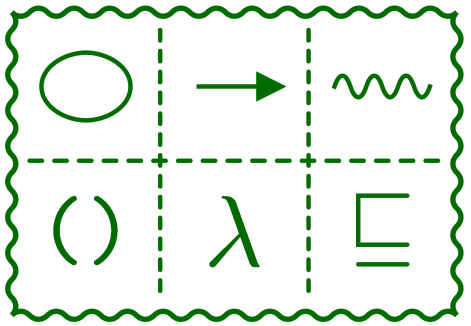
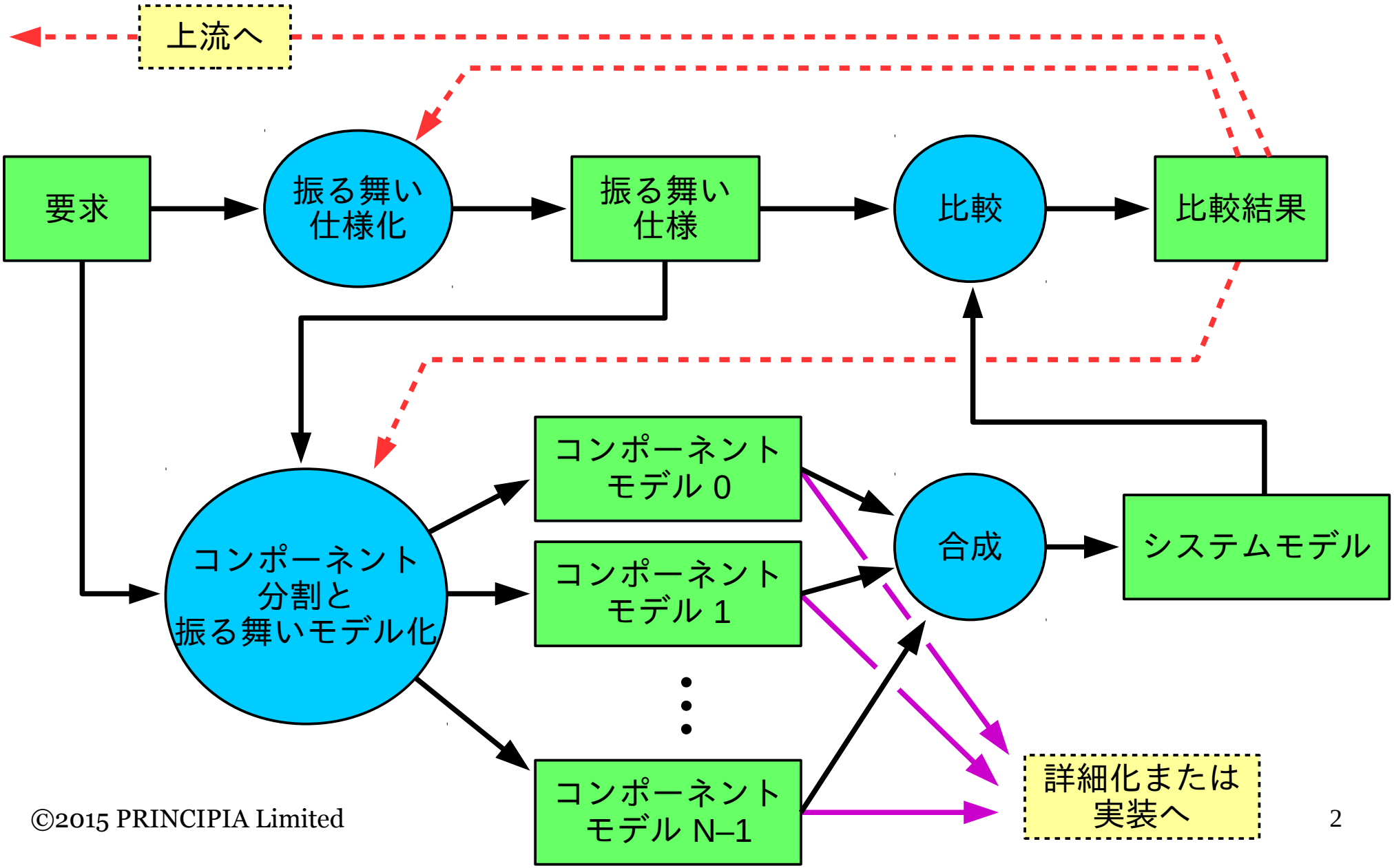


並行システムの検証と実装

第13章 並行システムの実装 2
不可分操作によるロックフリーアルゴリズムの実装

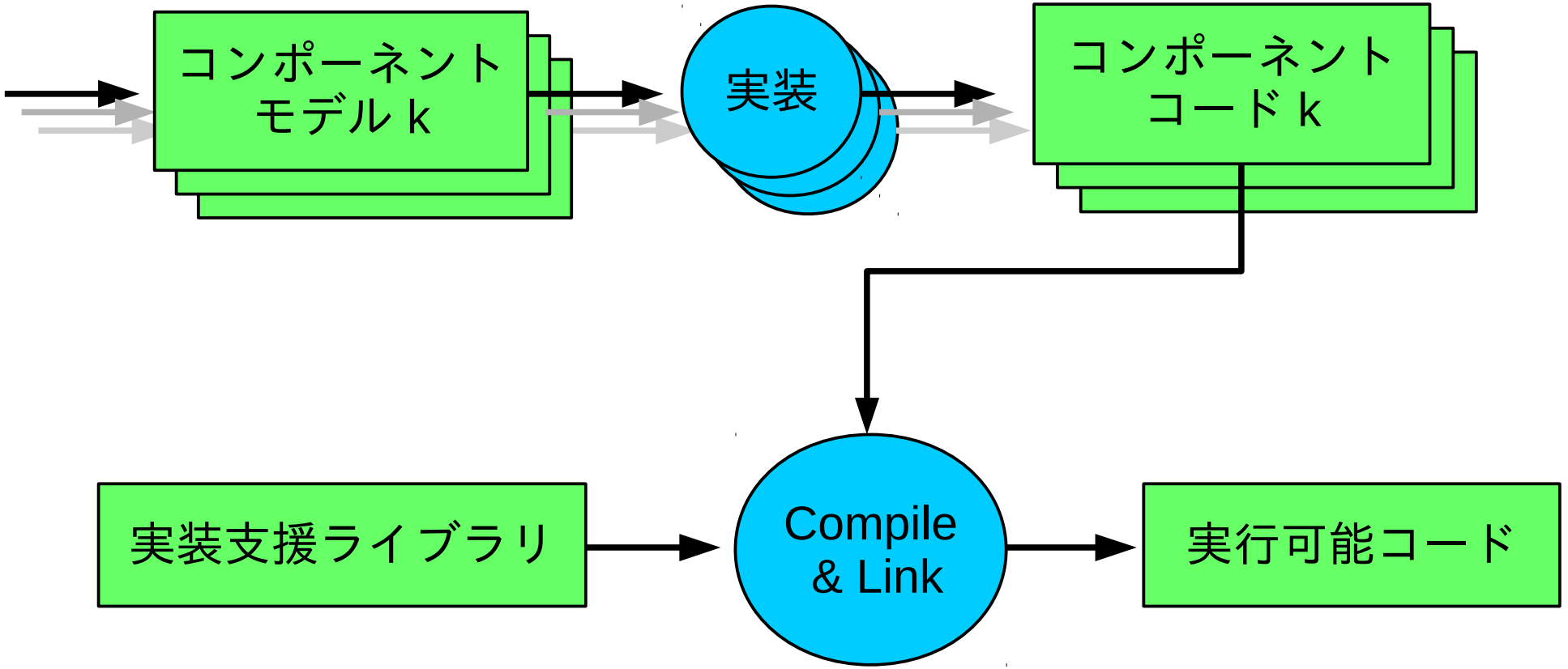
PRINCIPIA Limited
初谷 久史

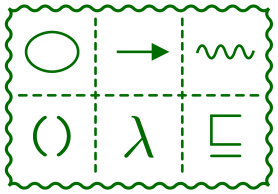
システムの設計（振る舞い側面）



モデルから実装へ

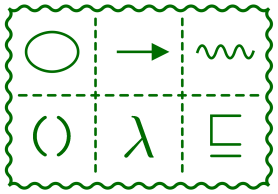
上流から





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



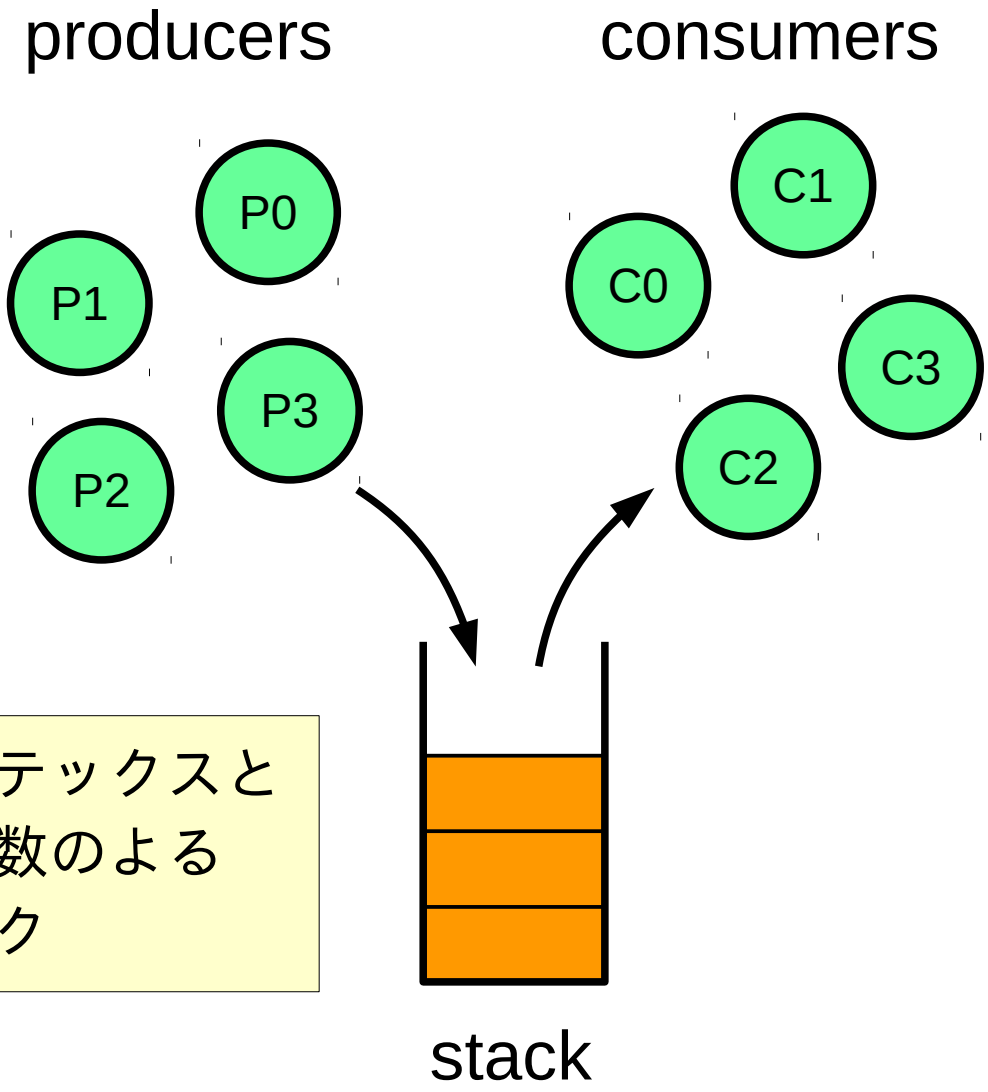
同期機構のオーバーヘッド

	sec
全実行時間	2.618
同期機構呼び出し	2.471
クリティカルセクション	0.018
その他	0.129

データ転送数 100,000
同期機構のオーバーヘッド
49.42μs/転送

同期機構:
pthread_mutex_lock
pthread_mutex_unlock
pthread_cond_wait
pthread_cond_signal

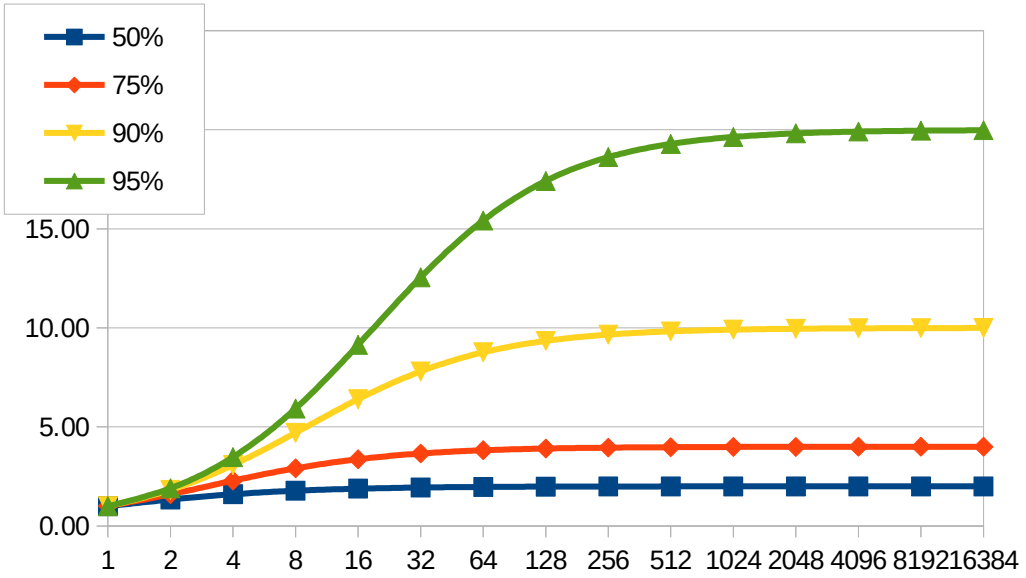
ミューテックスと
条件変数のよる
スタック

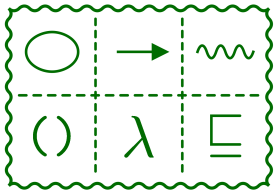


Amdahl の法則

$$\text{Speedup} = \frac{1}{\frac{p}{N} + (1 - p)}$$

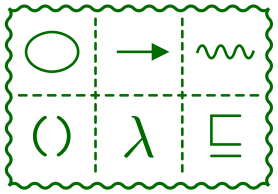
p : プログラム全体の中で並列化が可能な部分の比率



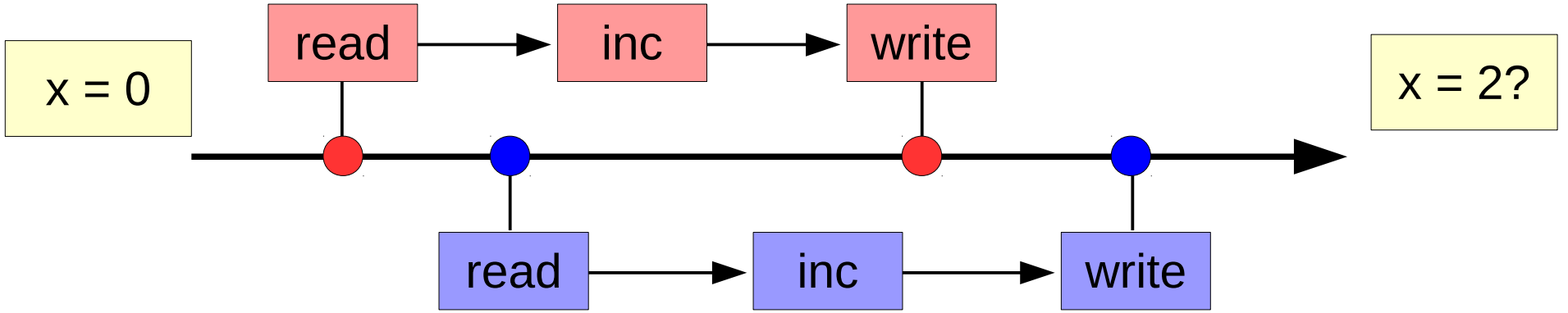


不可分（アトミック）操作

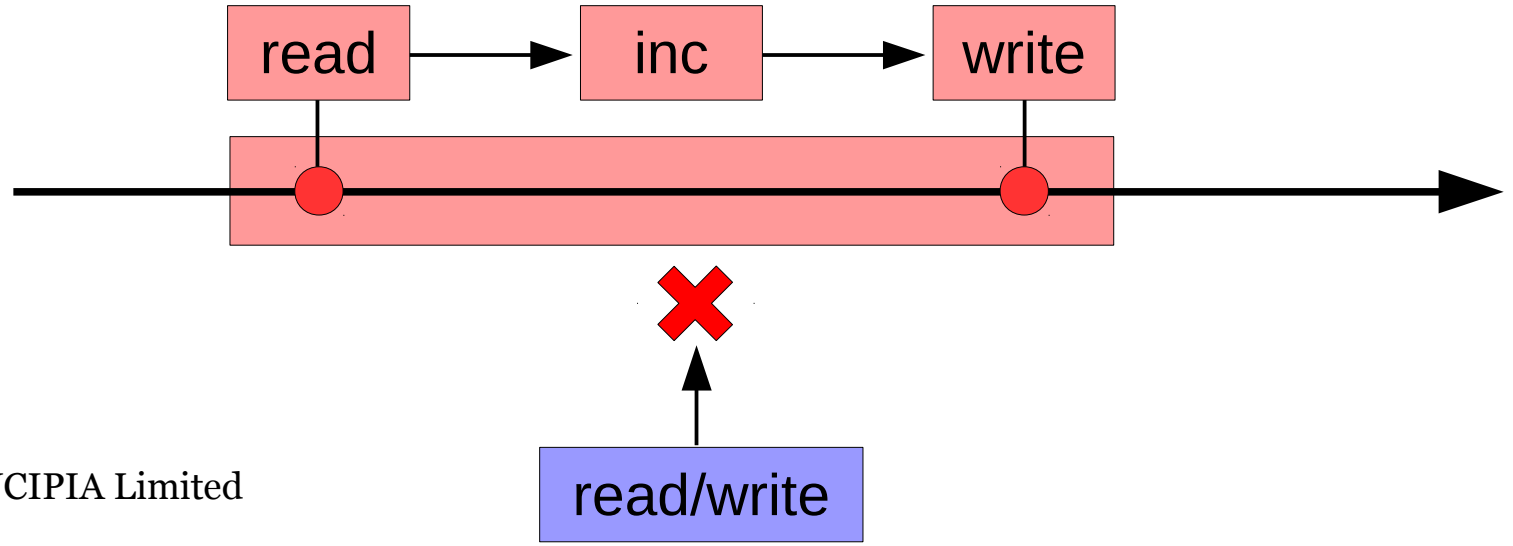
- Test and Set
- Exchange
- Fetch and Add
- **Compare and Swap**
- Load-Link / Store-Conditional
- Transactional Memory

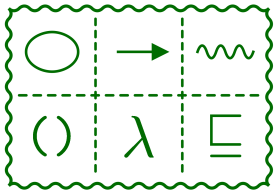


不可分操作



Read-Modify-Write

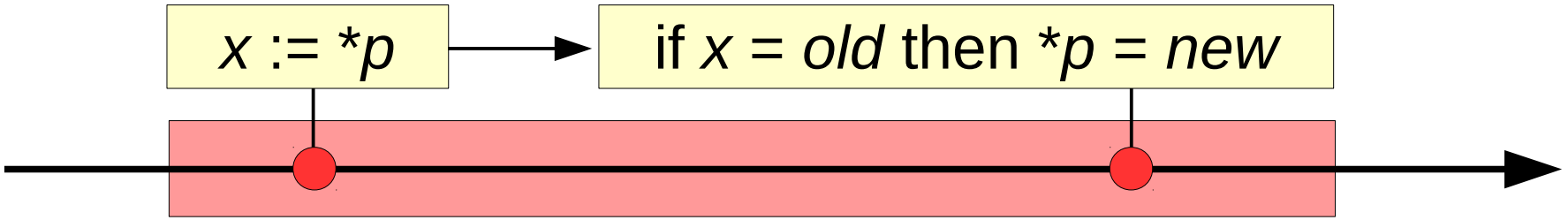




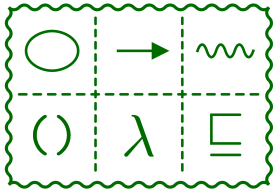
Compare-And-Swap (CAS)

```
T CAS(T *p, T old, T new)
```

- ポインタ *p* の指す値を読み込む
- 読み込んだ値と *old* を比較する（Compare）
- 値が一致した場合は *new* を書き込む（Swap）
- 最初に読み込んだ値を返す

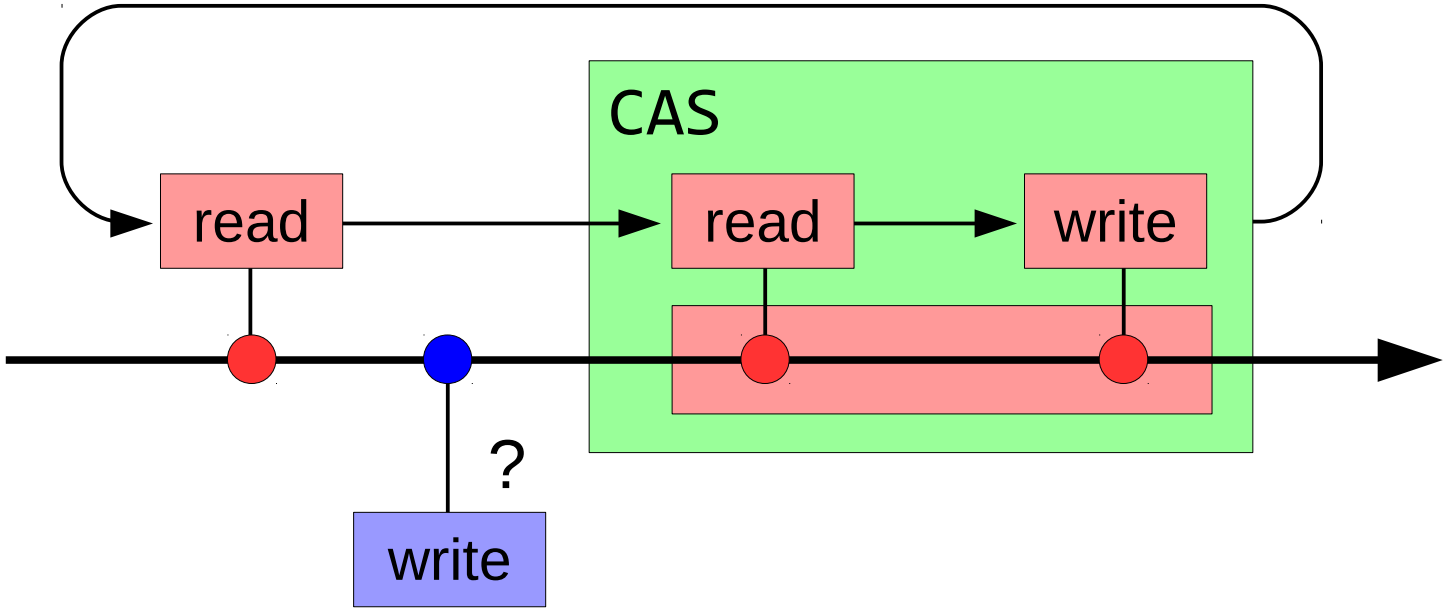


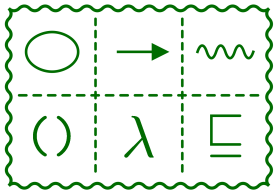
※ *x* ≠ *old* の場合は *x* を書き込むものもある



CAS によるインクリメント

```
do {
   x = *p;
   y = CAS(p, x, x + 1);
} while (y != x);
```





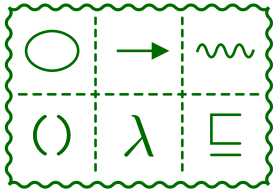
CAS によるミューテックス

```
int mutex = 0;
```

```
void lock(void) {
  while (1) {
    if (CAS(&mutex, 0, 1) == 0)
      break;
    sched_yield();
  }
}
```

Busy-wait

```
void unlock(void) {
  mutex = 0;
}
```



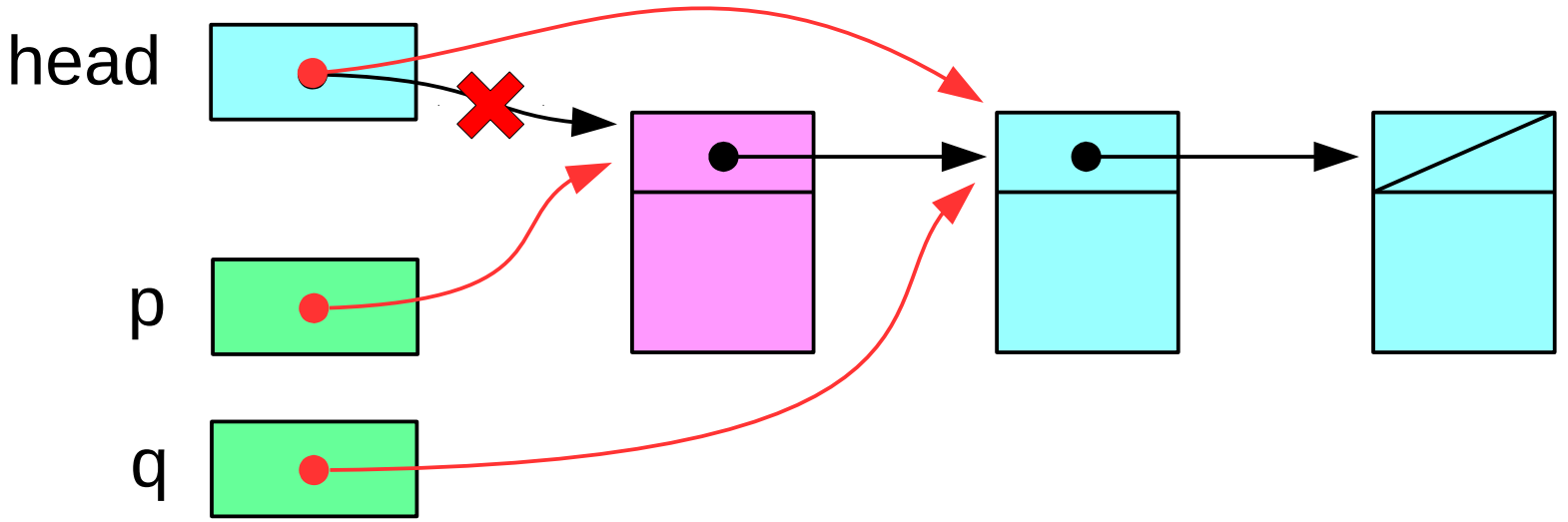
CAS によるリスト操作

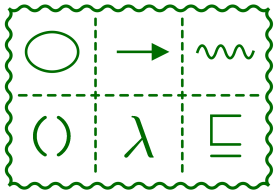
```
do {
   p = head;
   if (p == NULL)
      return NULL;
   q = p->next;
} while (CAS(&head, p, q) != p);
```

※ このコードには問題がある（後ほど修正）

head は共有変数，p，q はローカル変数

この時点でノードは解放されているかもしれない





Blocking / Non-Blocking

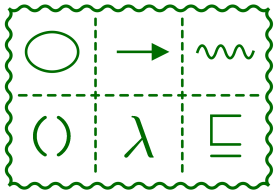
- 他のプロセスの振る舞いによって停止する可能性のある操作をブロック操作という
- 他のプロセスのいかなる振る舞いによっても停止しない操作をノンブロック操作という

ミューテックスと条件変数による pop

```
struct node *pop() {
  mutex_lock(&mutex);
  while (head == NULL)
    cond_wait(&cond, &mutex);
  p = head;
  head = head->next;
  mutex_unlock(&mutex);
  return p;
}
```

CAS よる pop

```
struct node *pop() {
  do {
    p = head;
    if (p == NULL)
      return NULL;
    q = p->next;
  } while (CAS(&head, p, q) != p);
  return p;
}
```



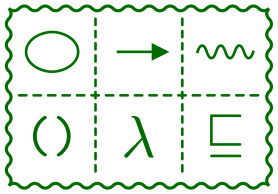
Lock-Free / Wait-Free

- 呼び出したすべてのプロセスが有限ステップのうちに完了するものを wait-free 操作という
- 呼び出したプロセスのうちのいずれかに必ず進捗があるもの lock-free 操作という

```
struct node *pop() {
  do {
    p = head;
    if (p == NULL)
      return NULL;
    q = p->next;
  } while (CAS(&head, p, q) != p);
  return p;
}
```

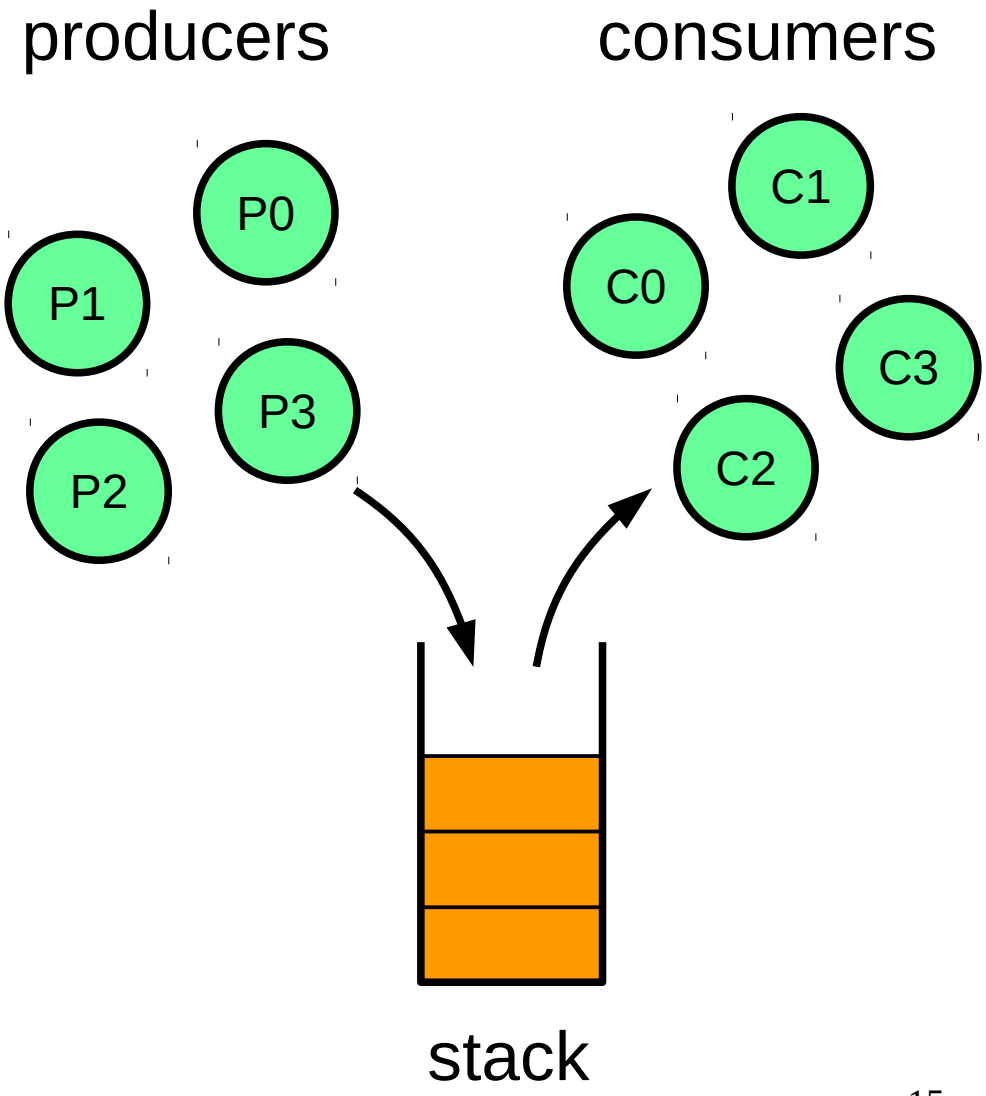
CAS よる pop は lock-free であるが wait-free ではない:

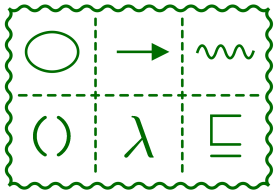
push プロセスと複数の pop プロセスが同時に走っている場合，ある pop プロセスだけが進捗しない場合がありうる



Lock-Free Stack

不可分操作 CAS を使って
スレッド間で共有できる
Lock-Free Stack を実装する





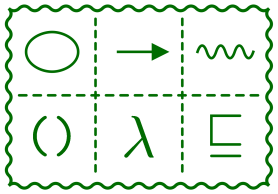
仕様のモデル化: イベント定義

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

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

NP	Producer 数
NC	Consumer 数
M	データの種類の数
L	スタックの容量

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

(define-channel push (x) DM)
(define-channel pop (x) DM)
```

イベント定義

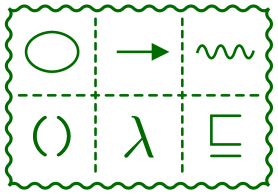


仕様のモデル化: STACK

```
(define-process (STACK xs)
  (alt
    (if (< (length xs) L)
        (? push (x) (STACK (cons x xs)))
        STOP)
    (if (null? xs)
        STOP
        (! pop ((car xs)) (STACK (cdr xs))))))
```

先頭に追加

先頭から取り出し



仕様のモデル化: SPEC

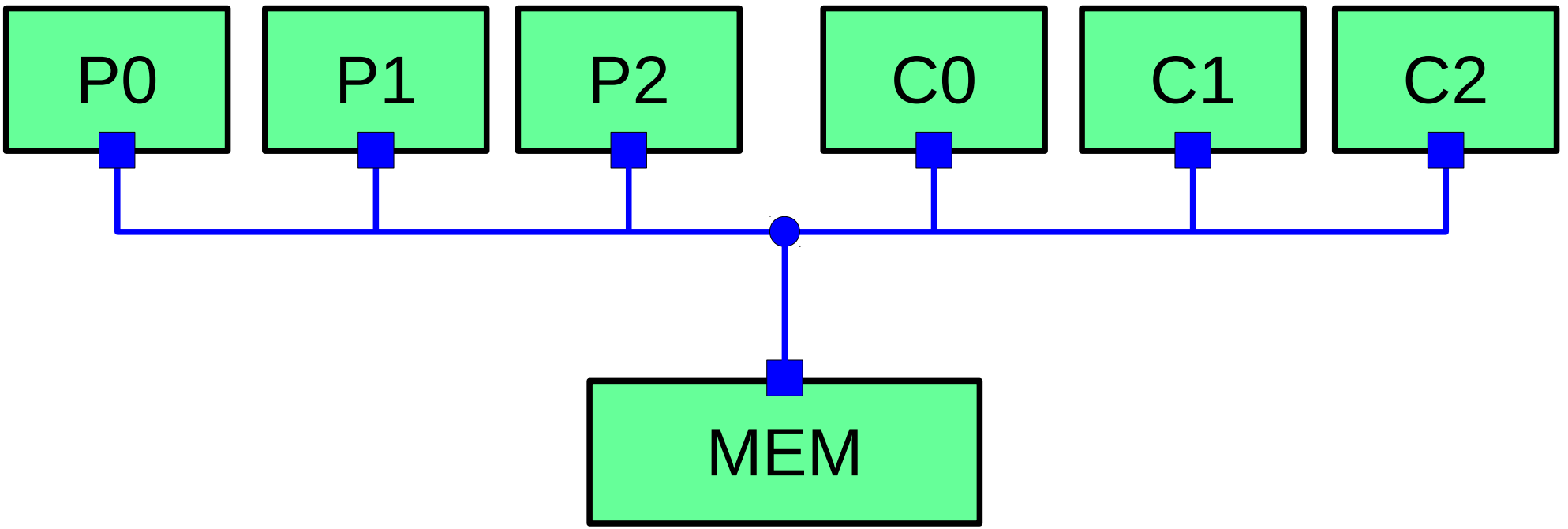
```
(define-process IN (? in (x) (! push (x) IN)))
```

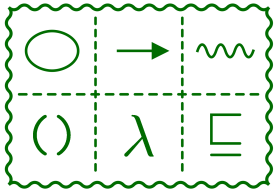
```
(define-process OUT (? pop (x) (! out (x) OUT)))
```

```
(define-process SPEC
  (hpar (list push pop)
    (par '()
      (if (= NP 1)
          IN
          (xpar k (interval 0 NP) '() IN))
      (if (= NC 1)
          OUT
          (xpar k (interval 0 NC) '() OUT)))
    (STACK '())))
```

push, pop は隠蔽
in, out を観測する

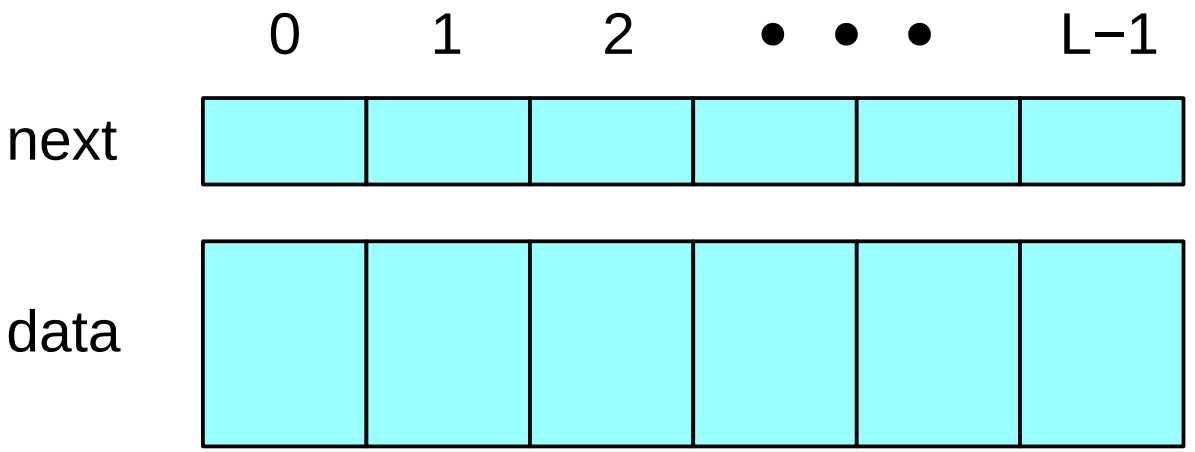
実装のモデル化

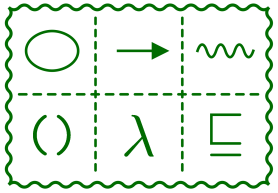




実装のモデル化

- メモリ空間を配列で表す
  - リストノードを2つの配列 next と data で表す
- ポインタをインデックスで表す
  - NULL ポインタは空リスト ( ) で表す





イベント定義: head

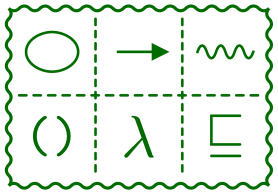
```
(define IL (interval 0 L))
(define DL (map list IL))
(define ILx (cons '() IL))
(define DLx (map list ILx))
(define DLxLxB (combinations (list ILx ILx '(#f #t))))
```

終端は空リスト () で表す

```
(define-channel rd-head (x) DLx)
(define-channel cas-head (old new b) DLxLxB)
```

対象は head と決まっているのでパラメータ p はない

パラメータ b で書き込みの成否を表す（同期型インターフェイス）



イベント定義: ノード

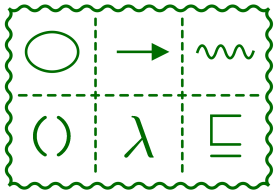
```
(define DLM (combinations (list IL IM)))
(define DLLx (combinations (list IL ILx)))
```

```
(define-channel rd (k x) DLM)
(define-channel wr (k x) DLM)
(define-channel rd-next (k p) DLLx)
(define-channel wr-next (k p) DLLx)
```

データと next ポインタの読み書き

```
(define-channel alloc (p) DL)
(define-channel free (p) DL)
```

空きノード管理



補助関数

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

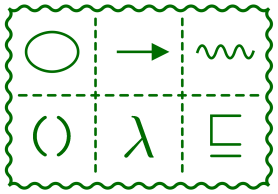
リスト xs の k 番目の要素を x に置き換えたリストを返す

配列の更新で使用する

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

昇順リスト xs に再び昇順となるように x を挿入する

空きノード管理で使用する



CAS のモデル化

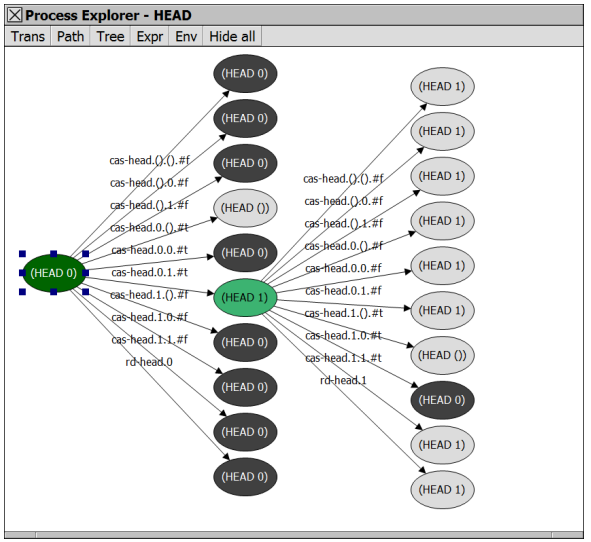
ポインタ head のモデル化例

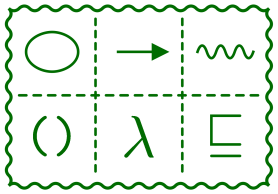
L=2

```
(define-process (HEAD head)
  (alt
    (! rd-head (head) (HEAD head))
    (? cas-head (old new b)
       (equal? b (equal? old head))
       (HEAD (if b new head)))))
```

Transitions

event	target
cas-head.().().#f	(HEAD 0)
cas-head.().0.#f	(HEAD 0)
cas-head.().1.#f	(HEAD 0)
cas-head.0.().#t	(HEAD ())
cas-head.0.0.#t	(HEAD 0)
cas-head.0.1.#t	(HEAD 1)
cas-head.1.().#f	(HEAD 0)
cas-head.1.0.#f	(HEAD 0)
cas-head.1.1.#f	(HEAD 0)
rd-head.0	(HEAD 0)





プロセス定義: メモリ MEM

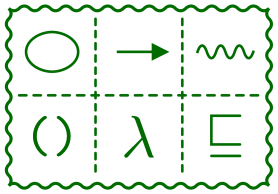
```
(define-process (MEM head ls ds fs)
  (alt
    (! rd-head (head) (MEM head ls ds fs))
    (? cas-head (old new b)
       (equal? b (equal? old head))
       (MEM (if b new head) ls ds fs))
    (? rd (k x) (equal? x (list-ref ds k))
       (MEM head ls ds fs))
    (? wr (k x)
       (MEM head ls (update ds k x) fs))
    (? rd-next (k p) (equal? p (list-ref ls k))
       (MEM head ls ds fs))
    (? wr-next (k p)
       (MEM head (update ls k p) ds fs))
    (if (null? fs)
        STOP
        (! alloc ((car fs))
          (MEM head ls ds (cdr fs))))
    (? free (p)
       (MEM head
            (update ls p '())
            (update ds p 0)
            (insert p fs)))))
```

head

ノードデータ

ノード next ポインタ

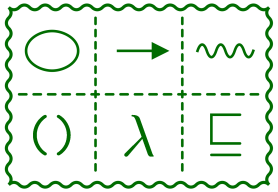
空きノード管理



メモリ: プロセスパラメータ

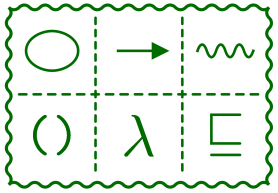
```
(define-process (MEM head ls ds fs)
  ...
```

head	head ポインタ
ls	next ポインタ配列（リスト）
ds	データ配列（リスト）
fs	フリーノードポインタの昇順リスト



メモリ: head

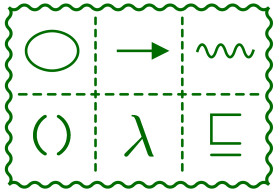
```
(define-process (MEM head ls ds fs)
  (alt
    (! rd-head (head) (MEM head ls ds fs))
    (? cas-head (old new b)
       (equal? b (equal? old head))
       (MEM (if b new head) ls ds fs))
    ...
```



メモリ: ノード

```
(define-process (MEM head ls ds fs)
  (alt
    ...
    (? rd (k x) (equal? x (list-ref ds k))
       (MEM head ls ds fs))
    (? wr (k x)
       (MEM head ls (update ds k x) fs))

    (? rd-next (k p) (equal? p (list-ref ls k))
       (MEM head ls ds fs))
    (? wr-next (k p)
       (MEM head (update ls k p) ds fs))
    ...
```

rd, rd-next は同期型インターフェイス



メモリ: 空きノード管理

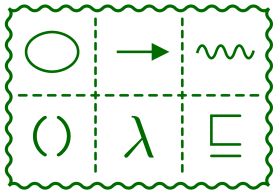
```
(define-process (MEM head ls ds fs)
  (alt
    ...
    (if (null? fs)
        STOP
        (! alloc ((car fs))
          (MEM head ls ds (cdr fs))))
    (? free (p)
       (MEM head
            (update ls p '())
            (update ds p 0)
            (insert p fs)))))
```

空きノード割り当て

ノード解放

未使用ノードを初期化して状態を少なくする

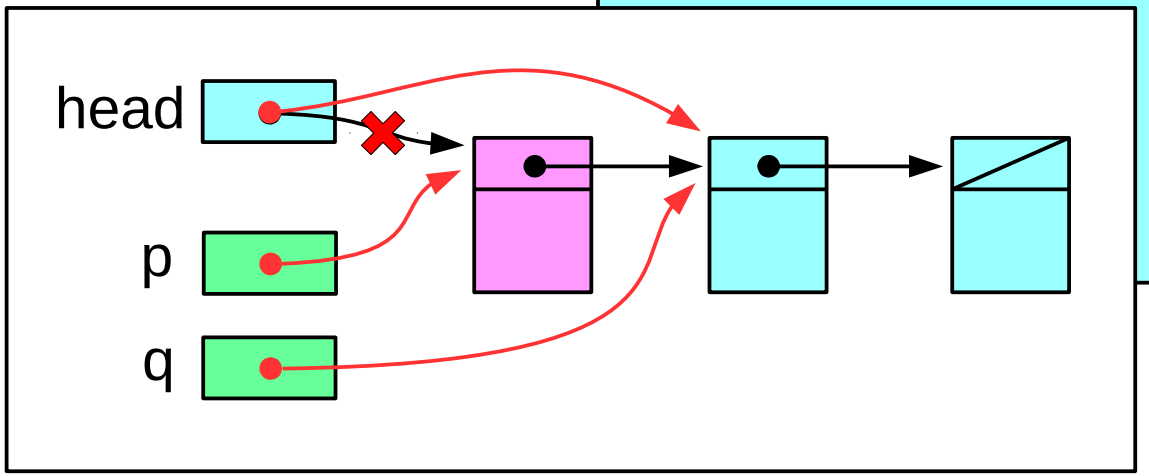
リストを昇順にして状態を少なくする

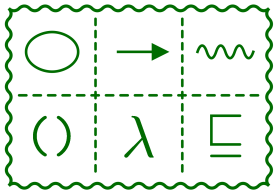


プロセス定義: 消費者

```
(define-process Q
  (? rd-head (p)
     (if (null? p)
         Q
         (? rd-next (pp q) (equal? pp p)
            (? cas-head (old new b)
               (and (equal? old p)
                    (equal? new q))
               (if (not b)
                   Q
                   (? rd (pp x) (equal? pp p)
                      (! free (p)
                         (! out (x) Q)))))))))
```

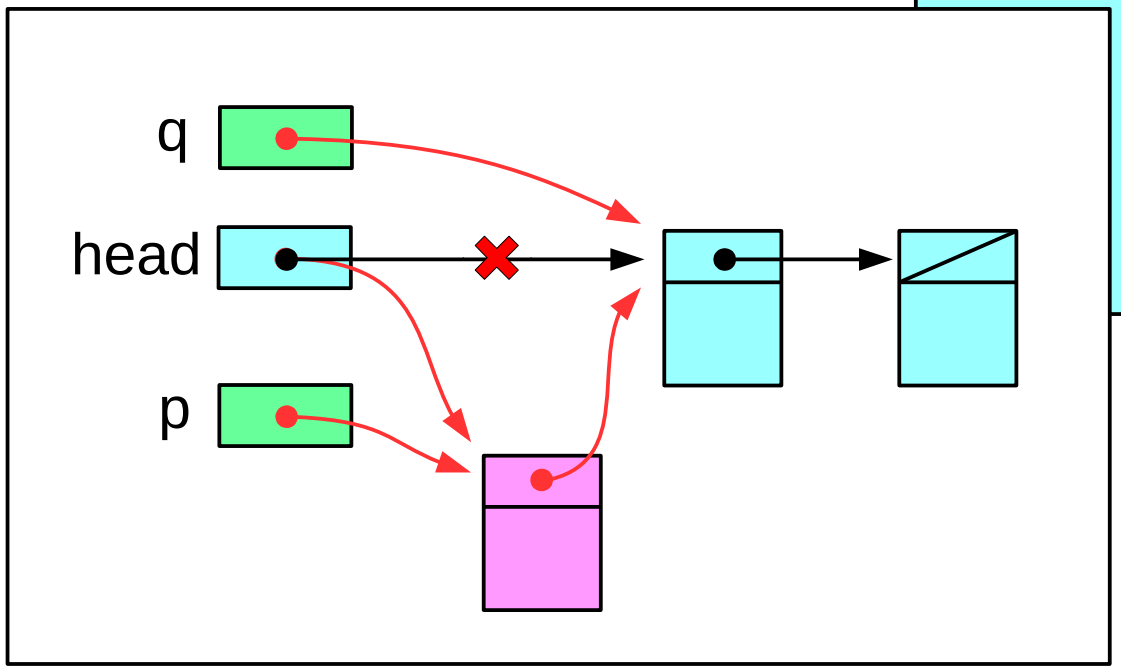
```
do {
  p = head;
  if (p == NULL)
    return NULL;
  q = p->next;
} while (CAS(&head, p, q) != p);
```

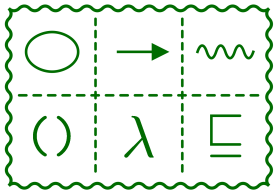




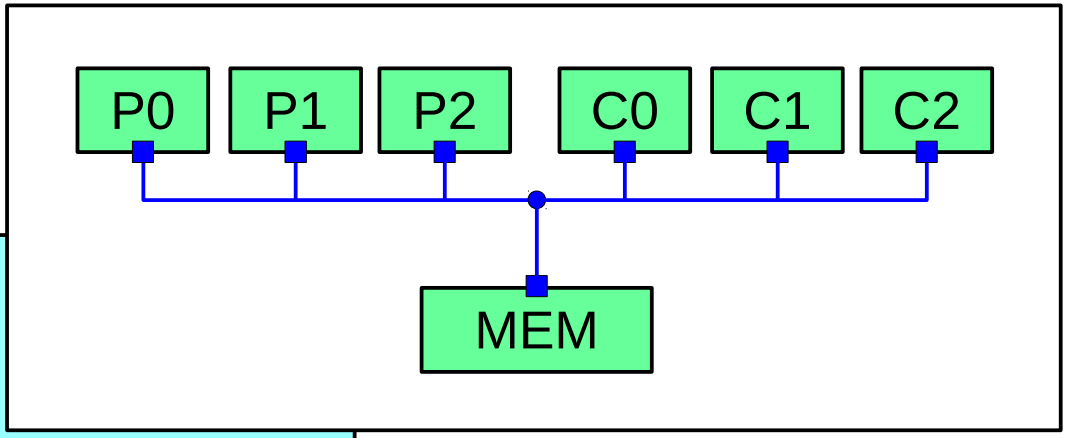
プロセス定義: 生産者

```
(define-process P
  (? in (x)
     (? alloc (p)
        (! wr (p x) (P-LOOP p)))))

(define-process (P-LOOP p)
  (? rd-head (q)
     (! wr-next (p q)
        (? cas-head (old new b)
           (and (equal? old q)
                (equal? new p))
           (if b P (P-LOOP p))))))
```





並行合成

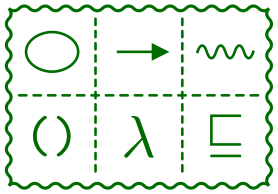



```
(define-process SYS
  (par X
    (par '()
      (if (= NP 1)
          P
          (xpar k (interval 0 NP) '() P))
      (if (= NC 1)
          Q
          (xpar k (interval 0 NC) '() Q)))
    (MEM '()
         (map (lambda (k) '()) IL)
         (map (lambda (k) 0) IL)
         IL)))
```

```
(define X
  (list rd-head cas-head
    rd wr rd-next wr-next
    alloc free)
```

メモリ初期値
head = ()
next ポインタはすべて ()
データはすべて 0
すべてのノードが空き

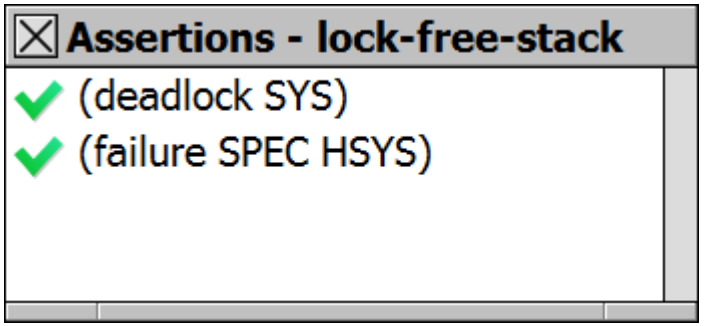
```
(define-process HSYS
  (hide X SYS))
```



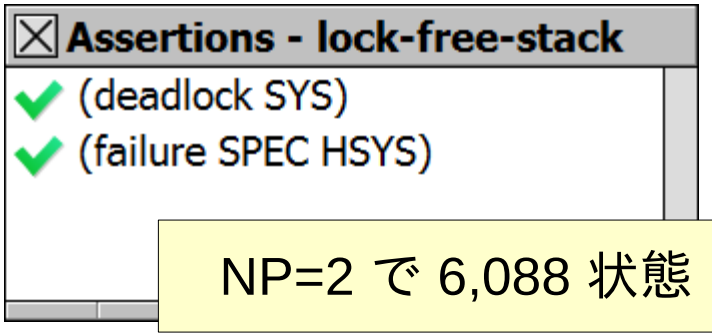
検査

NP	Producer 数
NC	Consumer 数
M	データの種類の数
L	スタックの容量

NP=1,2
NC=1,2
M=2
L=1



NP=1,2
NC=1
M=2
L=2

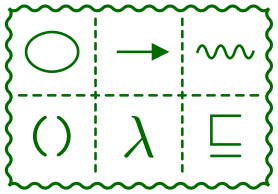


NP=2 で 6,088 状態

NP=1
NC=2
M=2
L=2

メモリ不足でエラー

状態爆発???



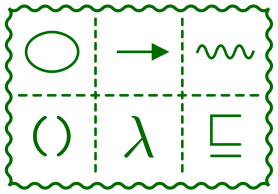
表明（assertion）を検査する

- 生産者・消費者に状態爆発する変数はない
- メモリが持つパラメータのうち head は有限，ls，ds は update による更新のみなので長さは変わらない．値が不正であればチャネル通信で定義域エラーになる
- 疑わしきは空きノードのポインタリスト fs
- 以下の assertion を仕掛けて検査する

```
(define-process (MEM head ls ds fs)
  (if (< L (length fs))
      STOP
      (alt
        ...
```

不変表明の一部

デッドロックさせる



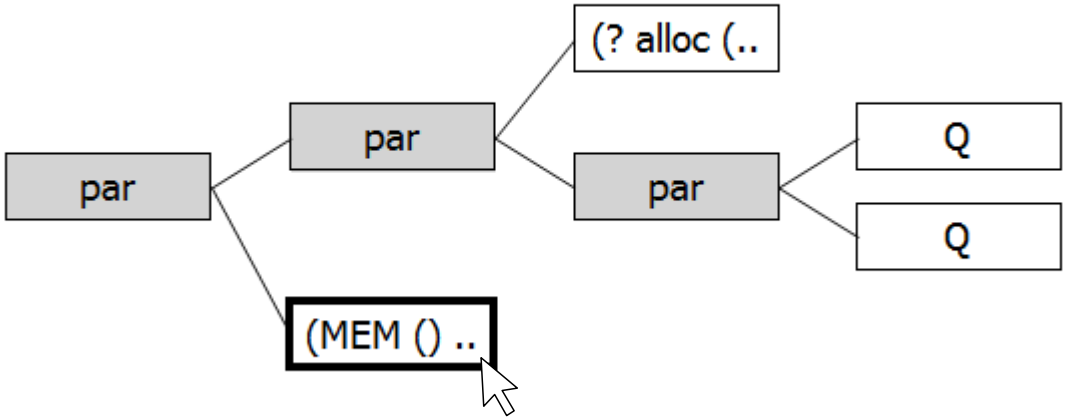
表明の検査

Assertions - lock-free-stack

- (deadlock SYS)
- (failure SPEC HSYS)

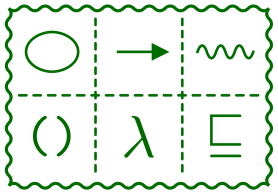
ノード 0 が2度解放されている

Process Tree



Environment

variable	value
ds	(0 0)
fs	(0 0 1)
head	()
ls	(() ())



分析

Path

	event	s
0	in.1	
1	alloc.0	
2	wr.0.1	
3	rd-head.()	
4	wr-next.0.()	
5	cas-head.().0.#t	
6	in.0	
7	alloc.1	
8	wr.1.0	
9	rd-head.0	
10	wr-next.1.0	
11	cas-head.0.1.#t	
12	rd-head.1	
13	rd-head.1	
14	rd-next.1.0	
15	cas-head.1.0.#t	
16	rd.1.0	
17	rd-next.1.0	
18	free.1	
19	out.0	
20	rd-head.0	
21	in.0	

Path

	event	s
21	in.0	
22	rd-next.0.()	
23	alloc.1	
24	cas-head.0.().#t	
25	wr.1.0	
26	rd-head.()	
27	wr-next.1.()	
28	cas-head.().1.#t	
29	cas-head.1.0.#t	
30	rd.1.0	
31	free.1	
32	out.0	
33	rd-head.0	
34	rd-next.0.()	
35	rd.0.1	
36	cas-head.0.().#t	
37	rd.0.1	
38	free.0	
39	in.1	
40	free.0	
41	out.1	
42	out.1	

push
node 0

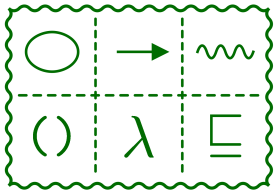
push.1

pop.1
消費者競合
一方は CAS 待ち

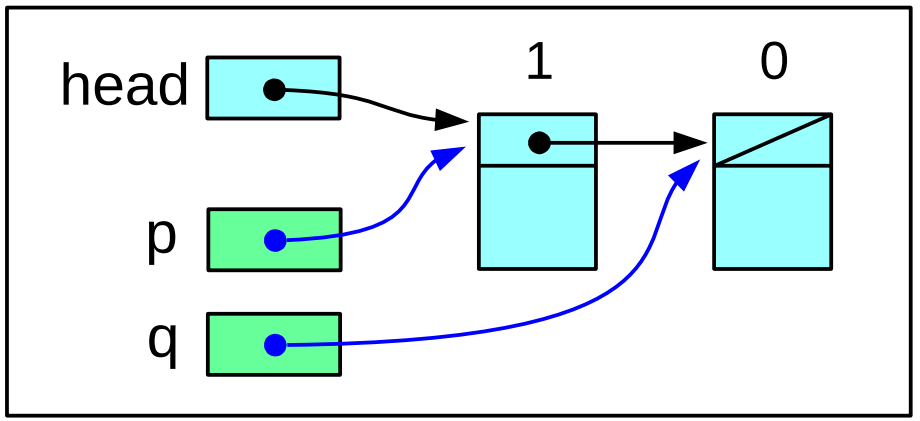
pop.0

push.1

ここで CAS 実行

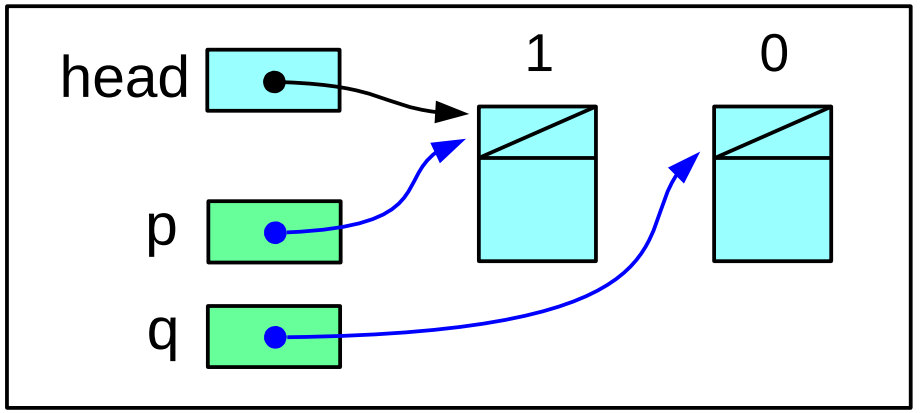


分析

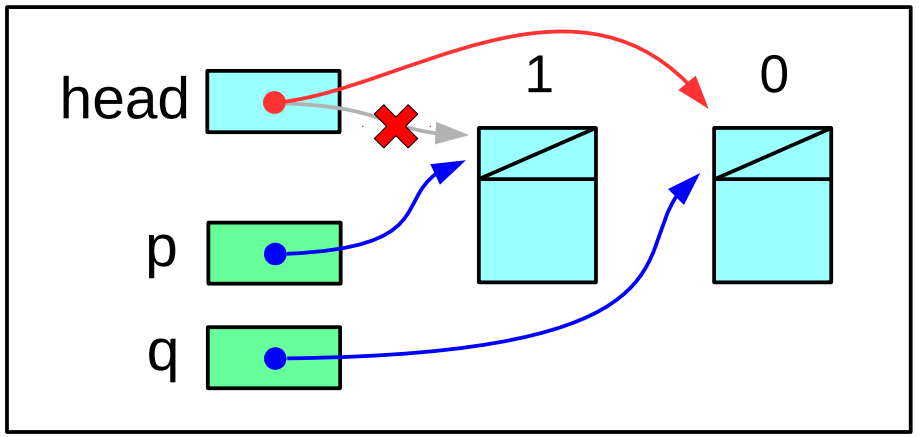


生産者が2回 push

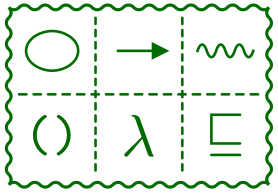
消費者 A が pop するために
p=1, q=0 を読んで CAS 待ち



消費者 B が2回 pop したのち
生産者が1回 push
ノード 0 は消費者 B による解放待ち

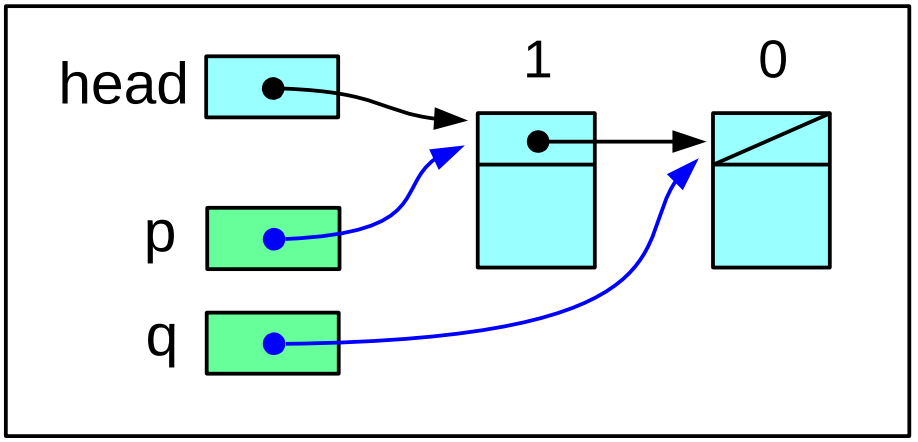


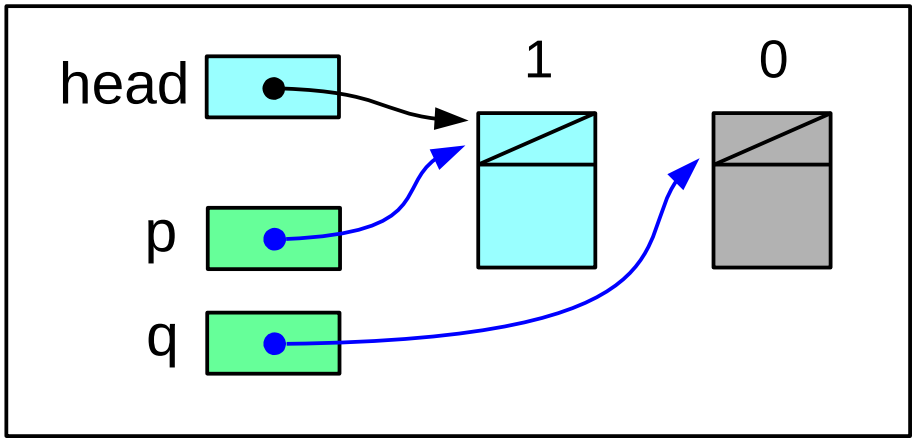
消費者 A が CAS を実行
head = p = 1 なので成功してしまう



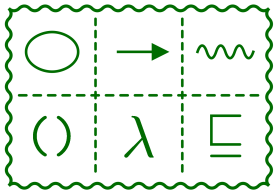
ABA 問題

- 2回の読出し結果が同じであることから，その間に変更はなかったと誤った判断をしてしまう問題





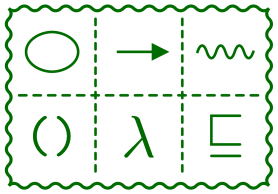
head = 1 だけを見ているのでリストの変更を識別できない



ABA 問題への対策

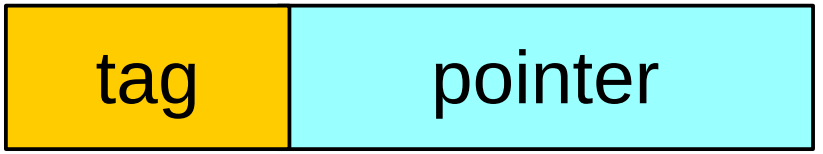
- タグ付きポインタ
- ハザードポインタ
- Load-Link / Store-Conditional
- Transactional Memory
- Double Compare-And-Swap

Garbage Collection のあるプログラミング言語では
ABA 問題は発生しない

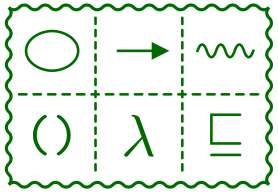


タグ付きポインタ

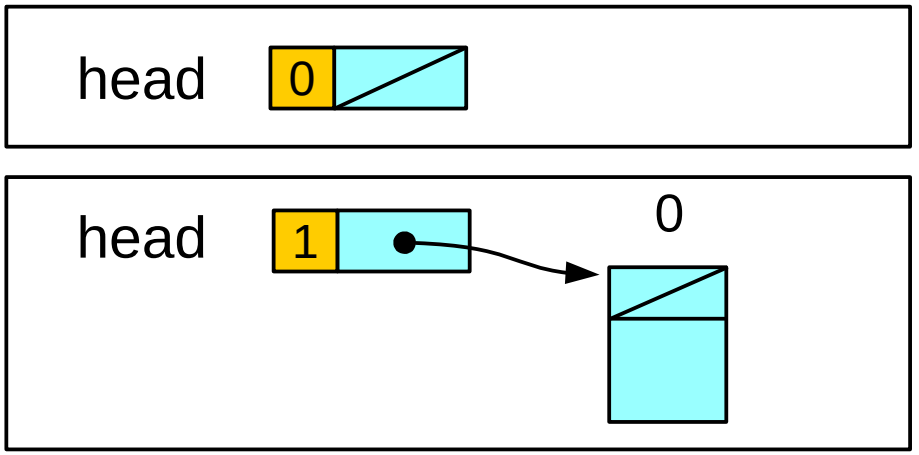
- CAS で比較の対象となるポインタ変数にタグをつけて拡張し，タグ込みで比較を行う
- ポインタ変数に値を書き込む際にはタグを変更する．こうすることで値が同じでも変更があったことを識別できる
- タグの変域は有限なので完全な解ではない．しかし ABA 問題が発生する可能性を減らせる

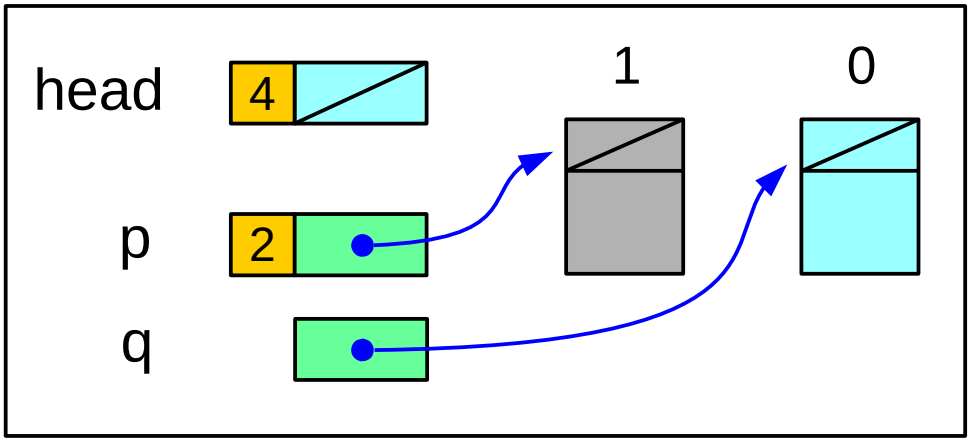


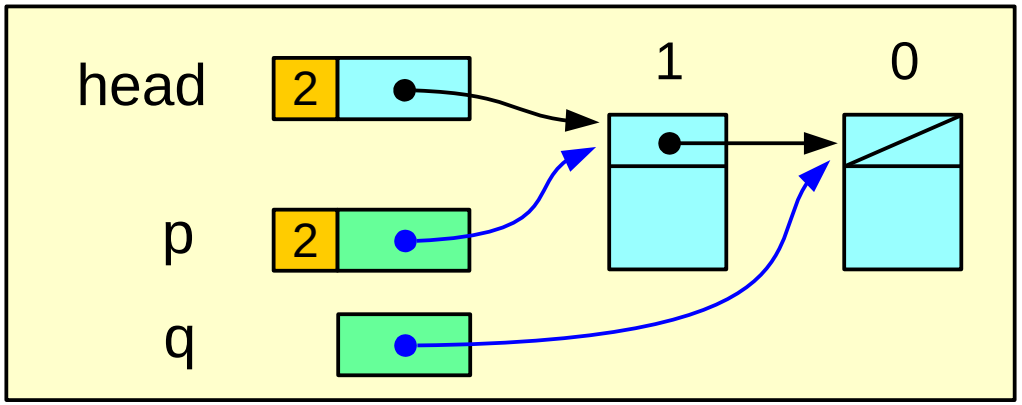
※ バリエーションがいくつか存在する

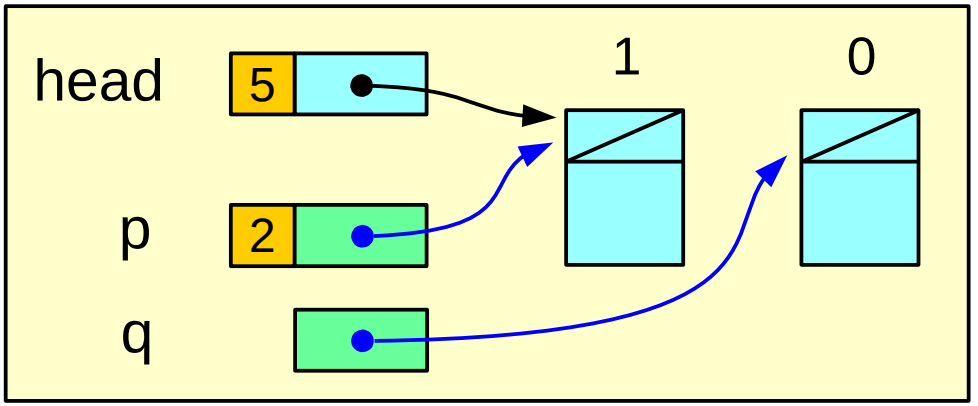


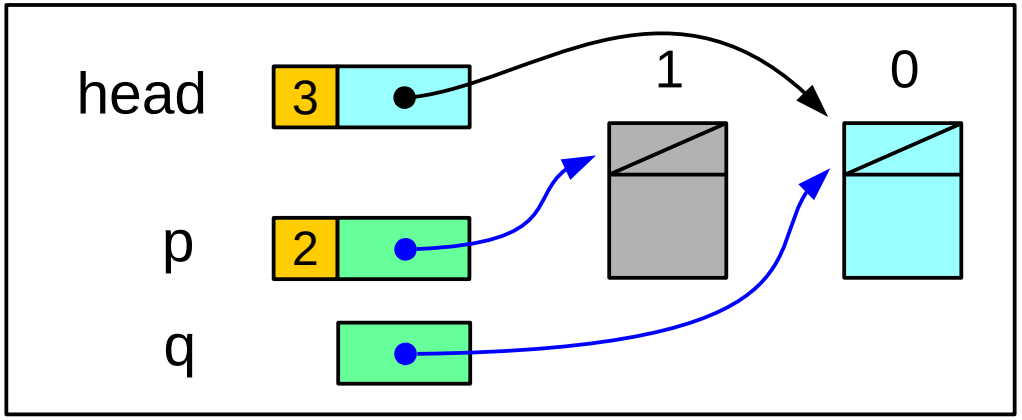
タグ付きポインタでのシナリオ

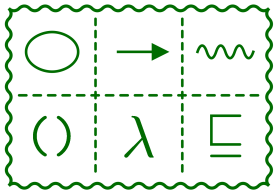












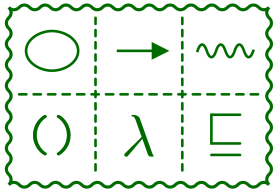
タグ付きポインタのモデル化

head をインデックスとタグのリスト (index tag) で表す

タグの変域は 0〜K−1

```
(define K 5)
(define IK (interval 0 K))
(define ILxK (combinations (list ILx IK)))
(define DLxK (map list ILxK))
(define DLxKLxB (combinations (list ILxK ILx '(#f #t))))
```

```
(define-channel rd-head (x) DLxK)
(define-channel cas-head (old new b) DLxKLxB)
```

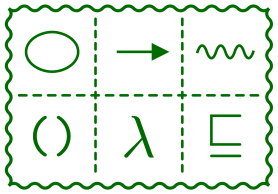


メモリ: head 部

```
(define-process (MEM head ls ds fs)
  (alt
   (! rd-head (head) (MEM head ls ds fs))
   (? cas-head (old new b)
      (equal? b (equal? old head))
      (let ((tag (mod (+ (cadr head) 1) K)))
        (MEM (if b (list new tag) head) ls ds fs)))
    ...
```

タグを含めて比較

タグを更新

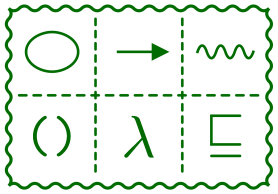


消費者

```
(define-process Q
  (? rd-head (pt)
     (if (null? (car pt))
         Q
         (? rd-next (pp q) (equal? pp (car pt))
            (? cas-head (old new b)
               (and (equal? old pt)
                    (equal? new q))
(if (not b)
    Q
    (? rd (pp x)
       (equal? pp (car pt))
       (! free (p)
          (! out (x) Q)))))))))
```

```
(define-process P
  (? in (x)
     (? alloc (p)
        (! wr (p x) (P-LOOP p)))))

(define-process (P-LOOP p)
  (? rd-head (qt)
     (! wr-next (p (car qt))
        (? cas-head (old new b)
           (and (equal? old qt)
                (equal? new p))
           (if b P (P-LOOP p))))))
```

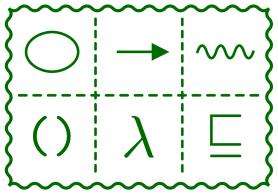
生産者



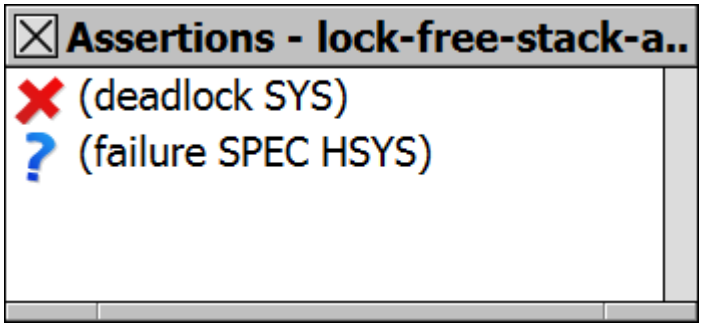
システムプロセス: head 初期値

```
(define-process SYS
  (par X
    (par '()
      (if (= NP 1)
          P
          (xpar k (interval 0 NP) '() P))
      (if (= NC 1)
          Q
          (xpar k (interval 0 NC) '() Q)))
    (MEM '(() 0)
         (map (lambda (k) '()) IL)
         (map (lambda (k) 0) IL)
         IL)))
```

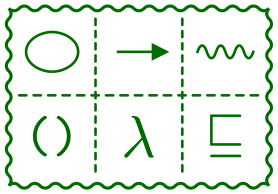
タグの初期値 0



検査



NP=1	Producer 数
NC=2	Consumer 数
M=2	データの種類の数
L=2	スタックの容量
K=5～10	タグの変域



分析

K = 10

Path		Path		Path		Path	
	event		event		event		event
0	in.0	20	wr.0.0	40	cas-head.(() 6).0.#t	59	wr-next.1
1	alloc.0	21	rd-head.(1 3)	41	in.0	60	cas-head.(0 9).1.#t
2	wr.0.0	22	wr-next.(	42	rd.1.0	61	out.0
3	rd-head.(() 0	23	out.0	43	free.1	62	rd-head.(1 0)
4	wr-next.0.()	24	cas-head.(1 3).0.#t	44	alloc.1	63	rd-next.1.0
5	cas-head.(() 0).0.#t	25	rd-head.(0 4	45	wr.1.0	64	cas-head.(1 0).0.#t
6	rd-head.(0 1)	26	rd-next.0.1	46	rd-head.(	65	cas-head.(0 1).1.#t
7	rd-head.(0 1)	27	cas-head.(0 4).1.#t	47	wr-next.1.0	66	rd.0.0
8	in.0	28	rd.0.0	48	cas-head.(0 7).1.#t	67	free.0
9	rd-next.0.()	29	rd-next.0.1	49	out.0	68	out.0
10	alloc.1	30		50	rd-head.(1 8)	69	rd-head.(1 2)
11	cas-head.(0 1).().#t	31		51	rd-next.1.0	70	rd-next.1.0
12	wr.1.0	32	in.0	52	cas-head.(1 8).0.#t	71	cas-head.(1 2).0.#t
13	rd-head.(() 2)	33	rd-head.(1 5)	53	rd.1.0	72	rd.1.0
14	wr-next.1.()	34	alloc.0	54	in.0	73	free.1
15	rd.0.0	35	rd-next.1.()	55	free.1	74	rd.1.0
16	cas-head.(() 2).1.#t	36	wr.0.0	56	alloc.1	75	free.1
17	free.0	37	cas-head.(1 5).().#t	57	wr.1.0	76	out.0
18	in.0	38	rd-head.(() 6)	58	rd-head.(0 9)	77	in.0
19	alloc.0	39	wr-next.0.()	59	wr-next.1.0	78	out.0

(0)

(0 1)

(1)

(1 0)

(0)

()

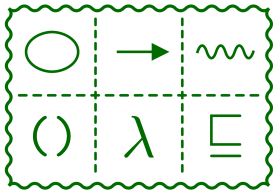
()

(1)

(1 0)

(0)

ゴミを読んでいる

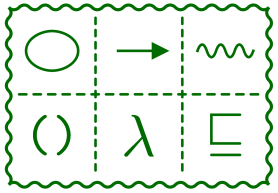


失敗シナリオ

- 消費者 A がある時点での head を読む
- 生産者と消費者 B が push，pop を繰り返し，タグの値が1周する
- 消費者 A が head の一致した時点で CAS を実行する

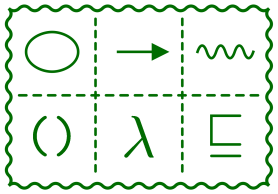
どんなに K の値を大きくとっても，必ず循環するケースが存在する

現実には K の値を大きくするにしたがって発生の可能性は減る



モデルの有限化

- 生産者・消費者ともに指定回数だけ実行して終了するモデルに変更する
- 与えられた実行回数 T に対し，十分大きな K をとれば問題が発生しないことを確認して良しとする



モデルの有限化

```
(define TP 3)
(define TC 2)
```

TP Producer の実行回数
TC Consumer の実行回数

仕様

```
(define-process (IN n)
  (if (= n 0)
      SKIP
      (? in (x) (! push (x)

(define-process (OUT n)
  (if (= n 0)
      SKIP
      (? pop (x) (! out (x)
```

消費者

```
(define-process (Q n)
  (if (= n 0)
      SKIP
      (? rd-head (pt)
         (if (null? (car pt))
             (Q n)
             (? rd-link (pp q) (equal?
                (? cas-head (old new b
                   (and (equal? old pt
                        (equal? new q)
                   (if (not b)
                       (Q n)
                       (? rd (pp x) (equal? pp (car pt))
                          (! free ((car pt))
                             (! out (x) (Q (- n 1)))))))))))))
```

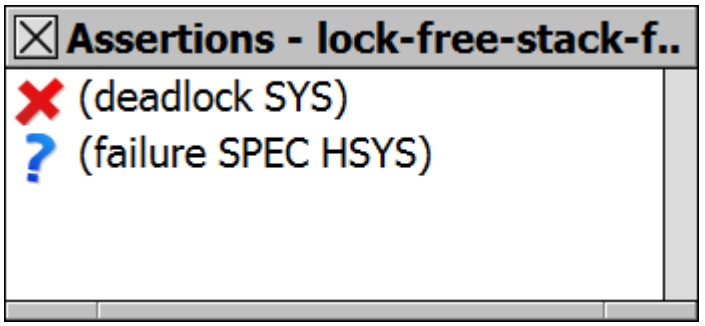
生産者

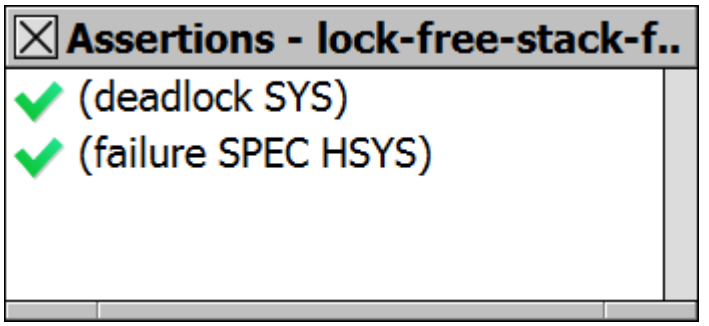
```
(define-process (P n)
  (if (= n 0)
      SKIP
      (? in (x)
         (? alloc (p)
            (! wr (p x)
               (P-LOOP n p))))))

(define-process (P-LOOP n p)
  (? rd-head (qt)
     (! wr-link (p (car qt))
        (? cas-head (old new b)
           (and (equal? old qt)
                (equal? new p))
           (if b
               (P (- n 1))
               (P-LOOP n p))))))
```

検査

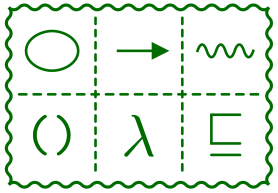
TP	Producer の実行回数
TC	Consumer の実行回数
K	タグの変域

NP=1	Producer 数
NC=2	Consumer 数
M=1	データの種類の数
L=2	スタックの容量





TP=3, TC=2, K=4	TP=3, TC=2, K=5
TP=4, TC=3, K=5	TP=4, TC=3, K=7
TP=5, TC=4, K=7	TP=5, TC=4, K=9

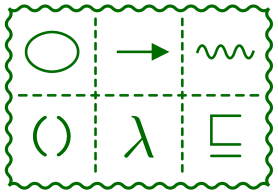


実装

stack-lockfree.c

- モデルと同様にノードは2つの配列で表現する
- ノードへのポインタは 32bit の符号付き整数で表す．終端は –1 で表す

```
#define L 4

volatile int32_t head;
int32_t next[L];
int data[L];
```



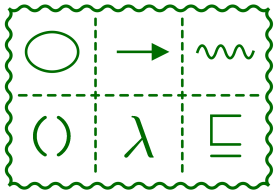
GCC 組み込みの CAS 関数

```
T __sync_val_compare_and_swap(T *p, T old, T new)
```

- intel x86 では 8, 16, 32, 64bit がサポートされている
- intel x86_64 では加えて 128bit がサポートされている
  - オプション `-mcx16` をつける

※ gcc 4.9.2 では legacy に分類されている．マニュアルを参照のこと
5.47 Built-in functions for atomic memory access
6.51 Legacy __sync Built-in Functions for Atomic Memory Access

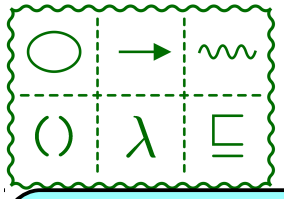
※ C11 / C++11 では atomic operations も用意されている
stdatomic.h と 関数 atomic_compare_exchange_* について調査のこと



関数 push

```
(define-process P
  (? in (x)
     (? alloc (p)
        (! wr (p x) (P-LOOP p)))))

(define-process (P-LOOP p)
  (? rd-head (q)
     (! wr-next (p q)
        (? cas-head (old new b)
           (and (equal? old q)
                (equal? new p))
           (if b P (P-LOOP p))))))
```

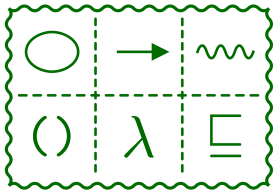
```
void push(int x)
{
    int32_t p, q;
    p = alloc_node();
    data[p] = x;
    do {
        q = head;
        next[p] = q;
    } while (__sync_val_compare_and_swap(&head, q, p) != q);
}
```



関数 pop

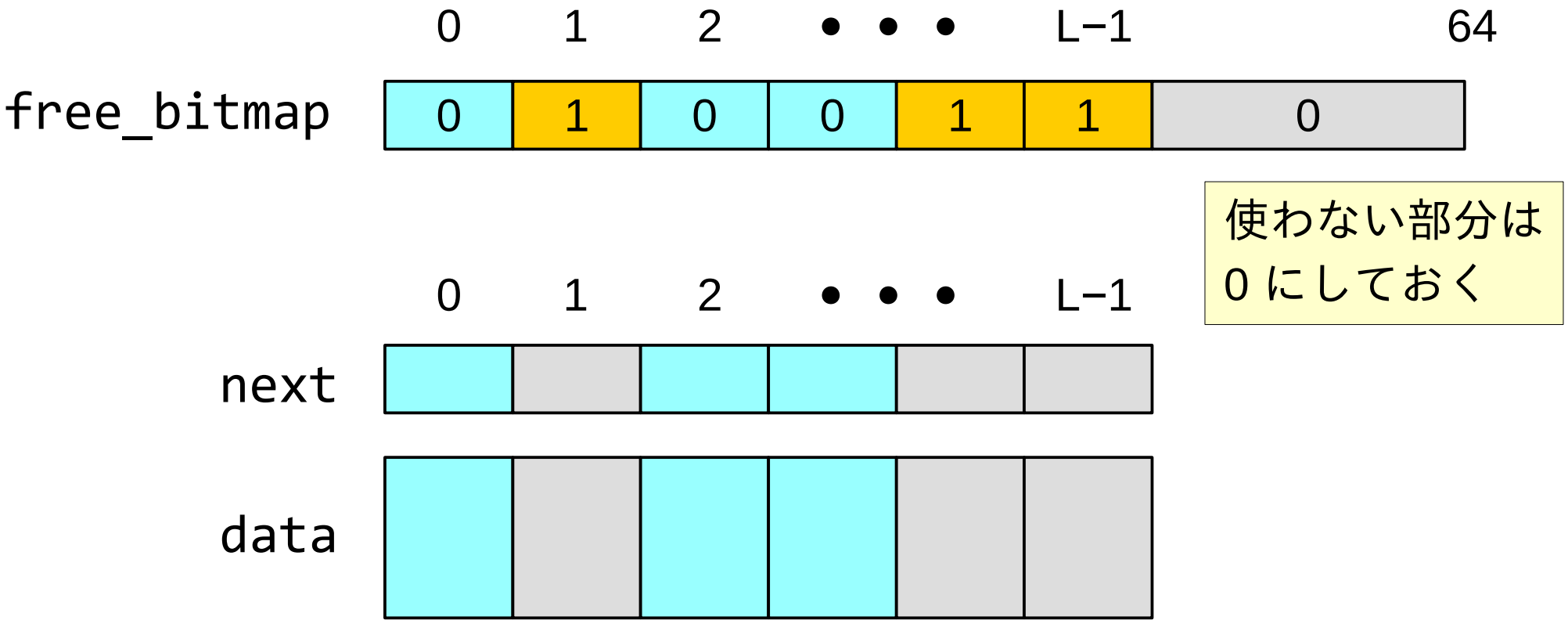
```
int pop(void) {
    int32_t p, q;
    int x;
    while (1) {
        p = head;
        if (p == -1) {
            sched_yield();
        } else {
            q = next[p];
            if (__sync_val_compare_and_swap(&head, p, q) == p)
                break;
        }
    }
    x = data[p];
    free_node(p);
    return x;
}
```

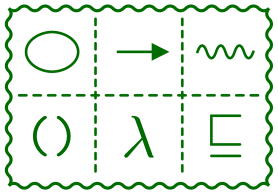
```
(define-process Q
  (? rd-head (p)
     (if (null? p)
         Q
         (? rd-next (pp q) (equal? pp p)
            (? cas-head (old new b)
               (and (equal? old p)
                    (equal? new q))
               (if (not b)
                   Q
                   (? rd (pp x) (equal? pp p)
                      (! free (p)
                         (! out (x) Q)))))))))
```



Bitmap Allocator

- インデックスに対応するビットの 0/1 で使用/未使用を表す



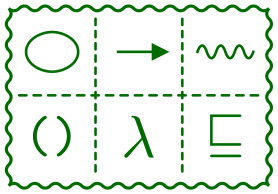


Bitmap Allocator: alloc_node

```
int32_t alloc_node(void)
{
    int32_t k;
    uint64_t x, y;
    while (1) {
        x = free_bitmap;
        if (x == 0) {
            sched_yield();
        } else {
            k = 63 - nlz(x);
            y = x & ~(1 << k);
            if (__sync_val_compare_and_swap(&free_bitmap, x, y) == x) {
                break;
            }
        }
    }
    return k;
}
```

ビットが 1 であるもっとも上位のビット番号を求める

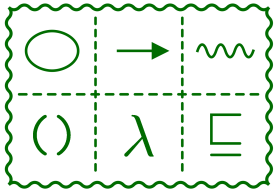
ビットを 0 （使用中）にする



Bitmap Allocator: free_node

```
void free_node(int32_t k)
{
    uint64_t x, y;
    do {
        x = free_bitmap;
        y = x | (1 << k);
    } while (__sync_val_compare_and_swap(&free_bitmap, x, y) != x);
}
```

ビットを１（未使用中）にする

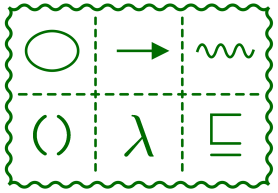


関数 nlz

x の上位（MSB）から連続する 0 のビット数を求める

```
int nlz(uint64_t x) {
    int n;
    if (x == 0)
      return 64;
    n = 0;
    if (x <= 0x00000000FFFFFFFFULL) { n += 32; x <<= 32; }
    if (x <= 0x0000FFFFFFFFFFFFULL) { n += 16; x <<= 16; }
    if (x <= 0x00FFFFFFFFFFFFFFULL) { n += 8; x <<= 8; }
    if (x <= 0x0FFFFFFFFFFFFFFFULL) { n += 4; x <<= 4; }
    if (x <= 0x3FFFFFFFFFFFFFFFULL) { n += 2; x <<= 2; }
    if (x <= 0x7FFFFFFFFFFFFFFFULL) { n += 1; }
    return n;
}
```

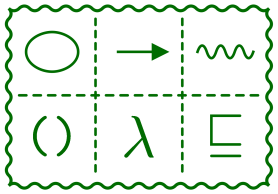
参考
ハッカーのたのしみ一本物のプログラマはいかにして問題を解くか
ジュニア,ヘンリー・S. ウォーレン



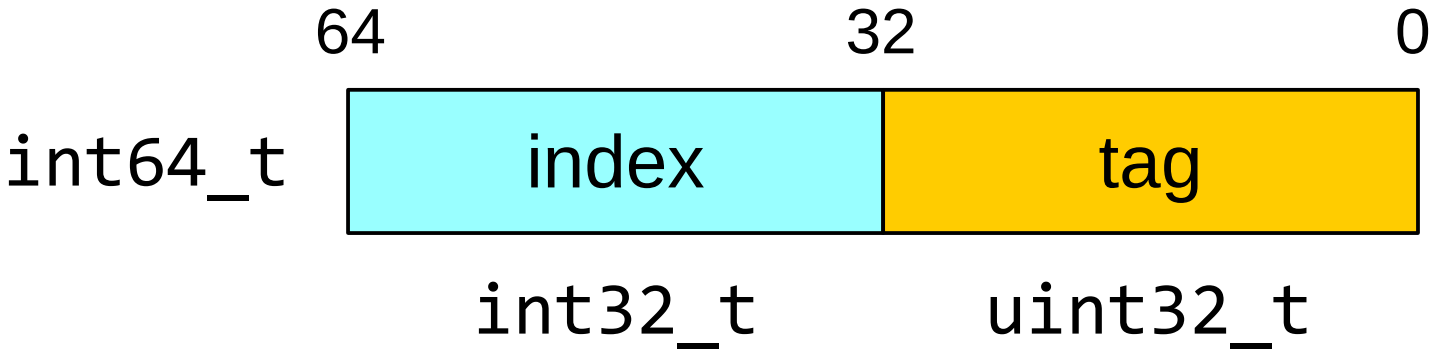
演習

stack-lockfree.c

- プログラムを実行し ABA 問題が発生することを確認してください
  - 必要に応じて問題検出の機構や発生の可能性が高まる修正を加えてください
- タグ付きポインタを実装し，ABA 問題に対する対策を行ってください
- ミューテックスと条件変数を使ったスタックを実装し，パフォーマンスを比較してください



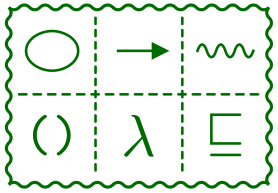
タグ付きポインタの表現




```
int64_t MAKEABAWORD(int32_t p, uint32_t tag) {
    int64_t x = (int64_t)p;
    return (x << 32) | tag;
}
```

```
int32_t IDX(int64_t x) { return x >> 32; }
```

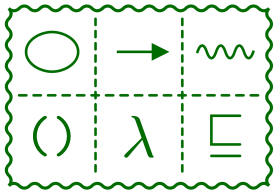
```
uint32_t TAG(int64_t x) { return x; }
```



関数 push

```
void push(int d) {
    uint64_t x, y;
    int32_t p, q;
    p = alloc_node();
    data[p] = d;
    do {
        x = head;
        q = IDX(x);
        y = MAKEABAWORD(p, TAG(x) + 1);
        next[p] = q;
    } while (__sync_val_compare_and_swap(&head, x, y) != x);
}
```

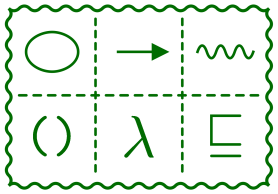
タグ更新



関数 pop

```
int pop(void) {
    uint64_t x, y;
    int32_t p, q;
    int d;
    while (1) {
        x = head;
        p = IDX(x);
        if (p == -1) {
            sched_yield();
        } else {
            q = next[p];
            y = MAKEABAWORD(q, TAG(x));
            if (__sync_val_compare_and_swap(&head, x, y) == x)
                break;
        }
    }
    d = data[p];
    free_node(p);
    return d;
}
```

タグをコピー



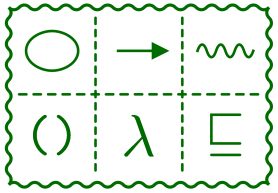
パフォーマンスの比較

ミューテックスと条件変数によるスタック

	sec
全実行時間	2.618
同期機構呼び出し	2.471
クリティカルセクション	0.018
その他	0.129

CAS によるロックフリースタック

	sec
全実行時間	0.107



まとめ

- 不可分操作を使うと Lock-Free, Wait-Free なアルゴリズムを実装することができ，同期機構を使う場合と比較してパフォーマンスを大きく改善することができる
- 不可分操作を使ったアルゴリズムの設計は難しい
  - ABA 問題のように発見/再現の難しい問題が存在する
- 不可分操作による実装においてもモデル検査は有効である
  - 問題を確実に発見し，原因を分析できる