Back to Basics: Lock-free

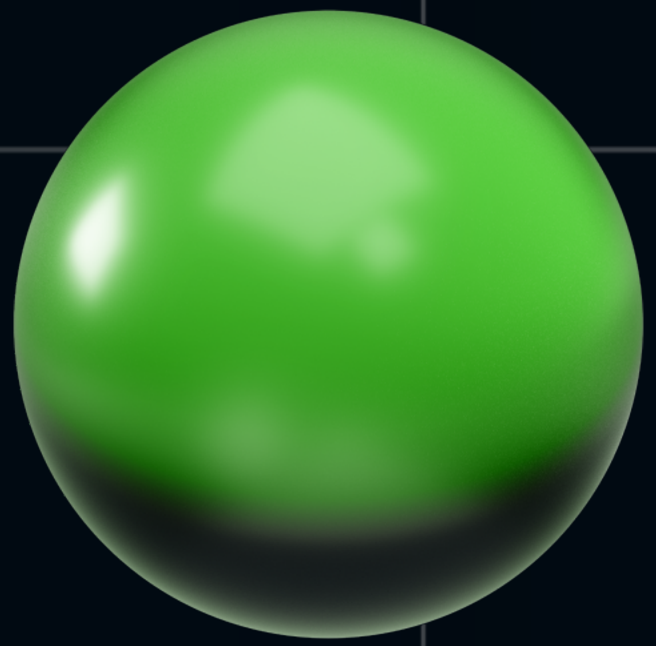
Марсель Галимуллин

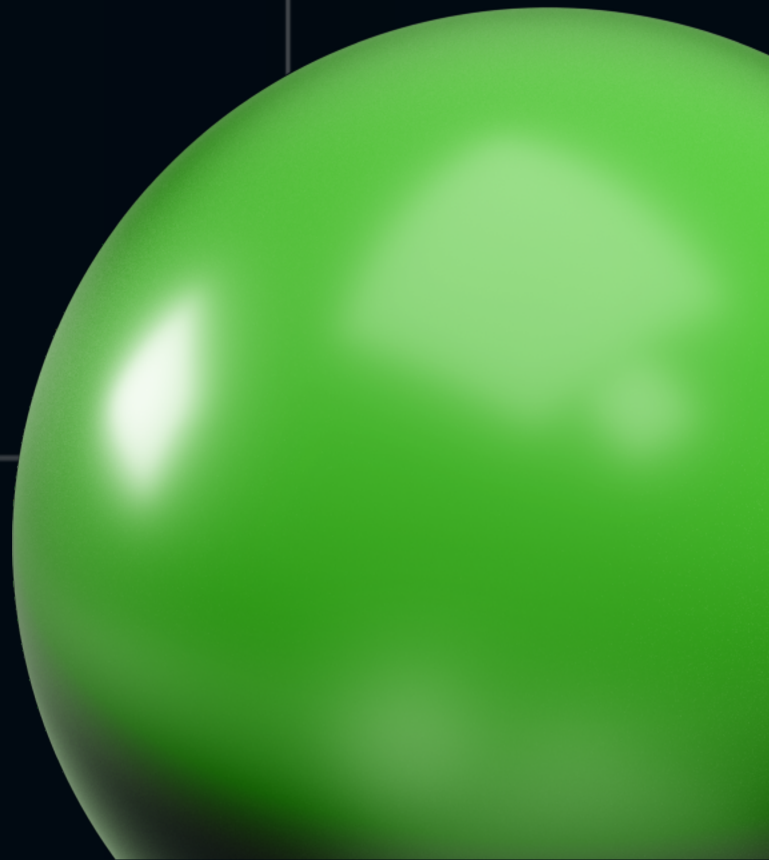
Яндекс Лавка

C++ Russia
2023

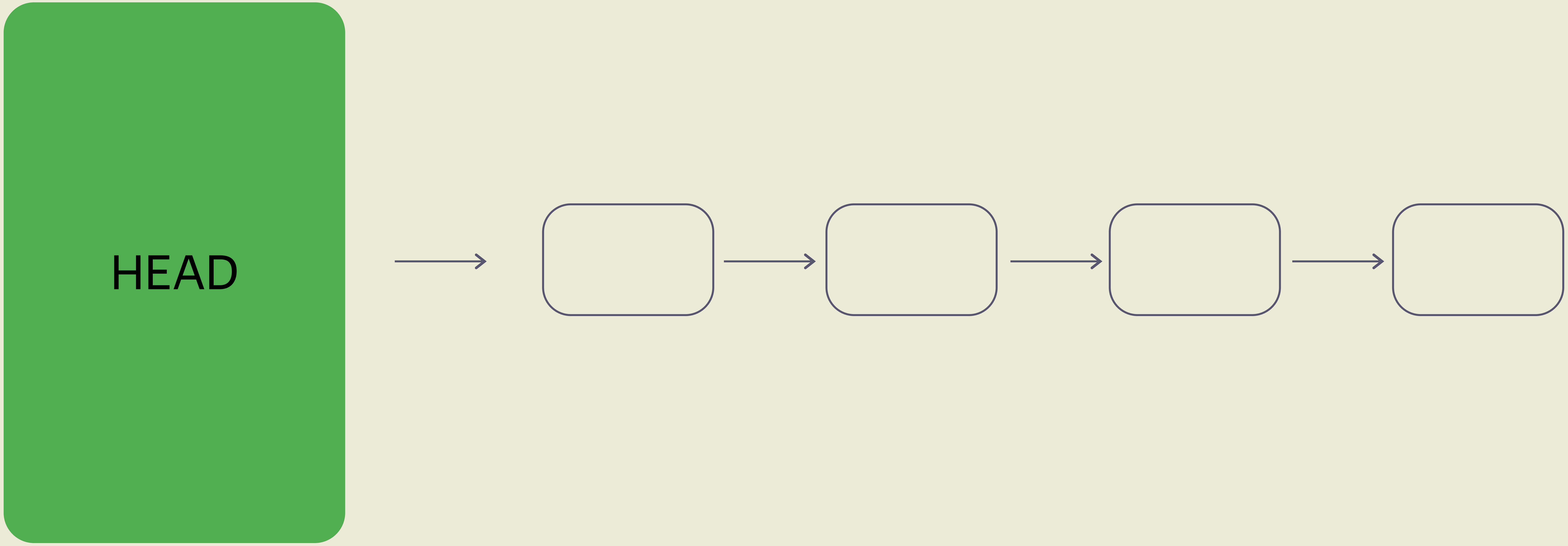
Содержание

- Список
- Гонка данных
- Мьютексы
- CAS-операции
- Hazard pointer
- Снимки атомарных регистров



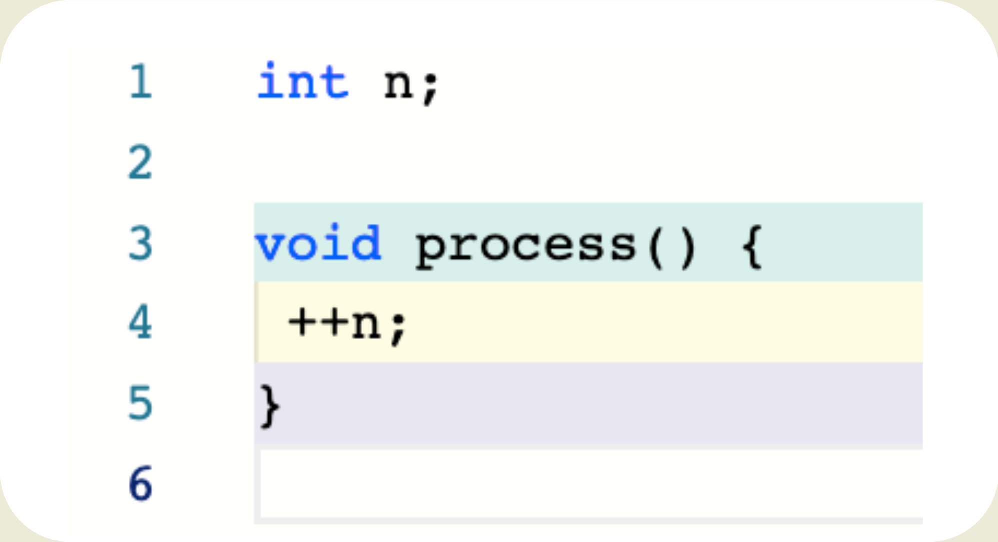


Список

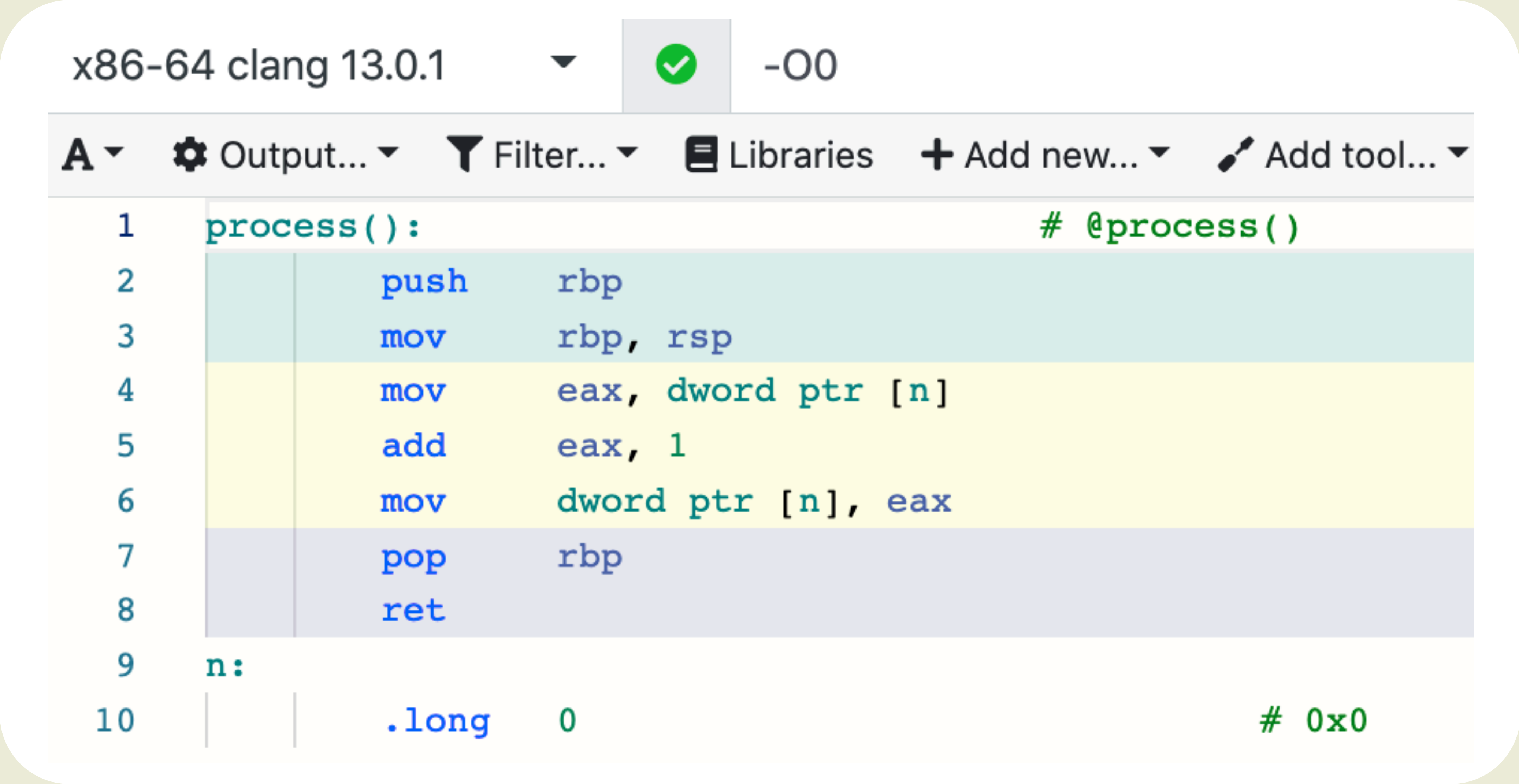


Гонка данных

Время	Thread 1	Thread 2
1	Read n (0)	
2		Read n (0)
3	++n (1)	++n (1)
4	Write n	Write n



```cpp
int n;

void process() {
 ++n;
}

```



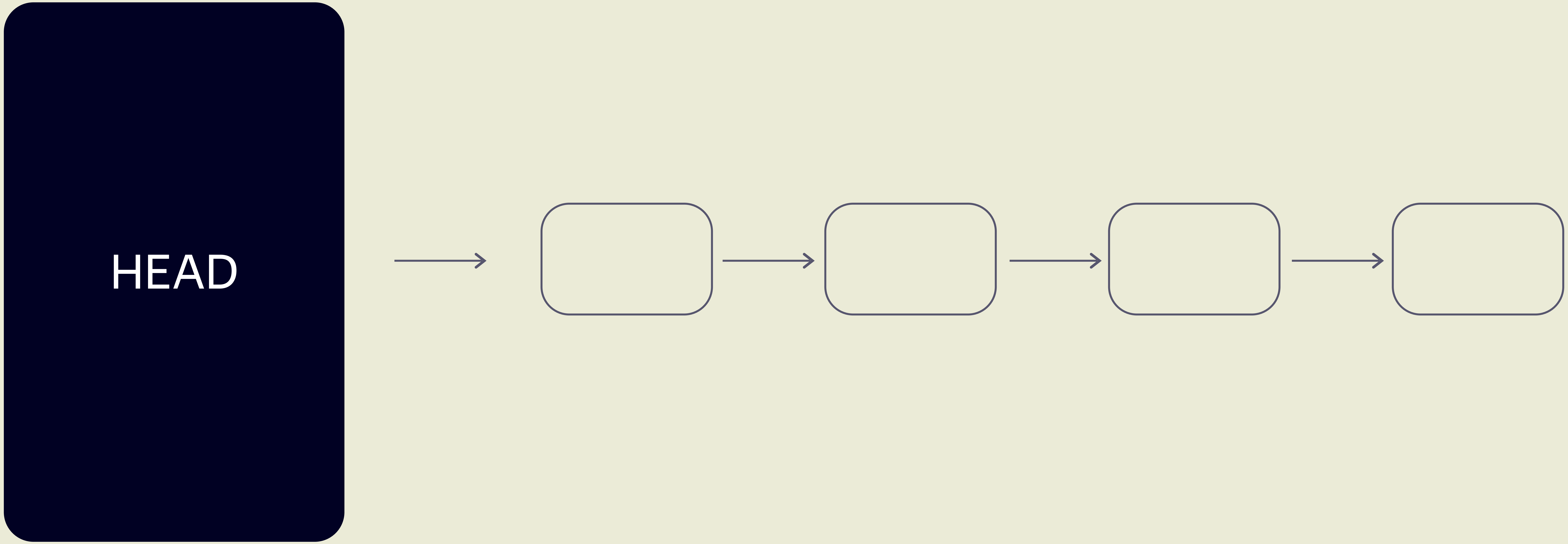
x86-64 clang 13.0.1 -O0

```asm
process():                              # @process()
        push    rbp
        mov     rbp, rsp
        mov     eax, dword ptr [n]
        add     eax, 1
        mov     dword ptr [n], eax
        pop     rbp
        ret
n:
        .long   0                               # 0x0
```

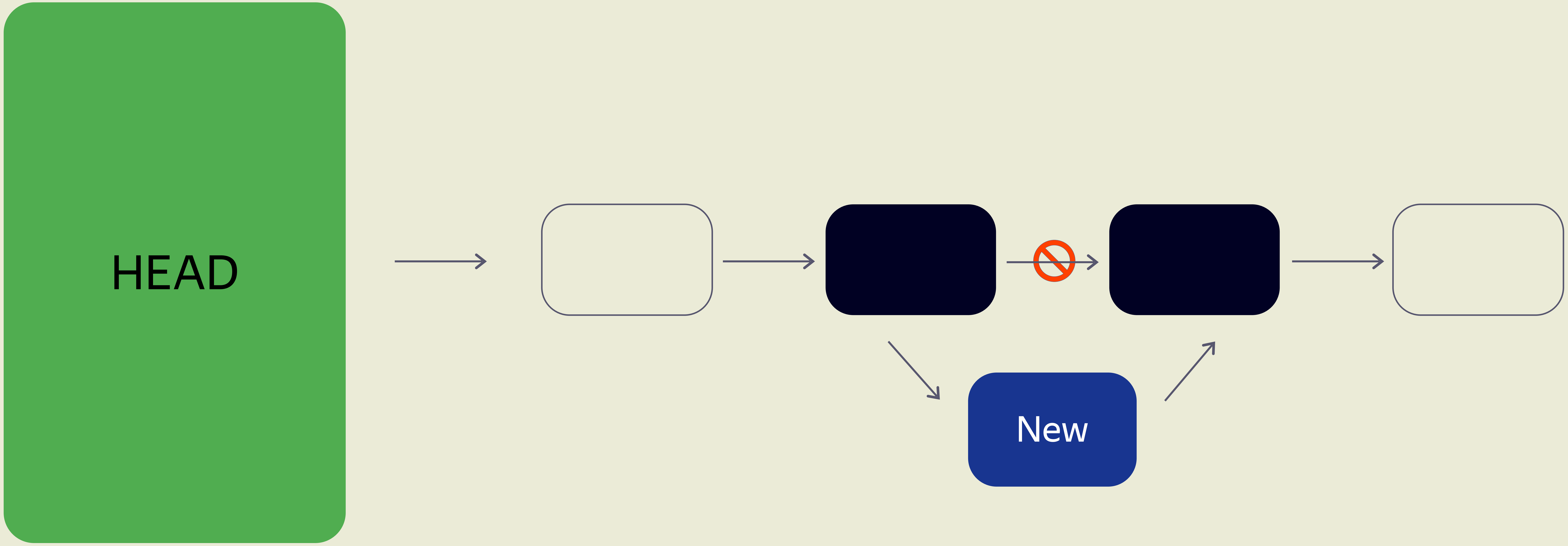
https://godbolt.org/z/5xz8hjj5E

Мьютексы

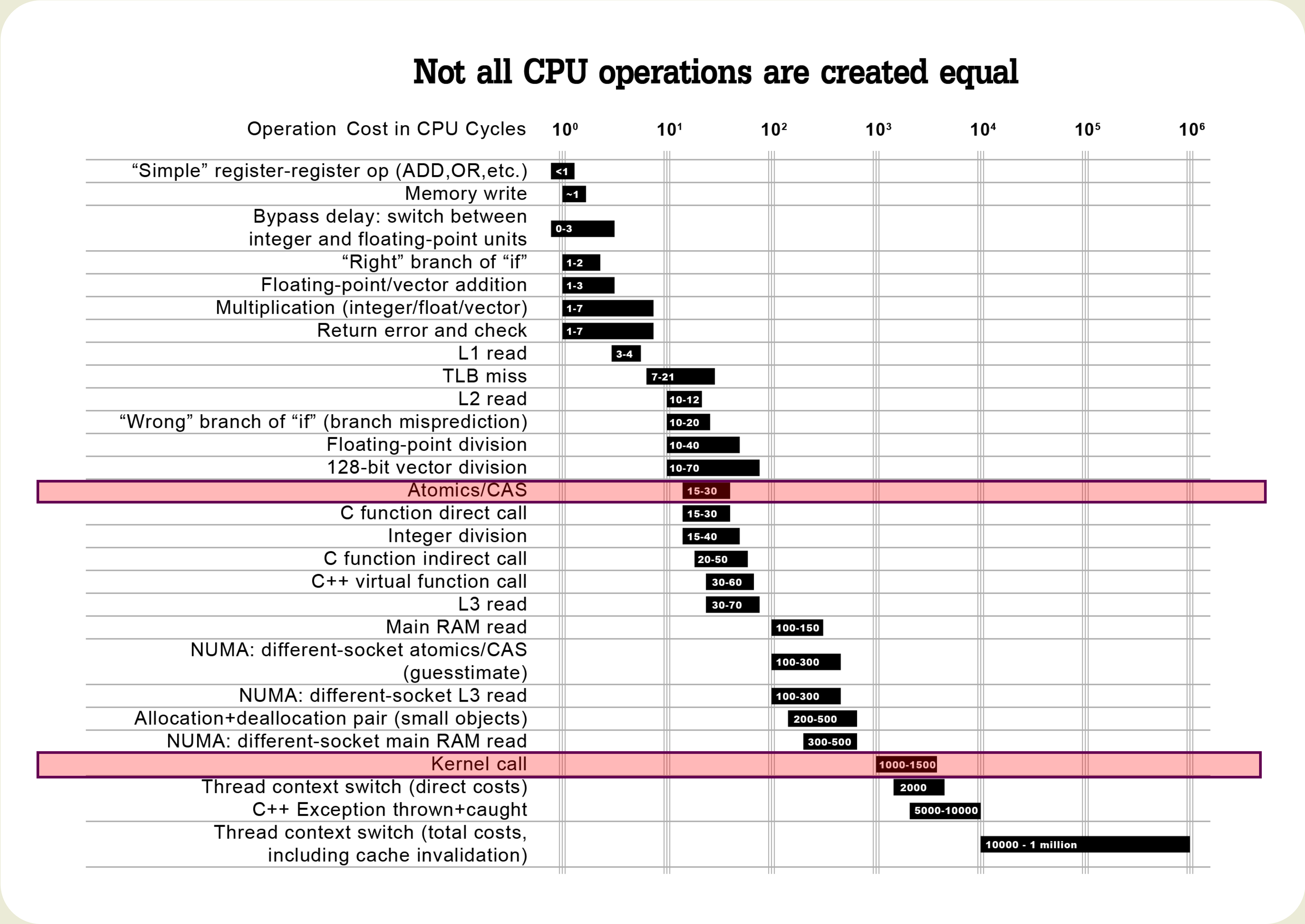
Общая блокировка



Гранулярная блокировка



CPU операции



Not all CPU operations are created equal

Operation	Cost in CPU Cycles
“Simple” register-register op (ADD,OR,etc.)	<1
Memory write	~1
Bypass delay: switch between integer and floating-point units	0-3
“Right” branch of “if”	1-2
Floating-point/vector addition	1-3
Multiplication (integer/float/vector)	1-7
Return error and check	1-7
L1 read	3-4
TLB miss	7-21
L2 read	10-12
“Wrong” branch of “if” (branch misprediction)	10-20
Floating-point division	10-40
128-bit vector division	10-70
Atomics/CAS	15-30
C function direct call	15-30
Integer division	15-40
C function indirect call	20-50
C++ virtual function call	30-60
L3 read	30-70
Main RAM read	100-150
NUMA: different-socket atomics/CAS (guesstimate)	100-300
NUMA: different-socket L3 read	100-300
Allocation+deallocation pair (small objects)	200-500
NUMA: different-socket main RAM read	300-500
Kernel call	1000-1500
Thread context switch (direct costs)	2000
C++ Exception thrown+caught	5000-10000
Thread context switch (total costs, including cache invalidation)	10000 - 1 million

http://ithare.com/infographics-operation-costs-in-cpu-clock-cycles/

CAS-операции

CAS

```
bool atomic_compare_exchange_weak(atomic<T>* obj, T* expected, T desired);
```

CAS

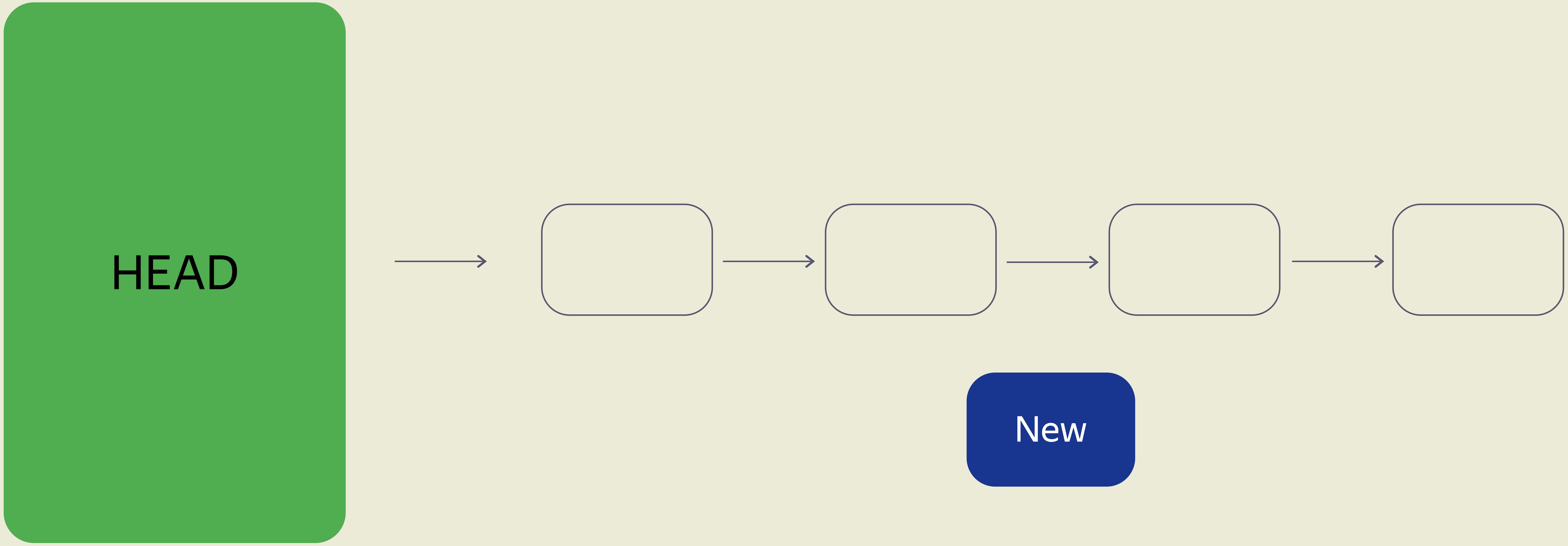
```
bool atomic_compare_exchange_weak(atomic<T>* obj, T* expected, T desired);

template <class T>
struct atomic {
    void store(T data);
    T load() const;
    T exchange(T data);
    bool compare_exchange_weak(T& expected, T desired);
};
```

CAS

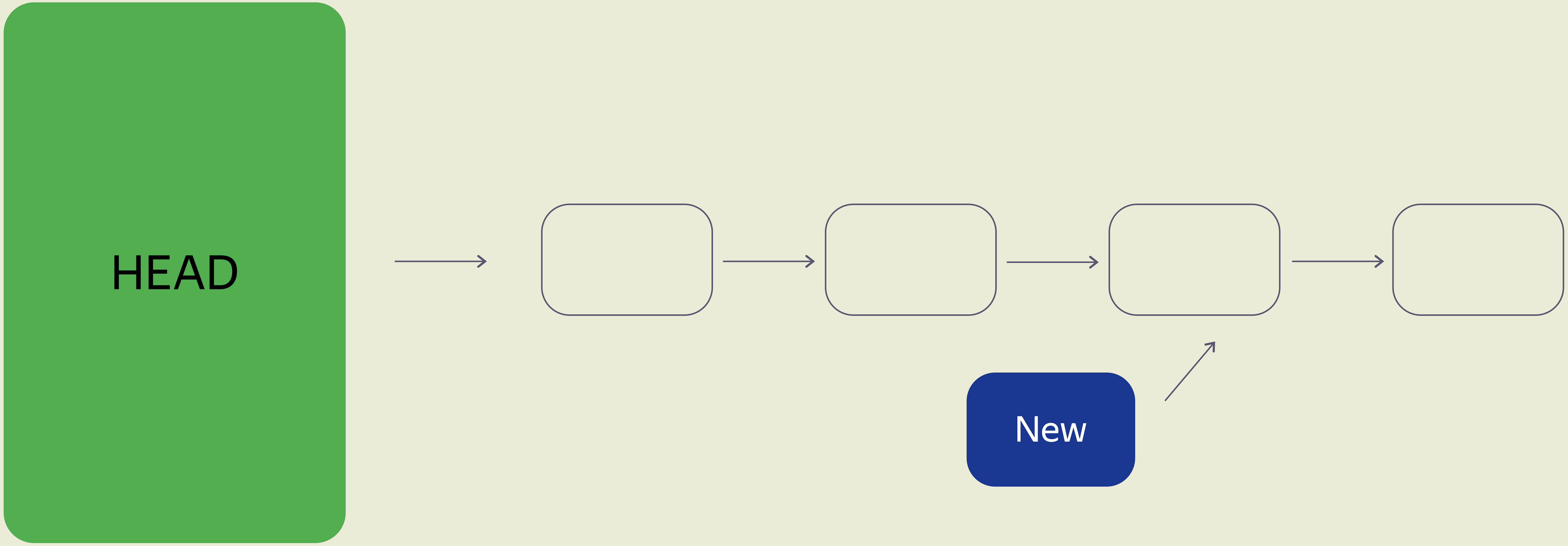
```
bool atomic_compare_exchange_weak(atomic<T>* obj, T* expected, T desired);
```

```
template<class T>
class stack{
   std::atomic<node<T>*> head;
  public:
   void push(const T& data){
     node<T>* new_node = new node<T>(data);
     new_node->next = head.load();
     while(!std::atomic_compare_exchange_weak(&head, &new_node->next, new_node));
   }
 };
```

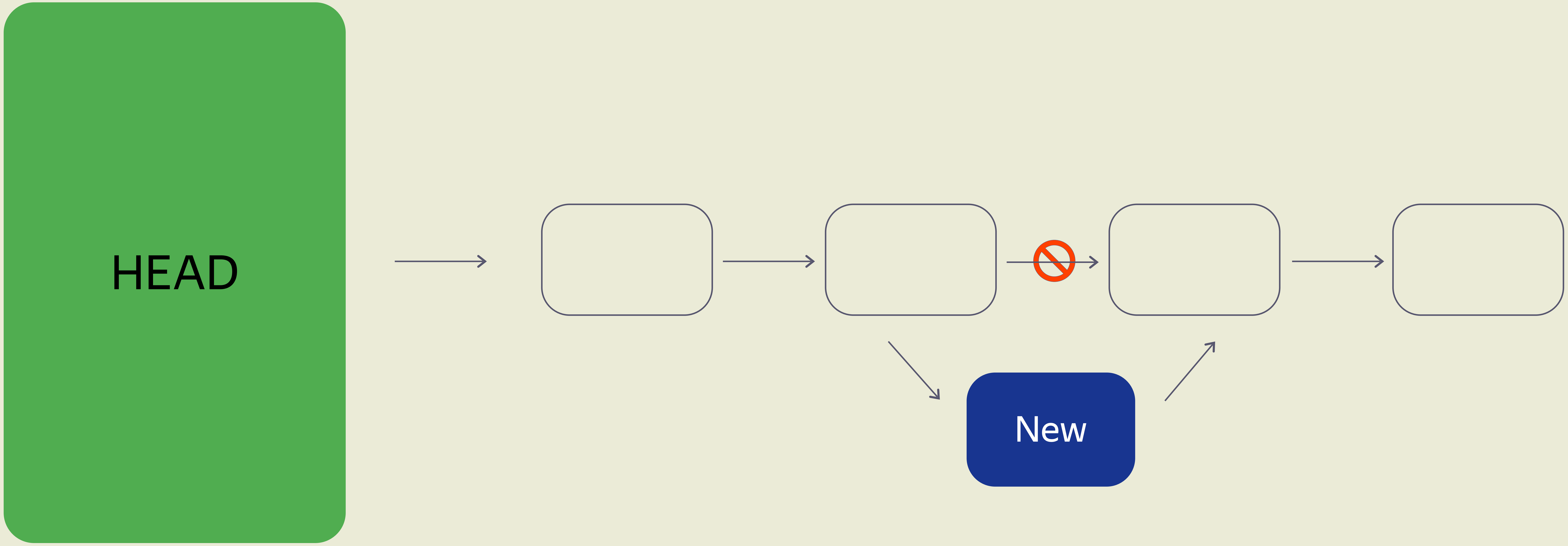
CAS



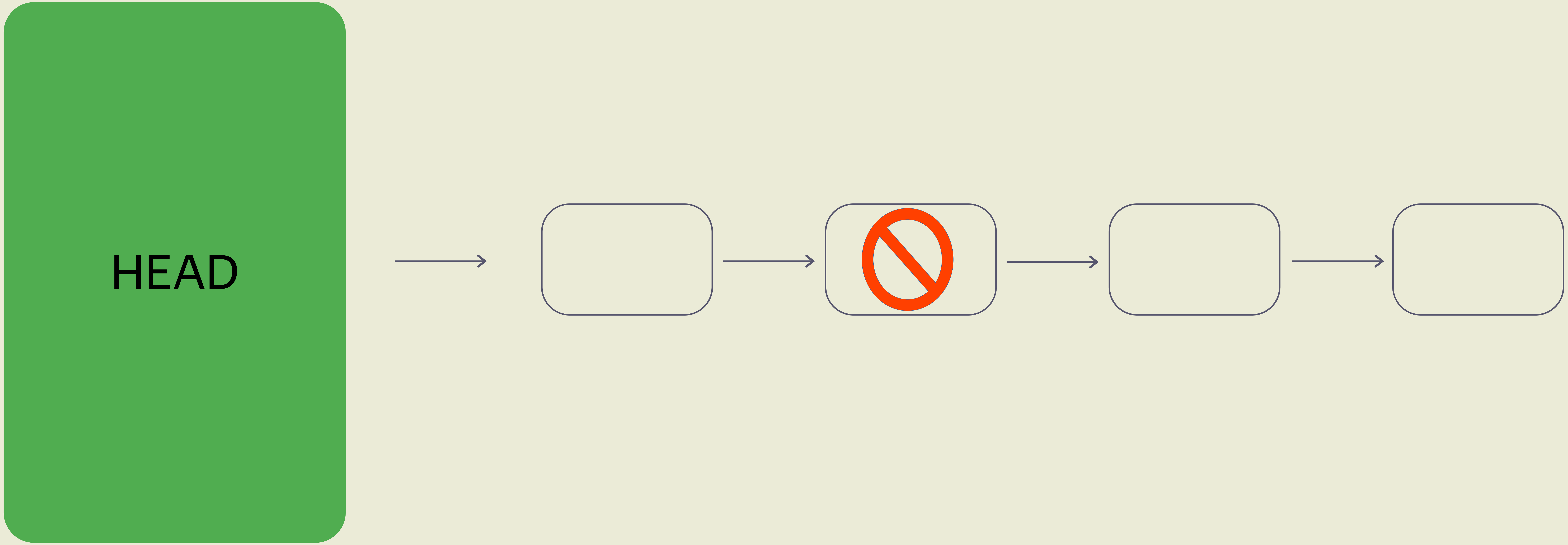
CAS



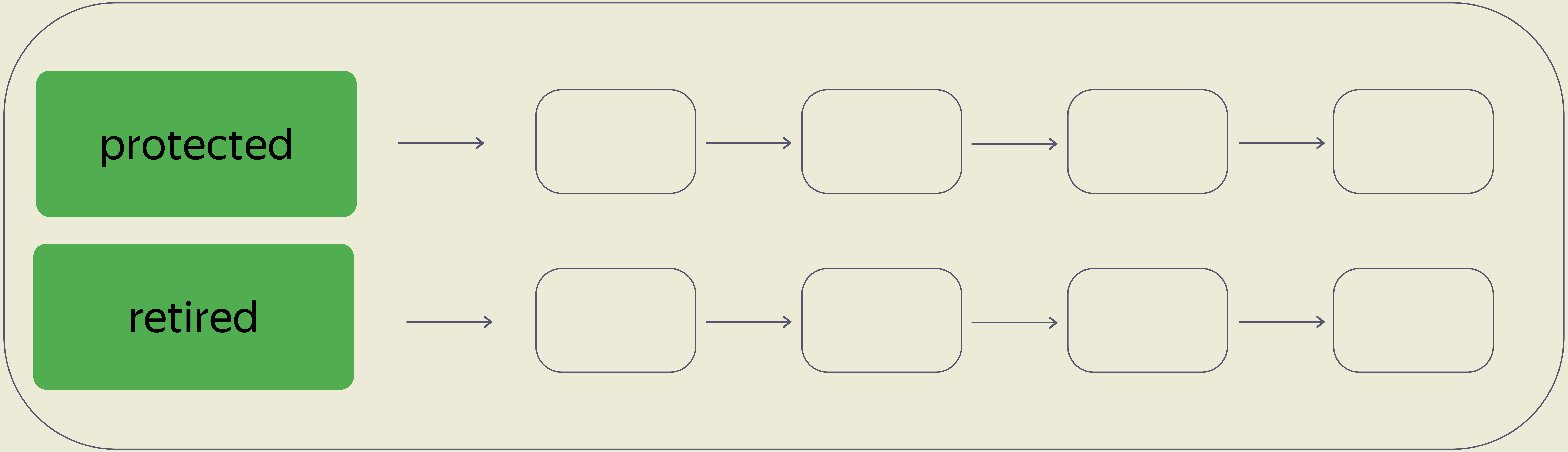
CAS

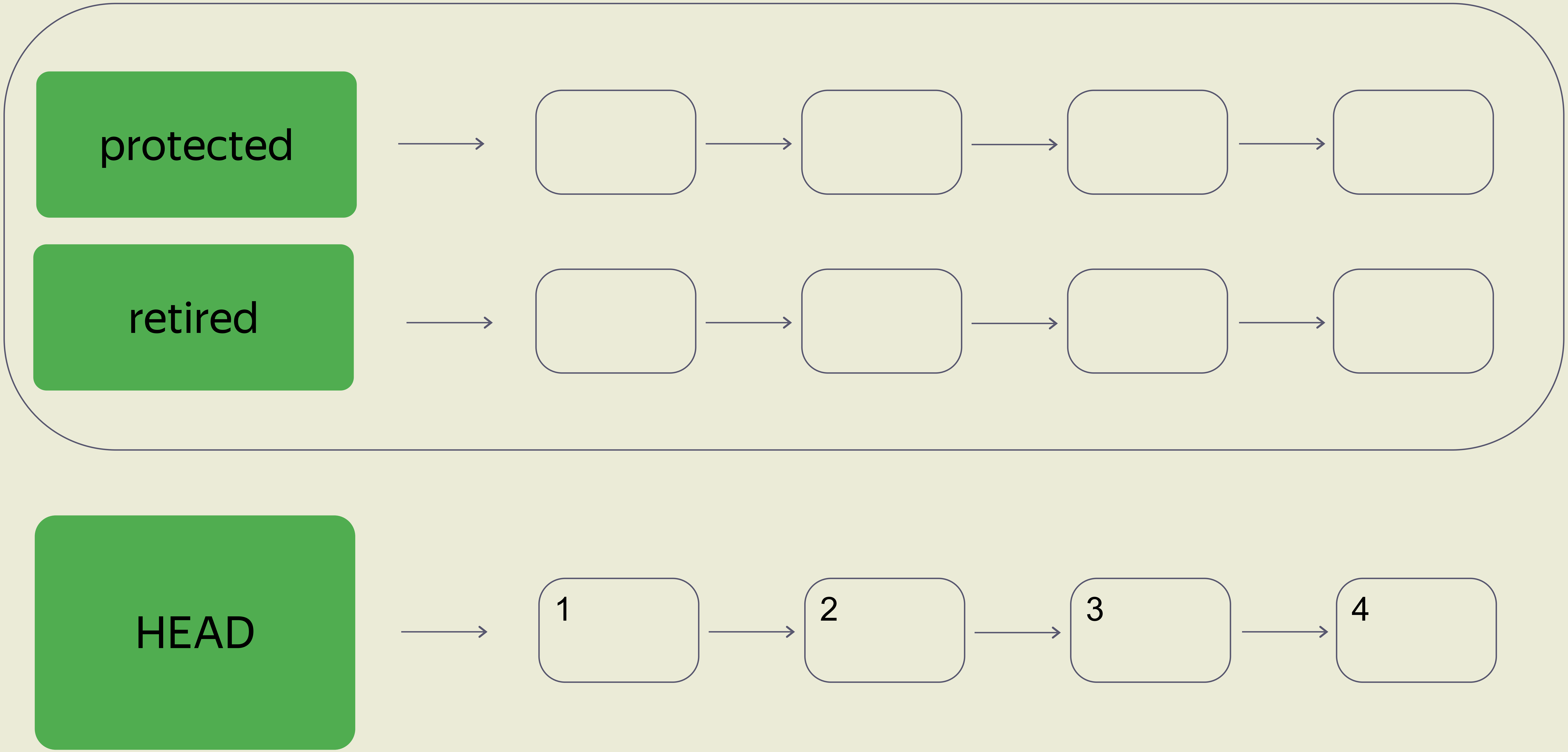


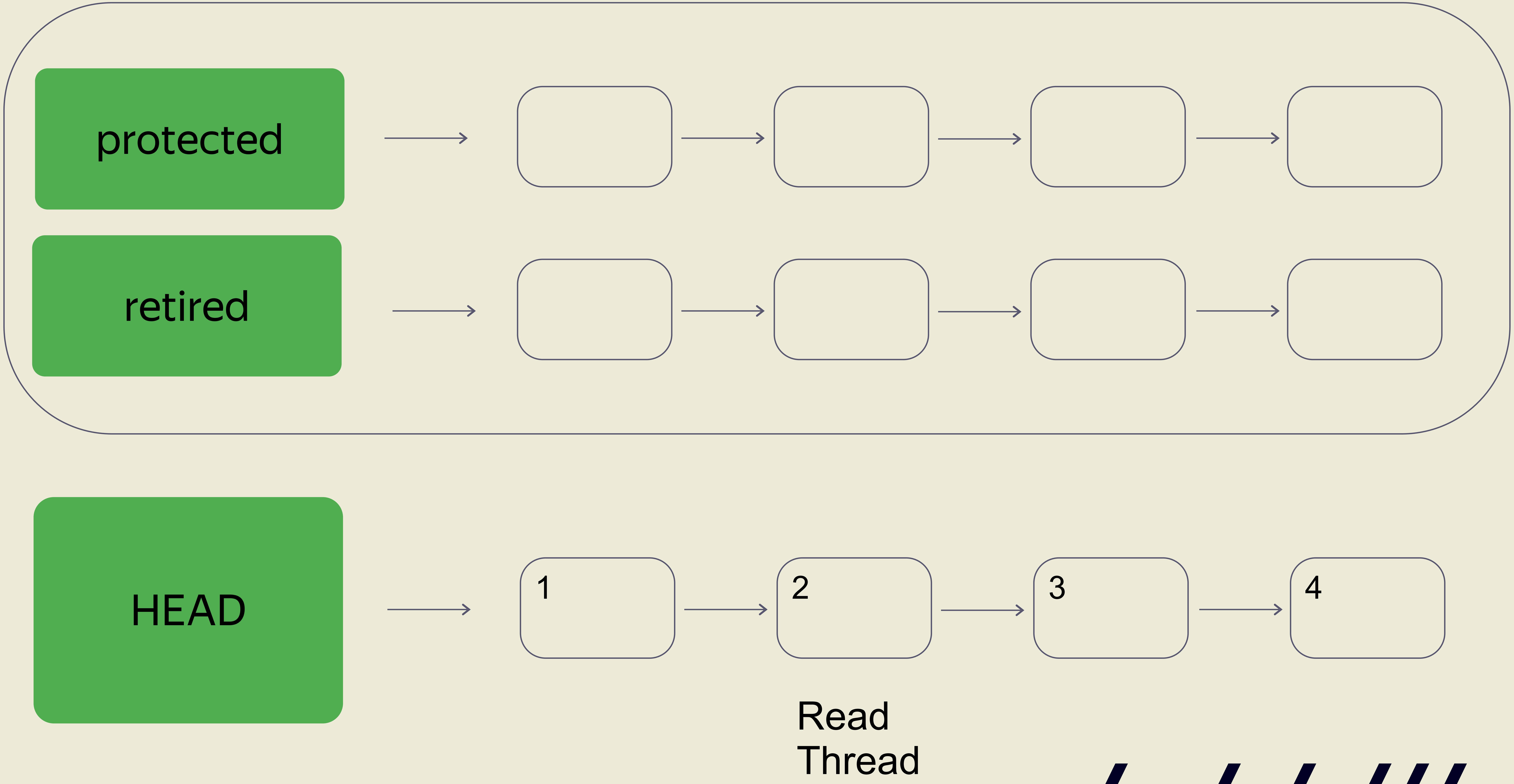
CAS как удалять?



Hazard pointer


protected
retired


protected
retired
HEAD
1
2
3
4



protected

retired

HEAD

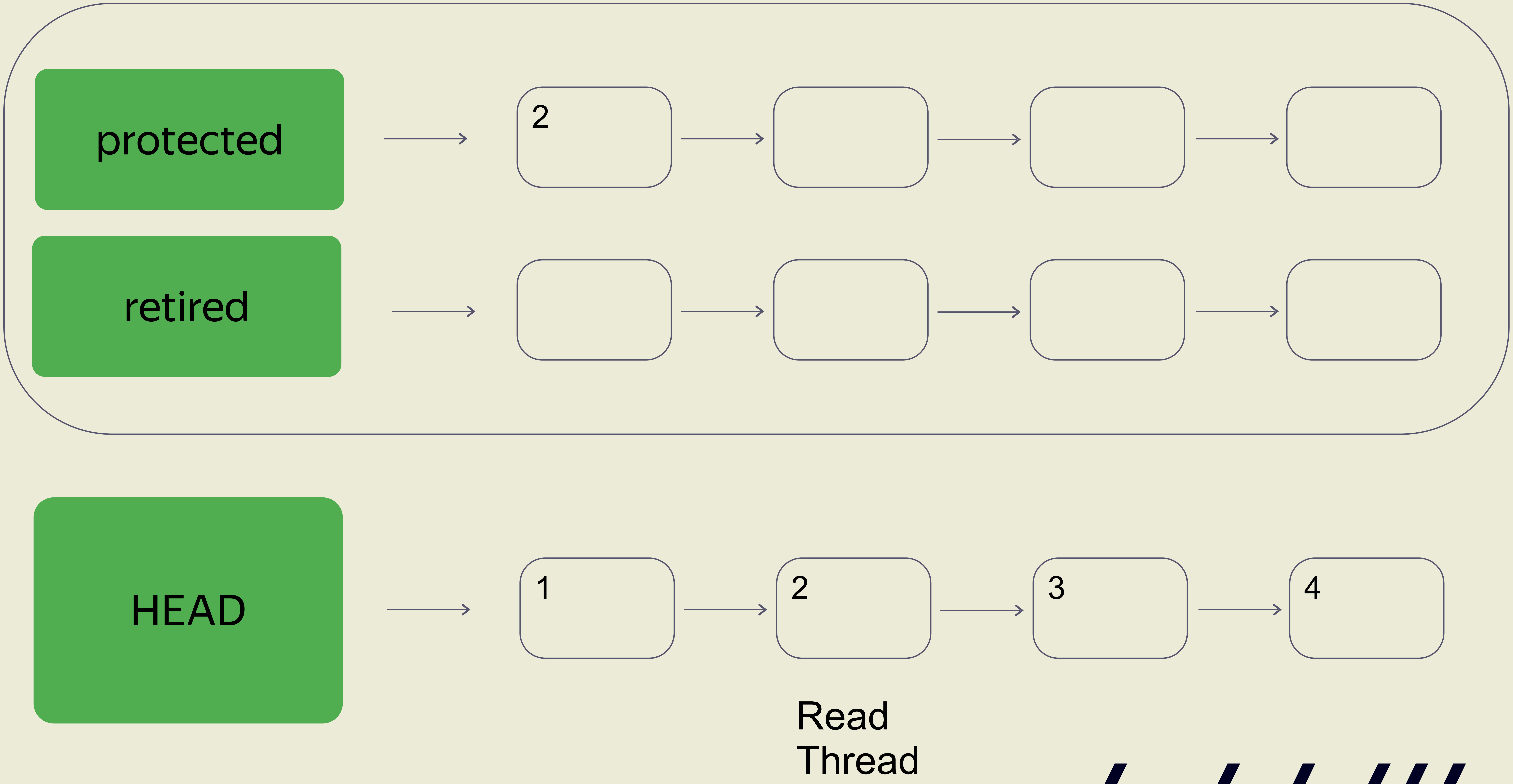
1

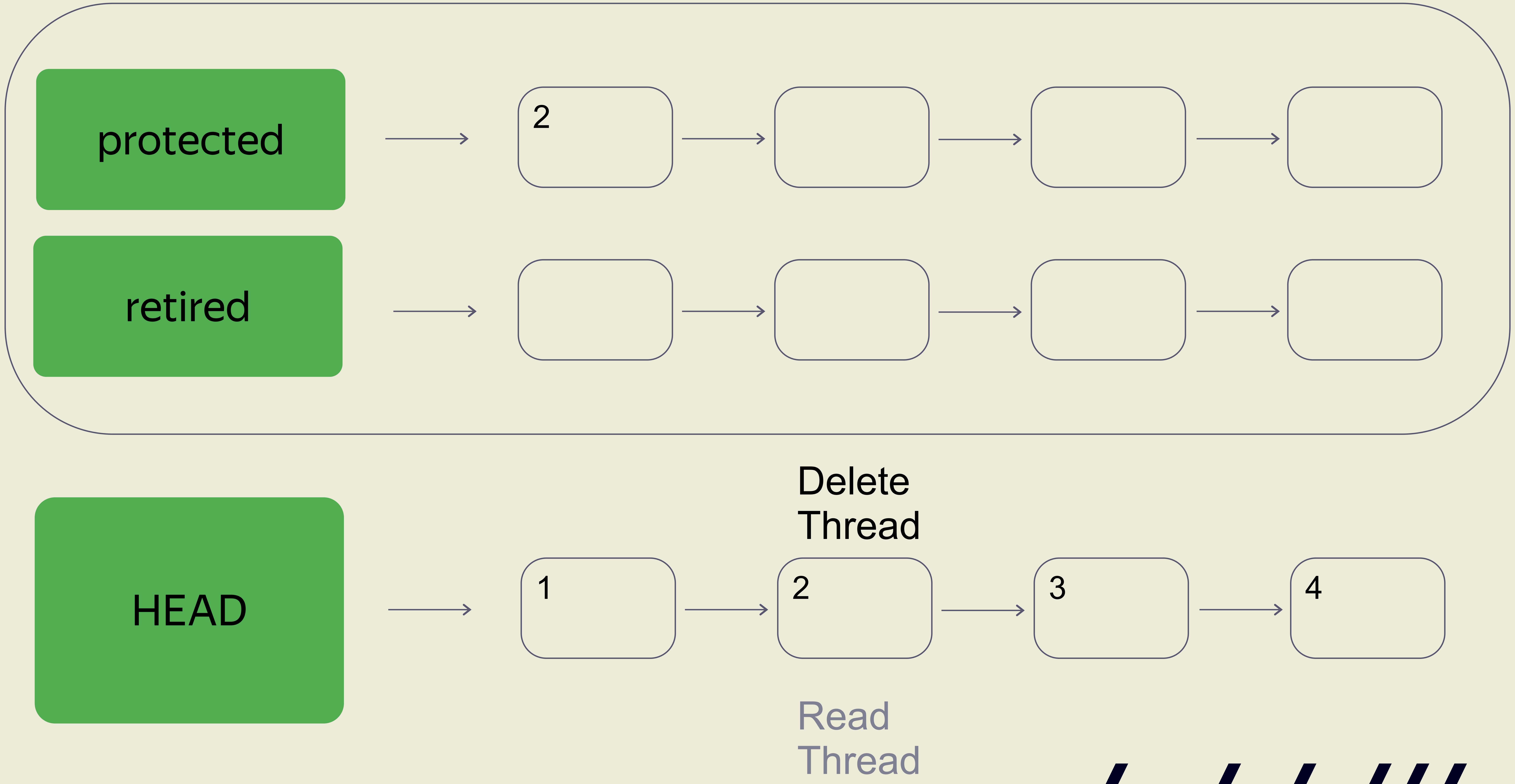
2

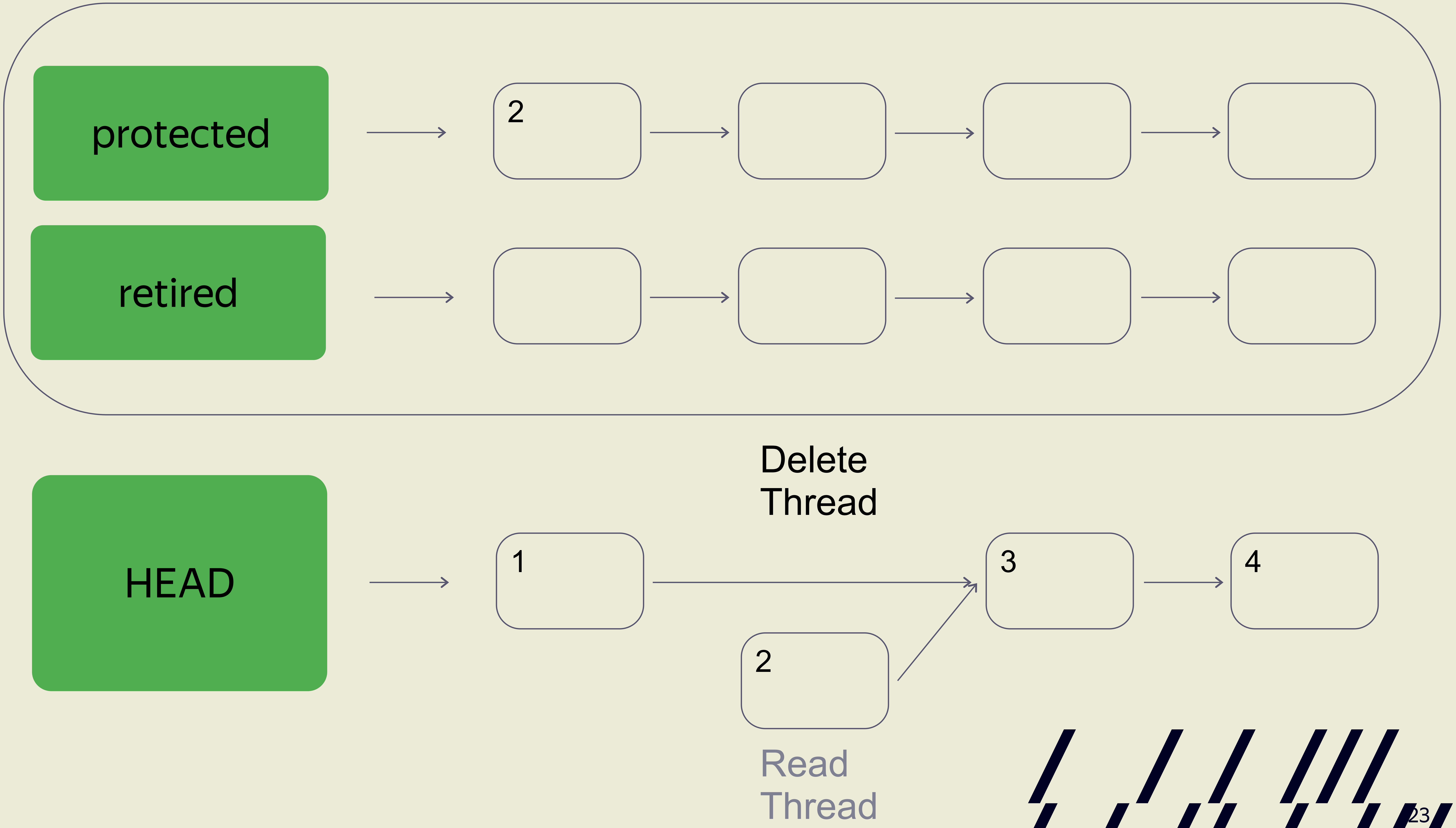
3

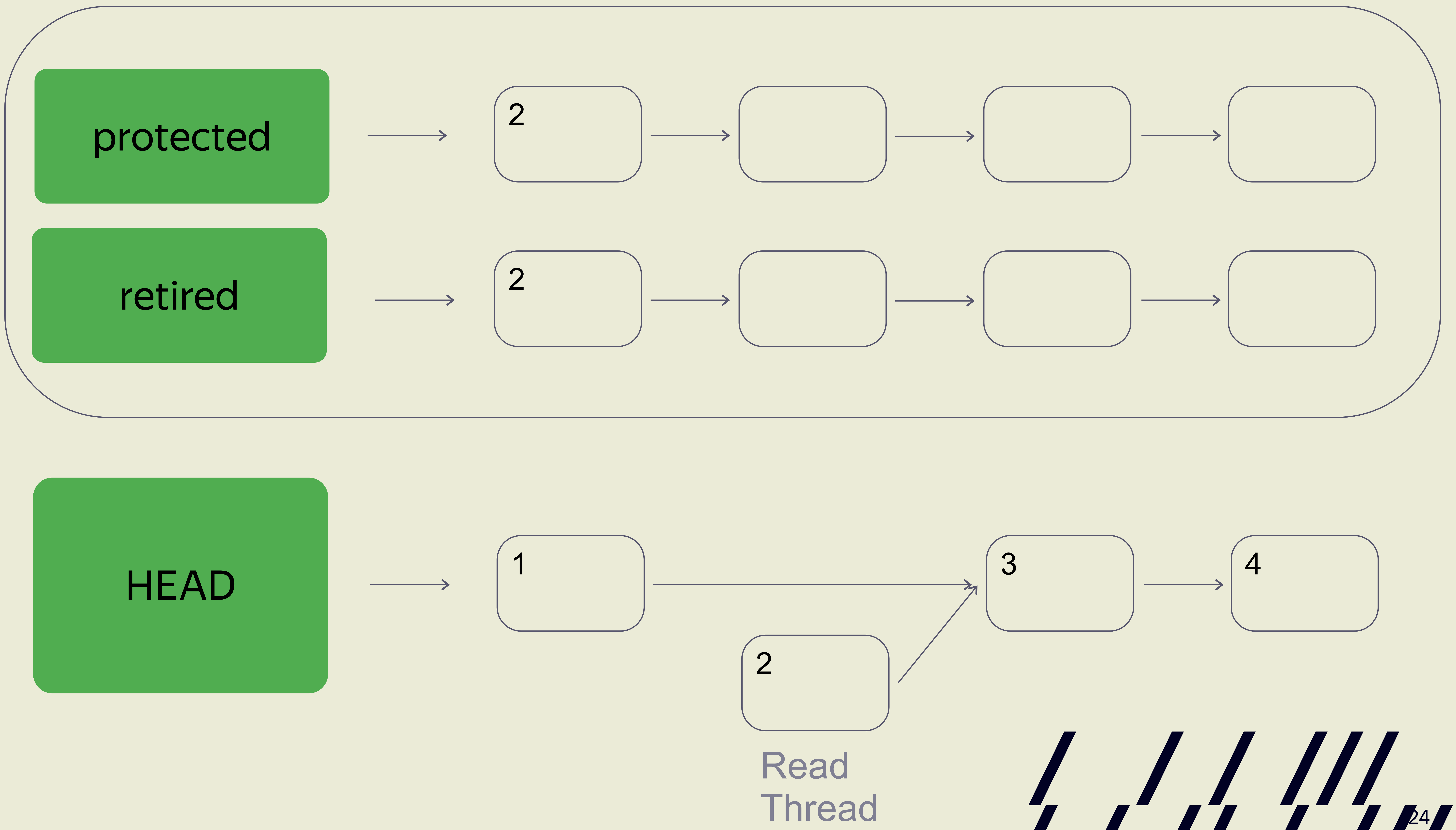
4

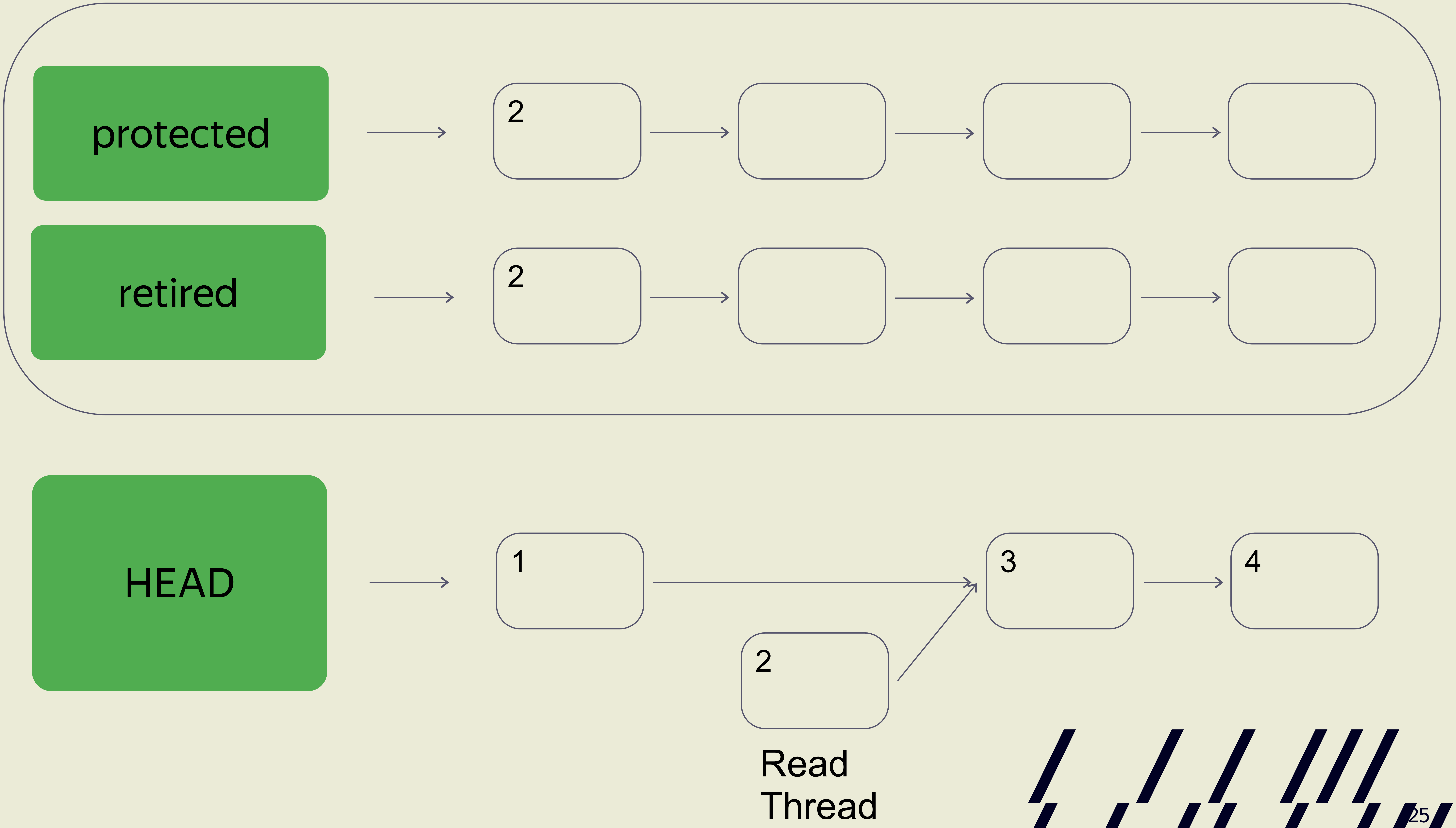
Read
Thread

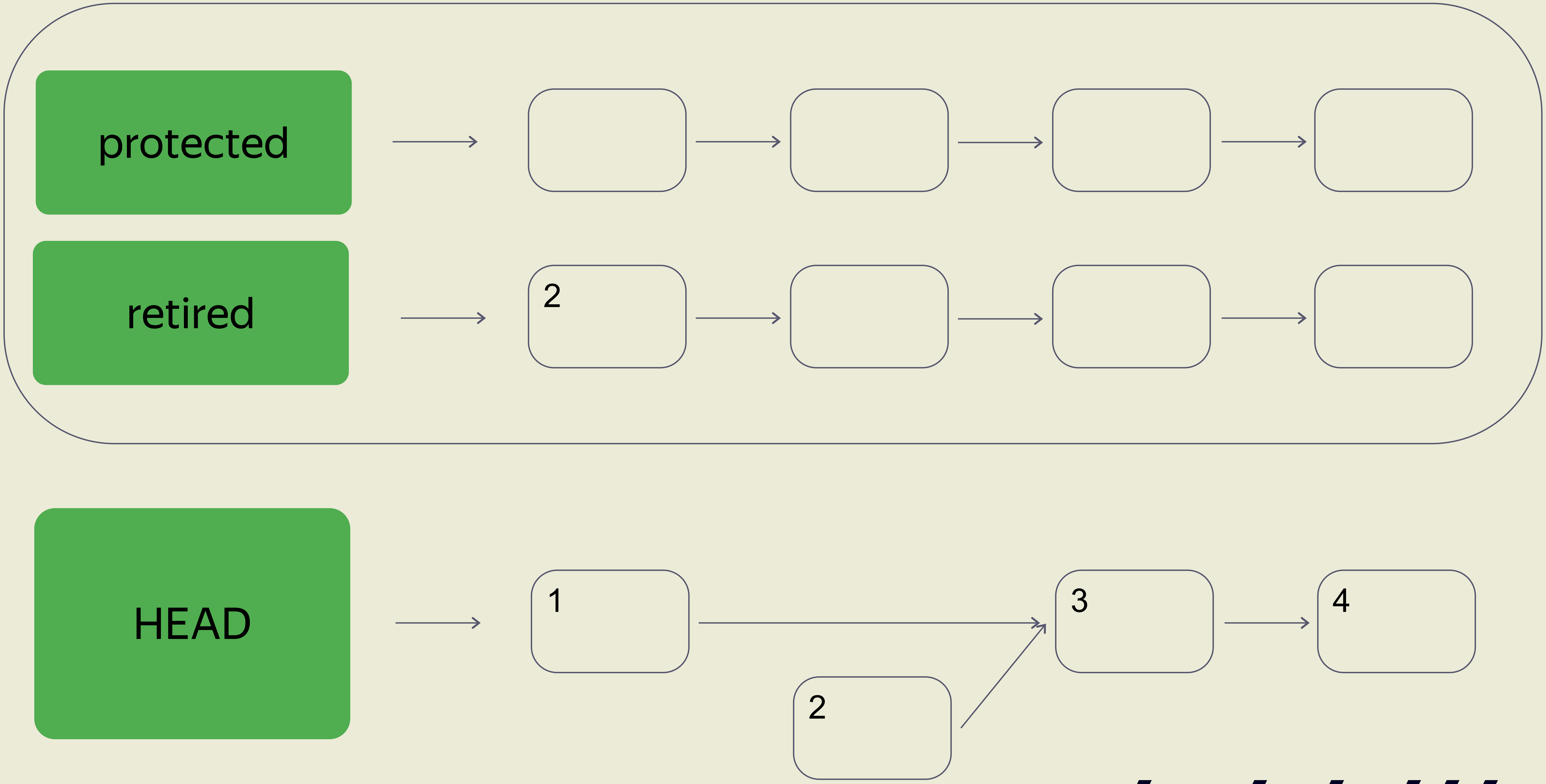

protected
2
retired
HEAD
1
2
3
4
Read Thread


protected
2
retired
Delete Thread
HEAD
1
2
3
4
Read Thread


protected
2
retired
Delete
Thread
HEAD
1
3
4
2
Read
Thread


protected
2
retired
2
HEAD
1
3
4
2
Read Thread


protected
2
retired
2
HEAD
1
3
4
2
Read Thread



protected

retired

2

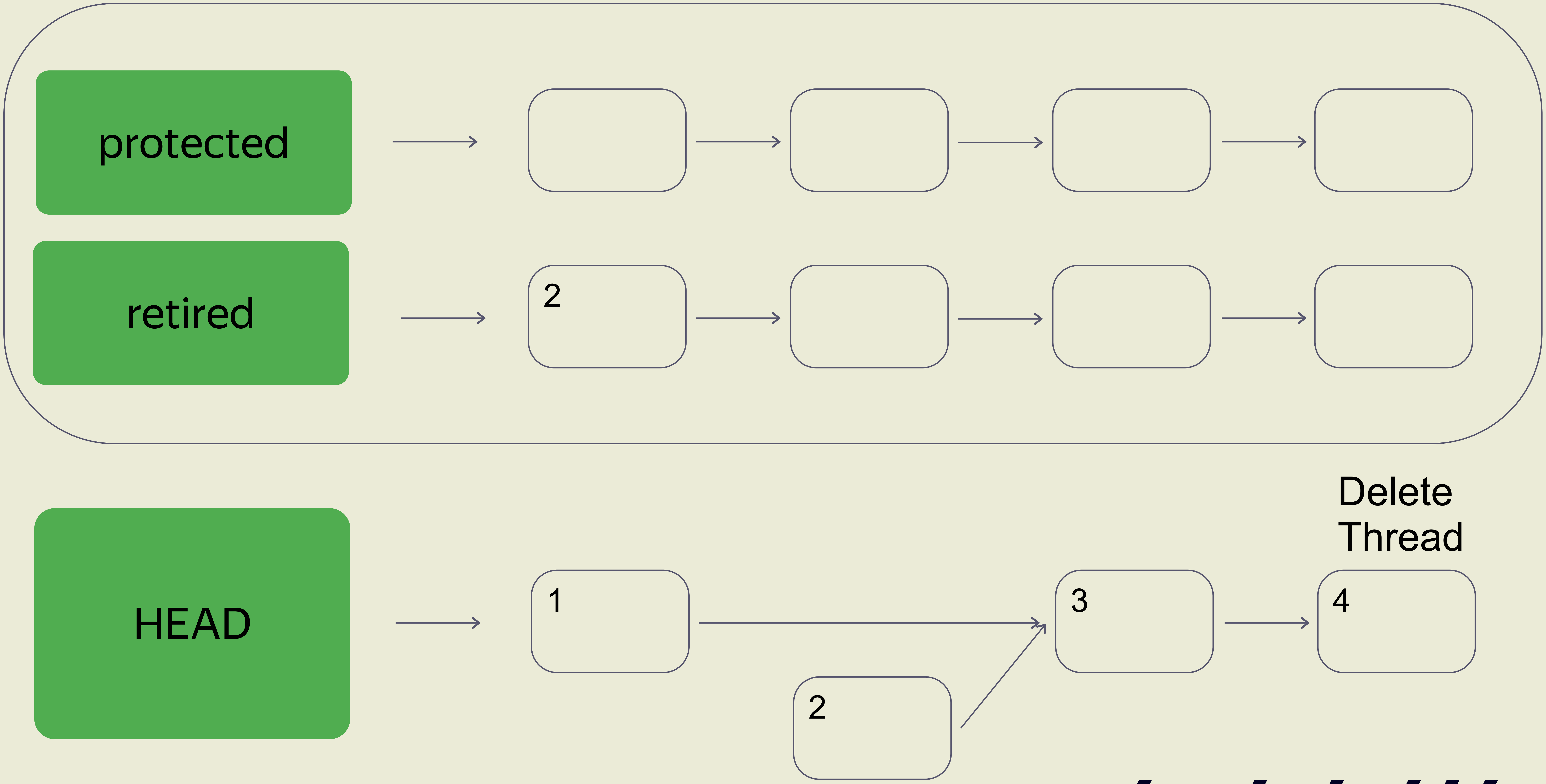
HEAD

1

3

4

2



protected

retired

2

HEAD

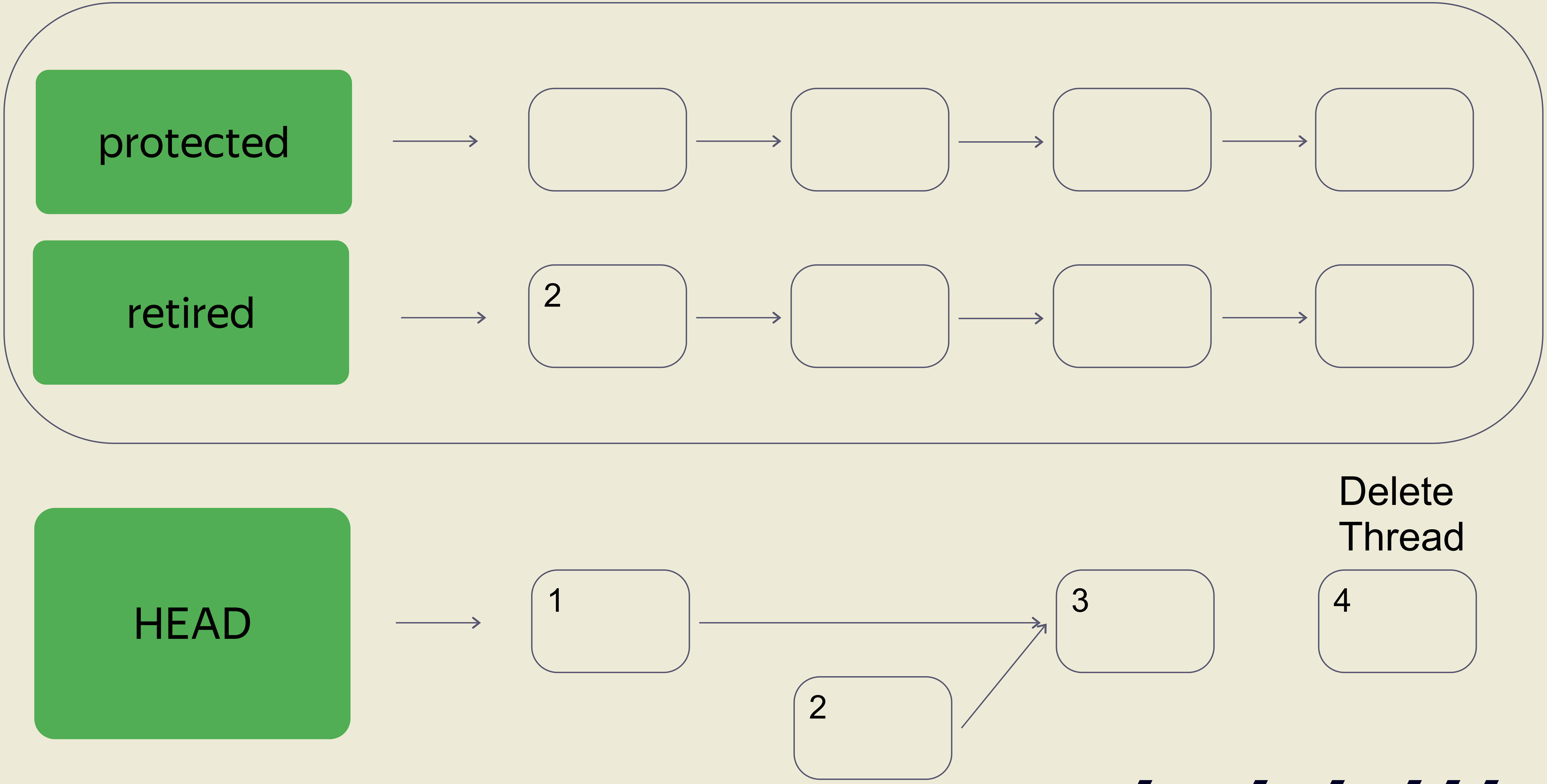
1

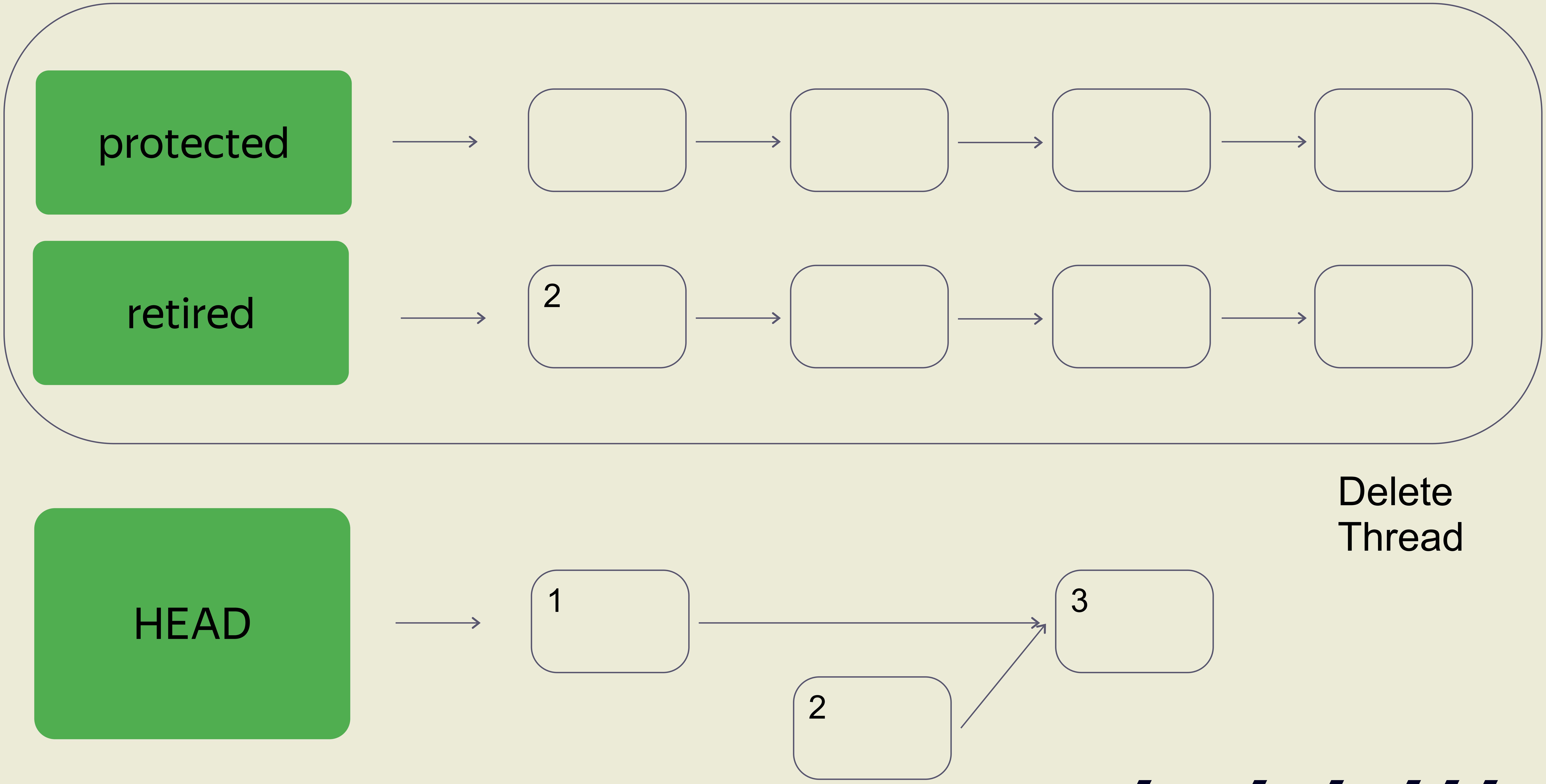
2

3

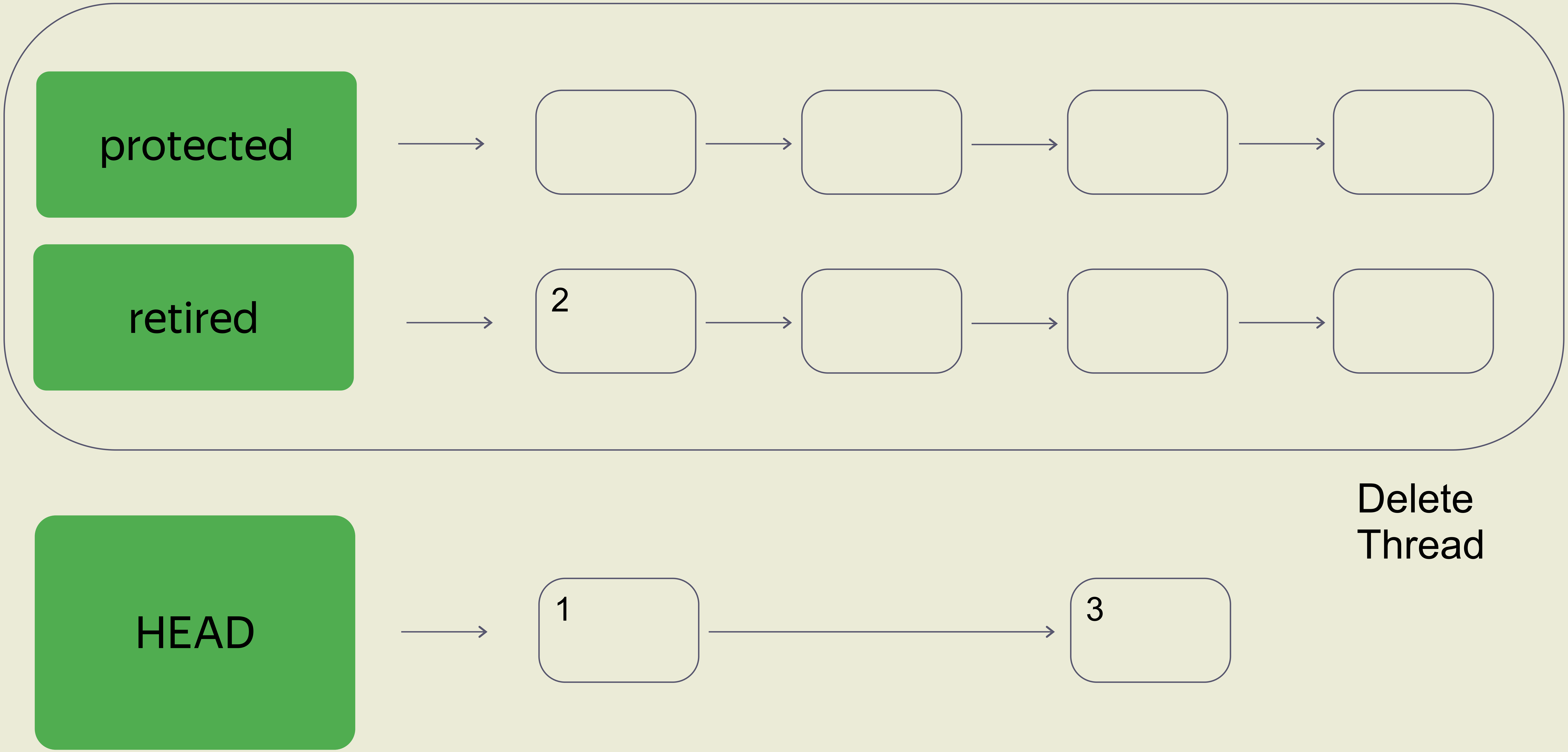
Delete Thread

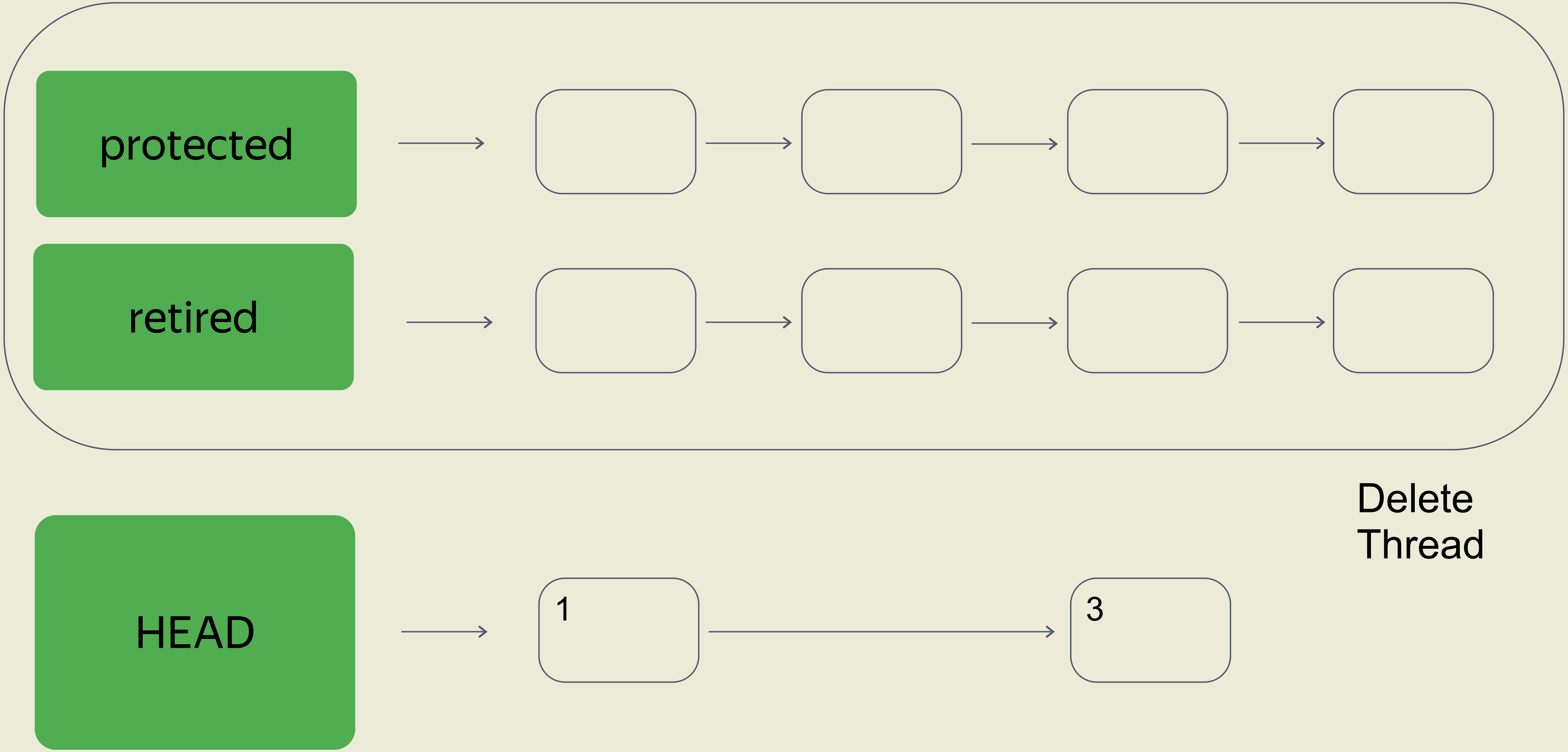
4


protected
retired
2
HEAD
1
3
Delete Thread
4
2



Delete Thread


protected
retired
2
Delete Thread
HEAD
1
3



protected

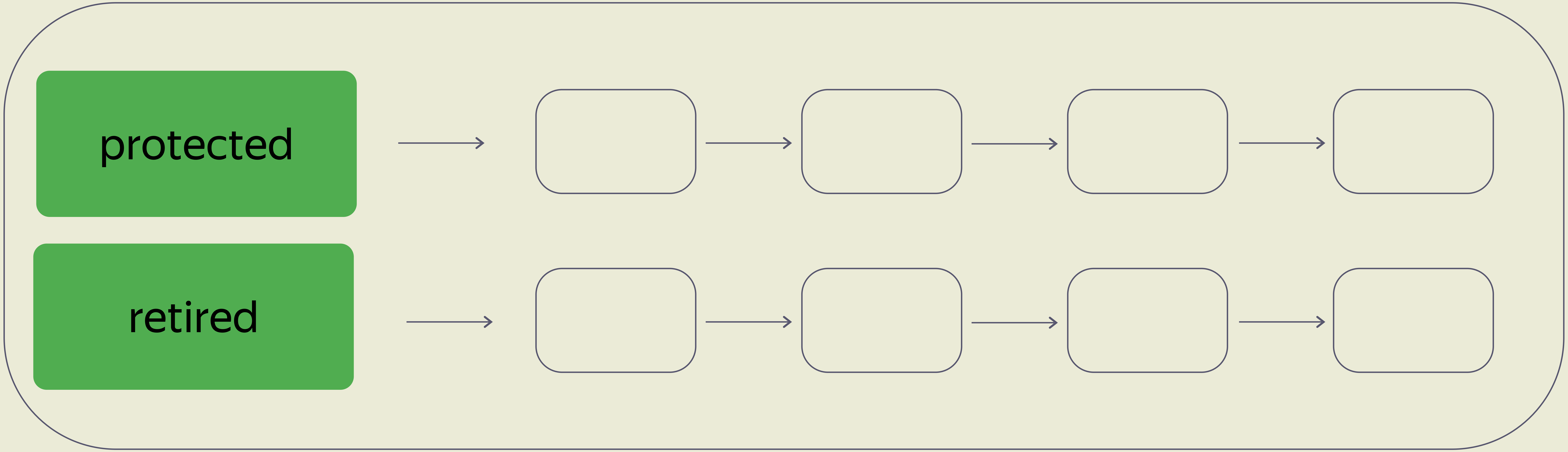
retired

HEAD

1

3

Delete
Thread



```
const unsigned max_hazard_pointers = 100;

struct hazard_pointer{
  std::atomic<std::thread::id> id;
  std::atomic<void*> pointer;
};

hazard_pointer hazard_pointers[max_hazard_pointers];
std::atomic<data_to_reclaim*> nodes_to_reclaim;
```

```
std::shared_ptr<T> pop(){
  std::atomic<void*>& hp = get_hazard_pointer_for_current_thread();
  node<T>* old_head = head.load();

  do {
    node<T>* temp;
    do{
      temp = old_head;
      hp.store(old_head);
      old_head = head.load();
    } while(old_head != temp)
  } while(old_head &&
          !head.compare_exchange_weak(old_head, old_head->next));

  hp.store(nullptr);

  std::shared_ptr<T> res;
  if (old_head){
    res.swap(old_head->data);
    if(outstanding_hazard_pointers_for(old_head)){
      reclaim_latter(old_head);
    } else {
      delete old_head;
    }
  }
  delete_nodes_with_no_hazards();
  return res;
}
```

```
std::shared_ptr<T> pop(){
  std::atomic<void*>& hp = get_hazard_pointer_for_current_thread();
  // ...
}
```

```
class hp_owner{
  hazard_pointer* hp_;
public:
  hp_owner(const hp_owner&) = delete;
  hp_owner operator=(const hp_owner&) = delete;

  hp_owner();
  ~hp_owner();

  std::atomic<void*>& get_pointer(){
    return hp_->pointer;
  }
};


std::atomic<void*>& get_hazard_pointer_for_current_thread(){
  thread_local static hp_owner hazard;
  return hazard.get_pointer();
}
```

```
const unsigned max_hazard_pointers = 100;

struct hazard_pointer{
  std::atomic<std::thread::id> id;
  std::atomic<void*> pointer;
};

hazard_pointer hazard_pointers[max_hazard_pointers];
```

```
hp_owner(): hp_(nullptr) {
  for(auto& hazard_pointer : hazard_pointers){
    std::thread::id old_id;
    if(hazard_pointer.id.compare_exchange_strong(old_id,
                                                 std::this_thread::get_id()))
    {
      hp_ = &hazard_pointer;
      break;
    }
  }
  if(!hp_) throw std::runtime_error(«No hp available»);
}
```

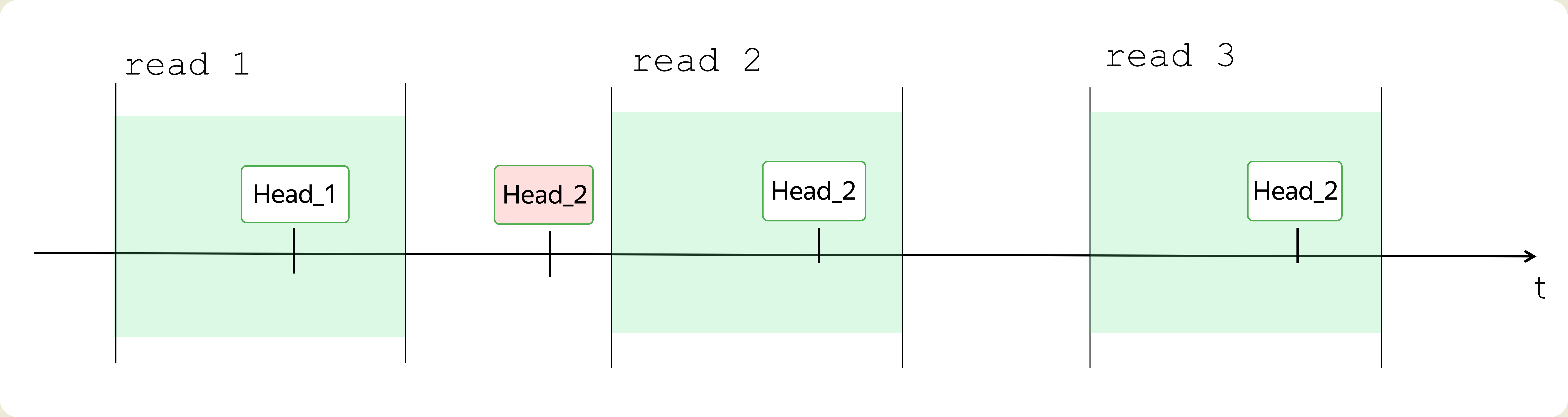
```
~hp_owner(){
  hp->pointer.store(nullptr);
  hp->id.store(std::thread::id());
}
```

```
class hp_owner{
  hazard_pointer* hp;
public:
  hp_owner(const hp_owner&) = delete;
  hp_owner operator=(const hp_owner&) = delete;

  hp_owner();
  ~hp_owner();

  std::atomic<void*>& get_pointer();
};


std::atomic<void*>& get_hazard_pointer_for_current_thread(){
  thread_local static hp_owner hazard;
  return hazard.get_pointer();
}
```

```
std::shared_ptr<T> pop(){
  std::atomic<void*>& hp = get_hazard_pointer_for_current_thread();
  node<T>* old_head = head.load();

  do {
    node<T>* temp;
    do{
      temp = old_head;
      hp.store(old_head);
      old_head = head.load();
    } while(old_head != temp)
  } while(old_head && !head.compare_exchange_weak(old_head, old_head->next));

  // ...
}
```

```
std::shared_ptr<T> pop(){
 std::atomic<void*>& hp = get_hazard_pointer_for_current_thread();
 node<T>* old_head = head.load();

 do {
  node<T>* temp;
  do{
   temp = old_head;
   hp.store(old_head);
   old_head = head.load();
  } while(old_head != temp)
 } while(old_head && !head.compare_exchange_weak(old_head, old_head->next));

 // ...
}
```

```
std::shared_ptr<T> pop(){
  std::atomic<void*>& hp = get_hazard_pointer_for_current_thread();
  node<T>* old_head = head.load();

  do {
    node<T>* temp;
    do{
      temp = old_head;
      hp.store(old_head);
      old_head = head.load();
    } while(old_head != temp)
  } while(old_head && !head.compare_exchange_weak(old_head, old_head->next));

  // ...
}
```



```
std::shared_ptr<T> pop(){
  std::atomic<void*>& hp = get_hazard_pointer_for_current_thread();
  node<T>* old_head = head.load();

  do {
      // ...
  } while(old_head &&
          !head.compare_exchange_weak(old_head, old_head->next));

  hp.store(nullptr);

  std::shared_ptr<T> res;
  if (old_head){
    res.swap(old_head->data);
    if(outstanding_hazard_pointers_for(old_head)){
      reclaim_latter(old_head);
    } else {
      delete old_head;
    }
  }
  delete_nodes_with_no_hazards();
  return res;
}
```

```
std::shared_ptr<T> pop(){
  // ..

  std::shared_ptr<T> res;
  if (old_head){
    res.swap(old_head->data);
    if(outstanding_hazard_pointers_for(old_head)){
      reclaim_latter(old_head);
    } else {
      delete old_head;
    }
  }
  delete_nodes_with_no_hazards();
  return res;
}
```

```cpp
const unsigned max_hazard_pointers = 100;

struct hazard_pointer{
  std::atomic<std::thread::id> id;
  std::atomic<void*> pointer;
};

hazard_pointer hazard_pointers[max_hazard_pointers];
```

```cpp
bool outstanding_hazard_pointers_for(void* p){
  for(auto & hazard_pointer : hazard_pointers){
    if(hazard_pointer.pointer.load() == p){
      return true;
    }
  }
  return false;
}
```

```
std::shared_ptr<T> pop(){
  // ..

  std::shared_ptr<T> res;
  if (old_head){
    res.swap(old_head->data);
    if(outstanding_hazard_pointers_for(old_head)){
      reclaim_latter(old_head);
    } else {
      delete old_head;
    }
  }
  delete_nodes_with_no_hazards();
  return res;
}
```

```
template<typename T>
void do_delete(void* p){
  delete static_cast<T*>(p);
}

struct data_to_reclaim
{
    void* data;
    std::function<void(void*)> deleter;
    data_to_reclaim* next;

    template<typename T>
    data_to_reclaim(T* p):
      data(p),
      deleter(&do_delete<T>),
      next(nullptr)
    {}

    ~data_to_reclaim(){
      deleter(data);
    }
};
```

```
std::atomic<data_to_reclaim*> nodes_to_reclaim;
```

```
void add_to_rlist(data_to_reclaim* node){
  node->next = nodes_to_reclaim.load();
  while(!nodes_to_reclaim.compare_exchange_weak(
                 node->next, node));
}

template<typename T>
void reclaim_later(T* data){
  add_to_reclaim_list(new data_to_reclaim(data));
}
```

```
template<typename T>
void do_delete(void* p){
  delete static_cast<T*>(p);
}

struct data_to_reclaim
{
    void* data;
    std::function<void(void*)> deleter;
    data_to_reclaim* next;

    template<typename T>
    data_to_reclaim(T* p):
      data(p),
      deleter(&do_delete<T>),
      next(nullptr)
    {}

    ~data_to_reclaim(){
      deleter(data);
    }
};
```

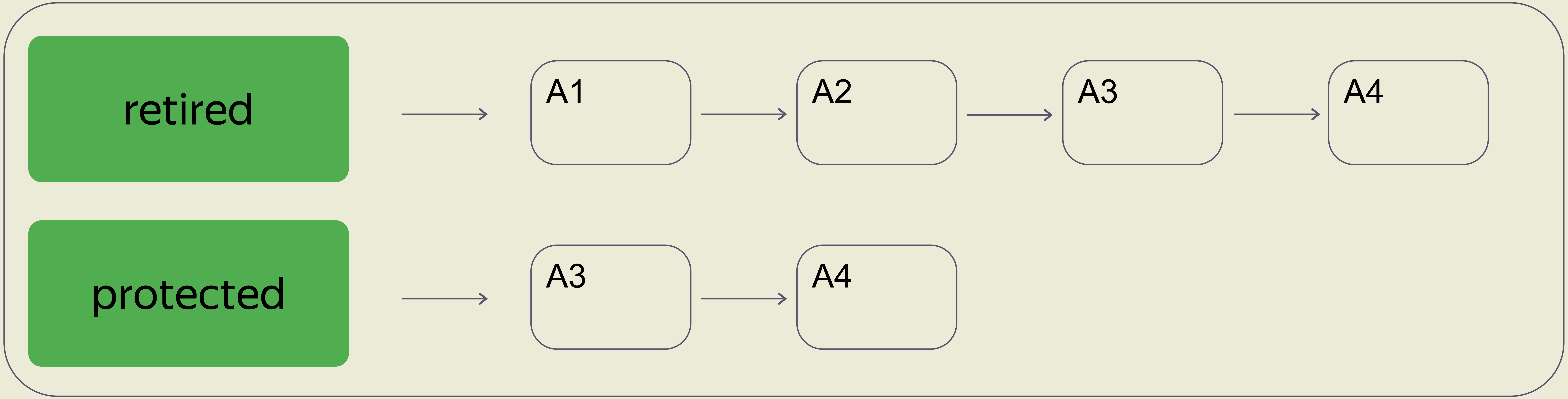
```
std::atomic<data_to_reclaim*> nodes_to_reclaim;
```

```
void add_to_rlist(data_to_reclaim* node){
  node->next = nodes_to_reclaim.load();
  while(!nodes_to_reclaim.compare_exchange_weak(
                 node->next, node));
}

template<typename T>
void reclaim_later(T* data){
  add_to_reclaim_list(new data_to_reclaim(data));
}
```

```
std::shared_ptr<T> pop(){
  // ...

  std::shared_ptr<T> res;
  // ...
  delete_nodes_with_no_hazards();
  return res;
}
```

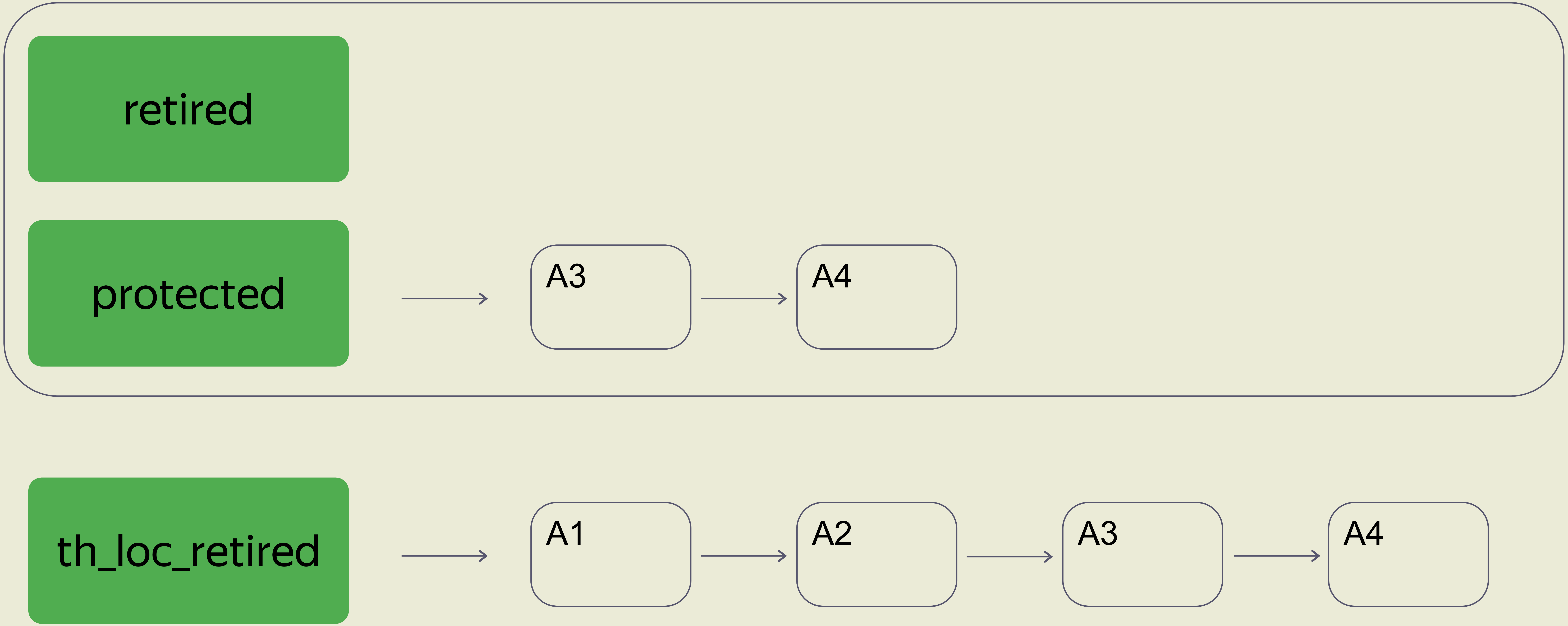
```
std::atomic<data_to_reclaim*> nodes_to_reclaim;
```

```
void delete_nodes_with_no_hazard(){
  data_to_reclaim* current = nodes_to_reclaim.exchange(nullptr);
  while (current)
  {
    data_to_reclaim* const next = current->next;
    if(!outstanding_hazard_pointers_for(current->data)){
      delete current;
    } else {
      add_to_reclaim_list(current);
    }
    current = next;
  }
}
```



retired → A1 → A2 → A3 → A4

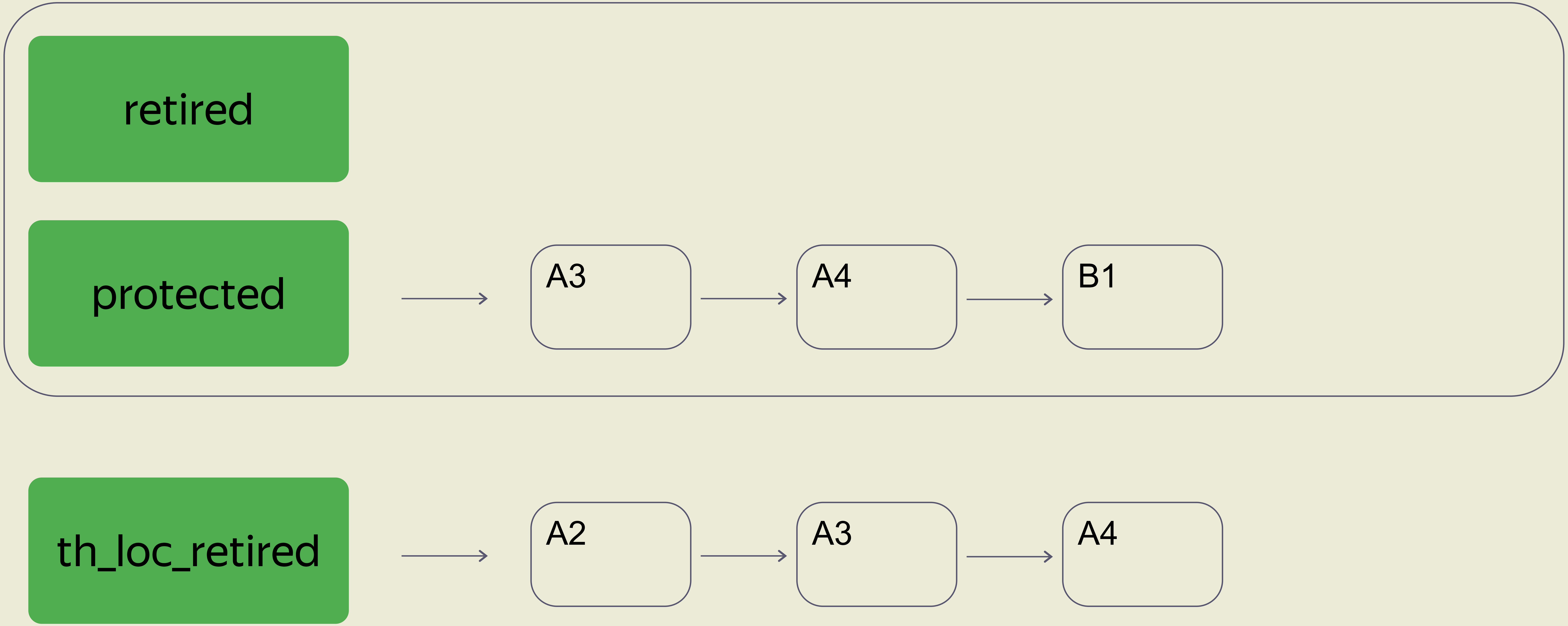
protected → A3 → A4



retired

protected → A3 → A4

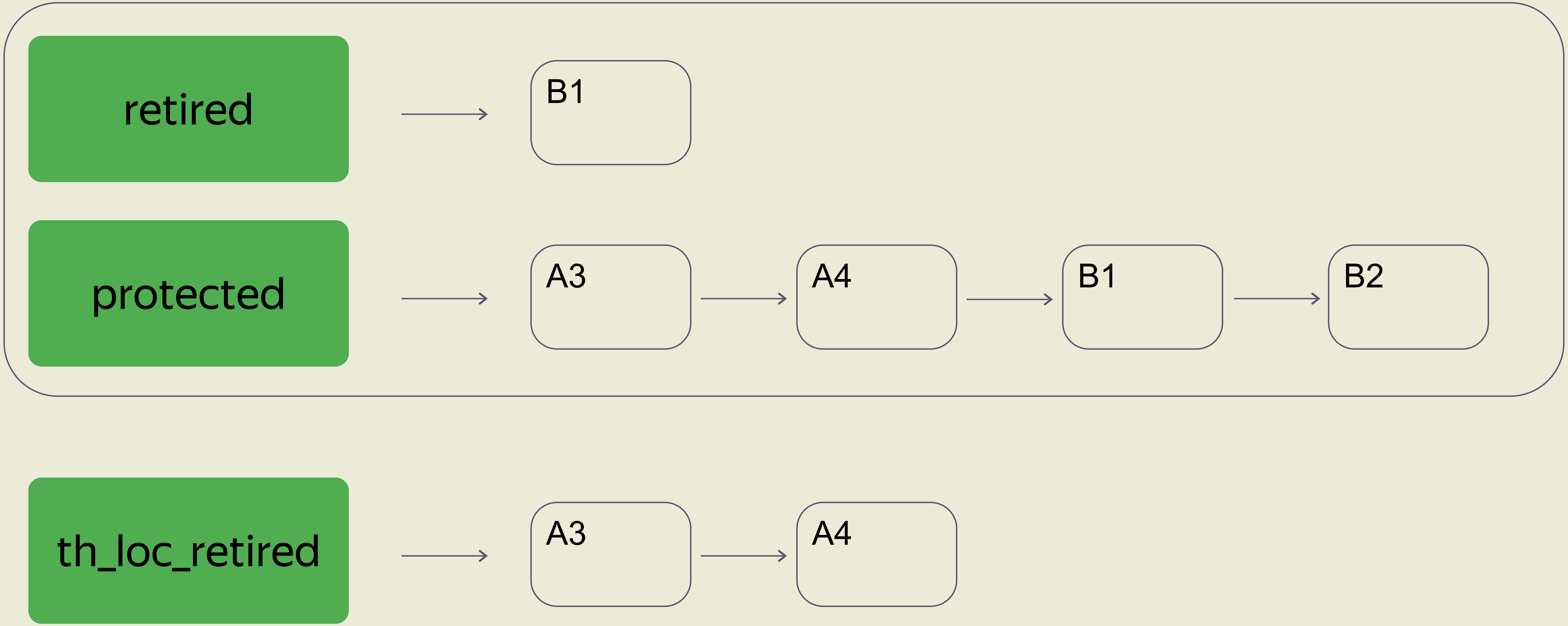
th_loc_retired → A1 → A2 → A3 → A4



retired

protected → A3 → A4 → B1

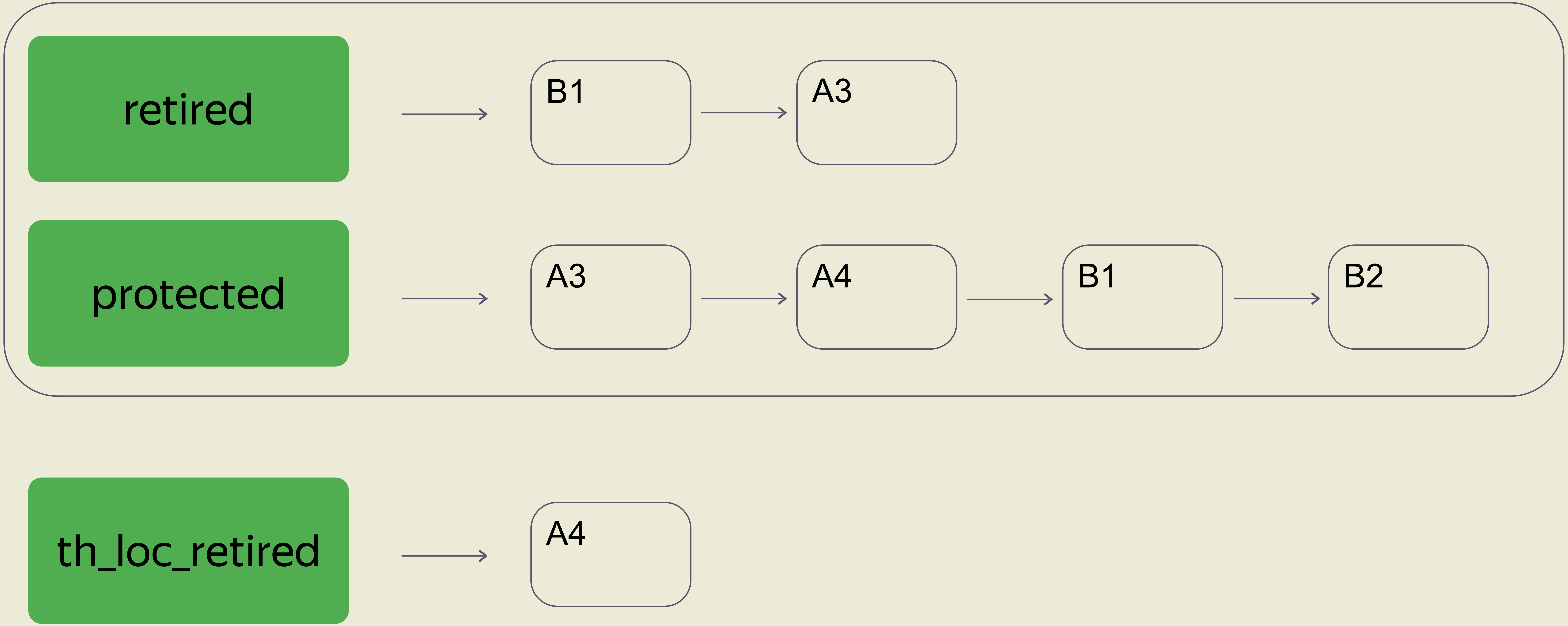
th_loc_retired → A2 → A3 → A4



retired → B1

protected → A3 → A4 → B1 → B2

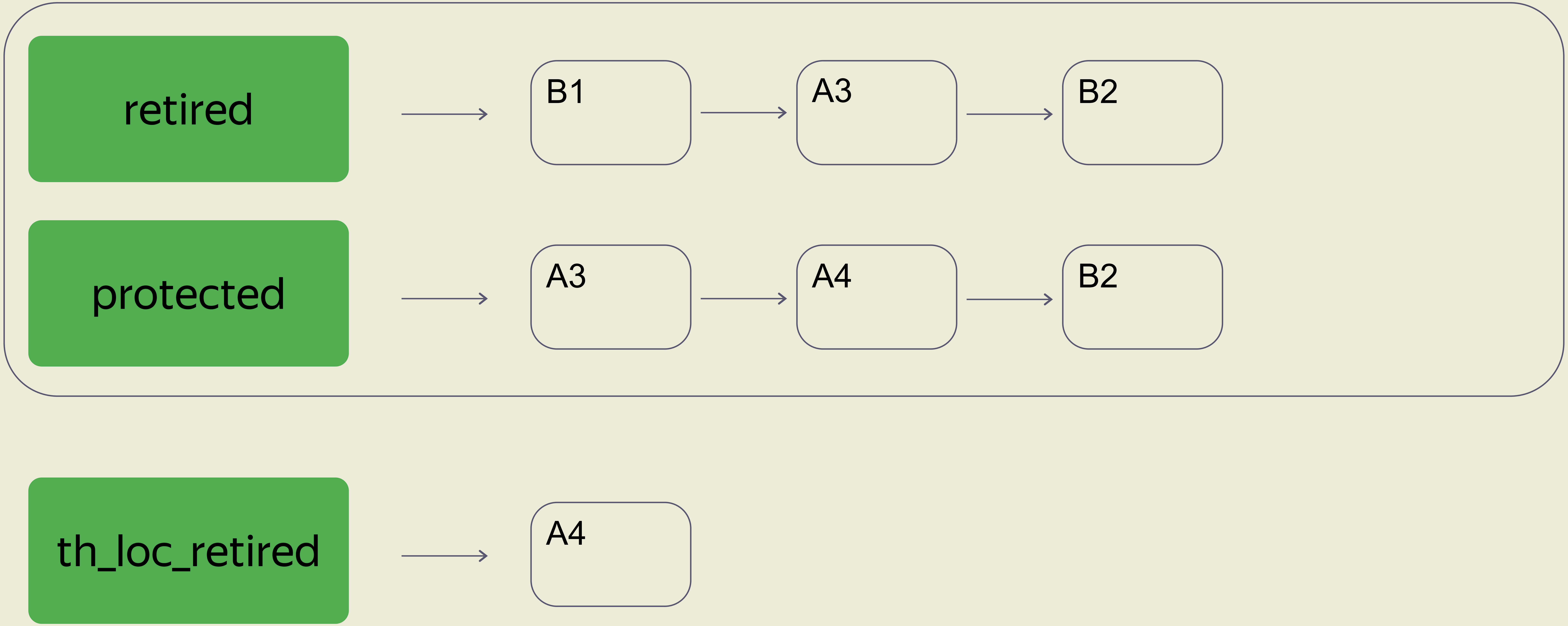
th_loc_retired → A3 → A4

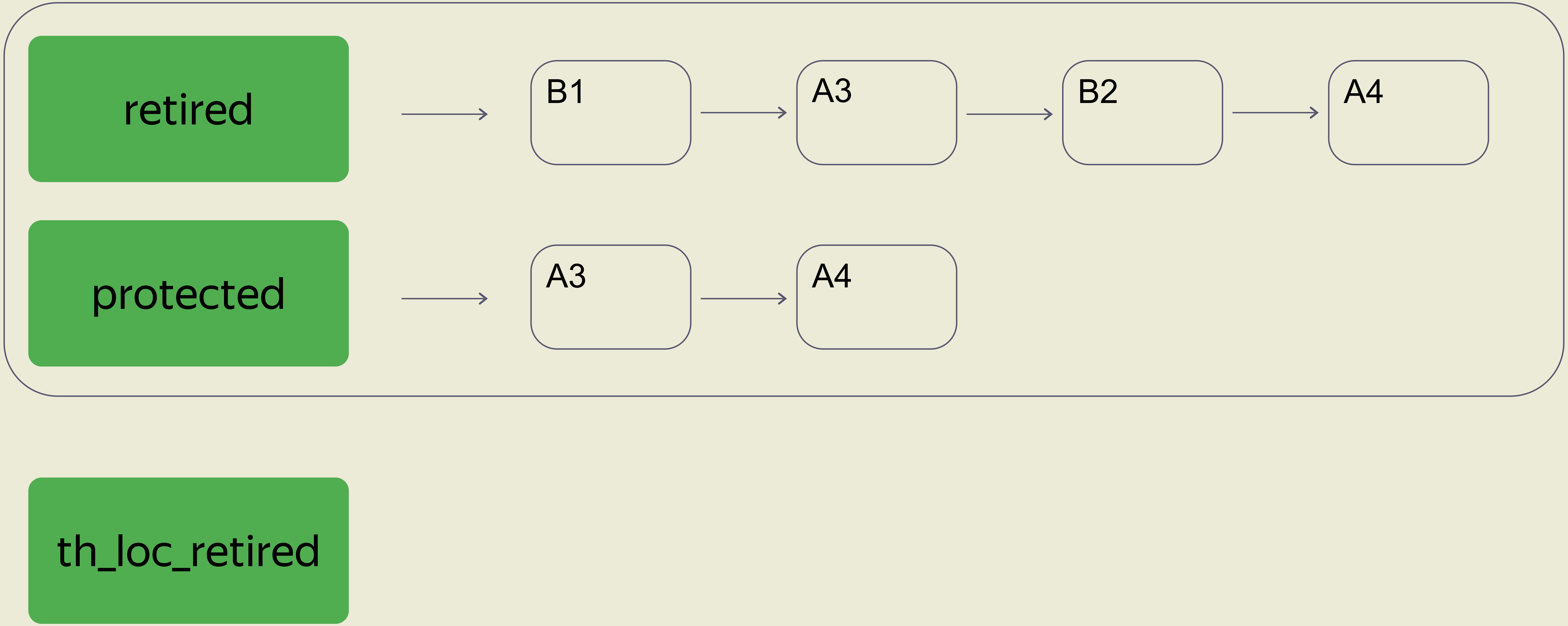


retired → B1 → A3

protected → A3 → A4 → B1 → B2

th_loc_retired → A4


retired
B1
A3
B2
protected
A3
A4
B2
th_loc_retired
A4



retired → B1 → A3 → B2 → A4

protected → A3 → A4

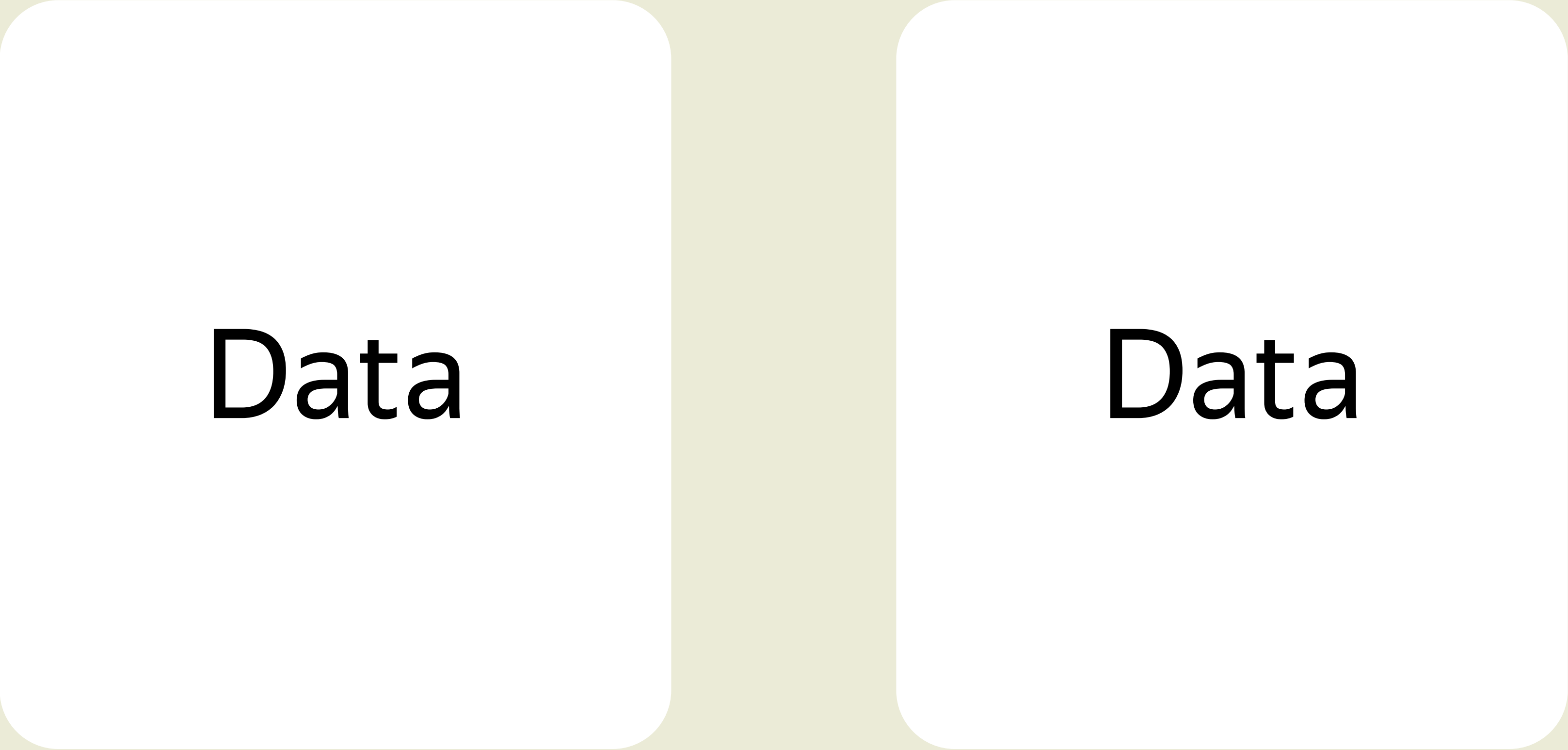
th_loc_retired

```
std::shared_ptr<T> pop(){
  std::atomic<void*>& hp = get_hazard_pointer_for_current_thread();
  node<T>* old_head = head.load();

  do {
    node<T>* temp;
    do{
      temp = old_head;
      hp.store(old_head);
      old_head = head.load();
    } while(old_head != temp)
  } while(old_head &&
          !head.compare_exchange_weak(old_head, old_head->next));

  hp.store(nullptr);

  std::shared_ptr<T> res;
  if (old_head){
    res.swap(old_head->data);
    if(outstanding_hazard_pointers_for(old_head)){
      reclaim_latter(old_head);
    } else {
      delete old_head;
    }
  }
  delete_nodes_with_no_hazards();
  return res;
}
```

```
// однопоточный stack
std::shared_ptr<T> pop(){
  if (head == nullptr)
    return nullptr;

  std::shared_ptr<T>
       res = head->data;

  head = head->next;
  return res;
}
```

Снимки атомарных регистров

Регистры LOCK-FREE

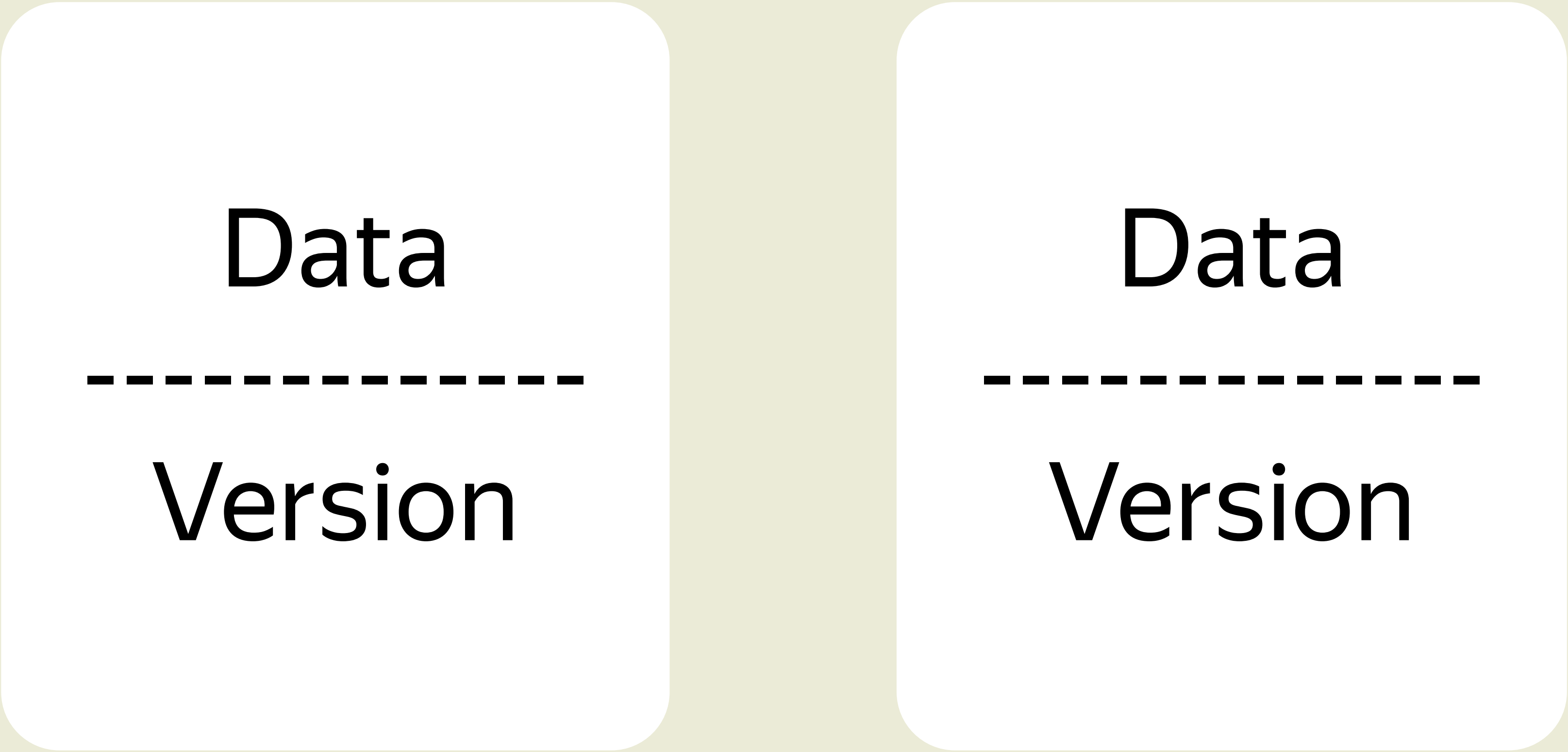


Data

Data

Время	(D)	(D)
1	(0)	(0)
2	(1)	(0)
3	(1)	(1)
4	(2)	(0)

Регистры LOCK-FREE



Data

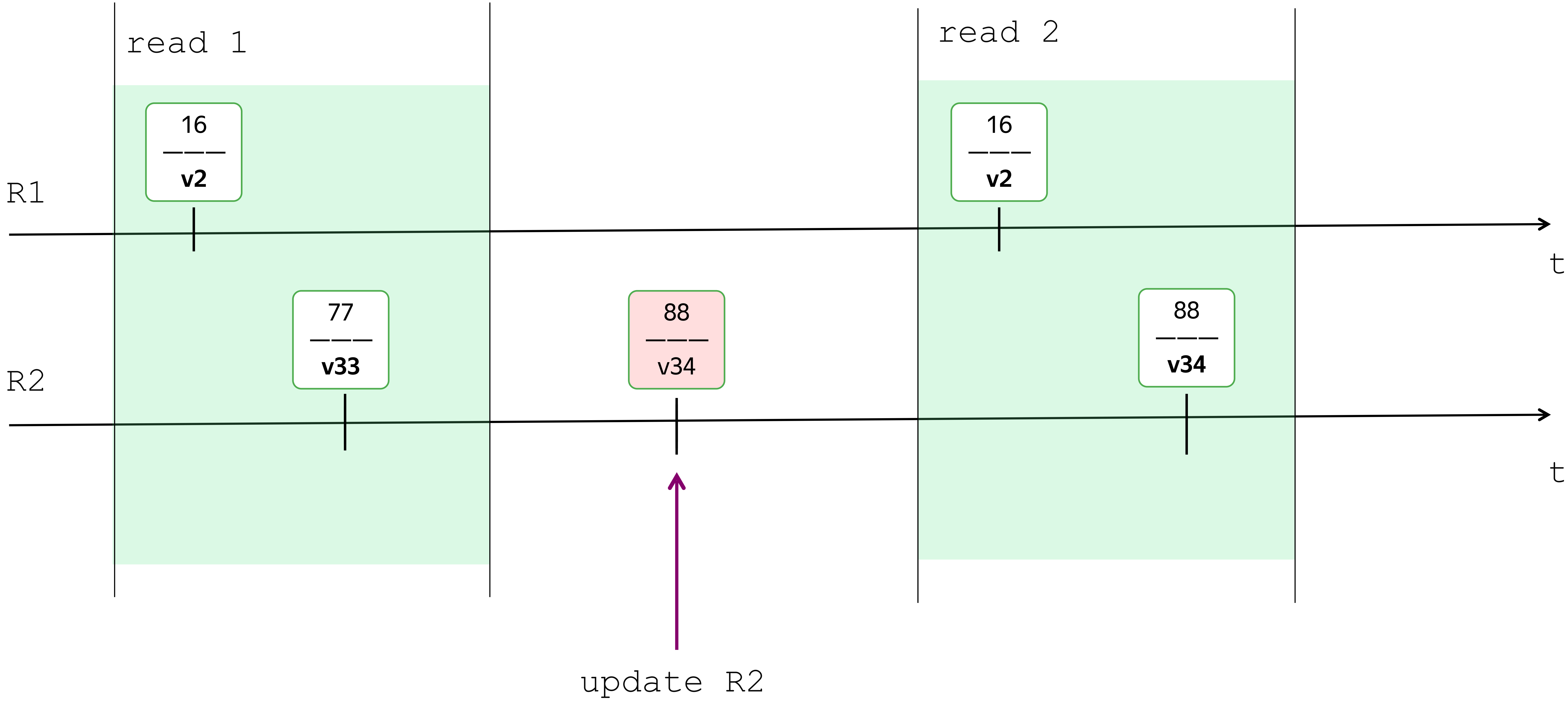
Version

Data

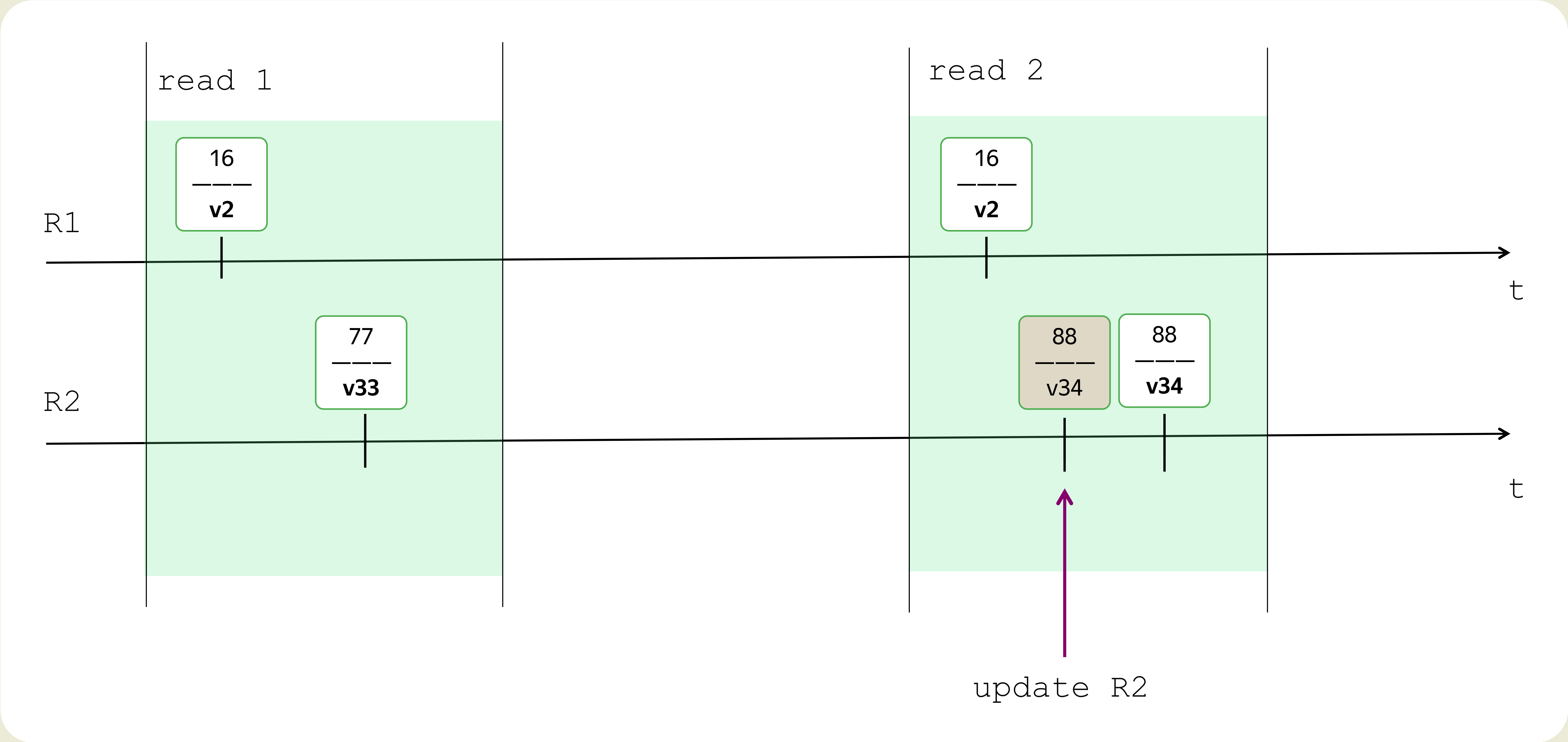
Version

Время	(D,V)	(D,V)
1	(0, v0)	(0, v0)
2	(1, v1)	(0, v0)
3	(1, v1)	(1, v1)
4	(2, v2)	(0, v2)

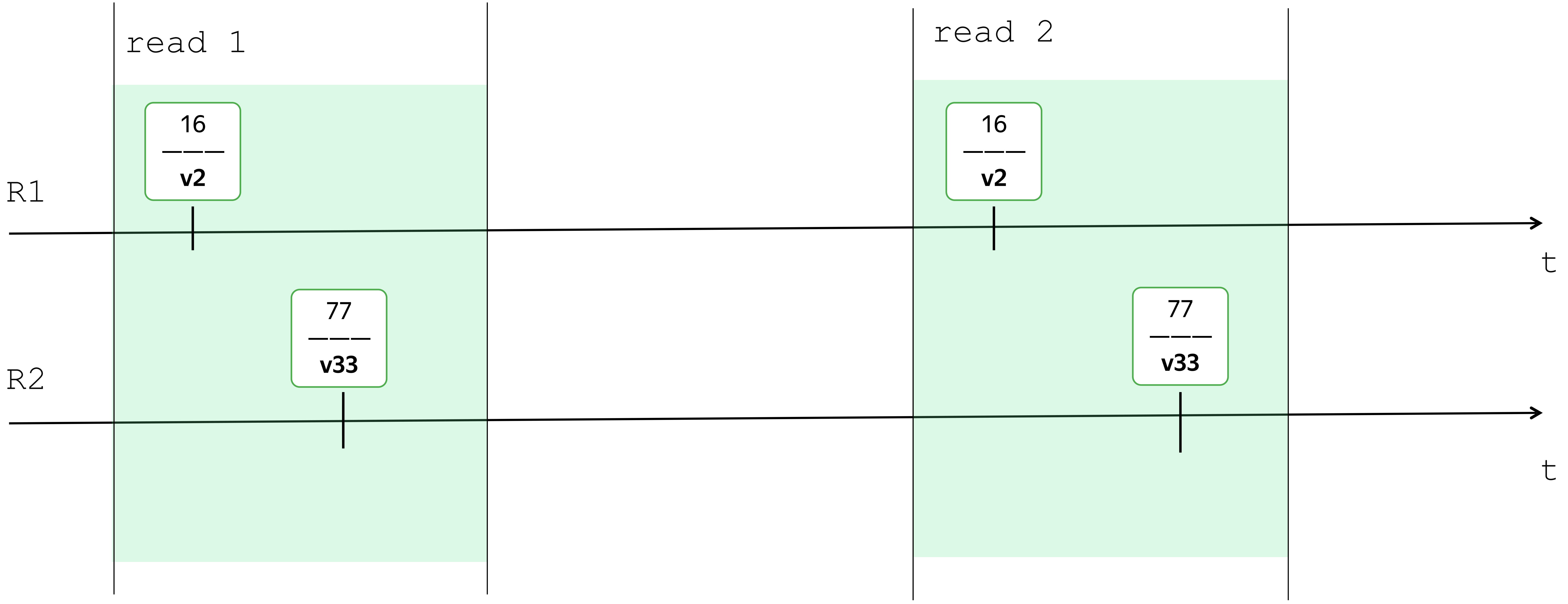
Регистры LOCK-FREE



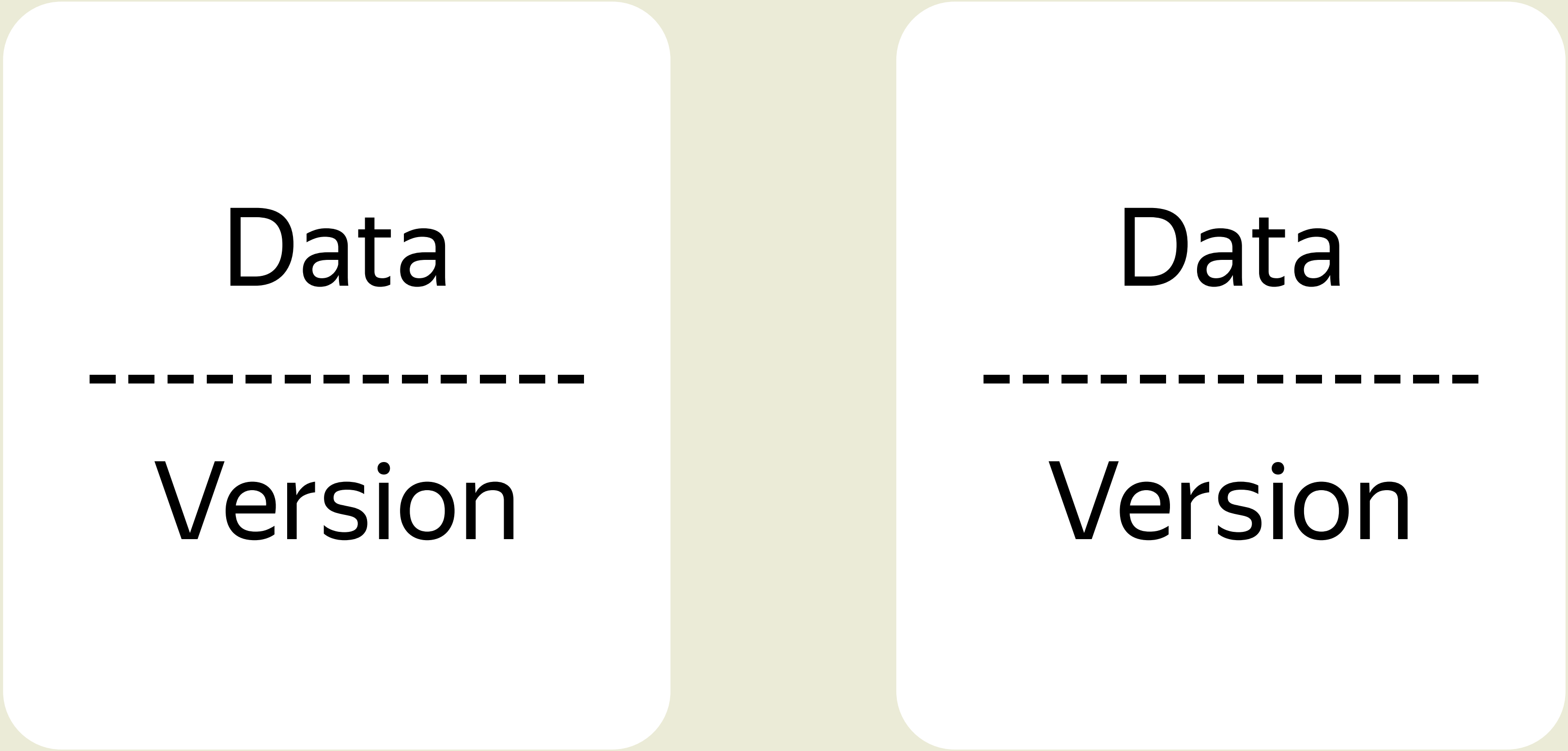
Регистры LOCK-FREE



Регистры LOCK-FREE



Регистры LOCK-FREE



Data

Version

Data

Version

SCAN() ->

(0, **v0**) (1, **v1**)

Время	(D,V)	(D,V)
1	(0, v0)	(0, v0)
2	(1, v1)	(0, v0)
3	(1, v1)	(1, v1)
4	(2, v2)	(0, v2)

Регистры LOCK-FREE

Data

Version

Data

Version

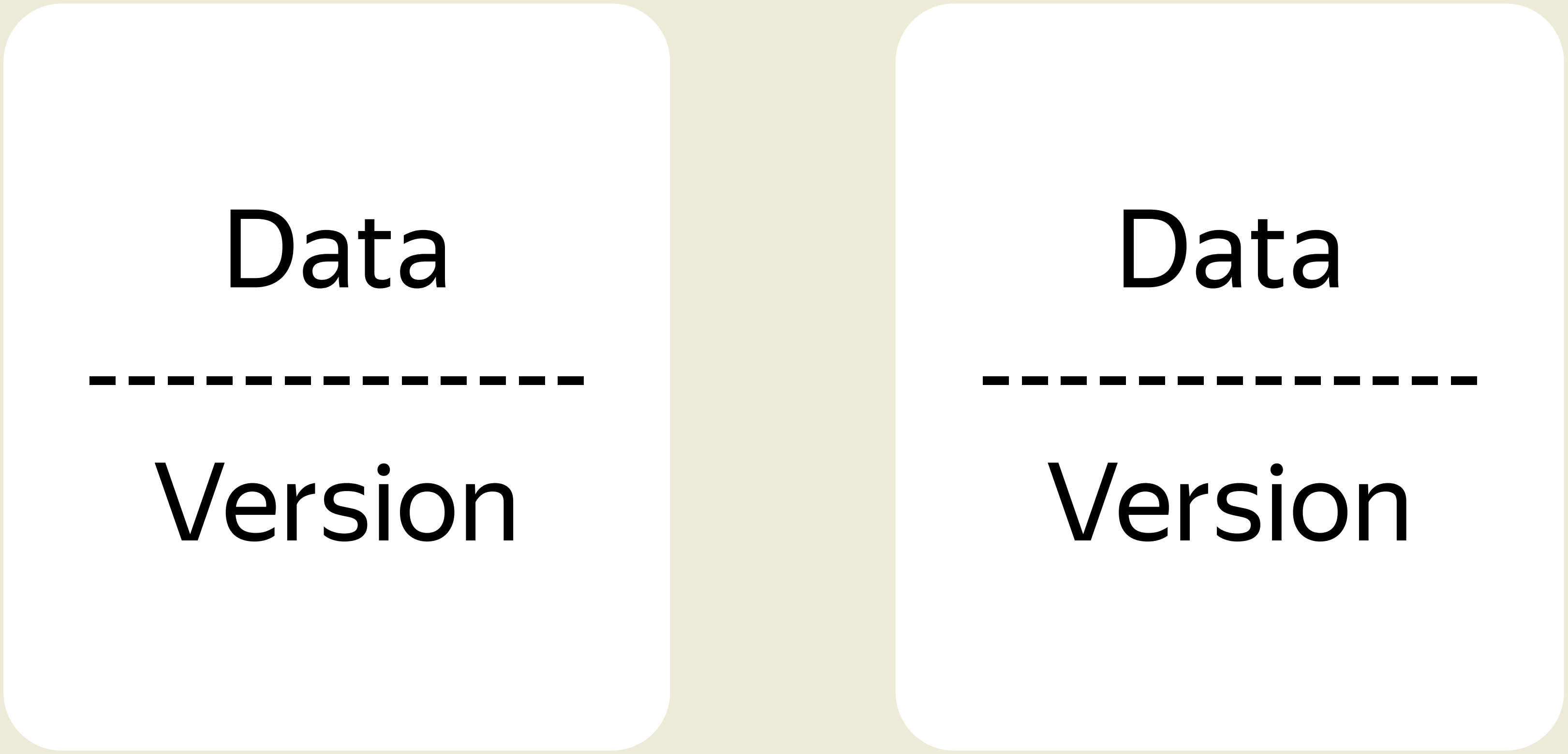
SCAN() ->

(0, **v0**) (1, **v1**)

(1, **v1**) (1, **v1**)

Время	(D,V)	(D,V)
1	(0, v0)	(0, v0)
2	(1, v1)	(0, v0)
3	(1, v1)	(1, v1)
4	(2, v2)	(0, v2)

Регистры LOCK-FREE

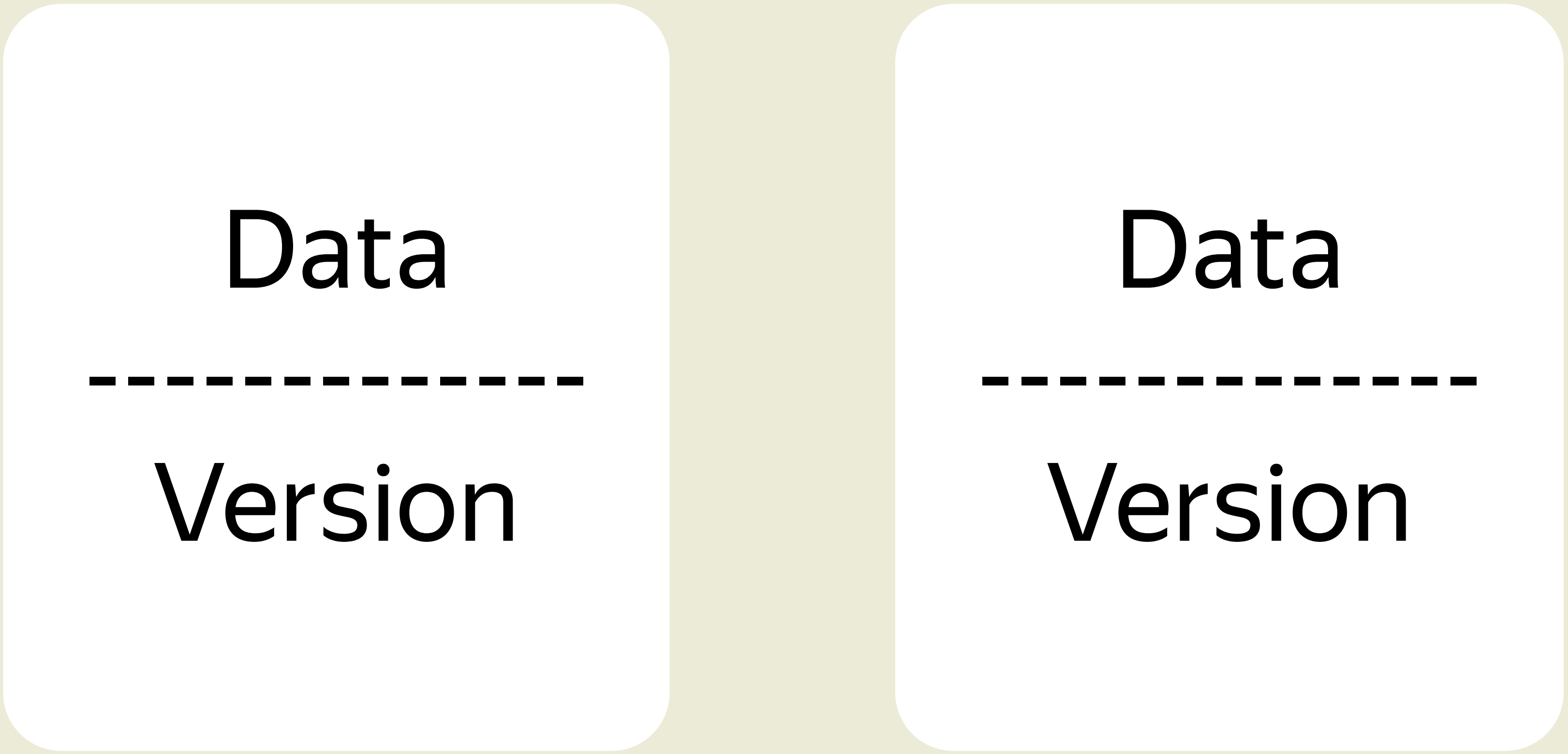


SCAN() ->

(0, **v0**) (1, **v1**)

(1, **v1**) (1, **v1**)

Регистры LOCK-FREE



SCAN() ->

(0, **v0**) (1, **v1**)

(1, **v1**) (1, **v1**)

(1, **v1**) (0, **v2**)

Время	(D,V)	(D,V)
1	(0, v0)	(0, v0)
2	(1, v1)	(0, v0)
3	(1, v1)	(1, v1)
4	(2, v2)	(0, v2)

Регистры LOCK-FREE

Data

Version

Data

Version

SCAN() ->

(0, **v0**) (1, **v1**)

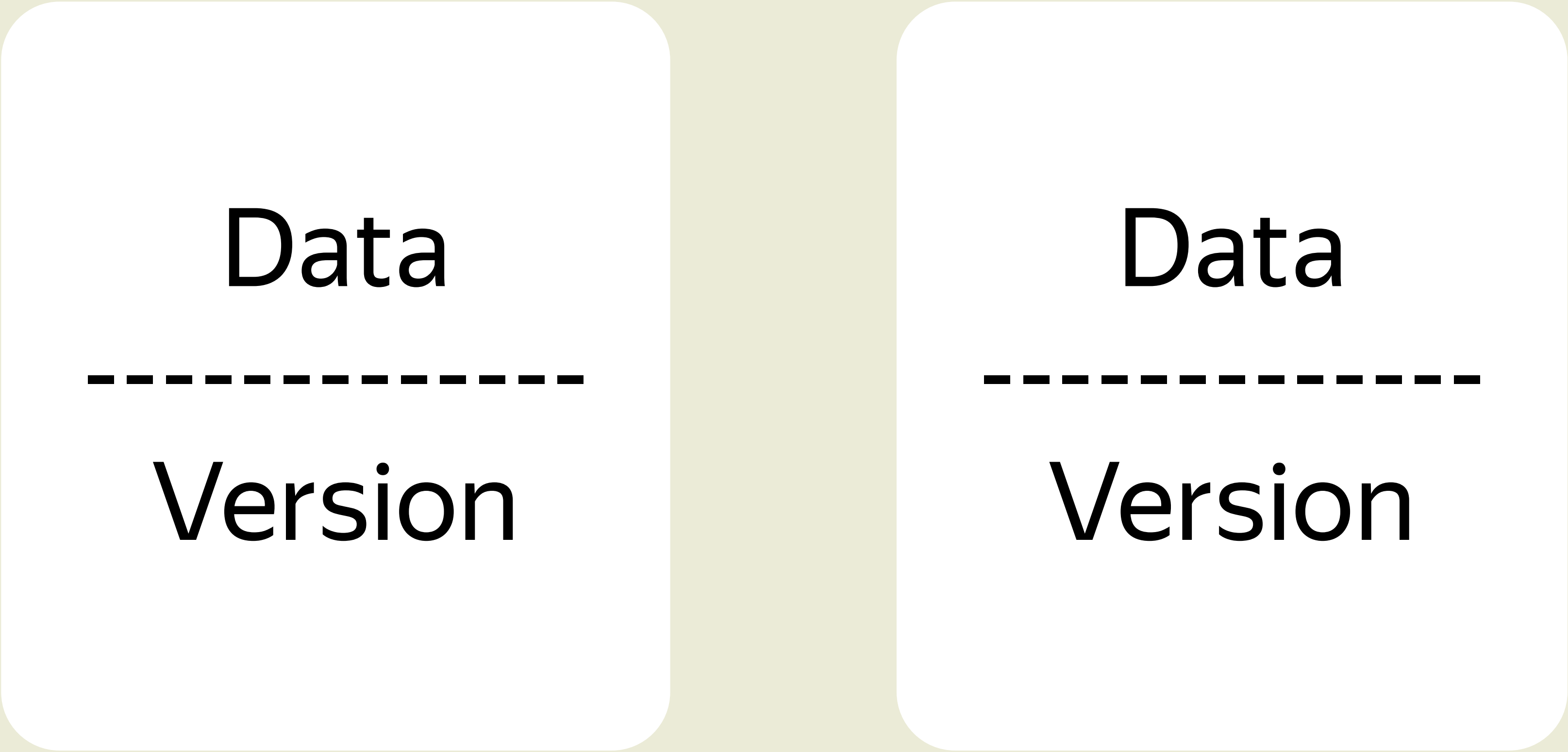
(1, **v1**) (1, **v1**)

(1, **v1**) (0, **v2**)

(2, **v2**) (0, **v2**)

Время	(D,V)	(D,V)
1	(0, v0)	(0, v0)
2	(1, v1)	(0, v0)
3	(1, v1)	(1, v1)
4	(2, v2)	(0, v2)

Регистры LOCK-FREE



Data

Version

Data

Version

SCAN() ->

(0, **v0**) (1, **v1**)

(1, **v1**) (1, **v1**)

(1, **v1**) (0, **v2**)

(2, **v2**) (0, **v2**)

(2, **v2**) (0, **v2**)

Время	(D,V)	(D,V)
1	(0, v0)	(0, v0)
2	(1, v1)	(0, v0)
3	(1, v1)	(1, v1)
4	(2, v2)	(0, v2)

Регистры SWMR WAIT-FREE

Data

Version

Data

Version

Время	(D,V)	(D,V)
1	(0, v0)	(0, v0)
2	(1, v1)	(0, v0)
3	(1, v1)	(1, v1)
4	(2, v2)	(0, v2)

Регистры SWMR WAIT-FREE

Data

Version

Snapshot

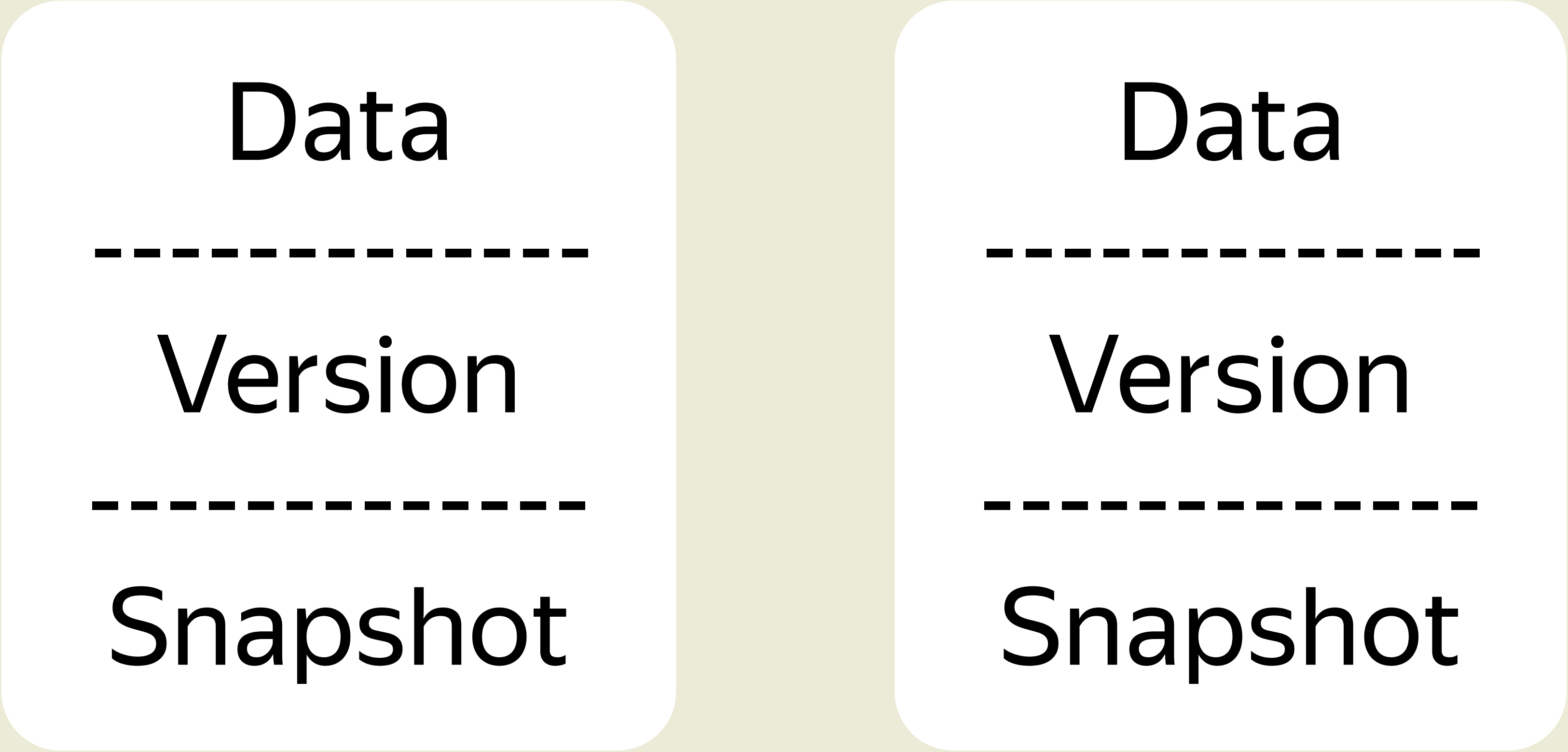
Data

Version

Snapshot

Время	(D,V,Snapshot)	(D,V,Snapshot)
1	(0, v0,[0;0])	(0, v0,[0;0])
2	(1, v1,**[0;0]**)	(0, v0,[0;0])
3	(1, v1,[0;0])	(1, v1,**[1;0]**)
4	(2, v2,**[1;1]**)	(0, v2,**[1;1]**)

Регистры SWMR WAIT-FREE



Data

Version

Snapshot

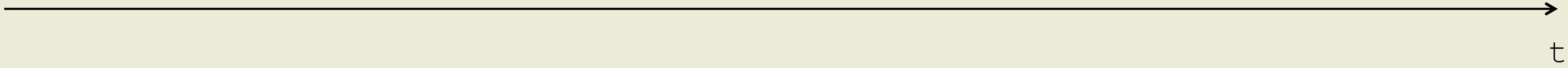
Data

Version

Snapshot

```
update(Data d) {
  Snapshot s = get_snapshot();
  write(d, v + 1, s);
}
```

```
get_snapshot():
```



Регистры SWMR WAIT-FREE

Data

Version

Snapshot

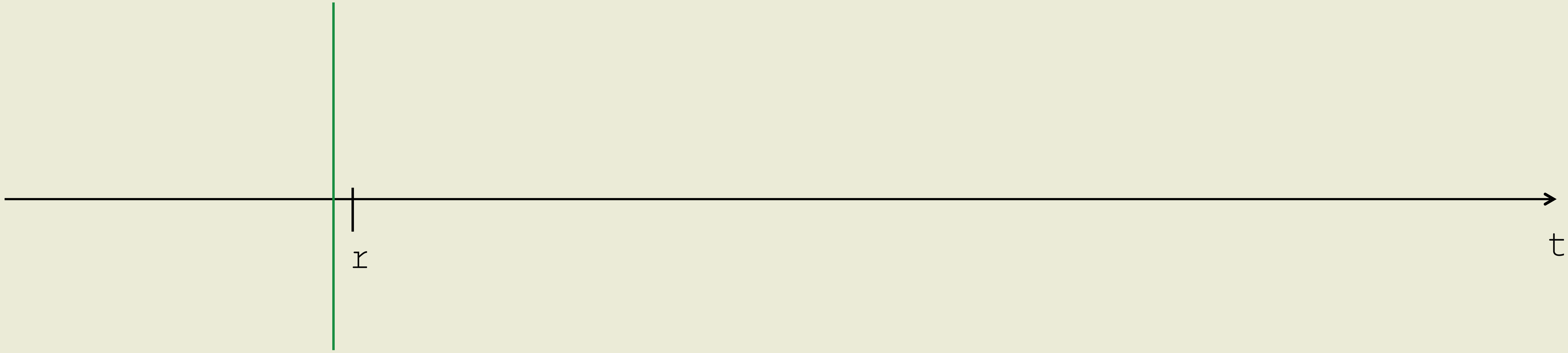
Data

Version

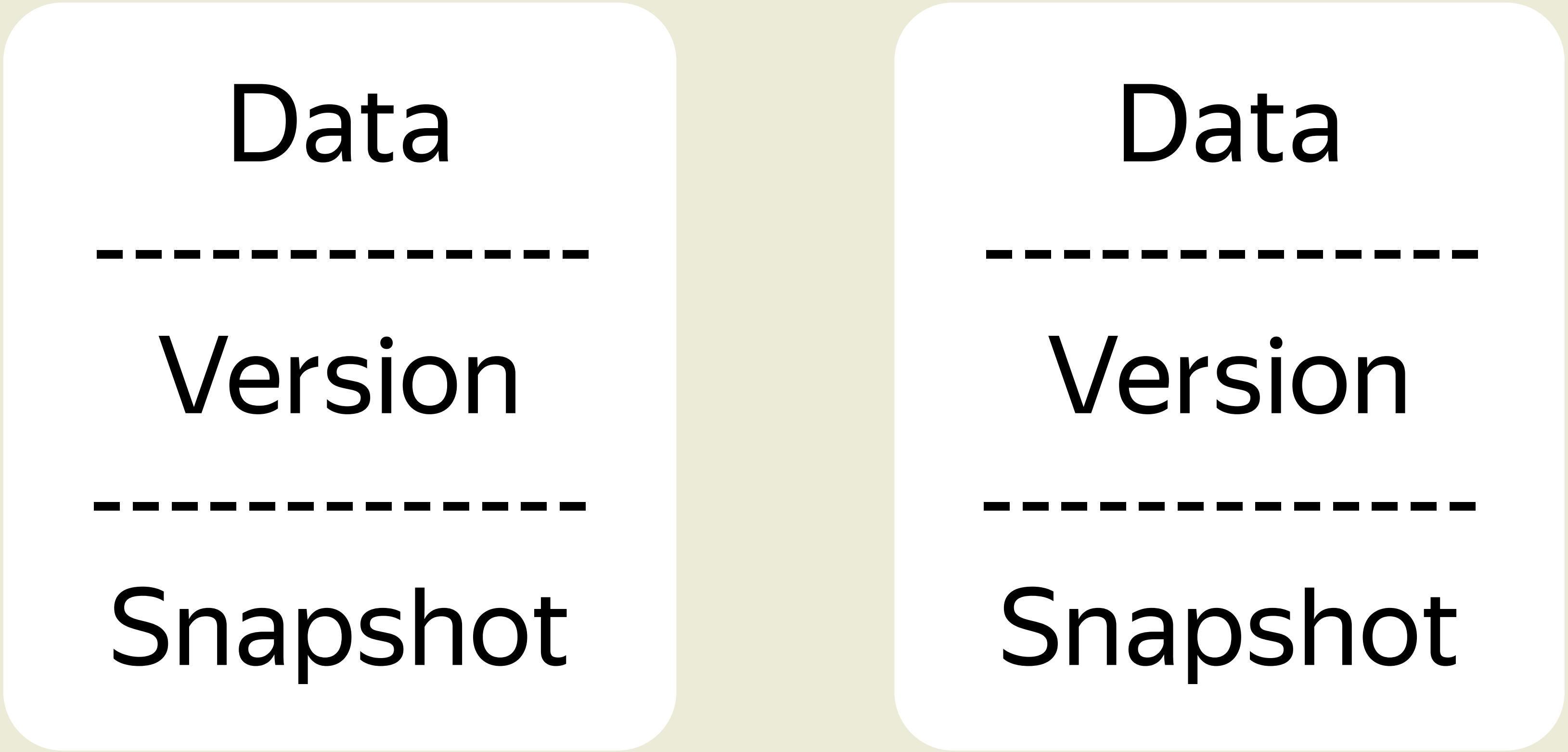
Snapshot

```
update(Data d) {
  Snapshot s = get_snapshot();
  write(d, v + 1, s);
}
```

```
get_snapshot():
```

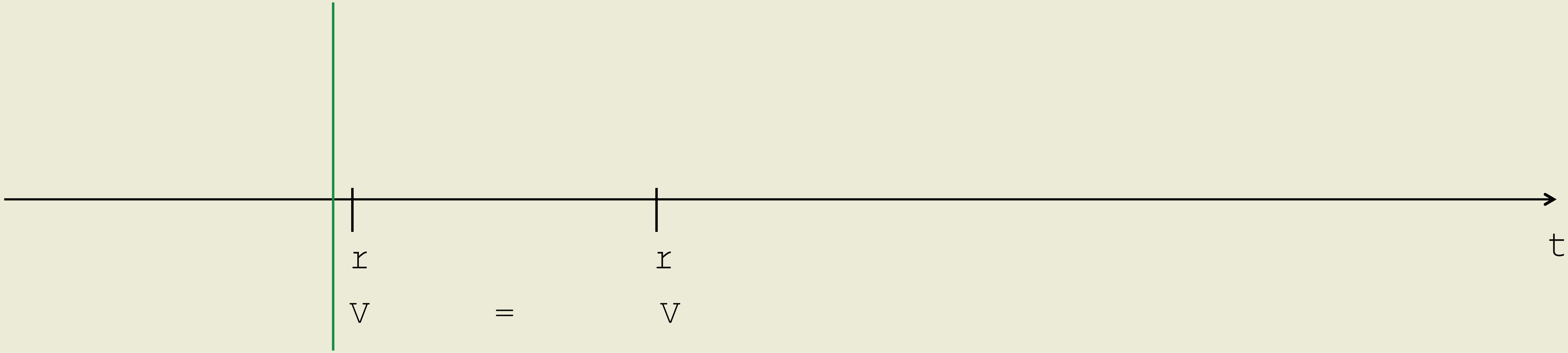


Регистры SWMR WAIT-FREE



```
update(Data d) {
  Snapshot s = get_snapshot();
  write(d, v + 1, s);
}
```

get_snapshot():



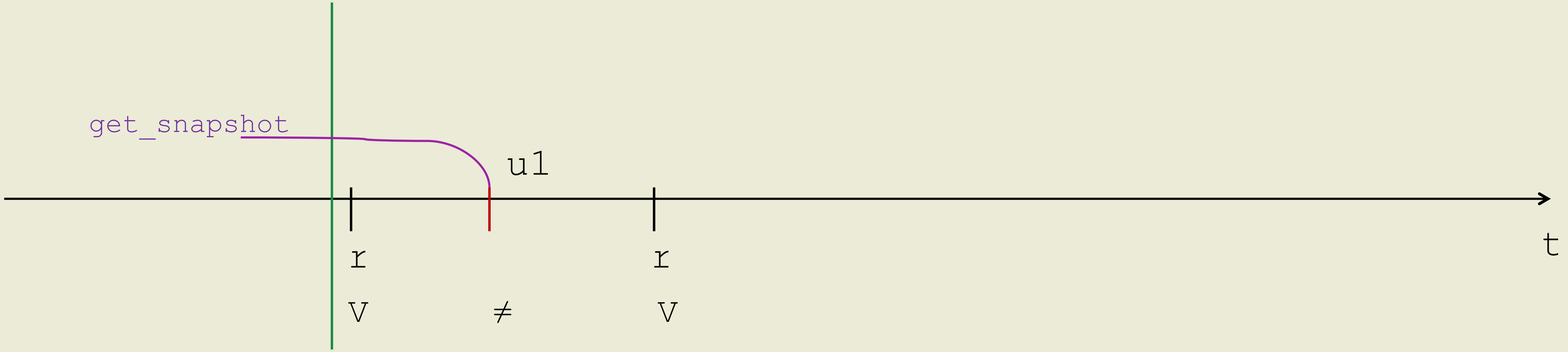
Регистры SWMR WAIT-FREE

Data
Version
Snapshot

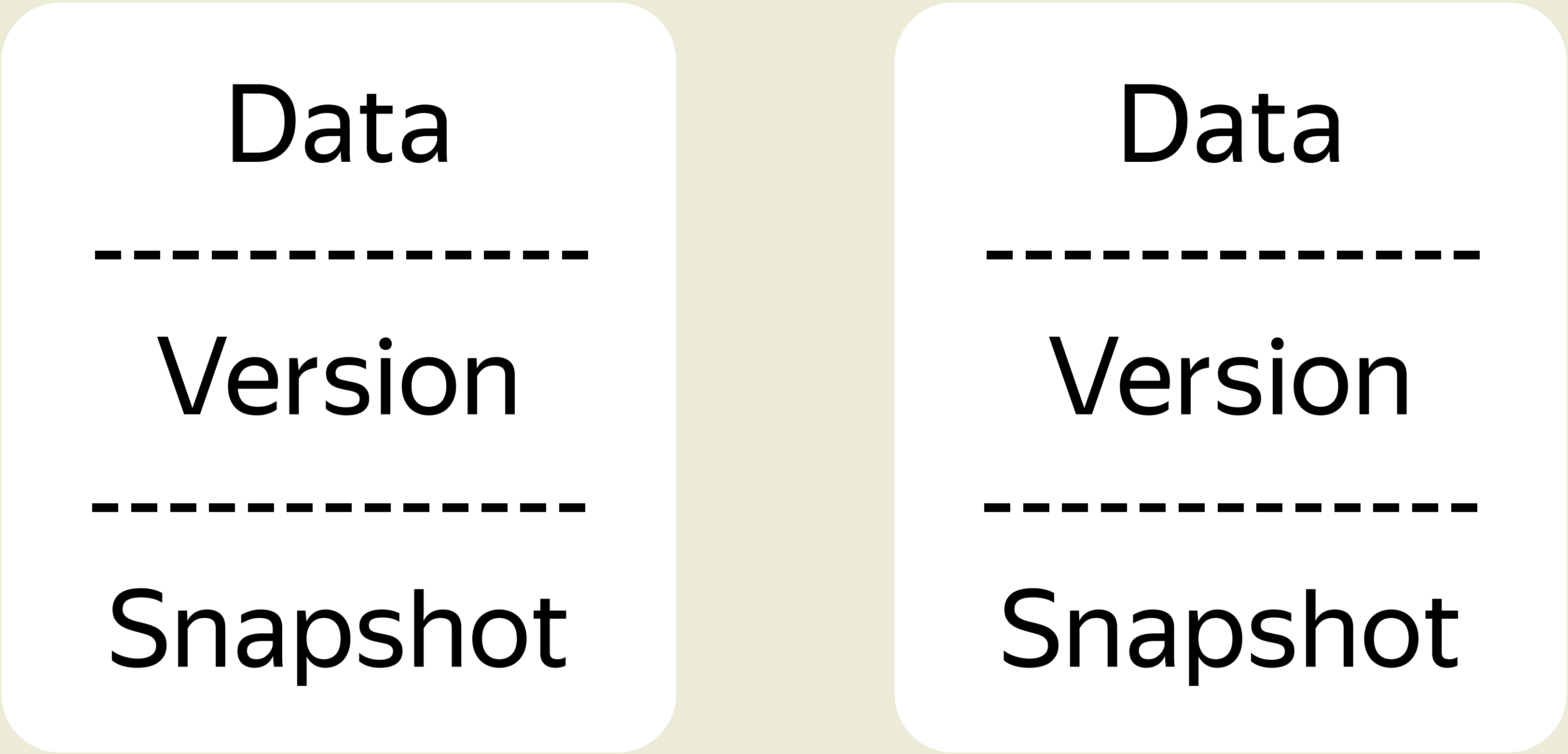
Data
Version
Snapshot

```
update(Data d) {
  Snapshot s = get_snapshot();
  write(d, v + 1, s);
}
```

get_snapshot():



Регистры SWMR WAIT-FREE



Data

Version

Snapshot

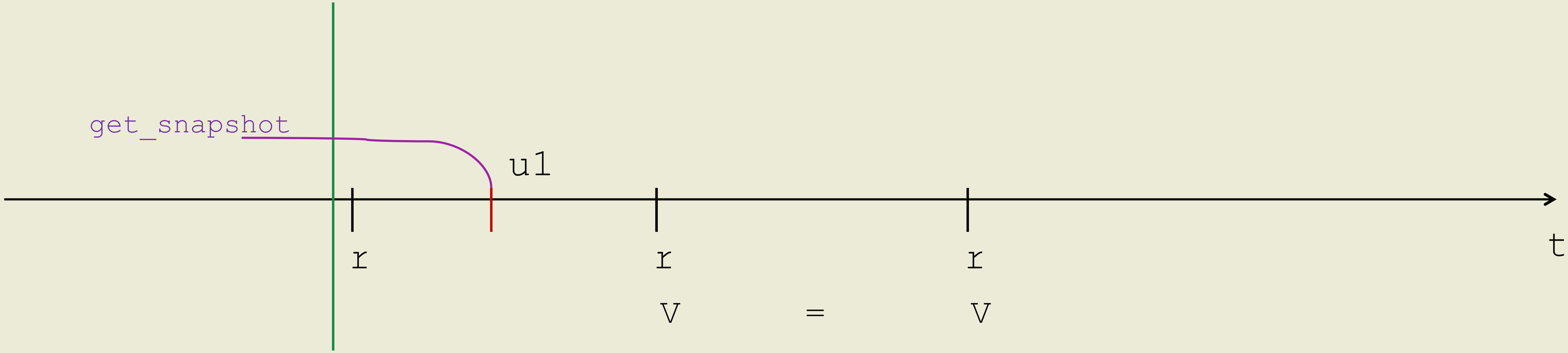
Data

Version

Snapshot

```
update(Data d) {
  Snapshot s = get_snapshot();
  write(d, v + 1, s);
}
```

get_snapshot():



Регистры SWMR WAIT-FREE

Data

Version

Snapshot

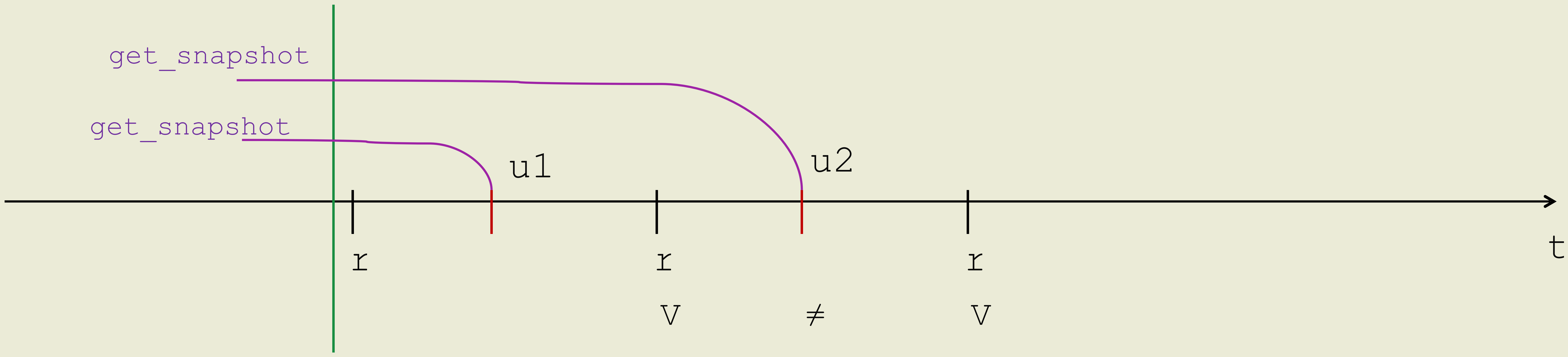
Data

Version

Snapshot

```
update(Data d) {
  Snapshot s = get_snapshot();
  write(d, v + 1, s);
}
```

get_snapshot():



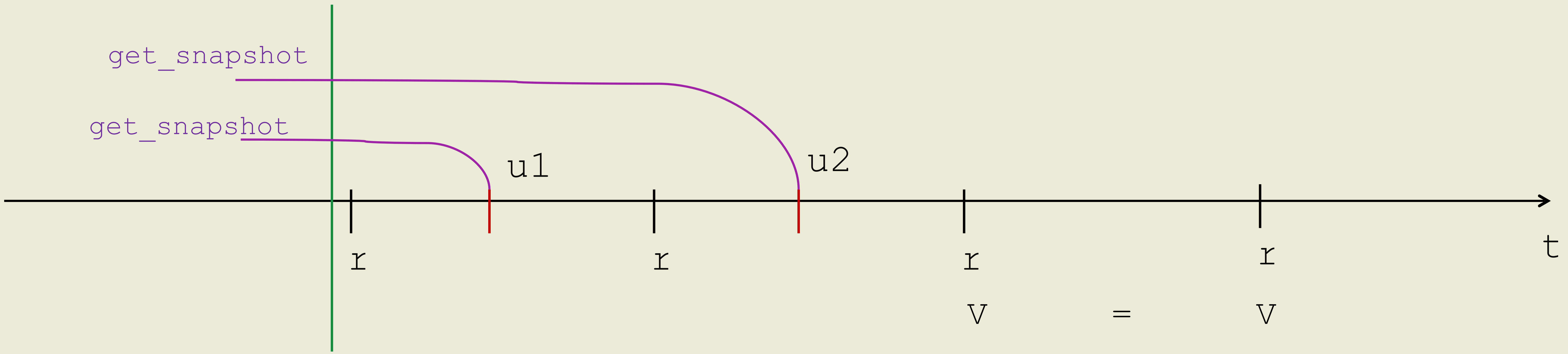
Регистры SWMR WAIT-FREE

Data
Version
Snapshot

Data
Version
Snapshot

```
update(Data d) {
  Snapshot s = get_snapshot();
  write(d, v + 1, s);
}
```

get_snapshot():



Регистры SWMR WAIT-FREE

Data

Version

Snapshot

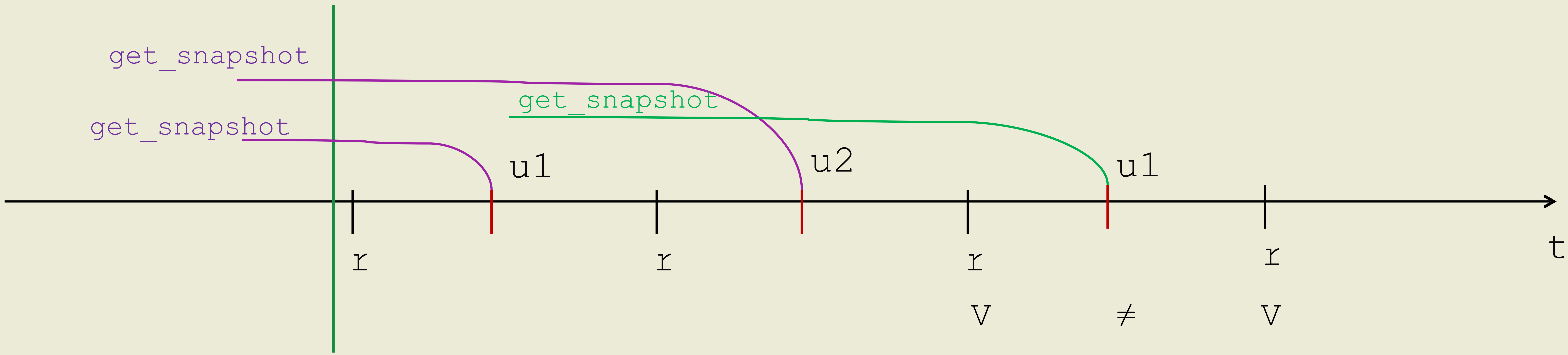
Data

Version

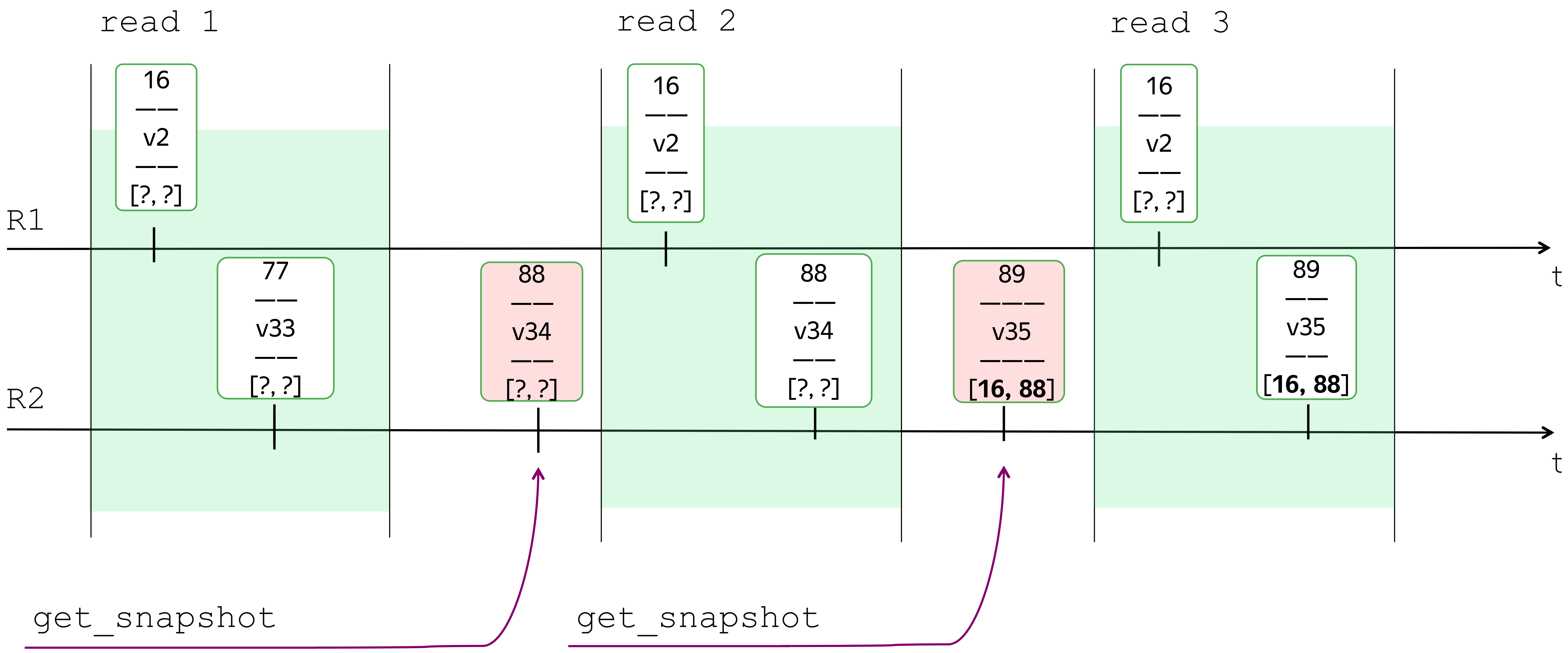
Snapshot

```
update(Data d) {
  Snapshot s = get_snapshot();
  write(d, v + 1, s);
}
```

get_snapshot():



Регистры SWMR WAIT-FREE



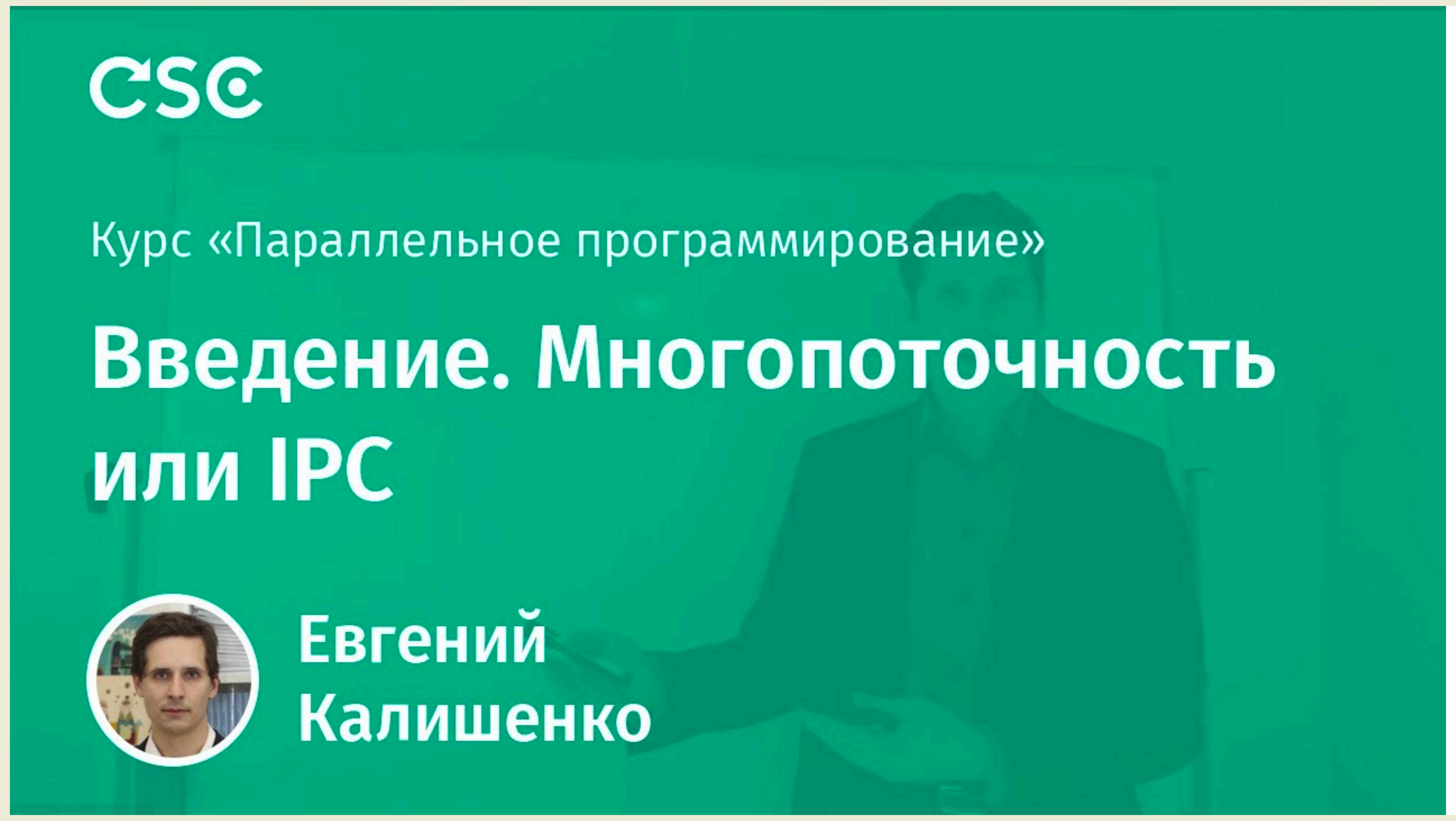
Конец

Параллельное программирование на С++ в действии

Энтони Уильямс

Практика разработки многопоточных программ

ДМК MANNING

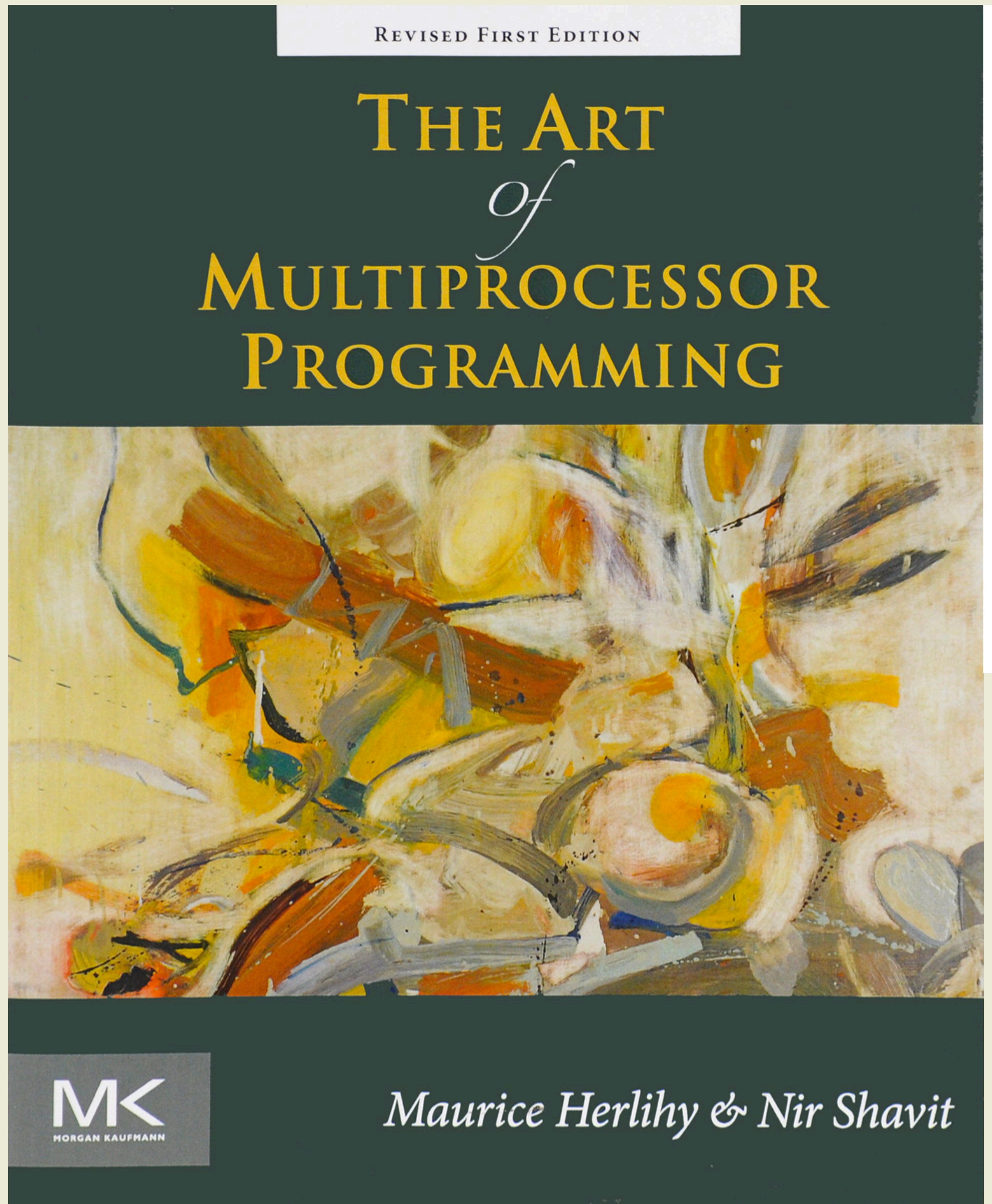


CSC

Курс «Параллельное программирование»

Введение. Многопоточность или IPC

Евгений Калишенко



Revised First Edition

The Art of Multiprocessor Programming

MK Morgan Kaufmann

Maurice Herlihy & Nir Shavit