```




```
(define NP 2)
(define NC 2)
(define L 2)
```





消費者 1 が wait

消費者 0 が wait

生産者がデータを格納し消費者 0 を起こす

生産者が wait

Path

	event
0	lock.2
1	ret.2
2	count.rd.0
3	wait0.2
4	lock.1
5	ret.1
6	in.1
7	lock.0
8	count.rd.0
9	wait0.1
10	ret.0
11	count.rd.0
12	putp.rd.0
13	buf.wr.0.1
14	putp.wr.0
15	count.wr.1
16	signal0
17	tau
18	unlock
19	in.0
20	lock.0
21	tau
22	ret.0
23	count.rd.1
24	wait0.0

Environment

variable	value
cs0	(1 2)
cs1	()
m	0
ms	()

Environment

variable	value
cs0	(2)
cs1	()
m	0
ms	(1)

分析

```
(define NP 1)
(define NC 2)
(define L 1)
```

Path

	event
25	ret.1
26	count.rd.1
27	getp.rd.0
28	buf.rd.0.1
29	getp.wr.0
30	count.wr.0
31	signal0
32	tau
33	unlock
34	ret.2
35	count.rd.0
36	out.1
37	wait0.2
38	lock.1
39	ret.1
40	count.rd.0
41	wait0.1

消費者 0 がデータを引き取り消費者 1 を起こす

Environment

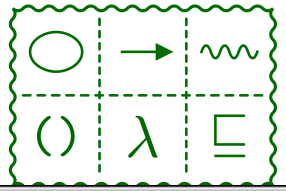
variable	value
cs0	(0 2)
cs1	()
m	1
ms	()

Environment

variable	value
cs0	(0)
cs1	()
m	1
ms	(2)

消費者 1 が wait

消費者 0 が wait



分析

```
(define NP 2)
(define NC 1)
(define L 1)
```

Path

	event	s
0	in.0	
1	lock.1	
2	ret.1	
3	count.rd.0	
4	putp.rd.0	
5	buf.wr.0.0	
6	putp.wr.0	
7	count.wr.1	
8	in.0	
9	signal0	
10	unlock	
11	lock.0	
12	ret.0	
13	count.rd.1	
14	in.0	
15	wait0.0	
16	lock.1	
17	ret.1	
18	lock.2	
19	count.rd.1	
20	wait0.1	

生産者 1
格納

生産者 0
wait

生産者 1
wait

Path

	event	s
21	ret.2	
22	count.rd.1	
23	getp.rd.0	
24	buf.rd.0.0	
25	getp.wr.0	
26	count.wr.0	
27	signal0	
28	tau	
29	unlock	
30	out.0	
31	lock.2	
32	tau	
33	ret.2	
34	count.rd.0	
35	wait0.2	

消費者
引き取り

生産者 0
wakeup

消費者
wait

Path

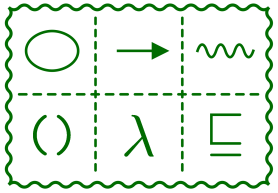
	event	s
36	ret.0	
37	count.rd.0	
38	putp.rd.0	
39	buf.wr.0.0	
40	putp.wr.0	
41	count.wr.1	
42	signal0	
43	tau	
44	unlock	
45	ret.1	
46	count.rd.1	
47	in.0	
48	wait0.1	
49	lock.0	
50	ret.0	
51	count.rd.1	
52	wait0.0	

生産者 0
格納

生産者 1
wakeup

生産者 1
wait

生産者 0
wait



デッドロックが起こる条件

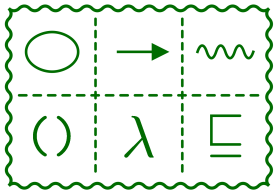
2L ≤ NP または 2L ≤ NC の場合はデッドロックがある

2L ≤ NP

1.生産者がフルになるまでデータを格納し，全員 wait
2.消費者がデータをすべて引き取り wait，その間に L 個の生産者を起こす
3.L 個の生産者がフルになるまでデータを格納し wait，その間に寝ている**生産者**を L 個起こす
4.バッファはフルなので起こされた生産者はすべて wait

2L ≤ NC

1.空なのですべての消費者が wait
2.生産者がフルになるまで格納し wait，その間に L 個の消費者を起こす
3.L 個の消費者がデータをすべて引き取り wait，その間に寝ている**消費者**を L 個起こす
4.バッファは空なので起こされた消費者はすべて wait



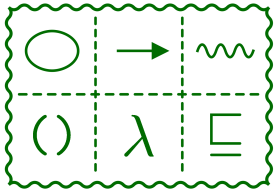
実装での再現

消費者がすべて wait するのを待つ

```
void *producer(void *arg) {
    int k = (intptr_t)arg;
    int i;
    sleep(1);
    for (i = 0; i < M; ++i) {
        put(k, k * M + i);
    }
    return NULL;
}
```

生産者を先に wait させる

```
int get(int k) {
    int x;
    pthread_mutex_lock(&mutex);
    while (count == 0) {
        pthread_cond_wait(&cond, &mutex);
        sleep(1);
    }
    x = buf[getp++];
    if (getp == L) getp = 0;
    count--;
    pthread_cond_signal(&cond);
    pthread_mutex_unlock(&mutex);
    return x;
}
```



まとめ

- 同期機構を使った並行システムの設計は難しい
  - コード上の小さな誤りが仕様違反やデッドロックなどの致命的な不具合を生み出す
  - 非決定性のためにテストでは発見しにくく，発見しても原因を調べることは難しい
- 同期機構のモデルを使った検査は有効である
  - 問題を確実に発見し，原因を分析できる