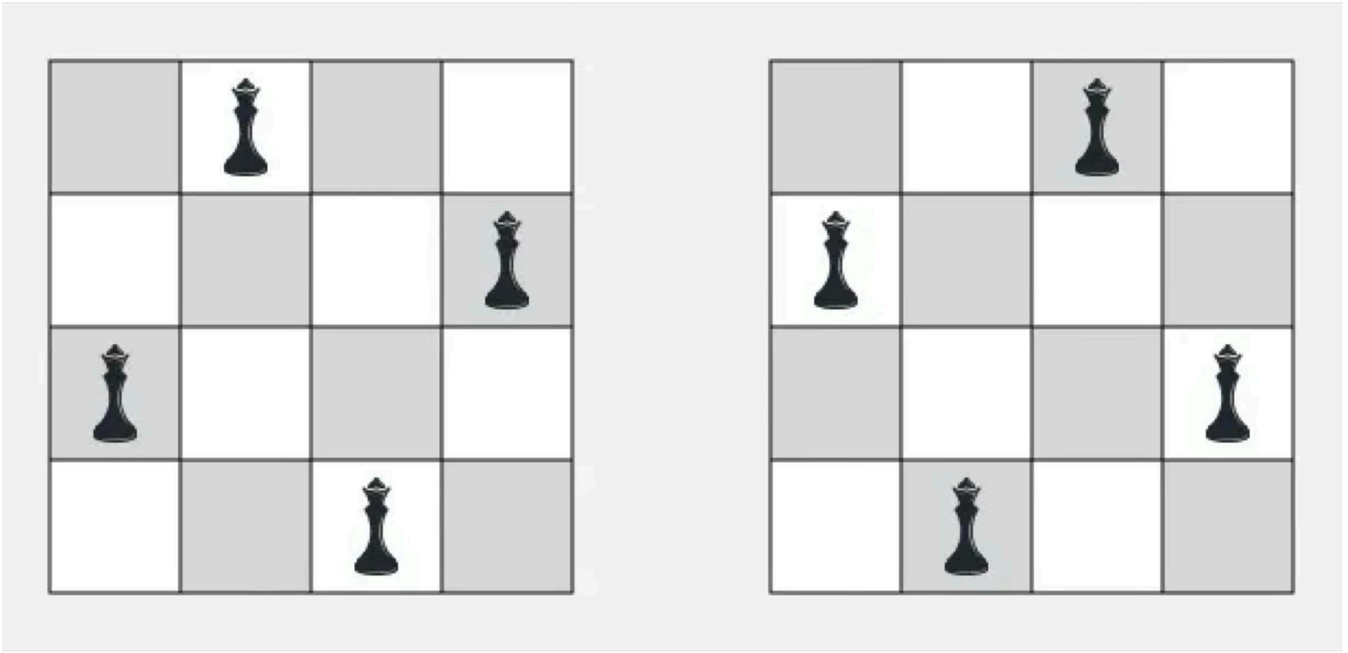
Virtualizing Continuations

Cong Ma, Max Jung, Yizhou Zhang
University of Waterloo

Background

Effect handlers and **multishot continuations** are powerful language features.



Motivating Example

NQueens

```
def safe(col, partial_sol) =
  ...

def place(size: int, row: int, partial_sol: int list): int =
  if row == size
    1
  else
    acc = 0
    for col in 0..size
      new_solution = col :: partial_sol
      if not safe(new_solution)
        continue
      acc += place(size, row + 1, new_solution)
    acc

def nqueens(n): int =
    place(n, 0, [])
```

First flavor

Two Flavors of NQueens

```
def safe(col, partial_sol) =
  ...

def place(size: int, row: int, partial_sol: int list): int =
  if row == size
    1
  else
    acc = 0
    for col in 0..size
      new_solution = col :: partial_sol
      if not safe(new_solution)
        continue
      acc += place(size, row + 1, new_solution)
    acc

def nqueens(n): int =
    place(n, 0, [])
```

First flavor

Two Flavors of NQueens

```
def safe(col, partial_sol) =
  ...

def place(size: int, row: int, partial_sol: int list): int =
  if row == size
    1
  else
    acc = 0
    for col in 0..size
      new_solution = col :: partial_sol
      if not safe(new_solution)
        continue
      acc += place(size, row + 1, new_solution)
    acc

def nqueens(n): int =
    place(n, 0, [])
```

First flavor

Two Flavors of NQueens

```
def place(s, r, part): int =
  if r == s
    1
  else
    acc = 0
    for c in 0..s
      if not safe(c::part)
        continue
      acc += place(s, r + 1, c::part)
    acc

def nqueens(n): int =
    place(n, 0, [])
```

First flavor

Two Flavors of NQueens

```
def place(s, r, part): int =
  if r == s
    1
  else
    acc = 0
    for c in 0..s
      if not safe(c::part)
        continue
      acc += place(s, r + 1, c::part)
    acc

def nqueens(n): int =
    place(n, 0, [])
```

First flavor

```
def place(s, r, part)
      :unit / {Fail, Choose} =
  if r == s
    ()
  else
    c = perform Choose(0..s)
    if not safe(c::part)
      perform Fail()
    place(s, r + 1, c::part)
```

Second flavor

Two Flavors of NQueens

```
def place(s, r, part): int =
  if r == s
    1
  else
    acc = 0
    for c in 0..s
      if not safe(c::part)
        continue
      acc += place(s, r + 1, c::part)
    acc

def nqueens(n): int =
    place(n, 0, [])
```

First flavor

```
def place(s, r, part)
      :unit / {Fail, Choose} =
  if r == s
    ()
  else
    c = perform Choose(0..s)
    if not safe(c::part)
      perform Fail()
    place(s, r + 1, c::part)
```

Second flavor

Two Flavors of NQueens

```
def place(s, r, part): int =
  if r == s
    1
  else
    acc = 0
    for c in 0..s
      if not safe(c::part)
        continue
      acc += place(s, r + 1, c::part)
    acc

def nqueens(n): int =
    place(n, 0, [])
```

First flavor

```
def place(s, r, part)
      :unit / {Fail, Choose} =
  if r == s
    ()
  else
    c = perform Choose(0..s)
    if not safe(c::part)
      perform Fail()
    place(s, r + 1, c::part)
```

- **Abstract away search strategy**

Second flavor

Two Flavors of NQueens

```
def place(s, r, part)
      :unit / {Fail, Choose} =
  if r == s
    ()
  else
    c = perform Choose(0..s)
    if not safe(c::part)
      perform Fail()
    place(s, r + 1, c::part)
```

- **Abstract away search strategy**

Second flavor

```
def nqueens(n): int =
  handle
    place(n, 0, [])
  with
  | () ⇒ 1
  | Fail(k) ⇒ 0
  | Choose(choices, k) ⇒
    acc = 0
    for choice in choices
      acc += resume k(choice)
    acc
```

- **Concretize search strategy**

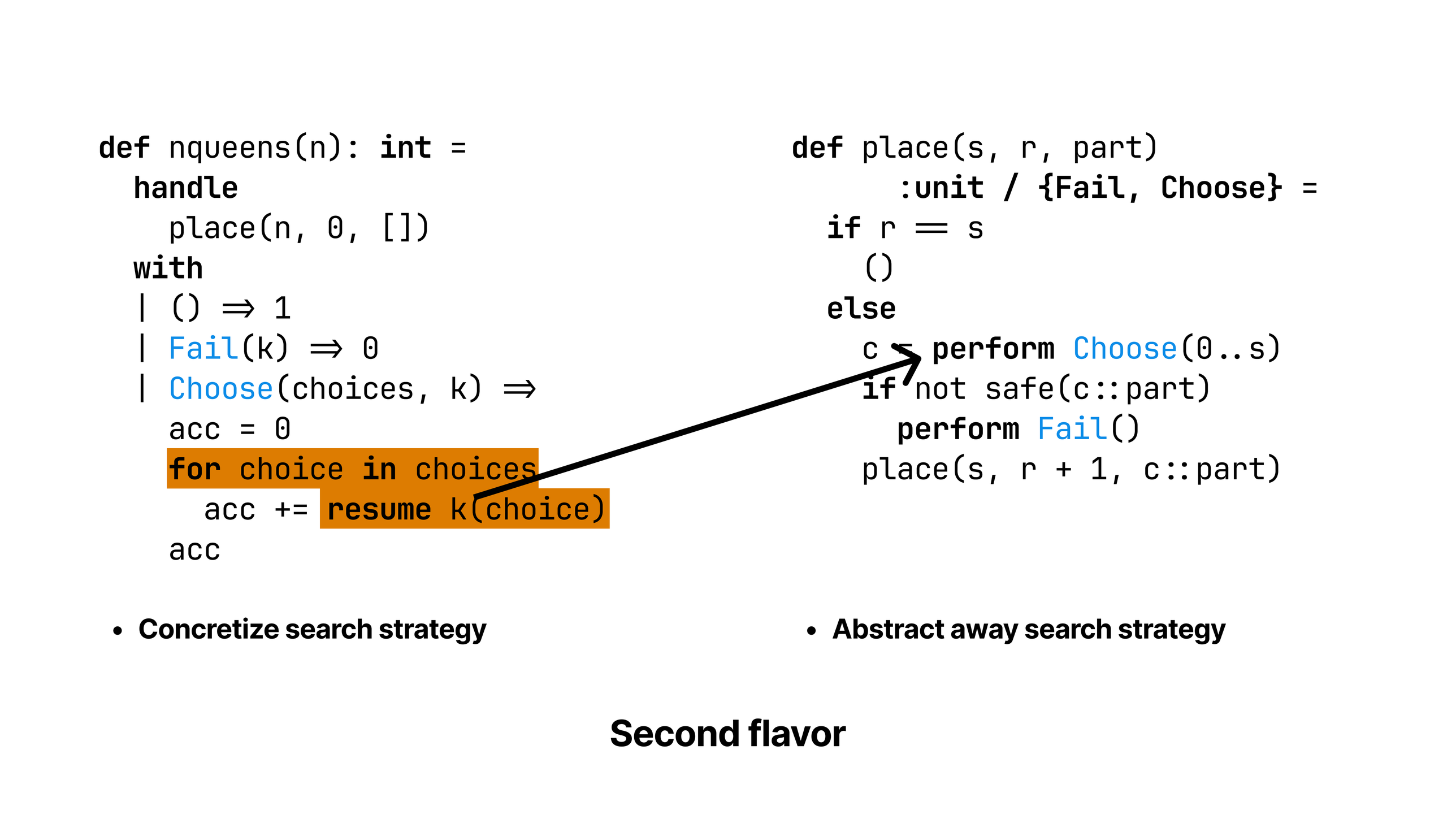
```
def place(s, r, part)
      :unit / {Fail, Choose} =
  if r == s
    ()
  else
    c = perform Choose(0..s)
    if not safe(c::part)
      perform Fail()
    place(s, r + 1, c::part)
```

- **Abstract away search strategy**

Second flavor

```
def nqueens(n): int =
  handle
    place(n, 0, [])
  with
  | () ⇒ 1
  | Fail(k) ⇒ 0
  | Choose(choices, k) ⇒
    acc = 0
    for choice in choices
      acc += resume k(choice)
    acc
```

```
def place(s, r, part)
      :unit / {Fail, Choose} =
  if r == s
    ()
  else
    c = perform Choose(0..s)
    if not safe(c::part)
      perform Fail()
    place(s, r + 1, c::part)
```



- **Concretize search strategy**
- **Abstract away search strategy**

Second flavor

```
def nqueens(n): int =
  handle
    place(n, 0, [])
  with
  | () ⇒ 1
  | Fail(k) ⇒ 0
  | Choose(choices, k) ⇒
    acc = 0
    for choice in choices
      acc += resume k(choice)
    acc
```

```
def place(s, r, part)
      :unit / {Fail, Choose} =
  if r == s
    ()
  else
    c = perform Choose(0..s)
    if not safe(c::part)
      perform Fail()
    place(s, r + 1, c::part)
```

- **Concretize search strategy**

Modularity

Second flavor

- **Abstract away search strategy**

```
def nqueens(n): int =
  handle
    place(n, 0, [])
  with
  | () ⇒ 1
  | Fail(k) ⇒ 0
  | Choose(choices, k) ⇒
    acc = 0
    for choice in choices
      acc += resume k(choice)
    acc
```

```
def place(s, r, part)
      :unit / {Fail, Choose} =
  if r == s
    ()
  else
    c = perform Choose(0..s)
    if not safe(c::part)
      perform Fail()
    place(s, r + 1, c::part)
```

Second flavor

```
def nqueens(n): int =
  handle
    place(n, 0, [])
  with
  | () ⇒ 1
  | Fail(k) ⇒ 0
  | Choose(choices, k) ⇒
    acc = 0
    for choice in choices
      acc += resume k(choice)
    acc
```

```
def place(s, r, part)
      :unit / {Fail, Choose} =
  if r == s
    ()
  else
    c = perform Choose(0..s)
    if not safe(c::part)
      perform Fail()
    place(s, r + 1, c::part)
```

Multishot Continuation

Second flavor

Short Digression to Semantics

Dynamically Scoped Handlers

When an effect is raised, the closest enclosing handler at the run time is used.

Java, Koka, OCaml

Lexically Scoped Handlers

When control enters a handler scope, a handler capability is generated. It is passed down the stack and used at raise site.

Effekt, Lexa

Short Digression to Semantics

Lexically Scoped Handlers

When control enters a handler scope, a handler capability is generated. It is passed down the stack and used at raise site.

Effekt, Lexa

```
def nqueens(n): int =
  handle
    place(n, 0, [])
  with
  | () ⇒ 1
  | Fail(k) ⇒ 0
  | Choose(choices, k) ⇒
    acc = 0
    for choice in choices
      acc += resume k(choice)
    acc
```

Short Digression to Semantics

Lexically Scoped Handlers

When control enters a handler scope, a handler capability is generated. It is passed down the stack and used at raise site.

Effekt, Lexa

```
def nqueens(n): int =
  handle cap
    place(n, 0, [], cap)
  with
  | () ⇒ 1
  | Fail(k) ⇒ 0
  | Choose(choices, k) ⇒
    acc = 0
    for choice in choices
      acc += resume k(choice)
    acc
```

Short Digression to Semantics

Lexically Scoped Handlers

When control enters a handler scope, a handler capability is generated. It is passed down the stack and used at raise site.

Effekt, Lexa

```
def place(s, r, part)
      :unit / {Fail, Choose} =
  if r == s
    ()
  else
    c = perform Choose(0..s)
    if not safe(c::part)
      perform Fail()
    place(s ,r + 1, c::part)
```

Short Digression to Semantics

Lexically Scoped Handlers

When control enters a handler scope, a handler capability is generated. It is passed down the stack and used at raise site.

Effekt, Lexa

```
def place(s, r, part, cap)
      :unit / {Fail, Choose} =
  if r == s
    ()
  else
    c = perform cap.Choose(0..s)
    if not safe(c::part)
      perform cap.Fail()
    place(s ,r + 1, c::part, cap)
```

Short Digression to Semantics

Lexically Scoped Handlers

When control enters a handler scope, a handler capability is generated. It is passed down the stack and used at raise site.

Effekt, Lexa

```
def place(s, r, part, cap)
      :unit / {Fail, Choose} =
  if r == s
    ()
  else
    c = perform cap.Choose(0..s)
    if not safe(c::part)
      perform cap.Fail()
    place(s ,r + 1, c::part, cap)
```

Lexically scoped handlers support strong reasoning principles while preserving the expressive power of effect handlers.

Background

Effect handlers are a unifying control flow mechanism.

Effect Handler

Background

Effect handlers are a unifying control flow mechanism.

They introduce first-class continuations into a language.

Effect Handler

Continuation

Single-shot
Multi-shot

Background

Effect handlers are a unifying control flow mechanism.

Effect Handler

They introduce first-class continuations into a language.

Multi-shot continuations are very expressive.

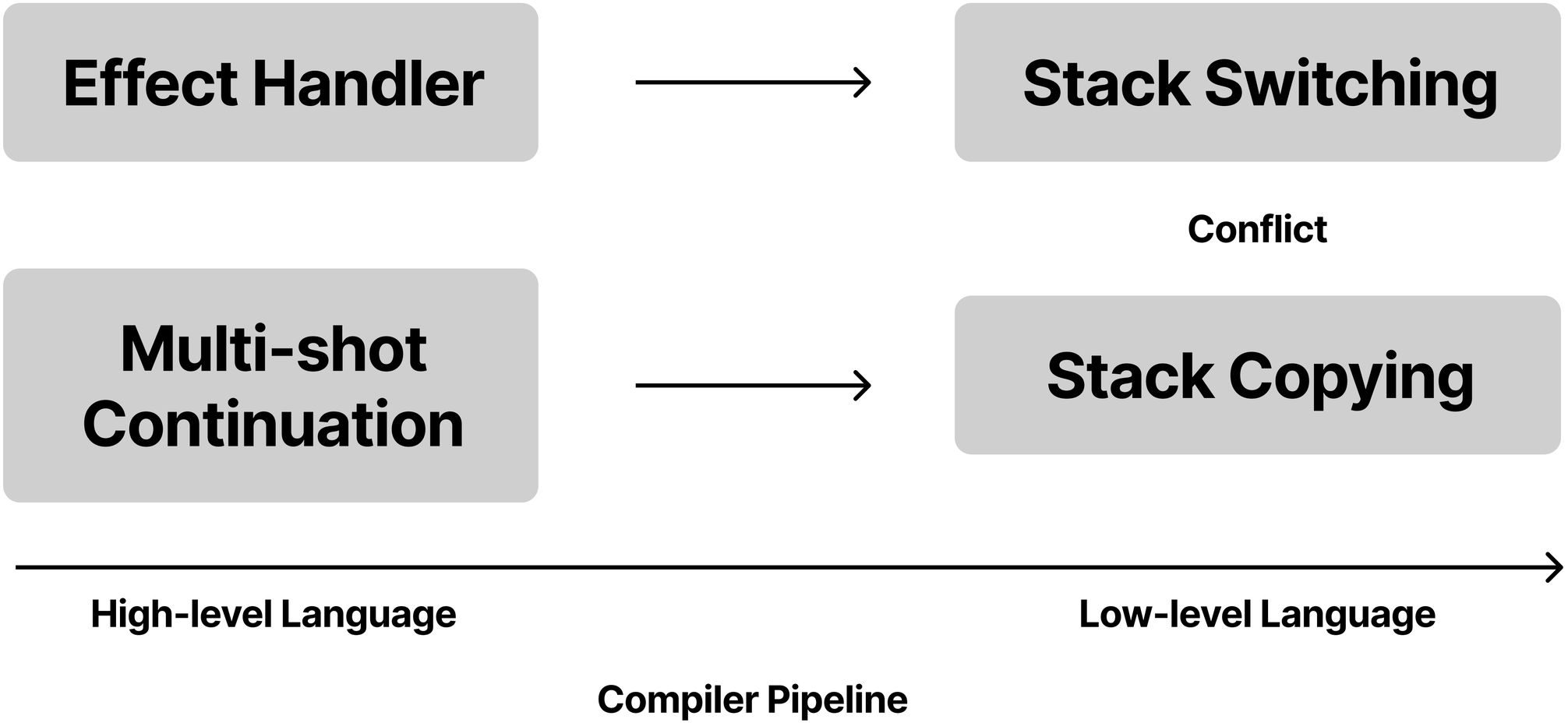
Multi-shot Continuation

Background

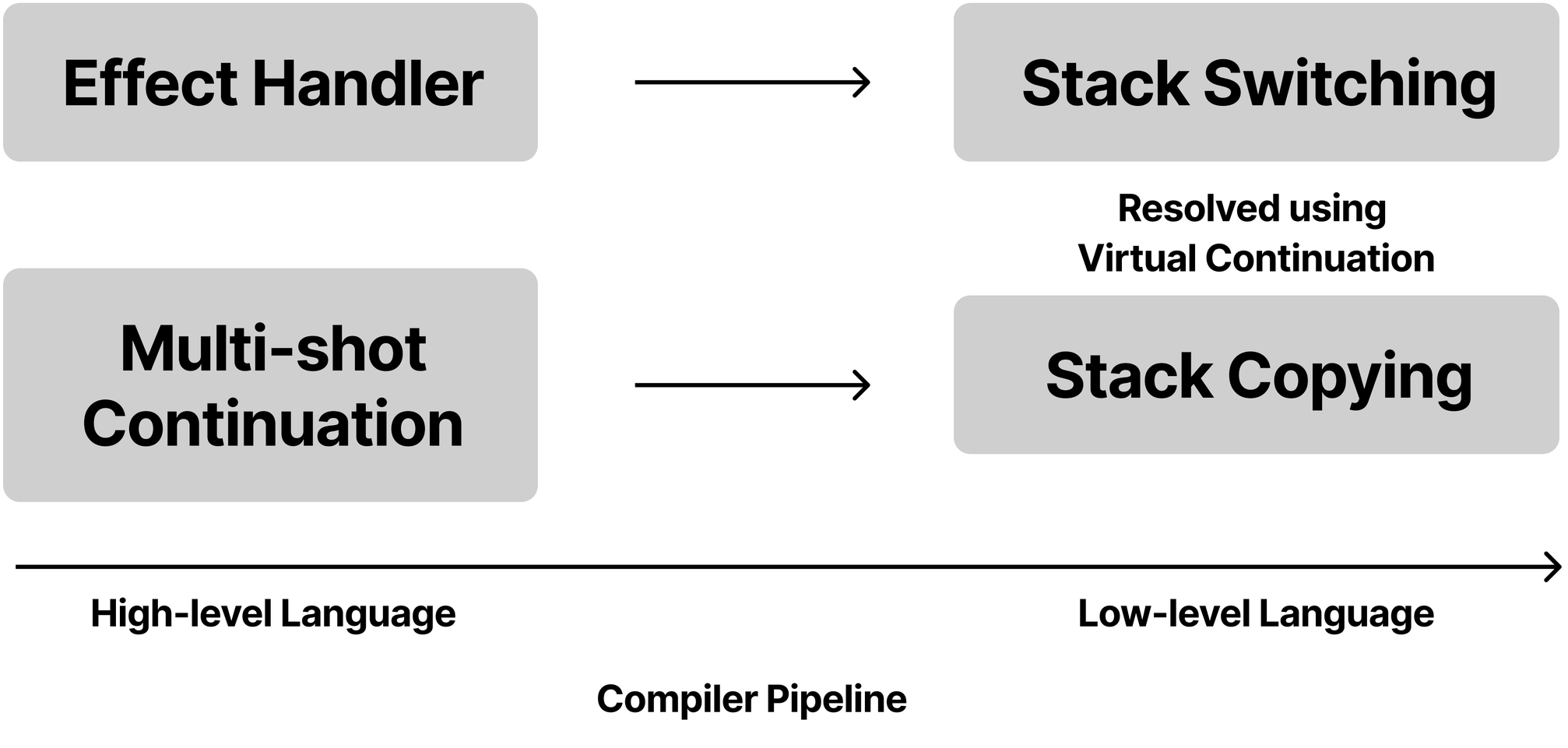
Effect Handler

Multi-shot Continuation

Background

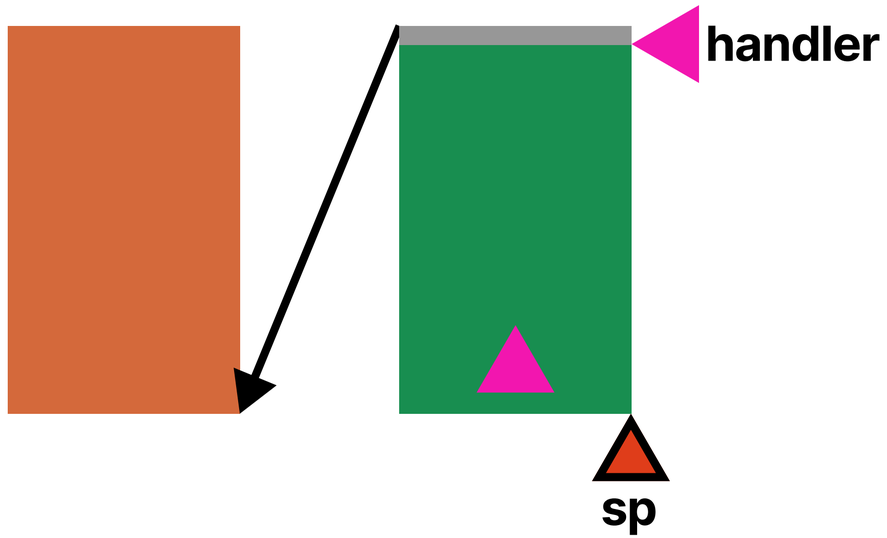

Effect Handler
Stack Switching
Conflict
Multi-shot Continuation
Stack Copying
High-level Language
Low-level Language
Compiler Pipeline

Background


Effect Handler
Stack Switching
Resolved using
Virtual Continuation
Multi-shot
Continuation
Stack Copying
High-level Language
Low-level Language
Compiler Pipeline

Stack Switching

Stack Copying



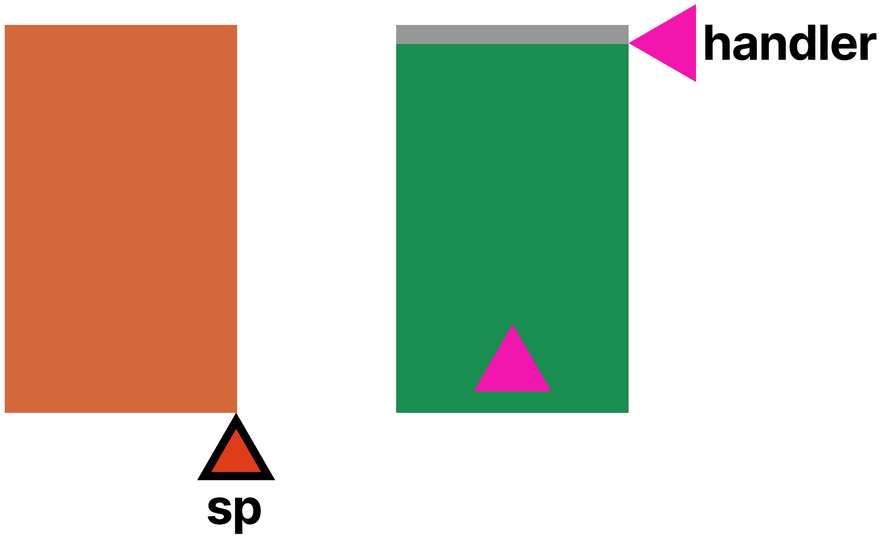
Stack Switching

Implementation of effect handlers

```
def nqueens(n): int =
  handle cap
    place(n, 0, [], cap)
  with
  | () ⇒ 1
  | Fail(k) ⇒ 0
  | Choose(choices, k) ⇒
    acc = 0
    for choice in choices
      acc += resume k(choice)
    acc

def place(size, r, part, cap)
      :unit / {Fail, Choose} =
  if r == size
    ()
  else
    c = perform cap.Choose(0..size)
    if not safe(c, part)
      perform cap.Fail()
    place(row + 1, c::part, cap)
```

ip



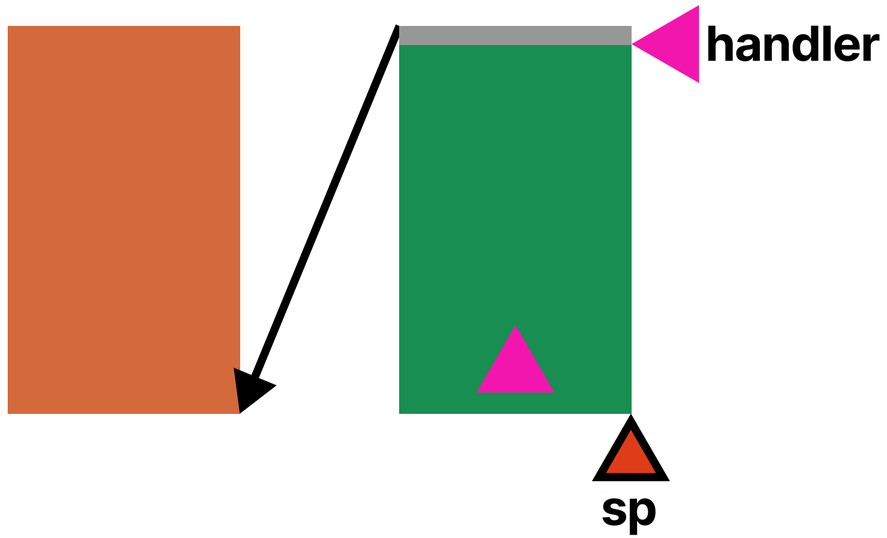
Stack Switching

Implementation of effect handlers

```
def nqueens(n): int =
  handle cap
    place(n, 0, [], cap)
  with
  | () ⇒ 1
  | Fail(k) ⇒ 0
  | Choose(choices, k) ⇒
    acc = 0
    for choice in choices
      acc += resume k(choice)
    acc

def place(size, r, part, cap)
      :unit / {Fail, Choose} =
  if r == size
    ()
  else
    c = perform cap.Choose(0..size)
    if not safe(c, part)
      perform cap.Fail()
    place(row + 1, c::part, cap)
```

ip



Stack Switching

Implementation of effect handlers

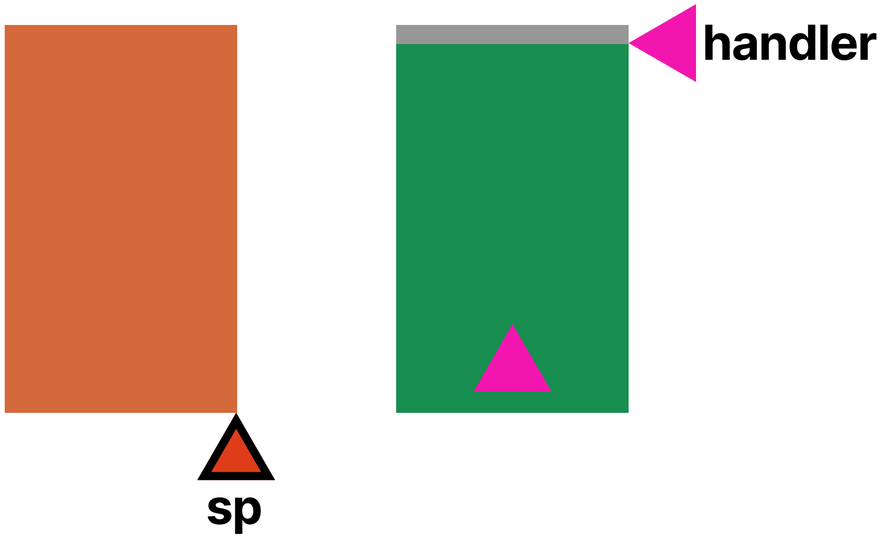
```
def nqueens(n): int =
  handle cap
    place(n, 0, [], cap)
  with
  | () ⇒ 1
  | Fail(k) ⇒ 0
  | Choose(choices, k) ⇒
    acc = 0
    for choice in choices
      acc += resume k(choice)
    acc

def place(size, r, part, cap)
      :unit / {Fail, Choose} =
  if r == size
    ()
  else
    c = perform cap.Choose(0..size)
    if not safe(c, part)
      perform cap.Fail()
    place(row + 1, c::part, cap)
```

ip

Stack Copying

Implementation of multishot continuations



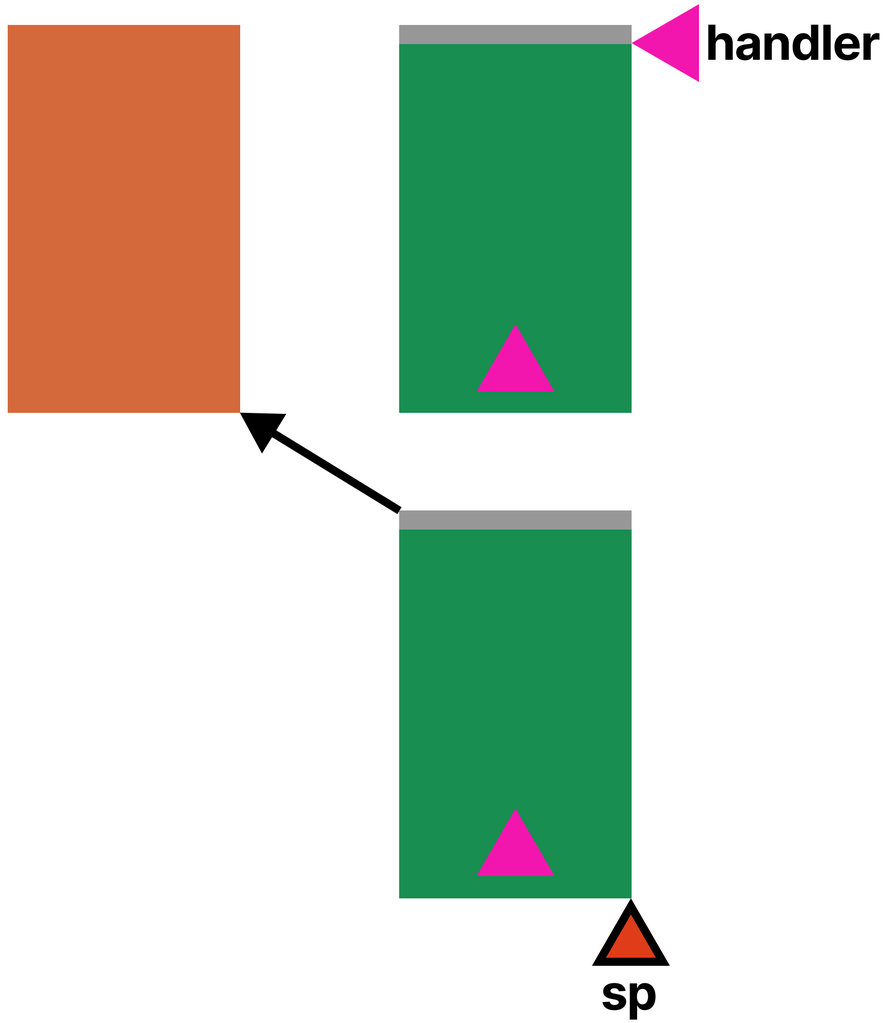
Stack Copying

Implementation of multishot continuations

```
def nqueens(n): int =
  handle cap
    place(n, 0, [], cap)
  with
  | () ⇒ 1
  | Fail(k) ⇒ 0
  | Choose(choices, k) ⇒
    acc = 0
    for choice in choices
      acc += resume k(choice)
    acc

def place(size, r, part, cap)
      :unit / {Fail, Choose} =
  if r == size
    ()
  else
    c = perform cap.Choose(0..size)
    if not safe(c, part)
      perform cap.Fail()
    place(row + 1, c::part, cap)
```

ip

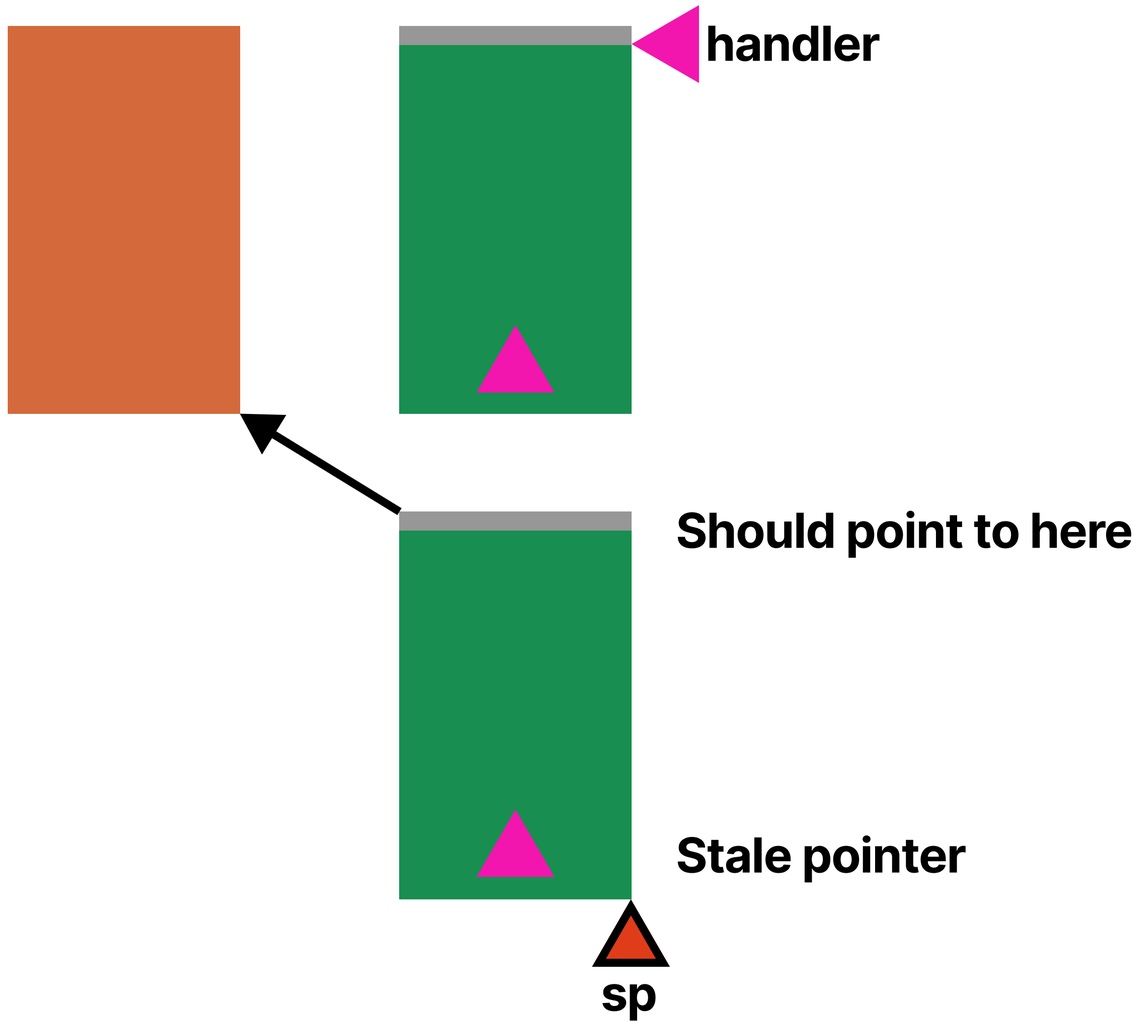


Stack Copying

Implementation of multishot continuations

```
def nqueens(n): int =
  handle cap
    place(n, 0, [], cap)
  with
  | () ⇒ 1
  | Fail(k) ⇒ 0
  | Choose(choices, k) ⇒
    acc = 0
    for choice in choices
      acc += resume k(choice)
    acc

def place(size, r, part, cap)
      :unit / {Fail, Choose} =
  if r == size
    ()
  else
    c = perform cap.Choose(0..size)
    if not safe(c, part)
      perform cap.Fail()
    place(row + 1, c::part, cap)
```



Stack Copying

Implementation of multishot continuations

```
def nqueens(n): int =
  handle cap
    place(n, 0, [], cap)
  with
  | () ⇒ 1
  | Fail(k) ⇒ 0
  | Choose(choices, k) ⇒
    acc = 0
    for choice in choices
      acc += resume k(choice)
    acc

def place(size, r, part, cap)
      :unit / {Fail, Choose} =
  if r == size
    ()
  else
    c = perform cap.Choose(0..size)
    if not safe(c, part)
      perform cap.Fail()
    place(row + 1, c::part, cap)
```

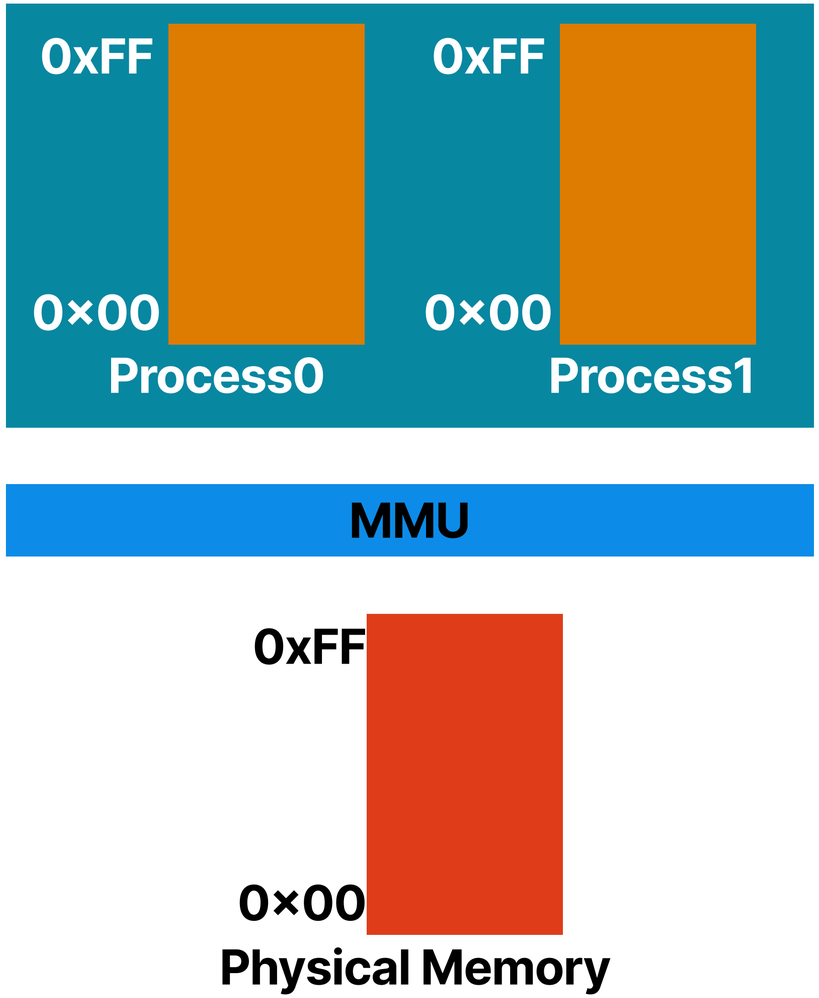
Challenge

The **stack switching** technique refers to handlers using stack addresses, which is incompatible with **stack copying**.

Idea

We borrow the idea of virtual memory management.

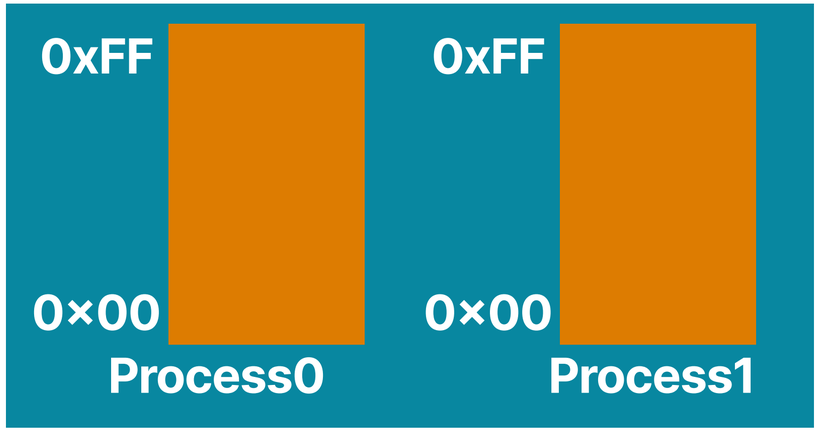
MMU offers address indirection.

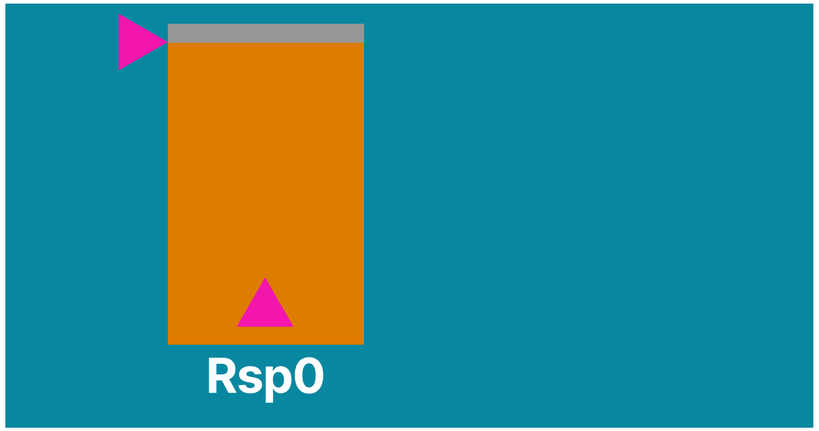


We borrow the idea of virtual memory management.

MMU offers address indirection.

Idea

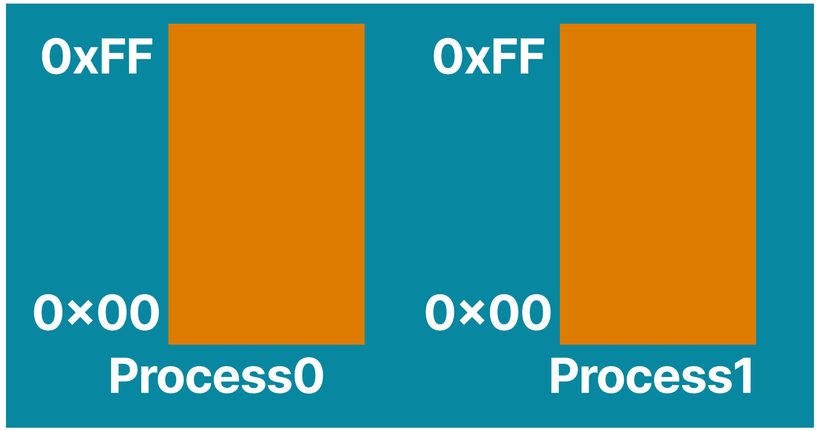


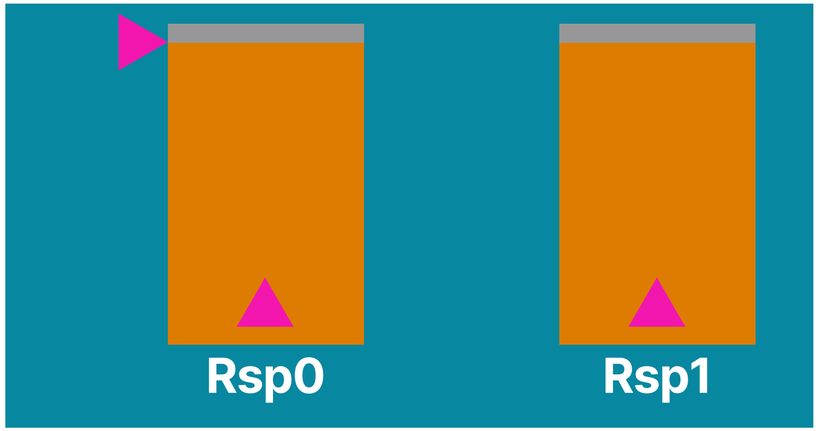


Idea

We borrow the idea of virtual memory management.

MMU offers address indirection.



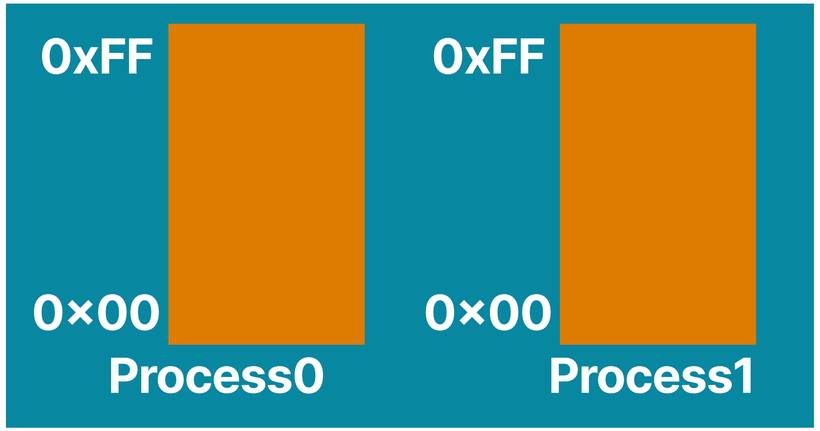


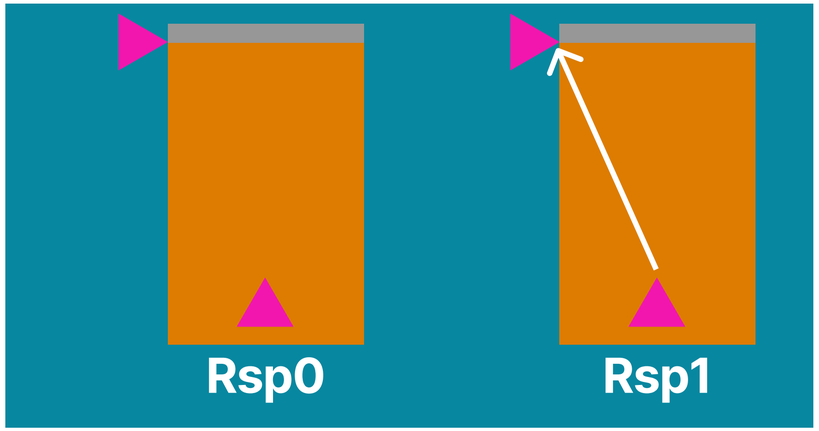
Stack addresses also need indirection.

Idea

We borrow the idea of virtual memory management.

MMU offers address indirection.

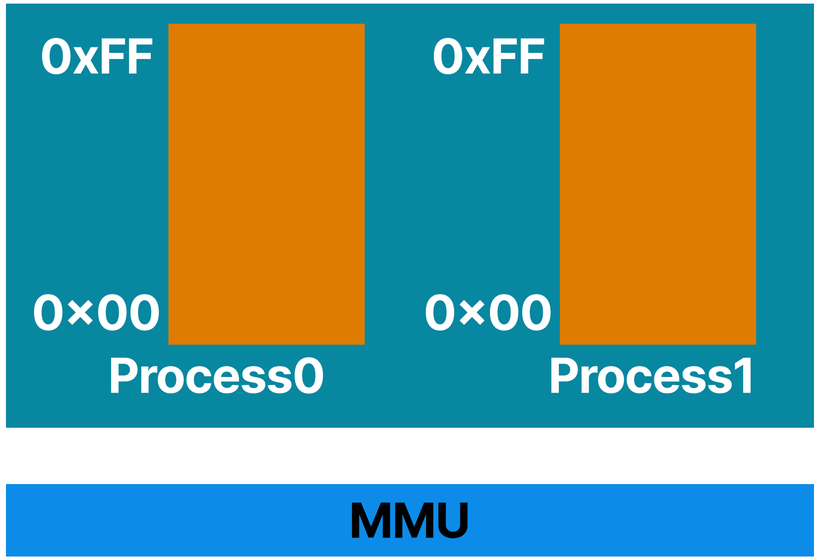


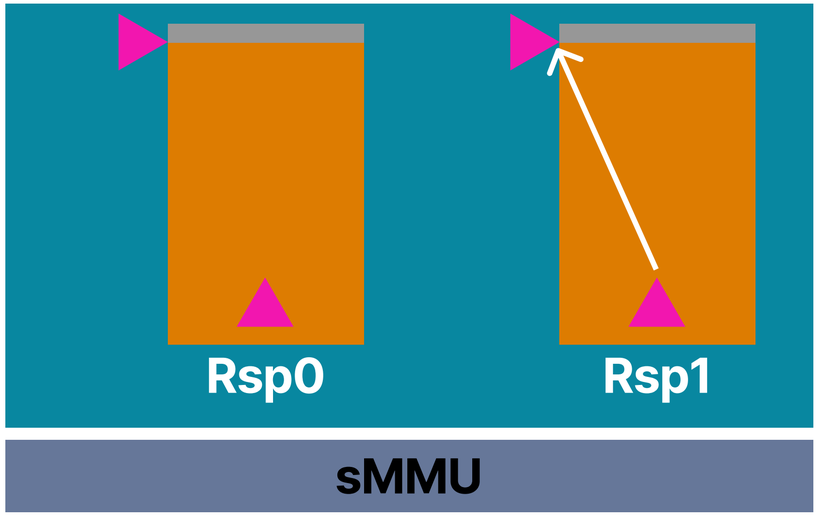


Stack addresses also need indirection.

We borrow the idea of virtual memory management.

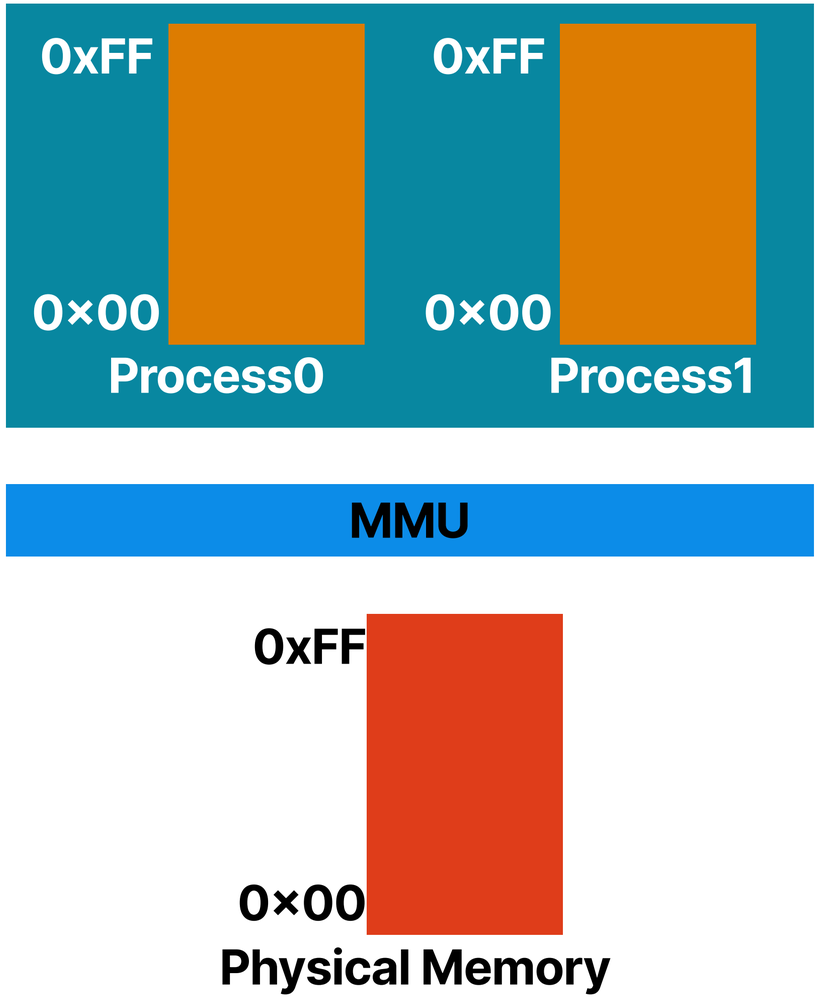
Idea

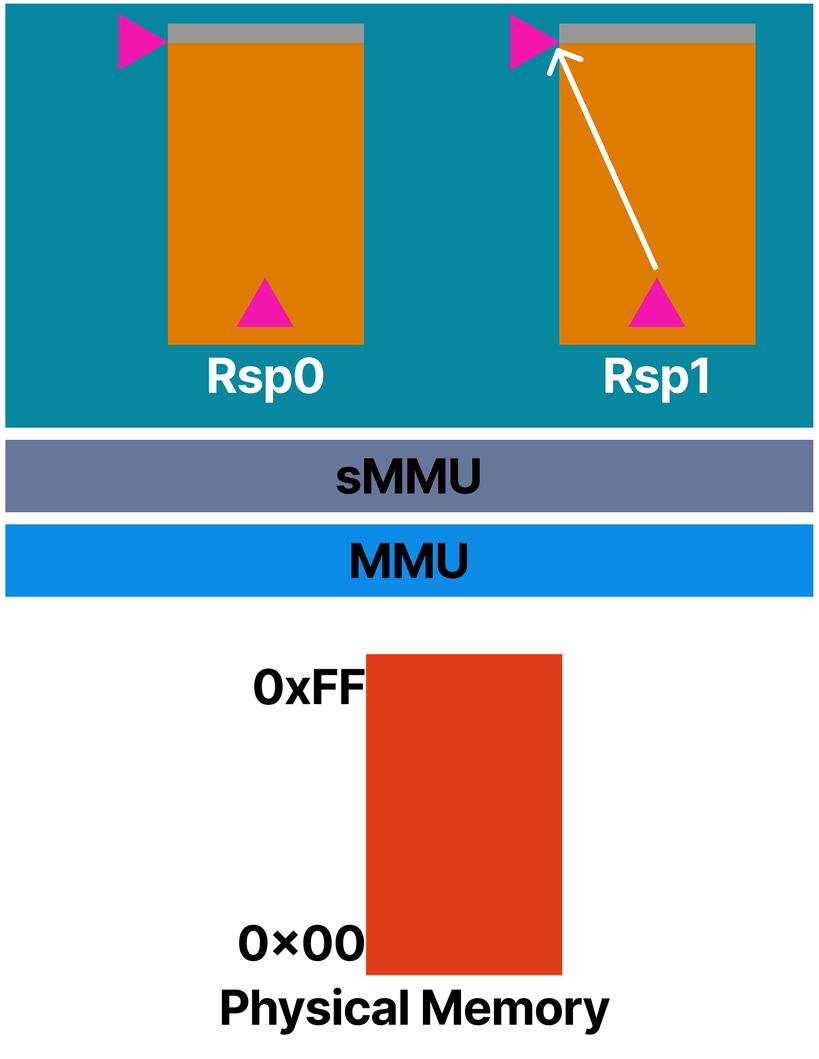




We borrow the idea of virtual memory management.

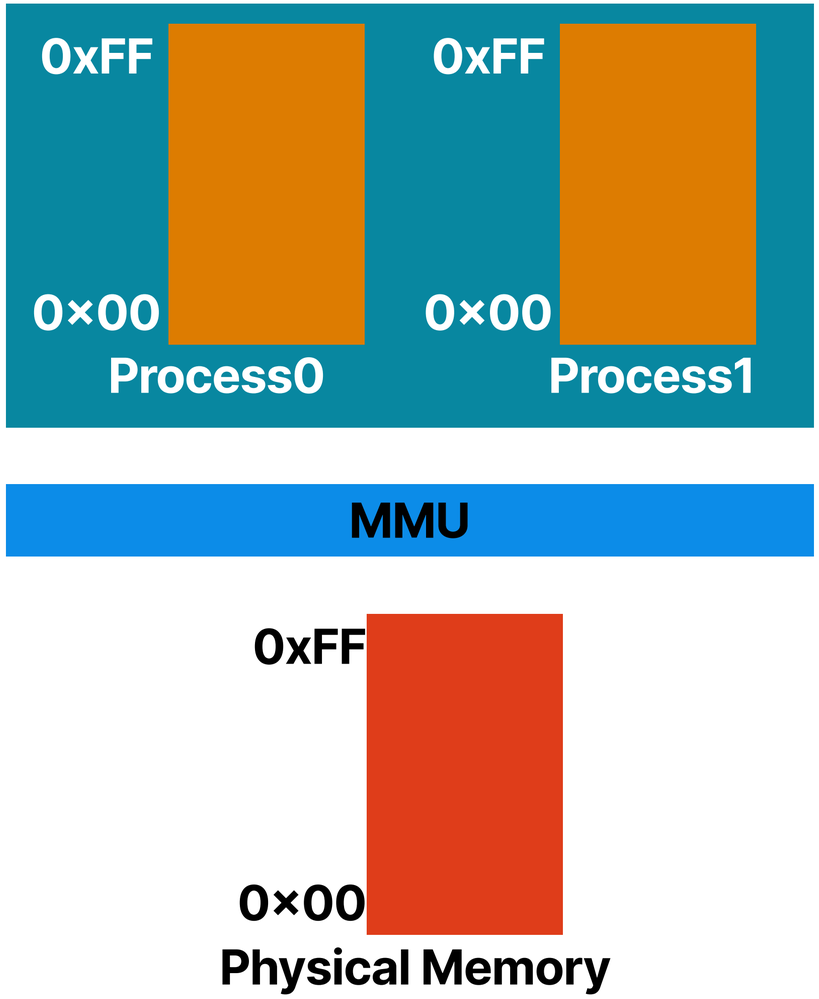
Idea

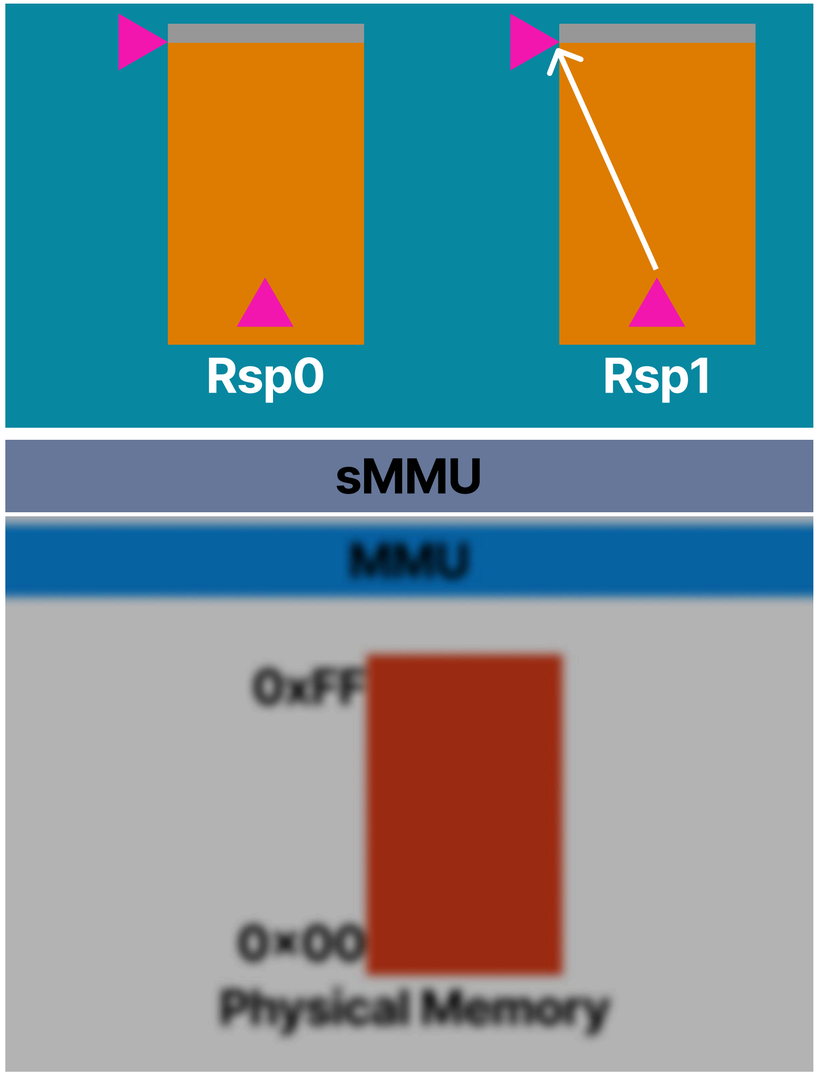




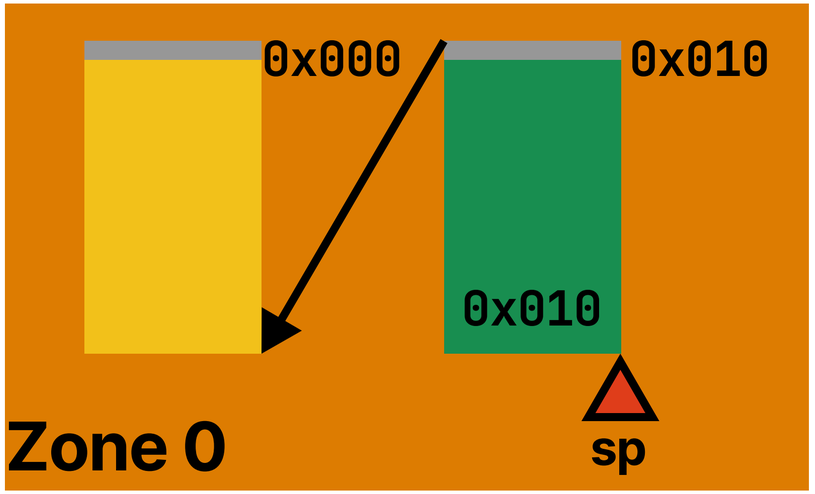
We borrow the idea of virtual memory management.

Idea





Virtualizing Continuations



0x73A

Zone number

Handler ID

Stack Offset

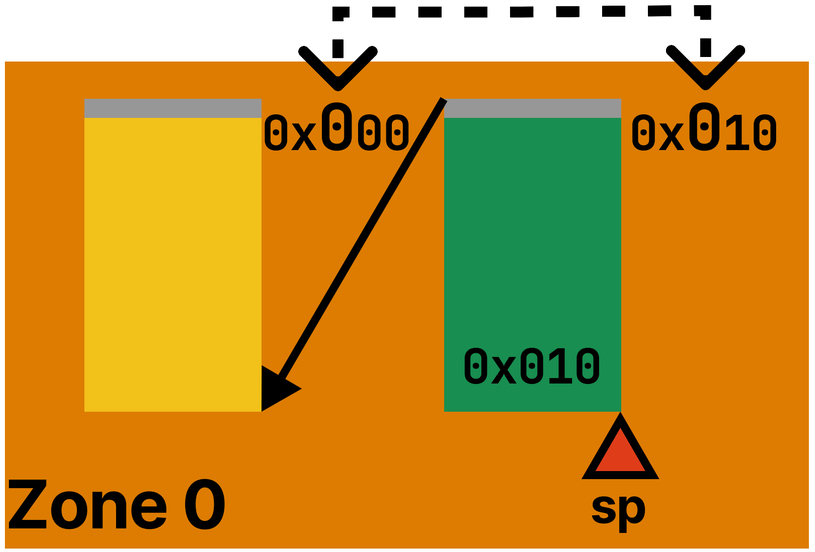
Virtual Addresses

Stacks and handlers live in virtual memory space, and all addresses are virtual addresses

Zoned Memory

Each continuation lives in a different zone; copies of the same handlers in different continuation live at the same relative address within zones

Virtualizing Continuations



0x73A

Zone number

Handler ID

Stack Offset

Virtual Addresses

Stacks and handlers live in virtual memory space, and all addresses are virtual addresses

Zoned Memory

Each continuation lives in a different zone; copies of the same handlers in different continuation live at the same relative address within zones

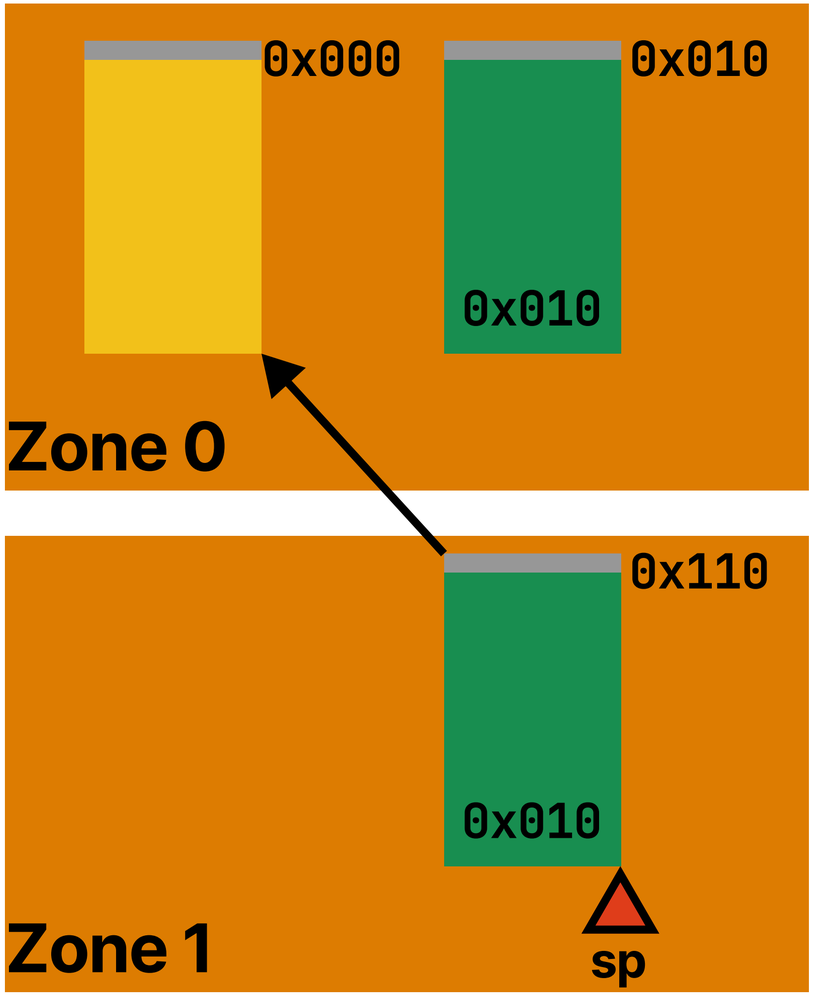
Virtualizing Continuations

0x73A

Zone number

Handler ID

Stack Offset



Virtual Addresses

Stacks and handlers live in virtual memory space, and all addresses are virtual addresses

Zoned Memory

Each continuation lives in a different zone; copies of the same handlers in different continuation live at the same relative address within zones

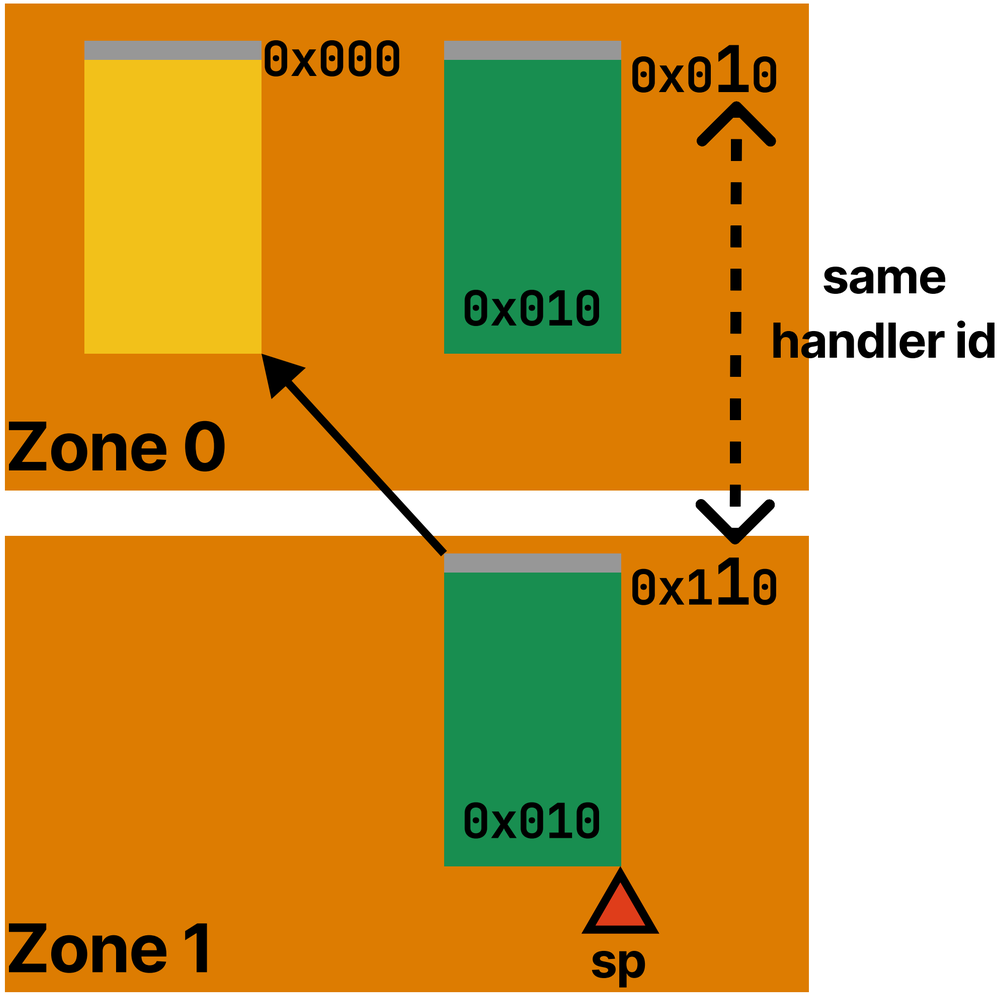
Virtualizing Continuations

0x73A

Zone number

Handler ID

Stack Offset



Virtual Addresses

Stacks and handlers live in virtual memory space, and all addresses are virtual addresses

Zoned Memory

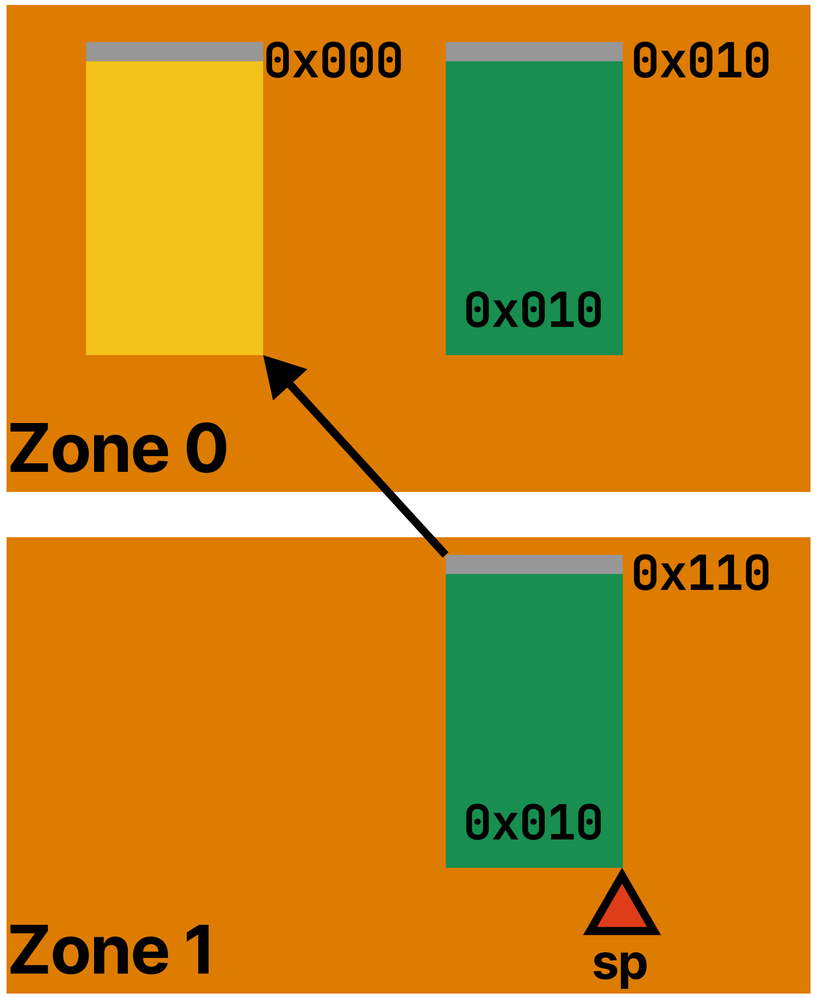
Each continuation lives in a different zone; copies of the same handlers in different continuation live at the same relative address within zones

Zone number

Handler ID

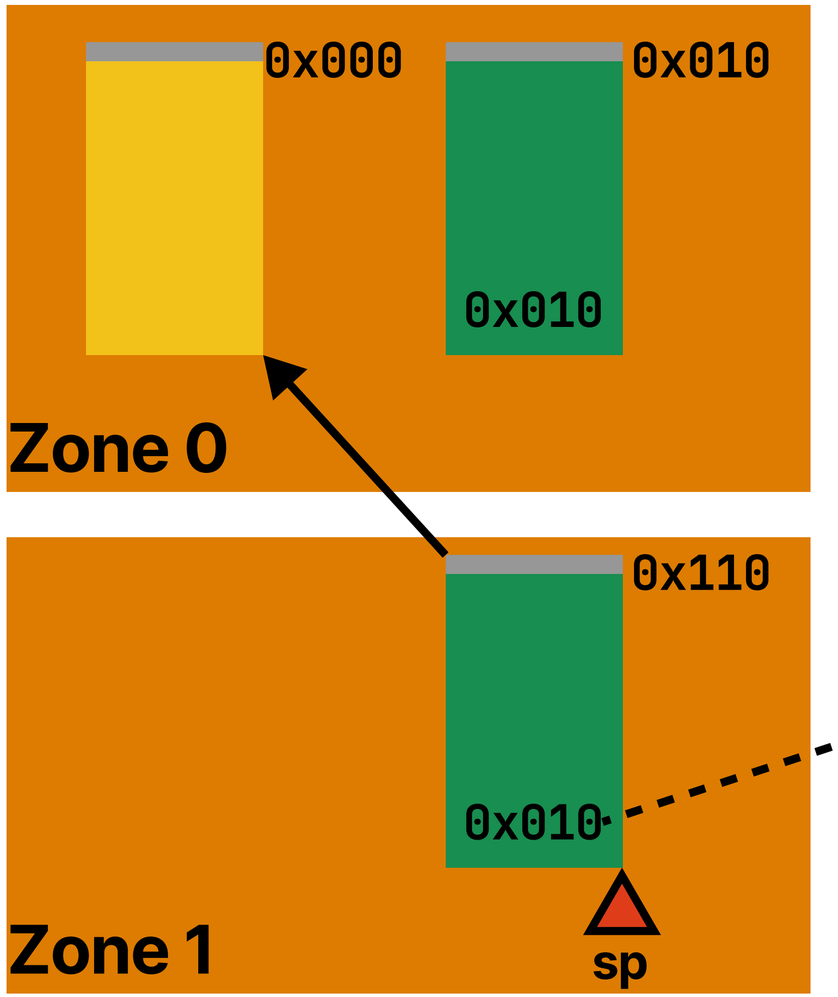
Stack Offset

Address Translation



Zone number
Handler ID
Stack Offset

Address Translation

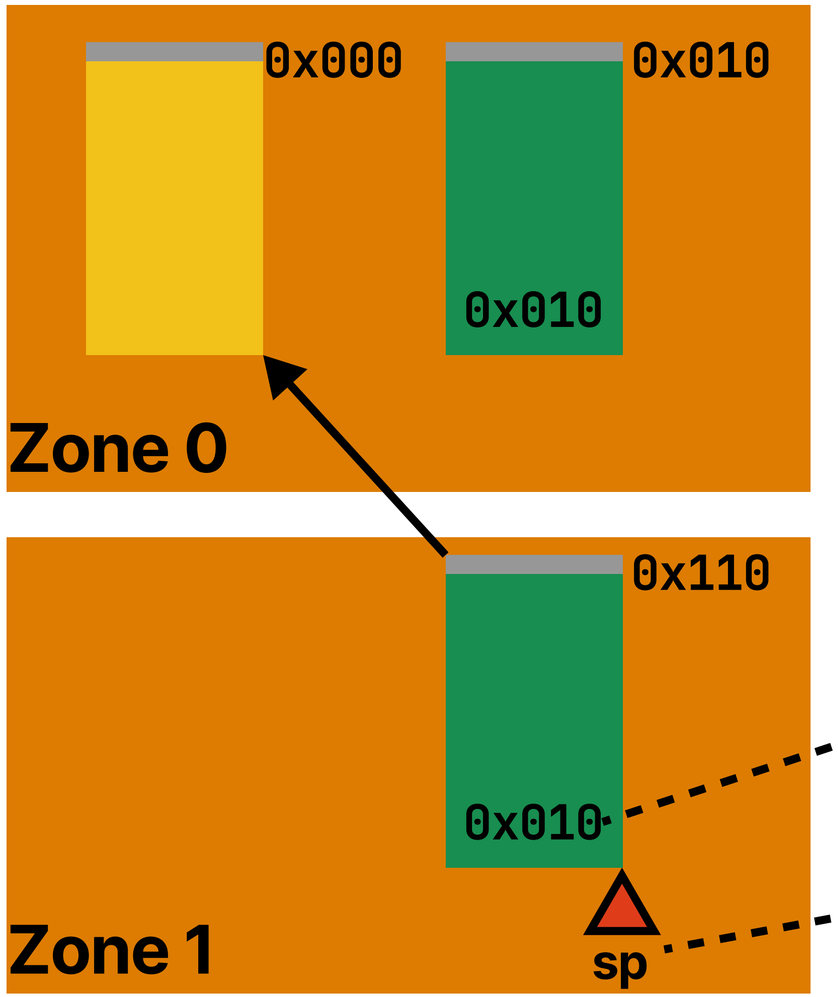


1. The **program** raises to handler at 0×010

→ 0x010

Address Translation

Zone number
Handler ID
Stack Offset



1. The **program** raises to handler at 0×010

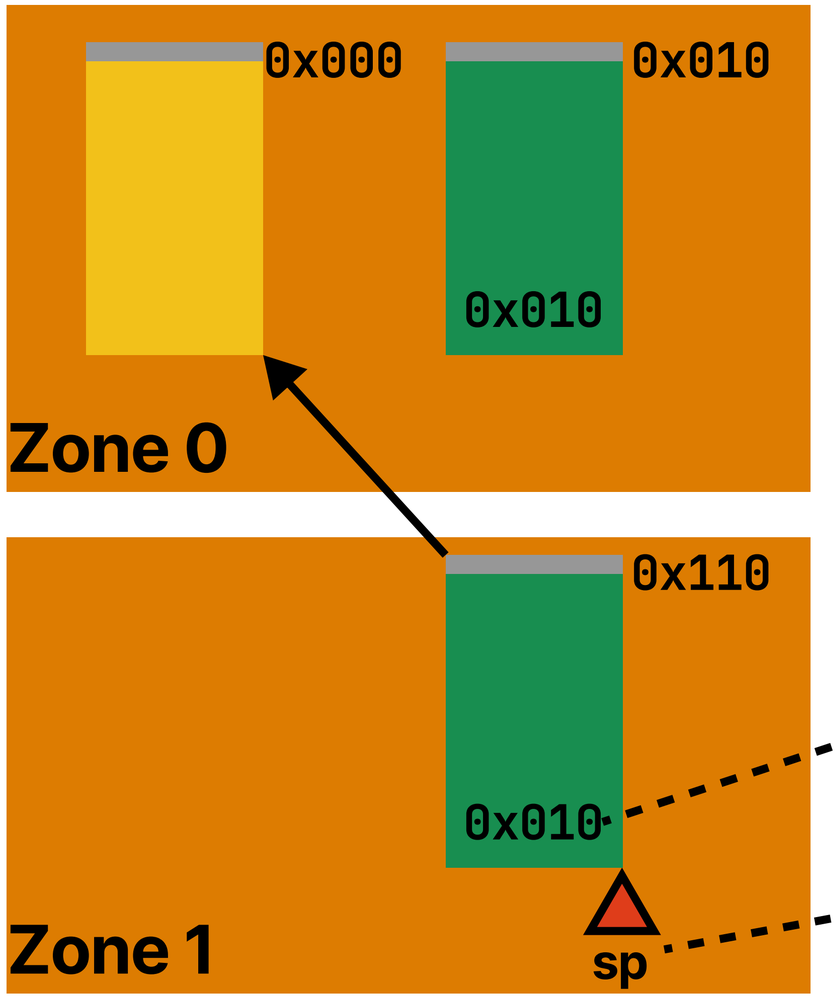
0x010

0x118

2. The **runtime** computes the current zone from sp

Zone number
Handler ID
Stack Offset

Address Translation



1. The **program** raises to handler at 0×010

0x010

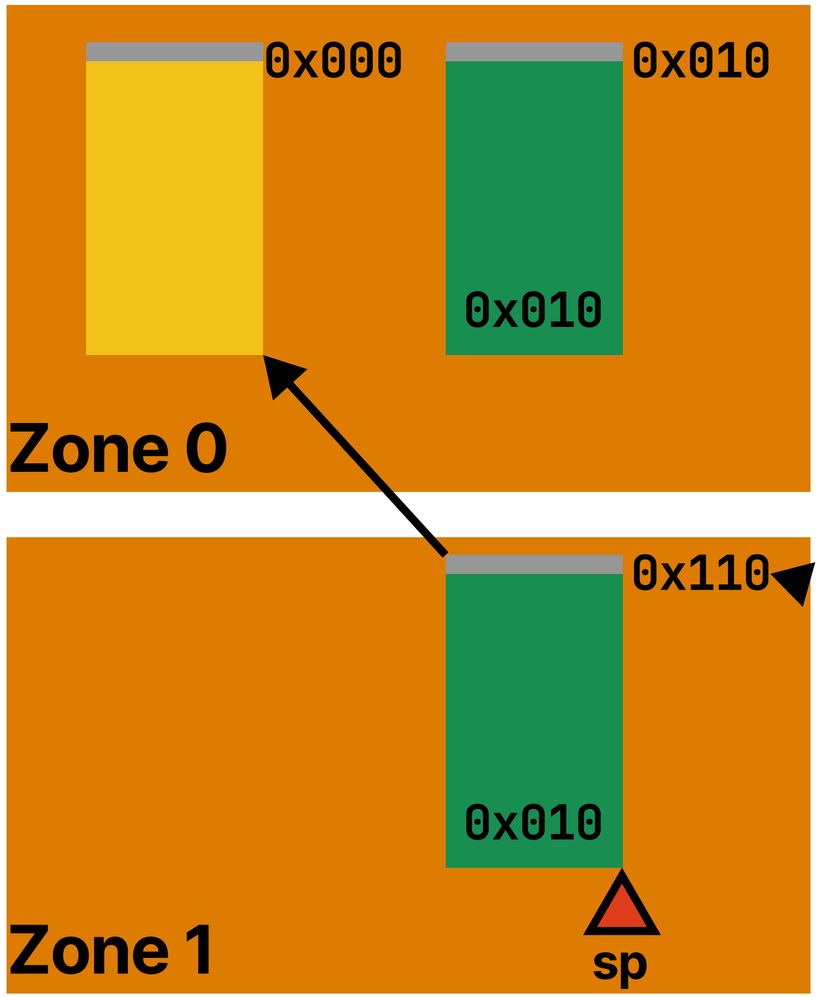
3. The **runtime** updates the zone number in the address

0x118

2. The **runtime** computes the current zone from sp

Zone number
Handler ID
Stack Offset

Address Translation



In our system, a stack pointer's upper part is meaningless.

It is contextualized at runtime according to the current execution point.

0x110

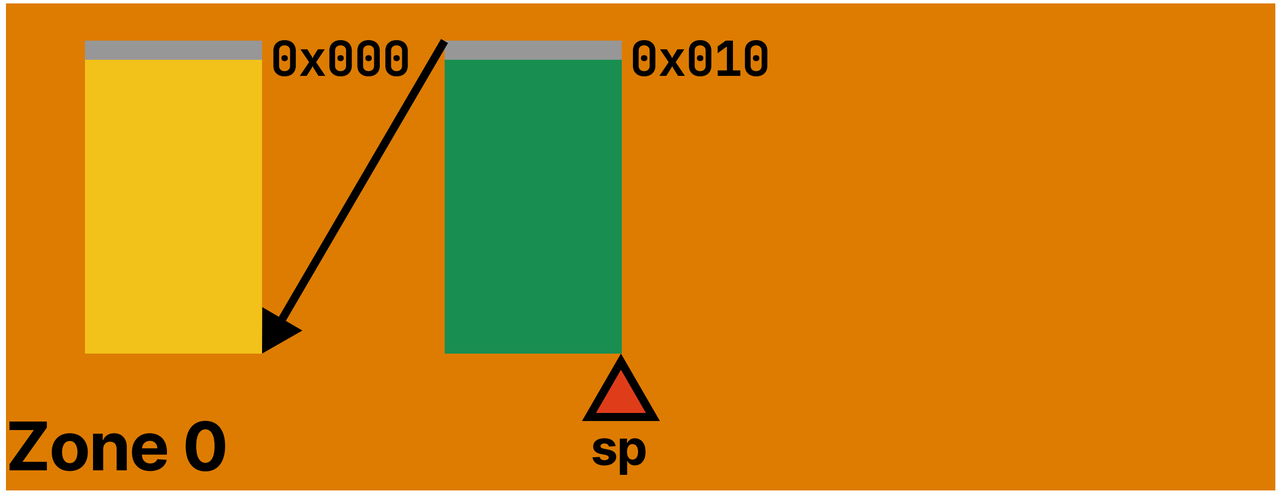
Translated Address

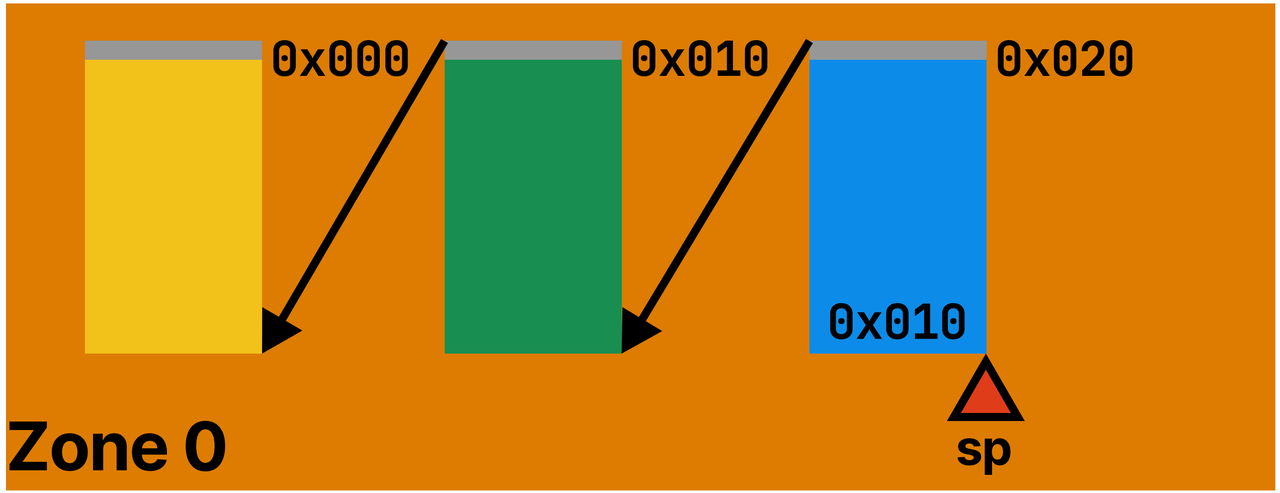
4. The **runtime** returns the translated address to the **program**

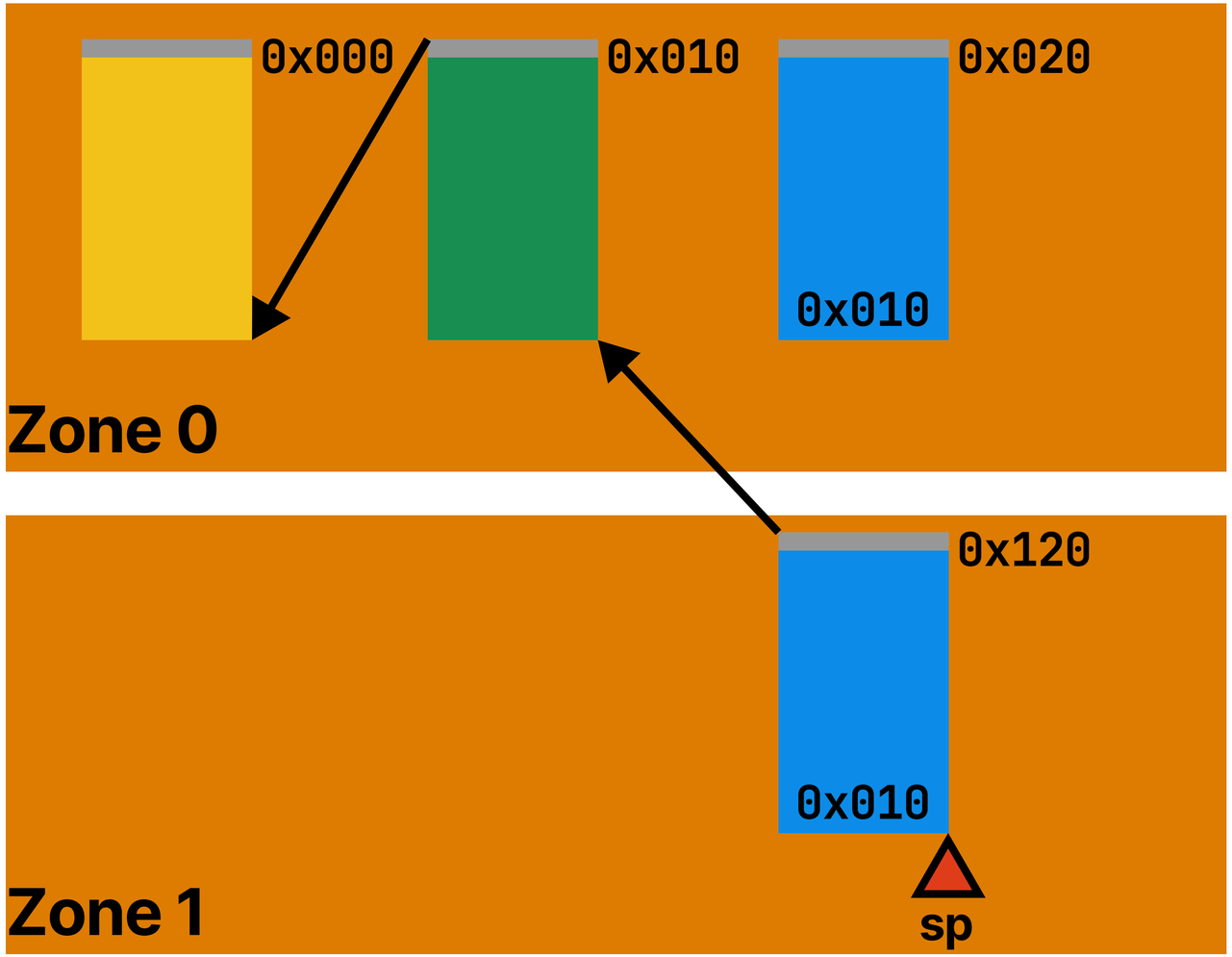
We’ve only covered the simple case.

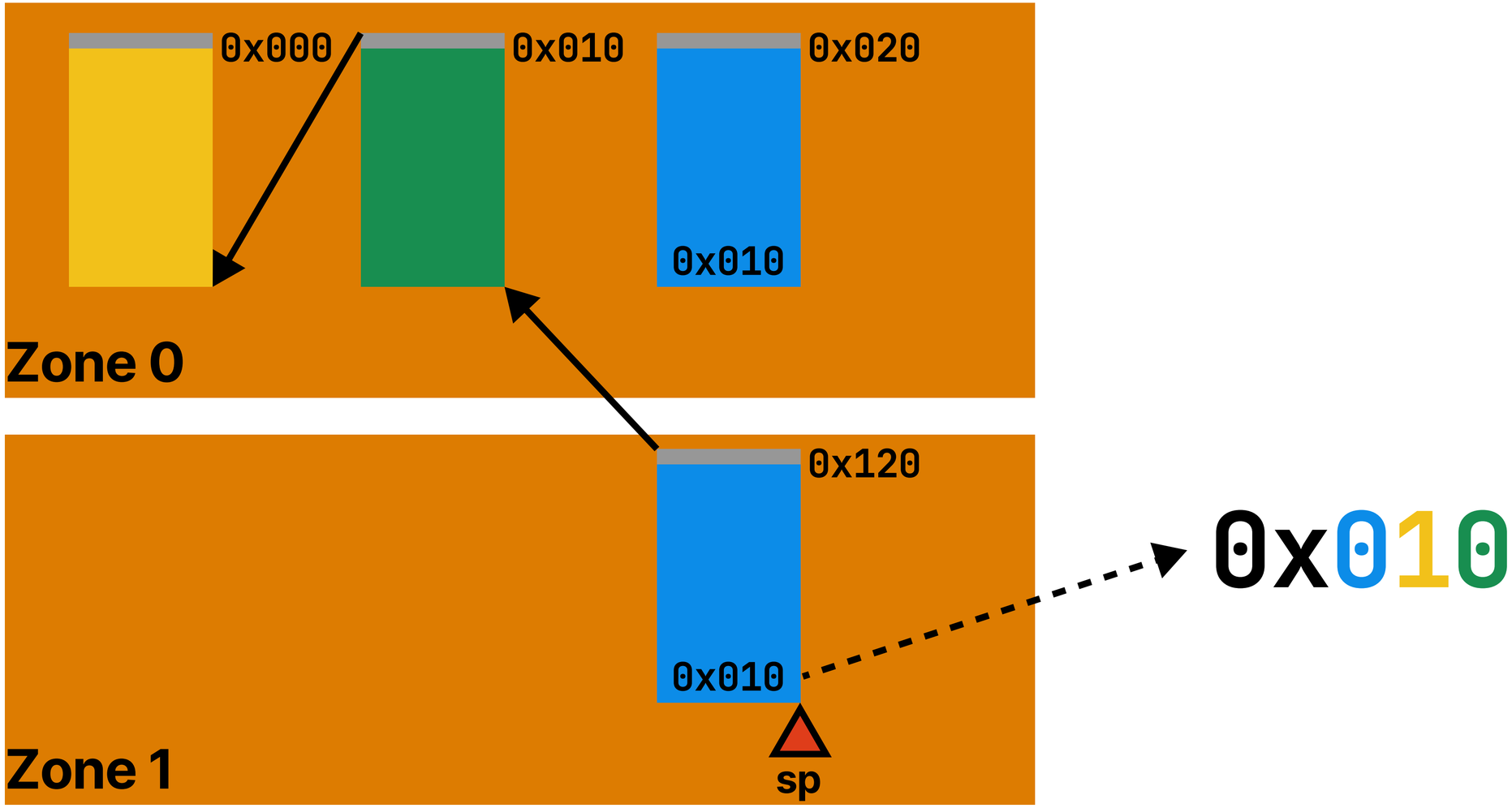
In general, the target handler can be in a different zone.

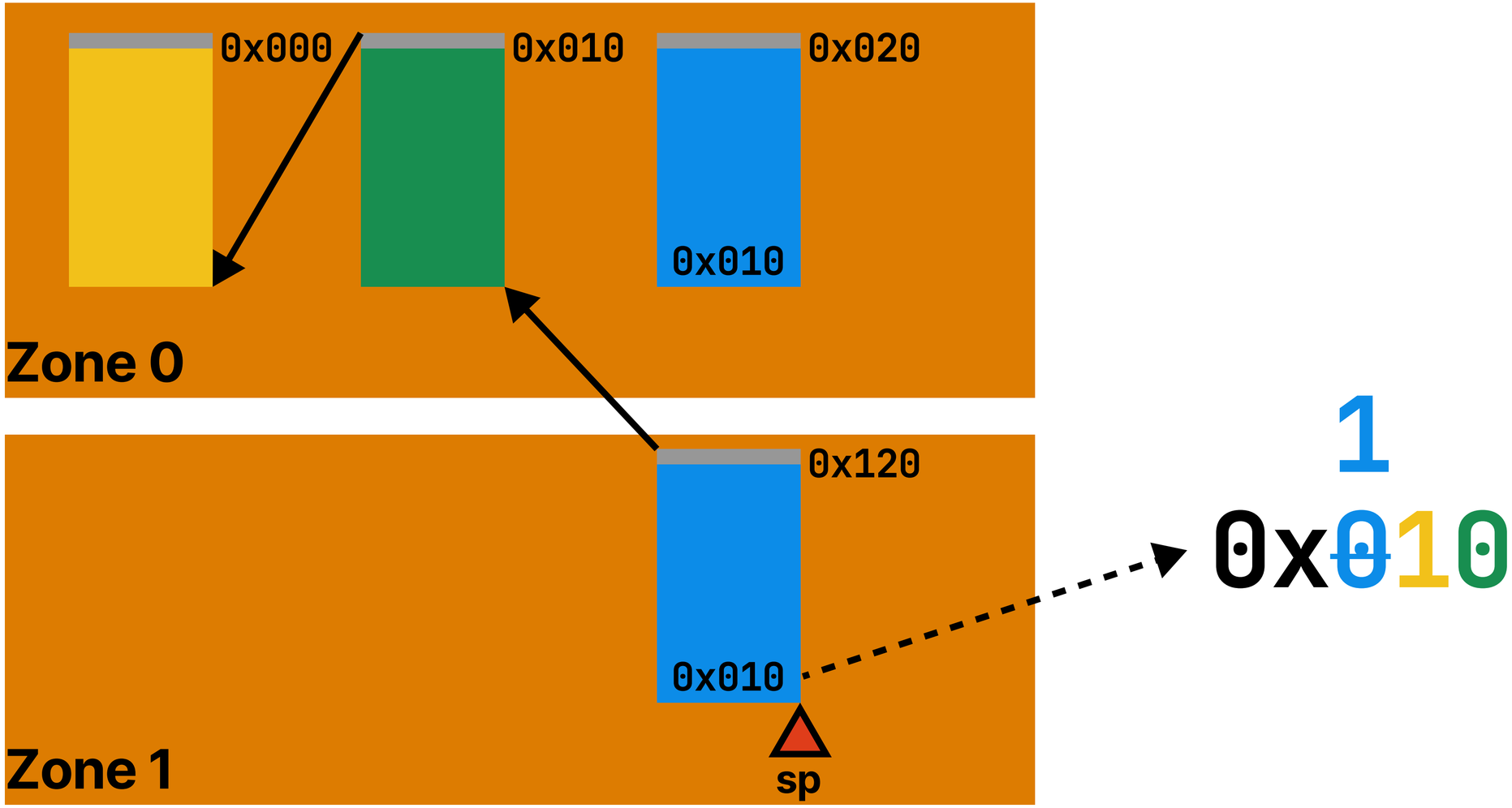

Zone 0sp

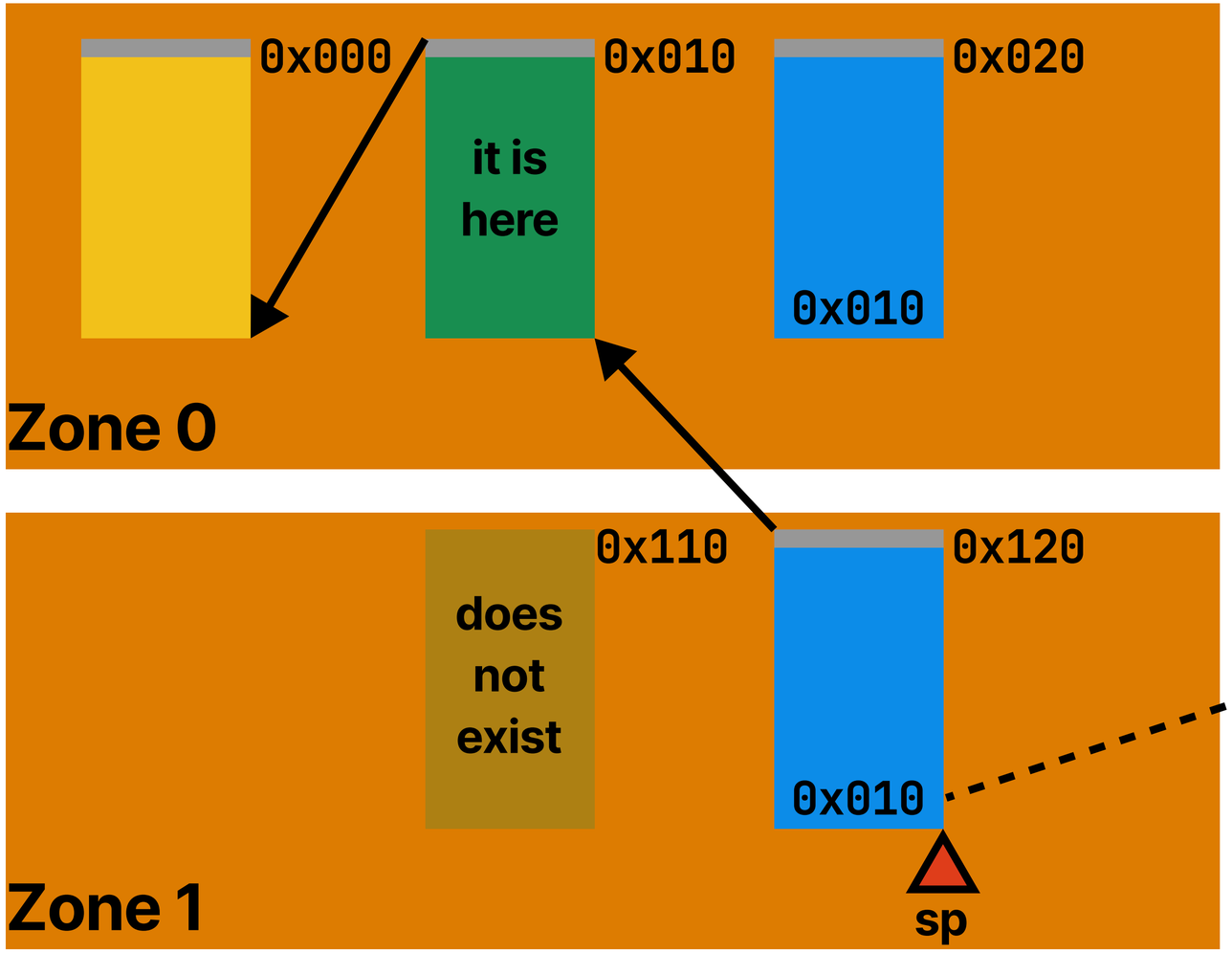

0x000
0x010
Zone 0
sp


0x000
0x010
0x020
0x010
Zone 0
sp


0x000
0x010
0x020
0x010
Zone 0
0x120
0x010
sp
Zone 1


0x000
0x010
0x020
0x010
Zone 0
0x120
0x010
sp
Zone 1
0x010


0x000
0x010
0x020
0x010
Zone 0
0x120
0x010
sp
Zone 1
1
0x010

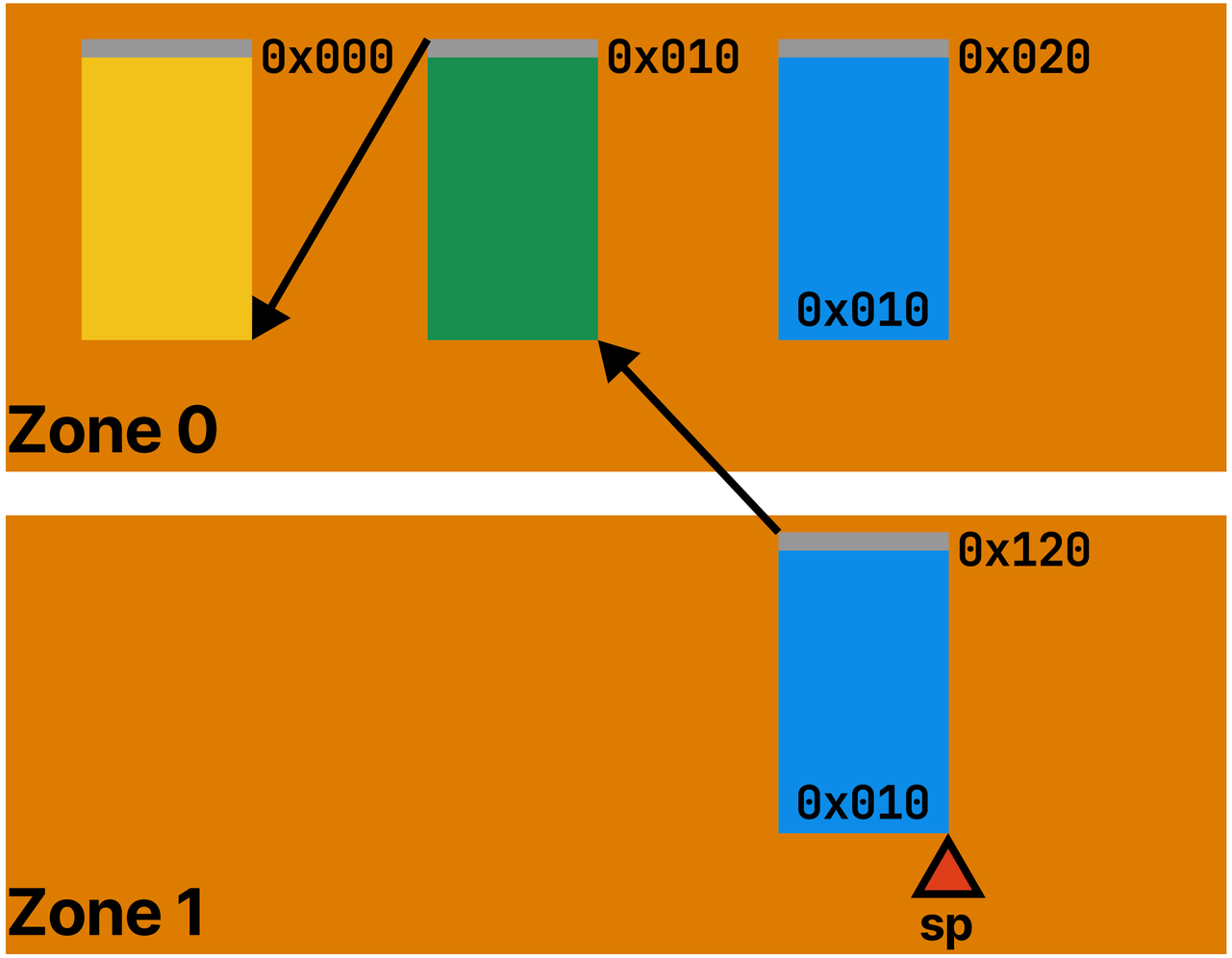


1

0x010

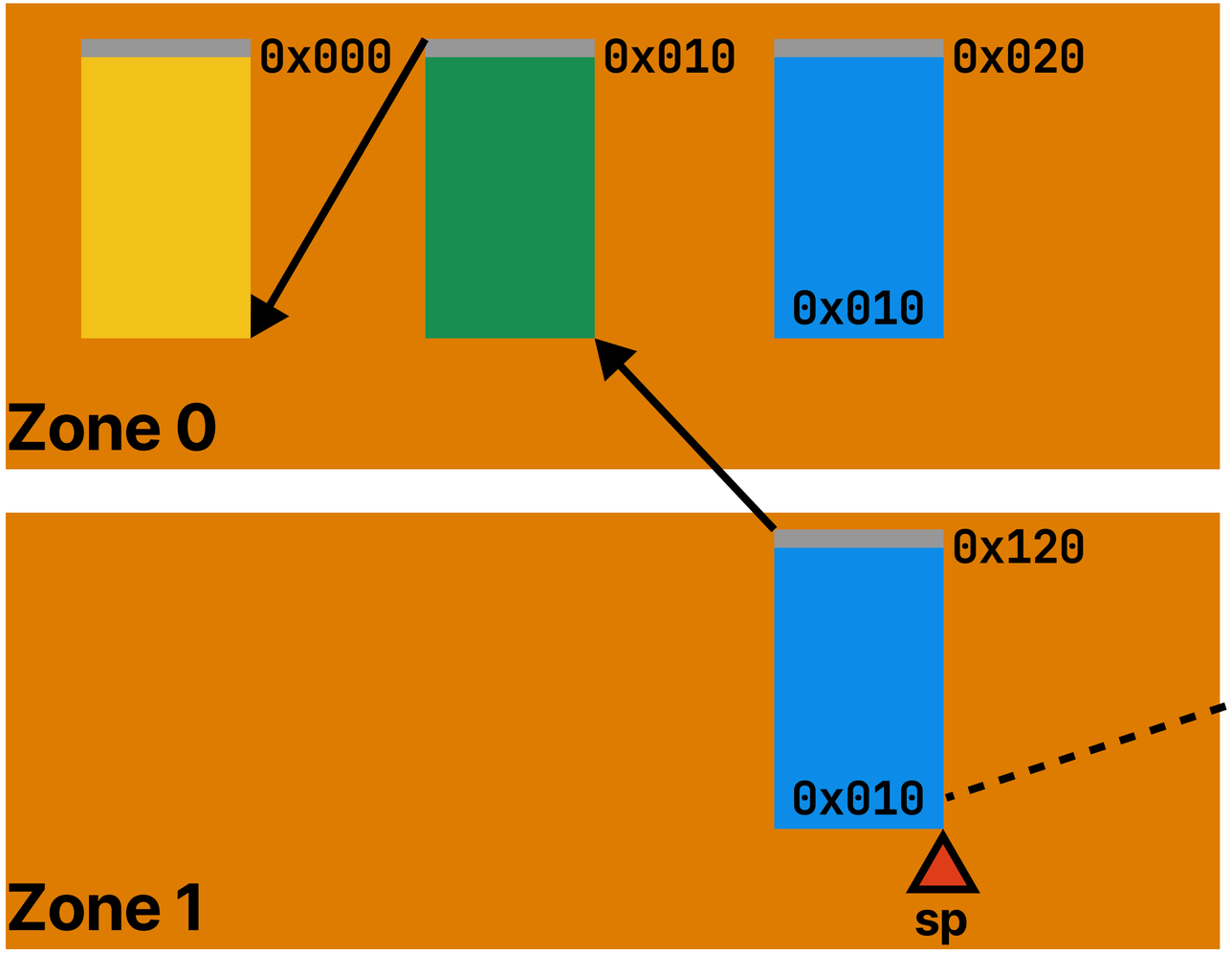
A naive address translation leads us to 0×110, which does not exist.

Zone Walk



Instead, for each translation, the runtime performs a zone walk.

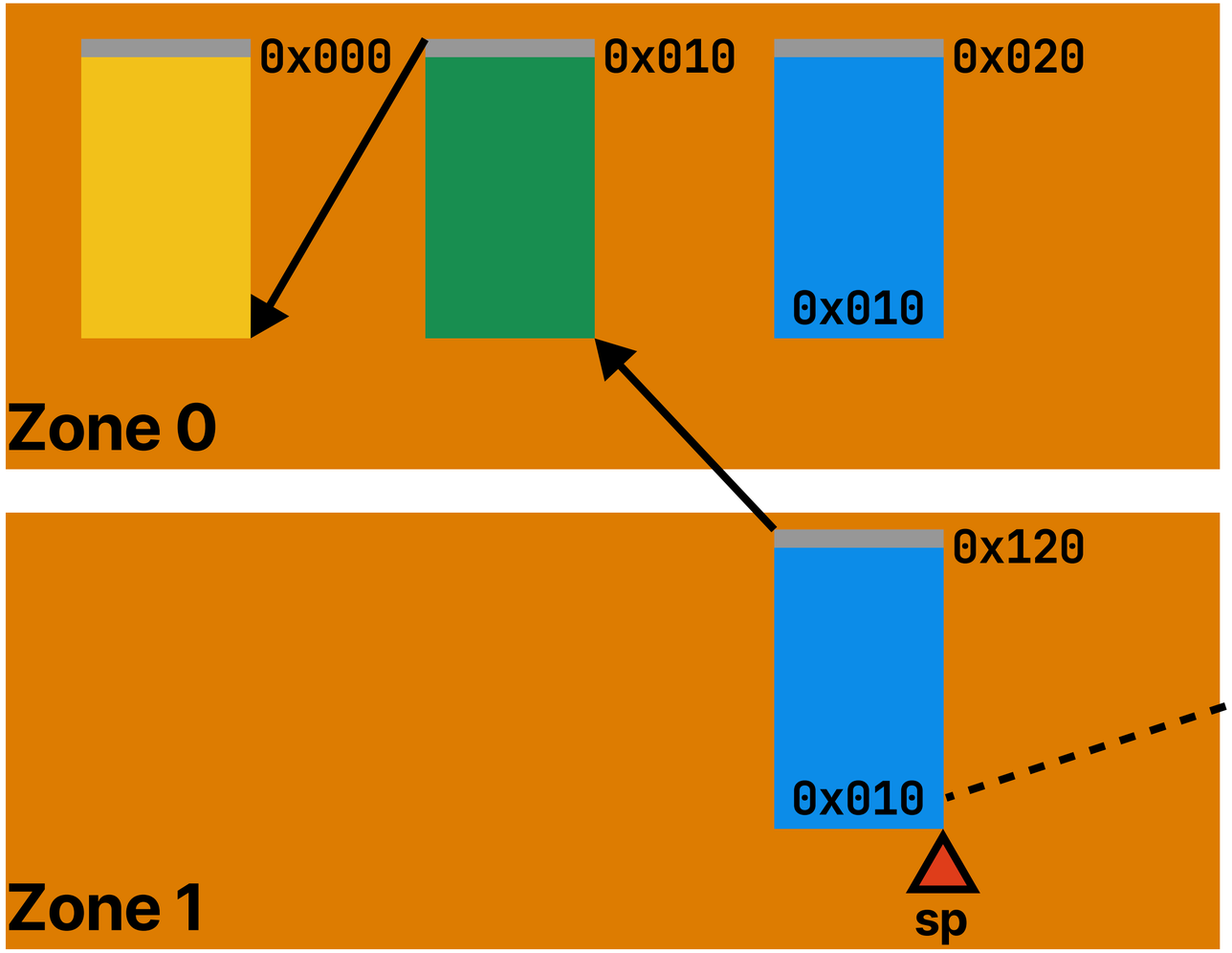
Zone Walk



Zone number
Handler ID
Stack Offset

0x010

Zone Walk



Zone number

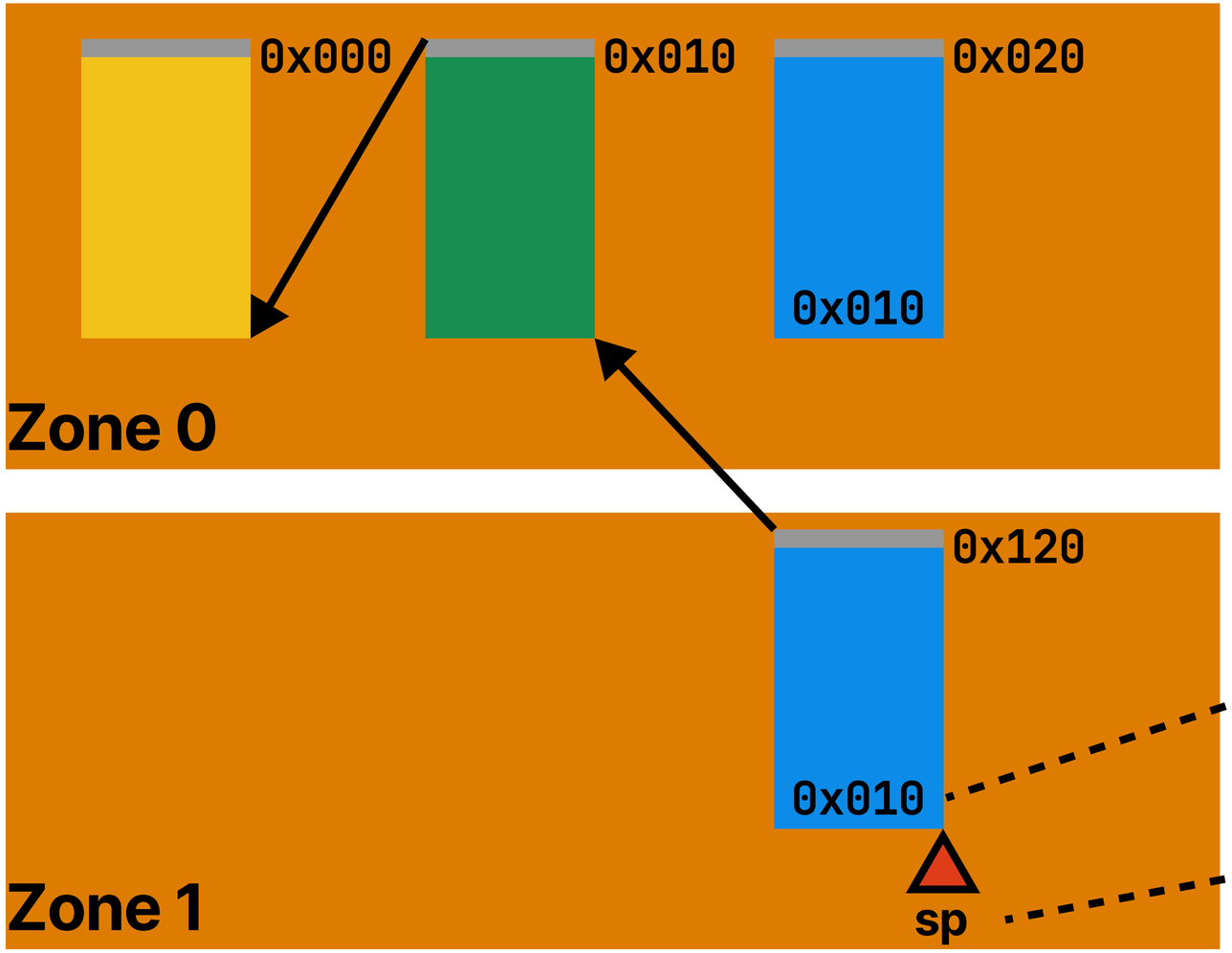
Handler ID

Stack Offset

1 target handler id

0x010

Zone Walk



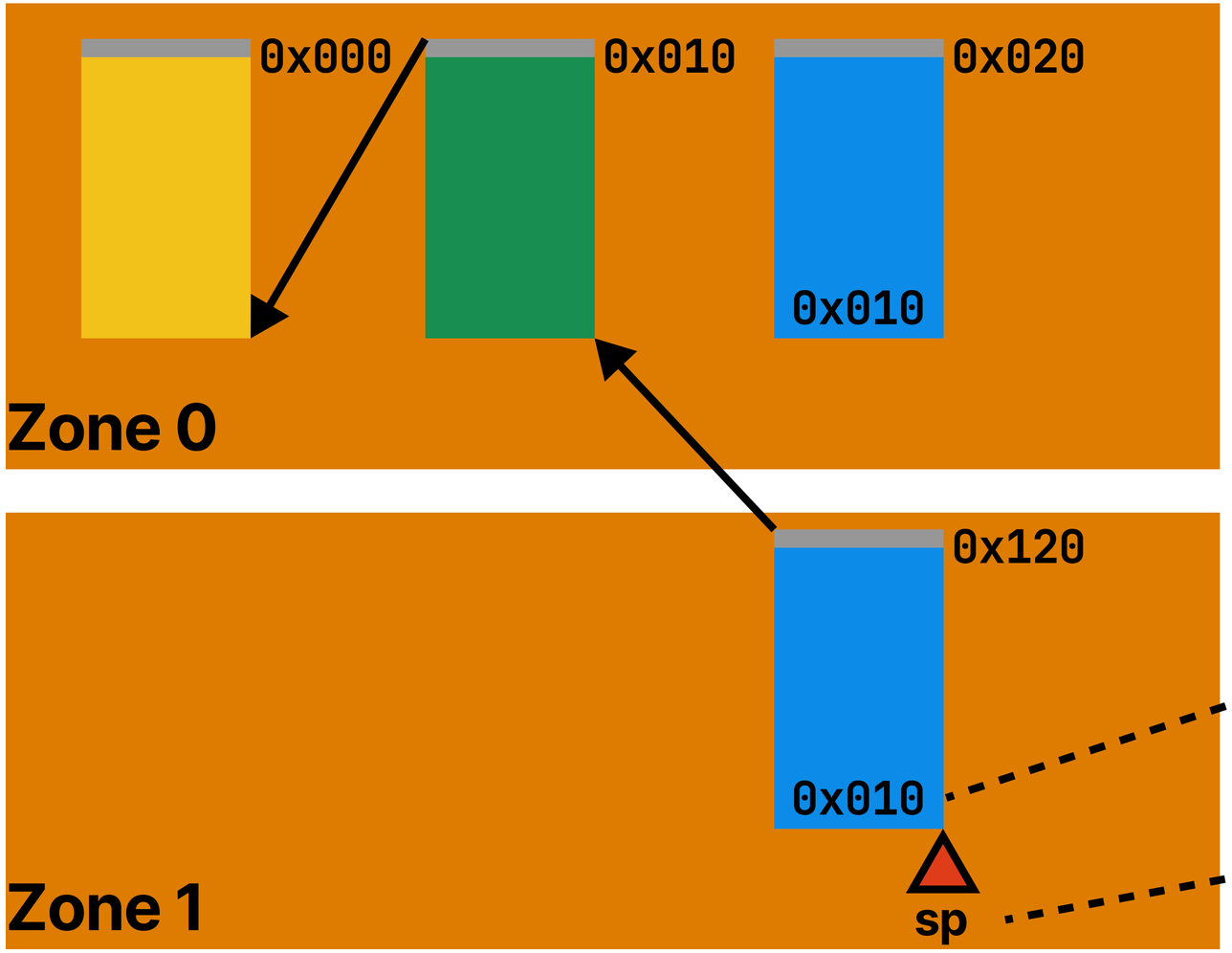
Zone number
Handler ID
Stack Offset

1 target handler id

0x010

0x118

Zone Walk



Zone number
Handler ID
Stack Offset

1 target handler id

0x010

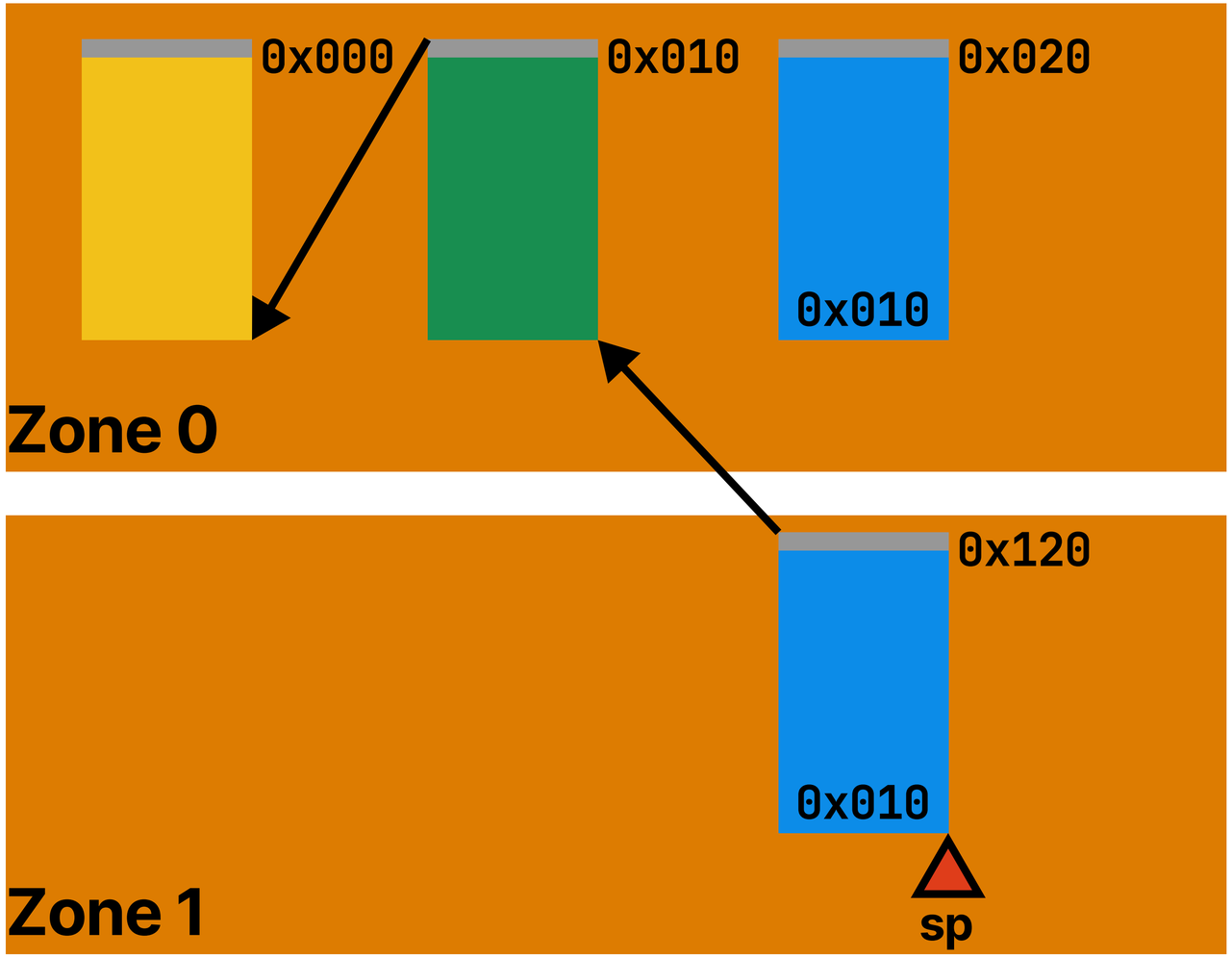
0x118

1 initial zone

Zone Walk

1 target handler id

1 initial zone



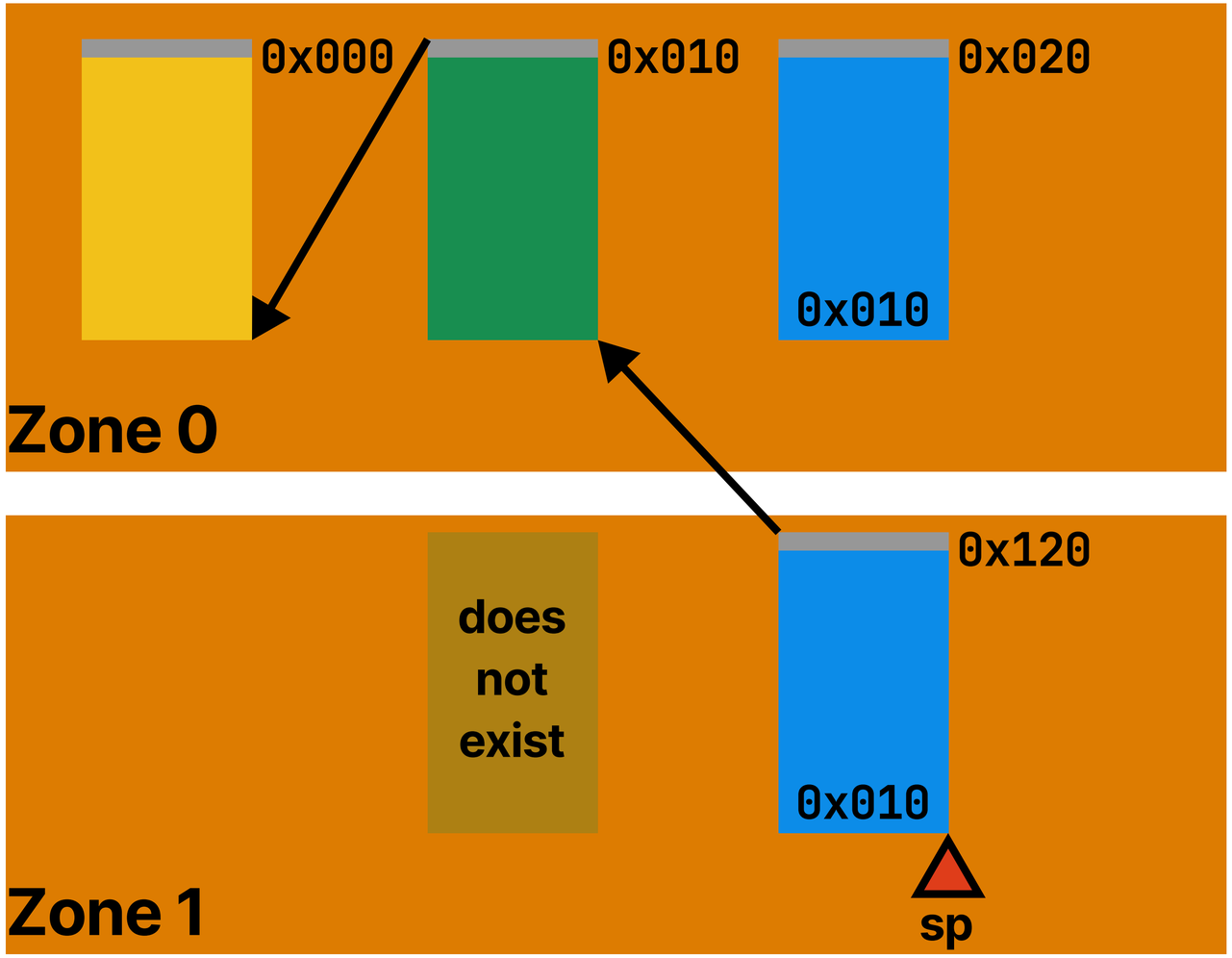
zone walker

Is there a handler with id 1 in this zone?

Zone Walk

1 target handler id

1 initial zone



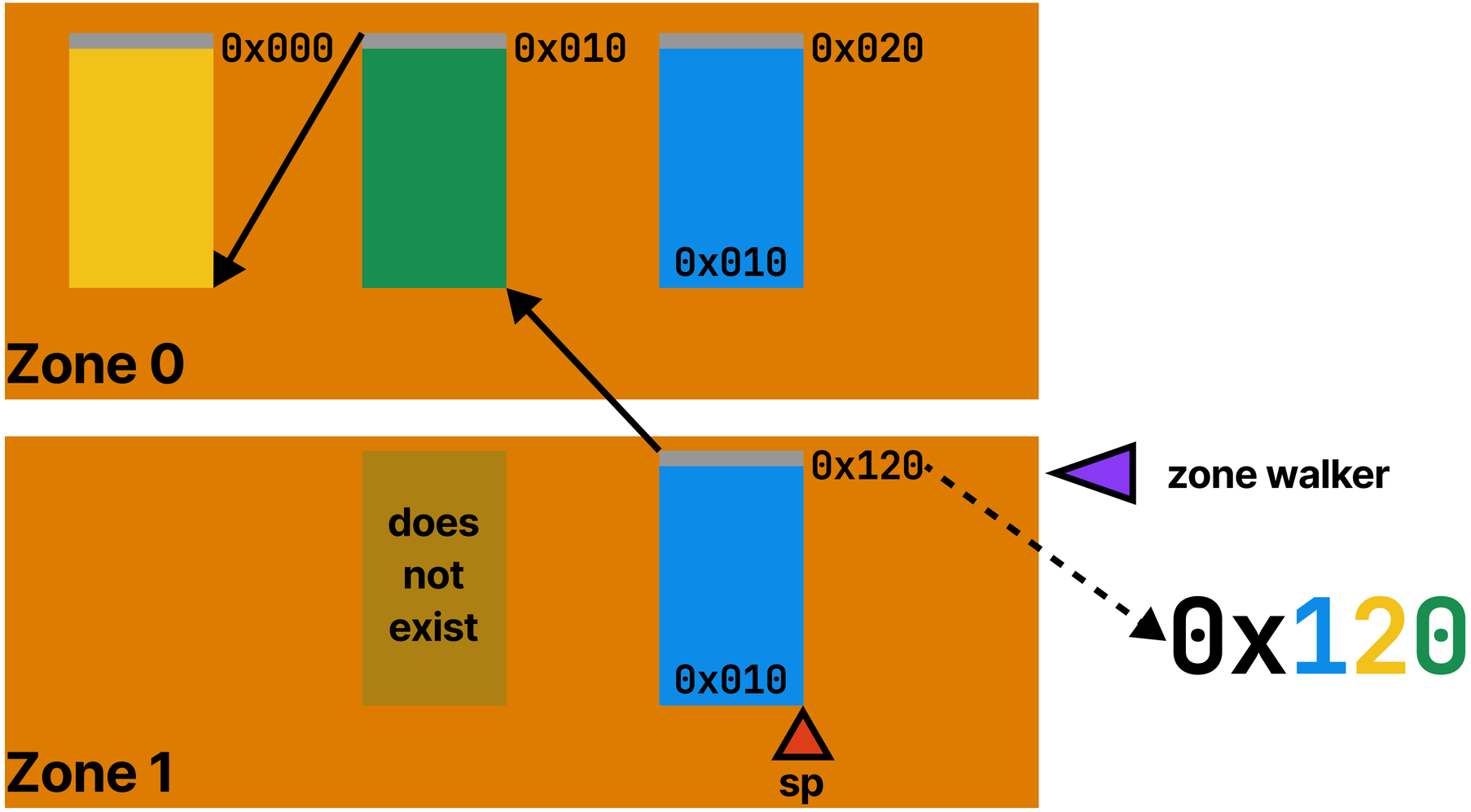
zone walker

Is there a handler with id 1 in this zone?
No, but how can we check?

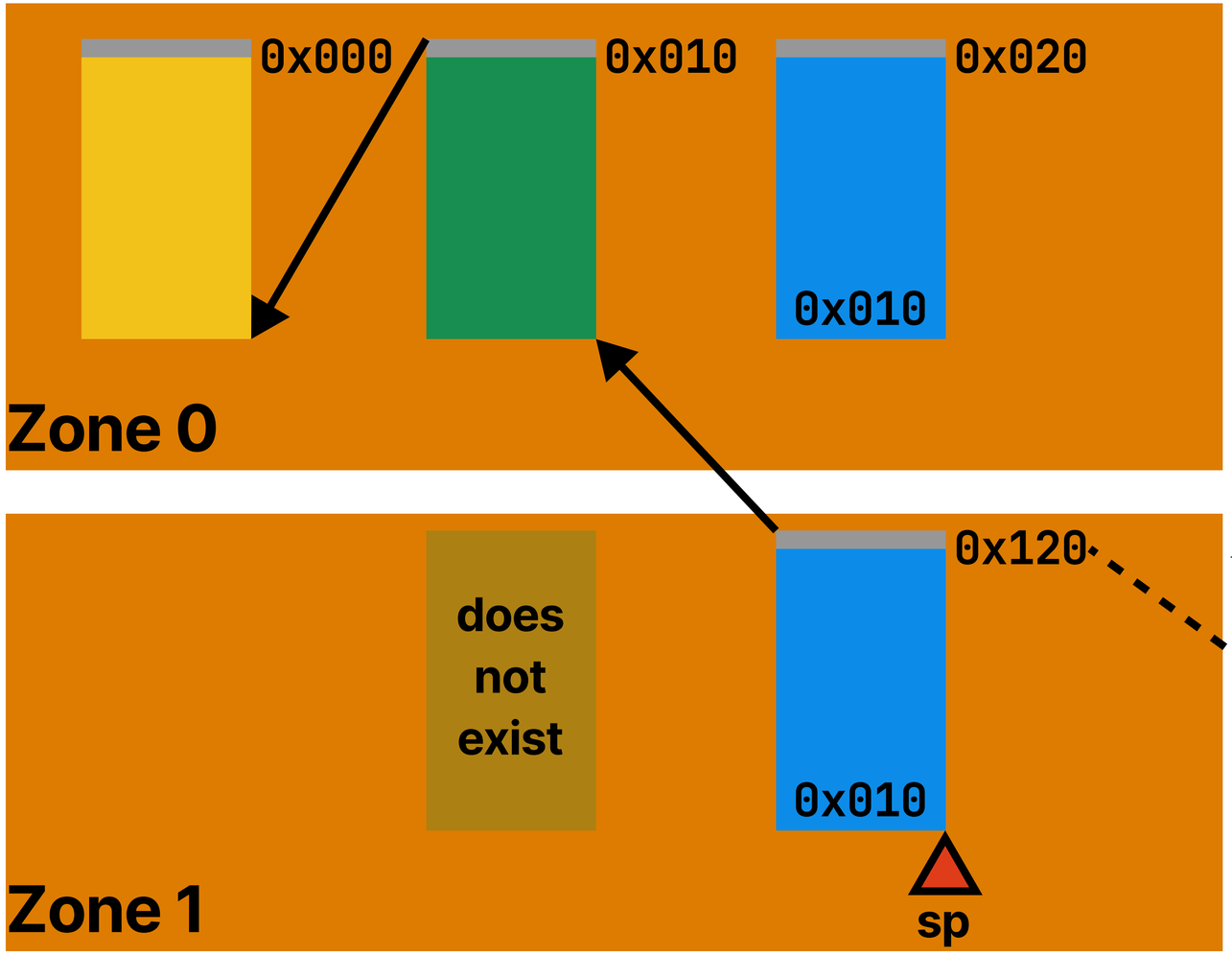
Zone Walk

1 target handler id

1 initial zone



Zone Walk



zone walker

0x120

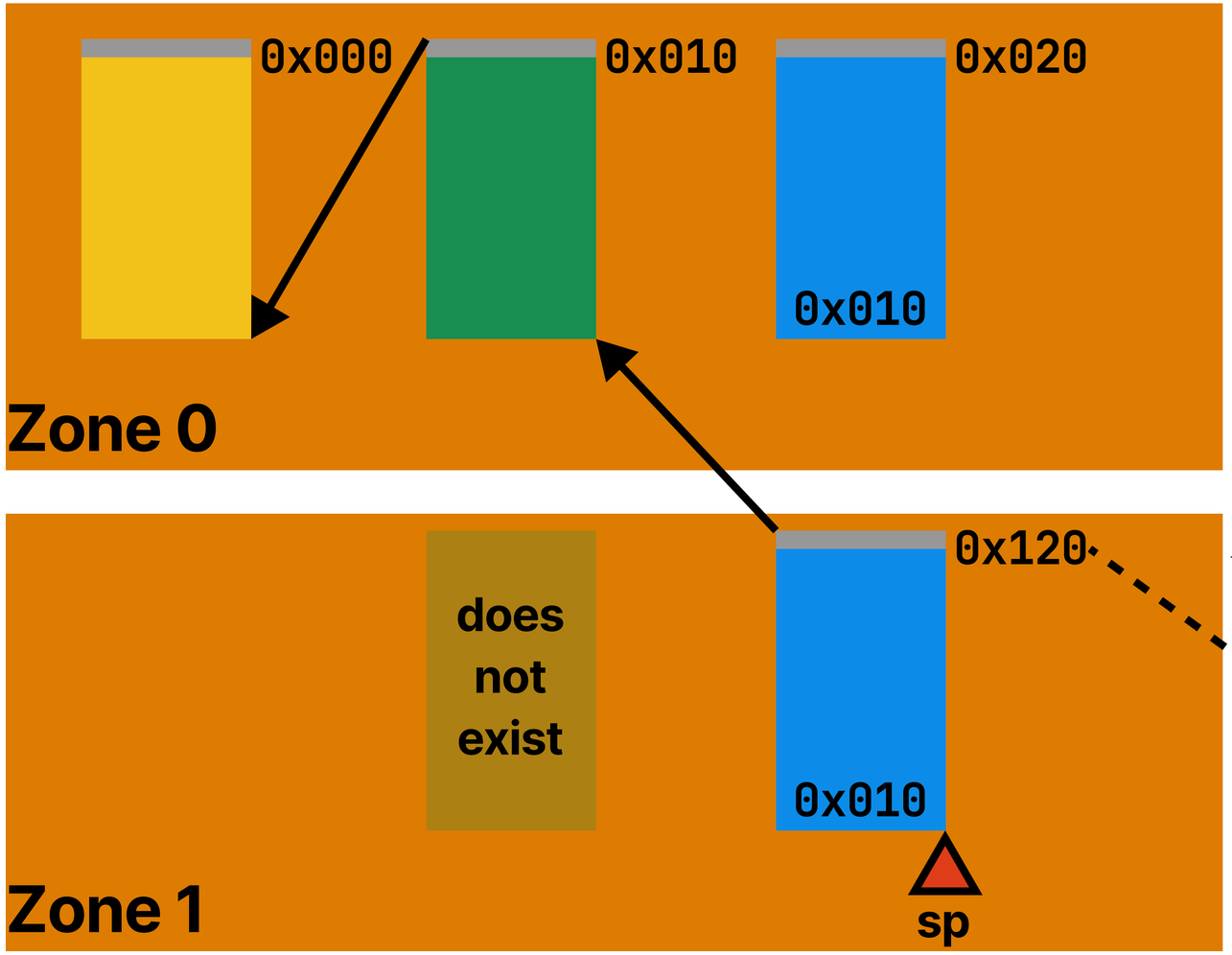
2 > 1 target handler id

1 target handler id

1 initial zone

Compare the first handler id with our target handler id.

Zone Walk



zone walker

0x120

2 > 1 target handler id

1 target handler id

1 initial zone

The target handler id is smaller,
so it is not in this zone. Why?

Zone Walk

2 > 1 **target handler id**

The target handler id is smaller, so it is not in this zone. **Why?**

Zone Walk

2 > 1 **target handler id**

The target handler id is smaller, so it is not in this zone. **Why?**

1. Handler IDs are assigned monotonically.

```
handle c1
  handle c2
    handle c3
      foo(c1, c2, c3)
    with
      ...
  with
    ...
with
  ...
```

Zone Walk

2 > 1 **target handler id**

The target handler id is smaller, so it is not in this zone. **Why?**

1. Handler IDs are assigned monotonically.

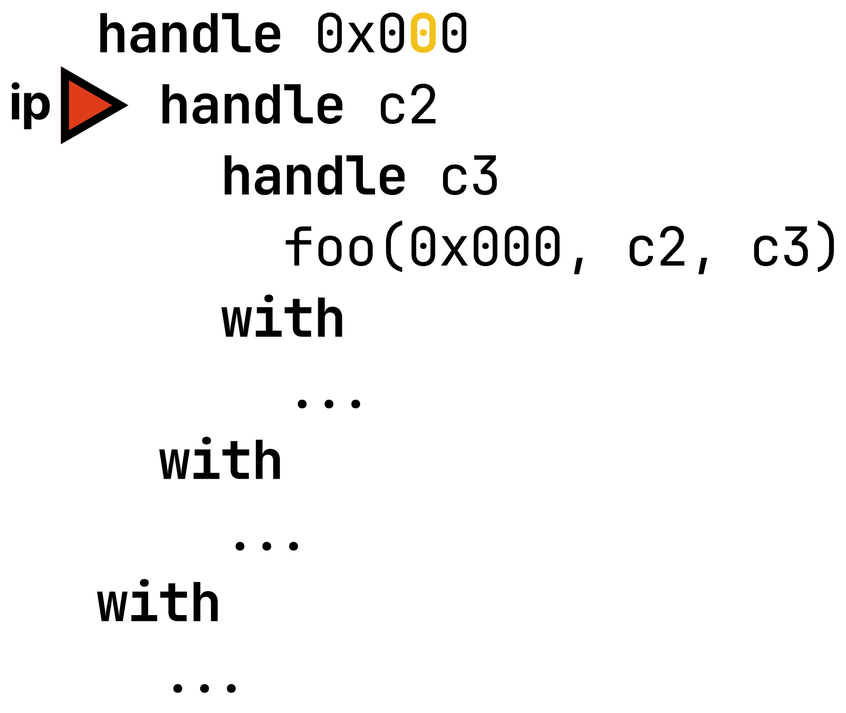
```
handle 0x000
  handle 0x010
    handle 0x020
      foo(0x000, 0x010, 0x020)
    with
       ...
  with
     ...
with
   ...
```

Zone Walk

$2 > 1$ **target handler id**

The target handler id is smaller, so it is not in this zone. **Why?**

1. Handler IDs are assigned monotonically.
2. Handler caps are scoped lexically.

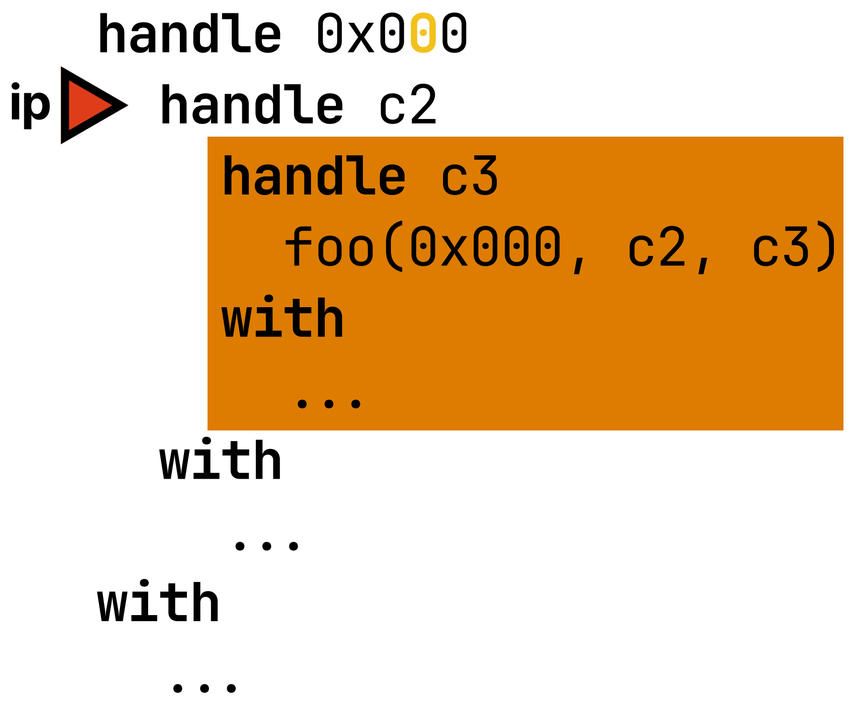


Zone Walk

2 > 1 **target handler id**

The target handler id is smaller, so it is not in this zone. **Why?**

1. Handler IDs are assigned monotonically.
2. Handler caps are scoped lexically.



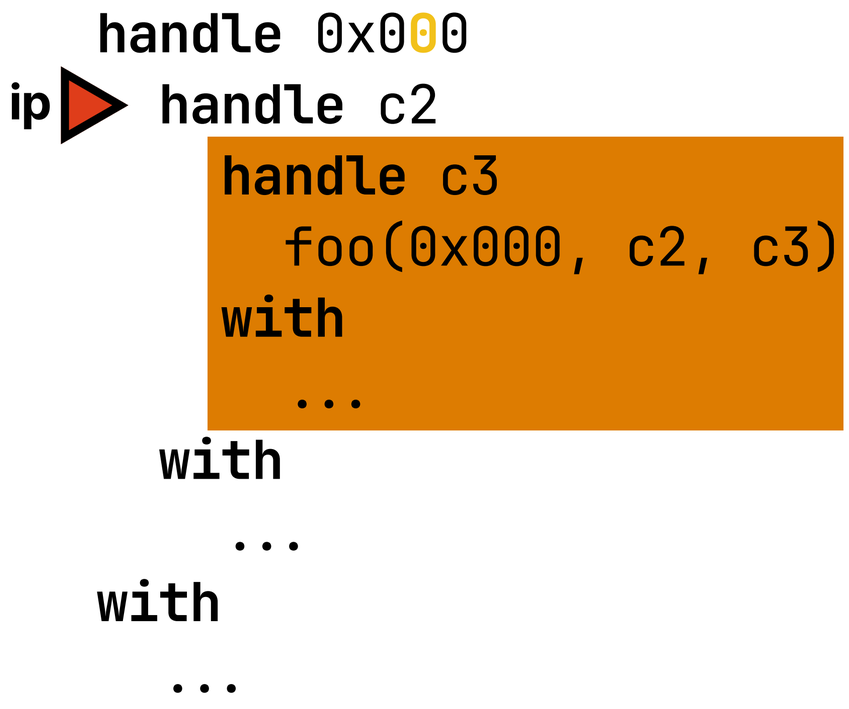
Installing a handler corresponds to the start of the lifetime of a continuation.

Zone Walk

2 > 1 **target handler id**

The target handler id is smaller, so it is not in this zone. **Why?**

1. Handler IDs are assigned monotonically.
2. Handler caps are scoped lexically.



Installing a handler corresponds to the start of the lifetime of a continuation.

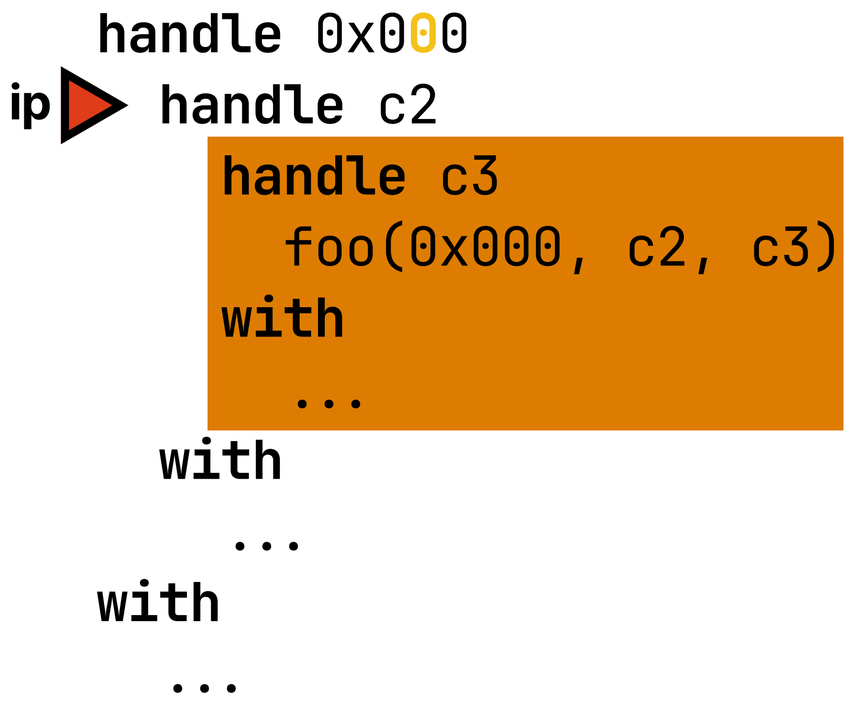
Due to lexically scoping, handlers that this continuation can ever reach to must have been allocated already, with smaller IDs.

Zone Walk

2 > 1 **target handler id**

The target handler id is smaller, so it is not in this zone. **Why?**

1. Handler IDs are assigned monotonically.
2. Handler caps are scoped lexically.



Installing a handler corresponds to the start of the lifetime of a continuation.

Due to lexically scoping, handlers that this continuation can ever reach to must have been allocated already, with smaller IDs.

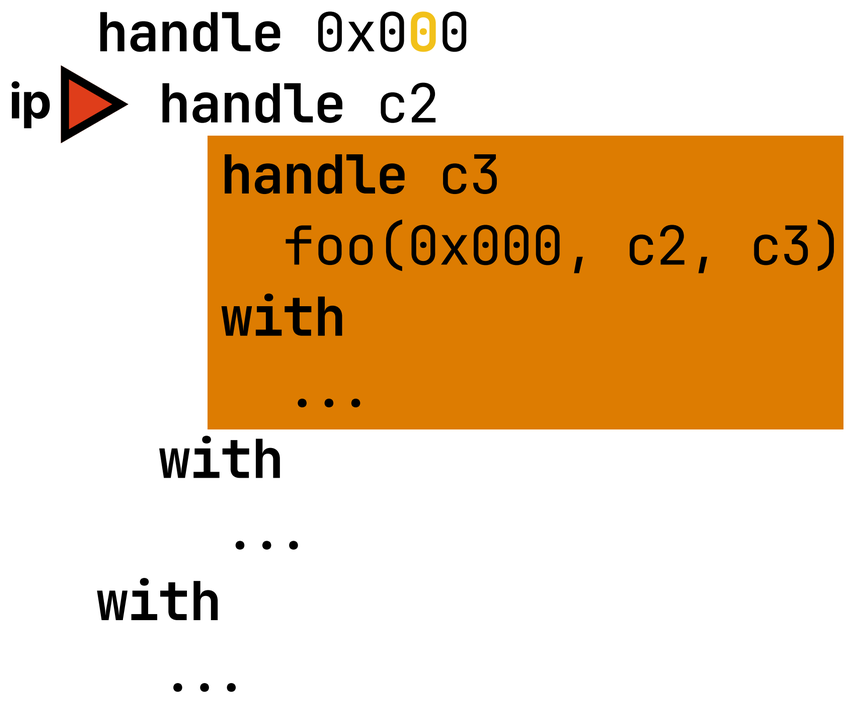
Handlers within this continuations are allocated later, with greater IDs.

Zone Walk

2 > 1 target handler id

The target handler id is smaller, so it is not in this zone. **Why?**

1. Handler IDs are assigned monotonically.
2. Handler caps are scoped lexically.



Installing a handler corresponds to the start of the lifetime of a continuation.

Due to lexically scoping, handlers that this continuation can ever reach to must have been allocated already, with smaller IDs.

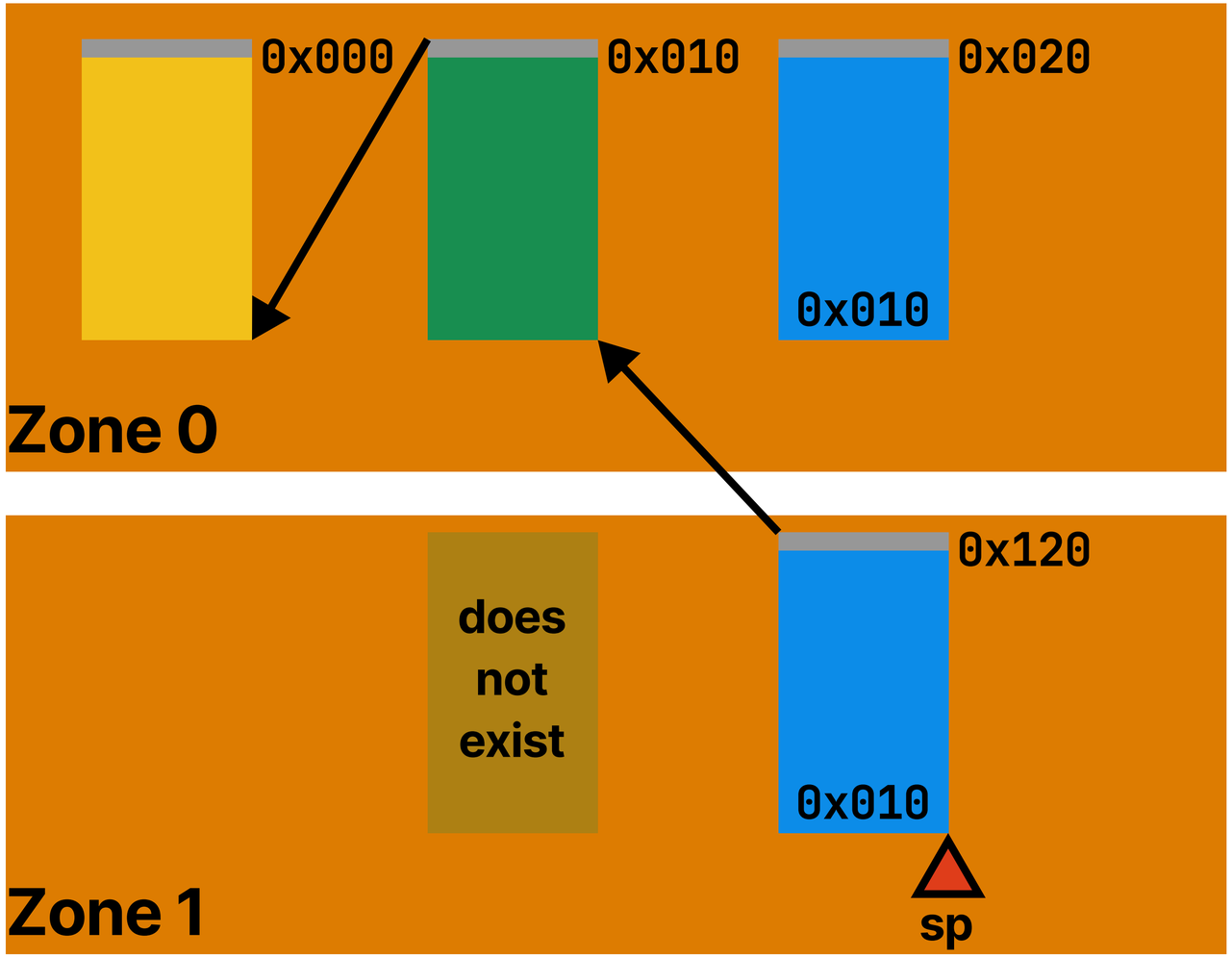
Handlers within this continuations are allocated later, with greater IDs.

Every handler demarcates the inside and outside continuations, with inside cont having smaller IDs and outside cont having greater IDs.

Zone Walk

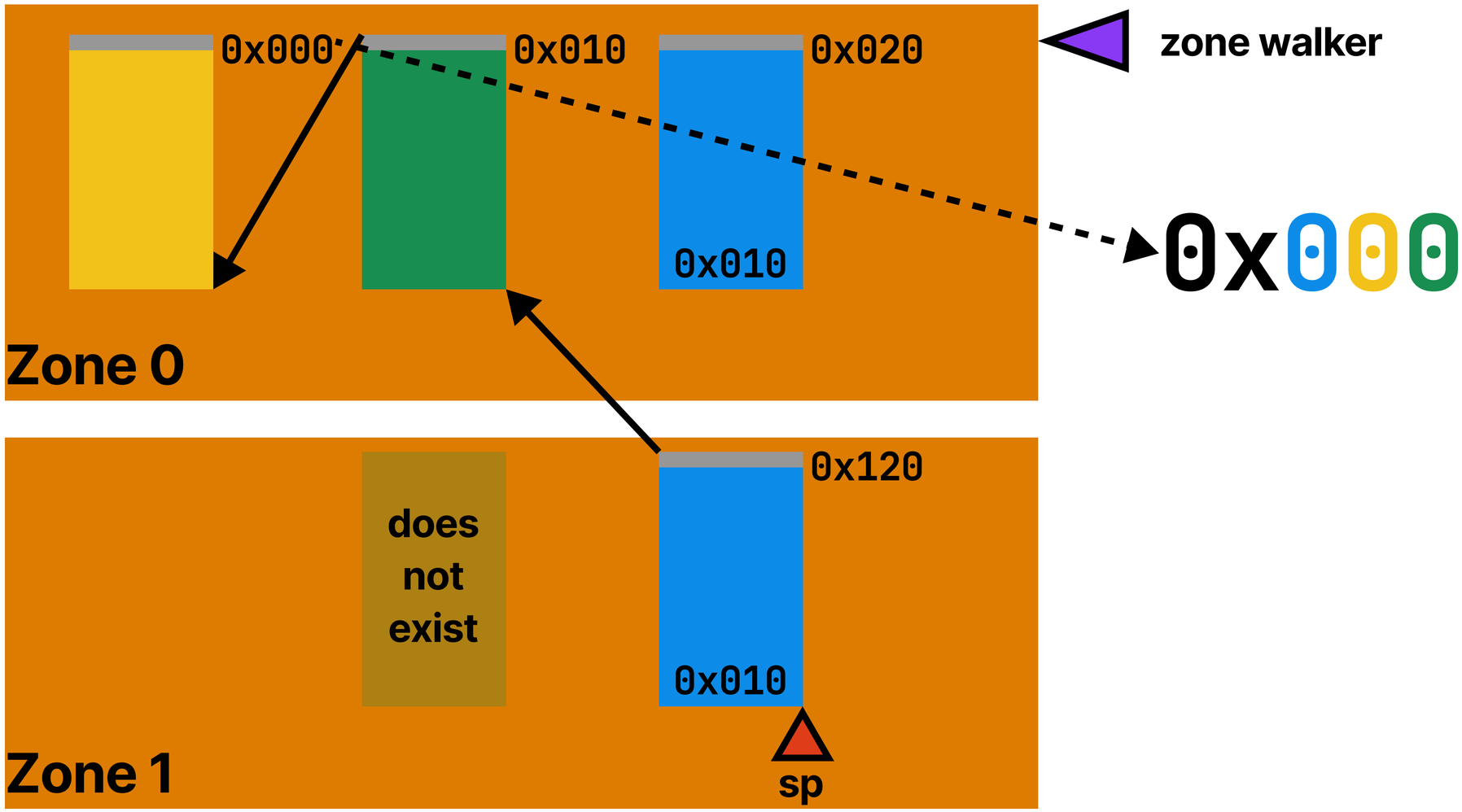
1 target handler id

1 initial zone



Zone 1 does not have a handler with ID 1.

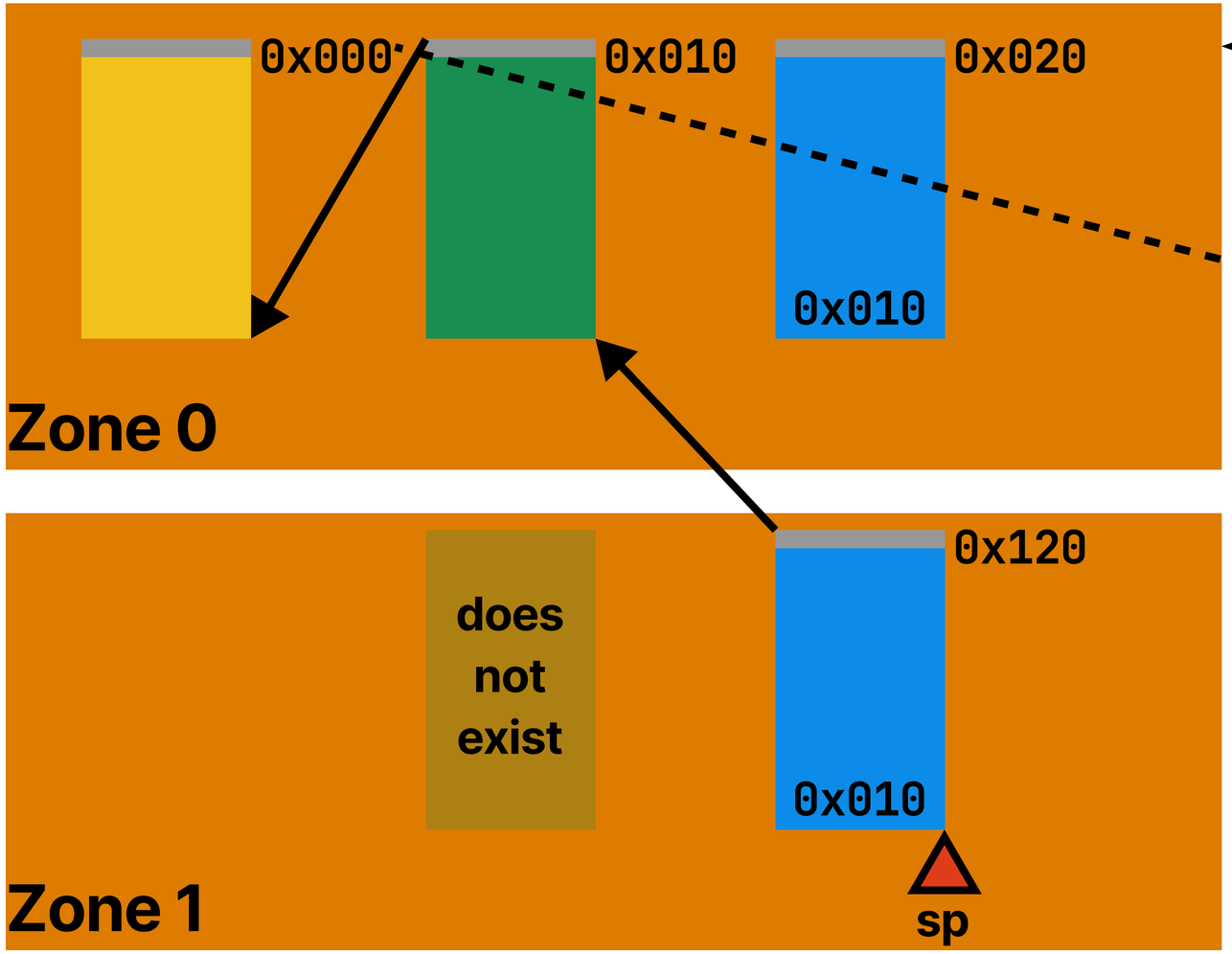
Zone Walk



1 target handler id

1 initial zone

Zone Walk



zone walker

1 target handler id

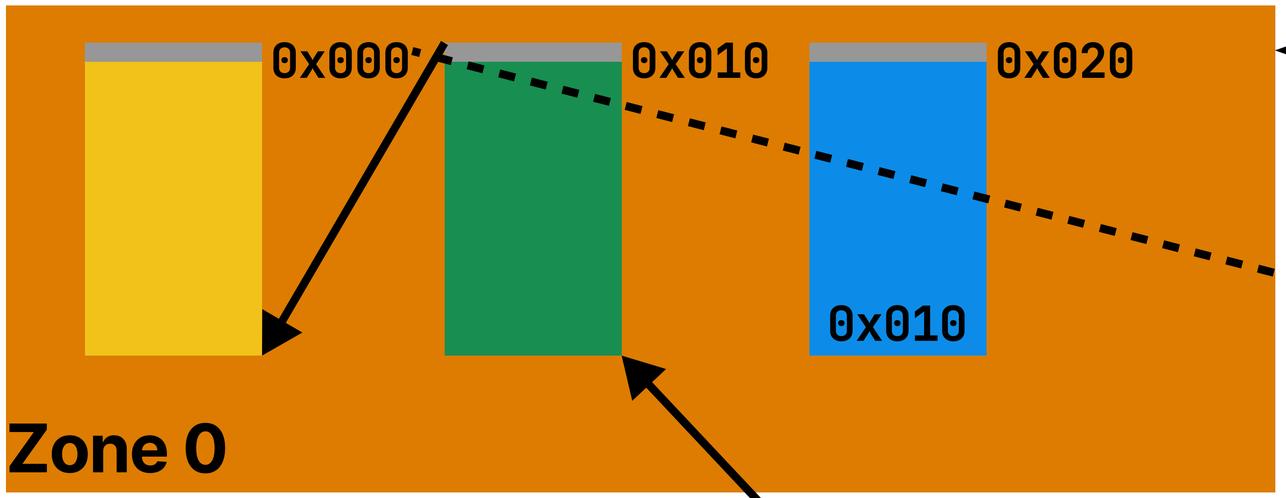
1 initial zone

0x000

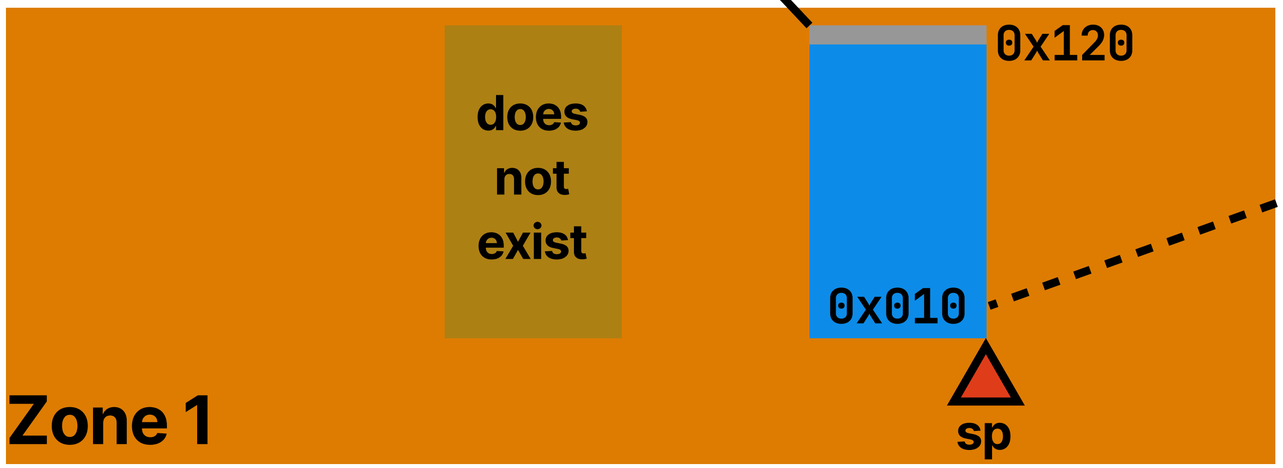
0 < 1 target handler id

The target handler id is greater, so it must be in this zone. Why?

Zone Walk



zone walker



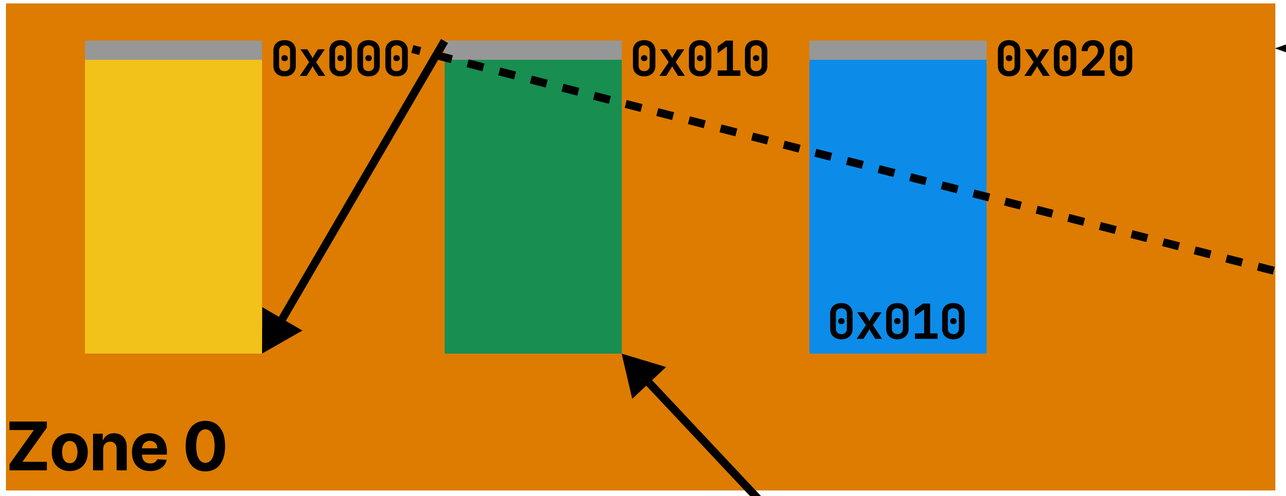
1 target handler id

1 initial zone

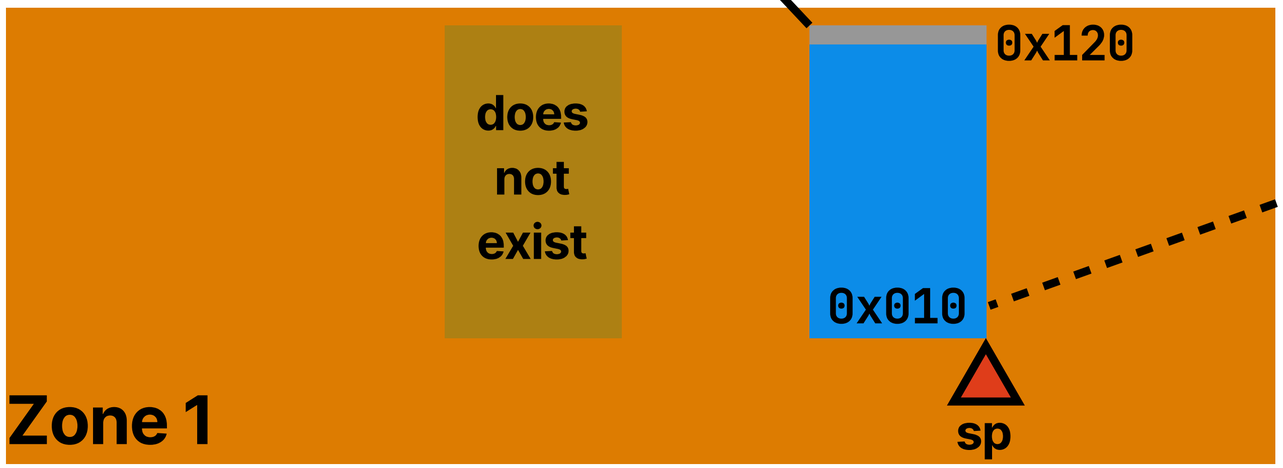
0x000

0x010 Update the zone number in the address

Zone Walk



zone walker



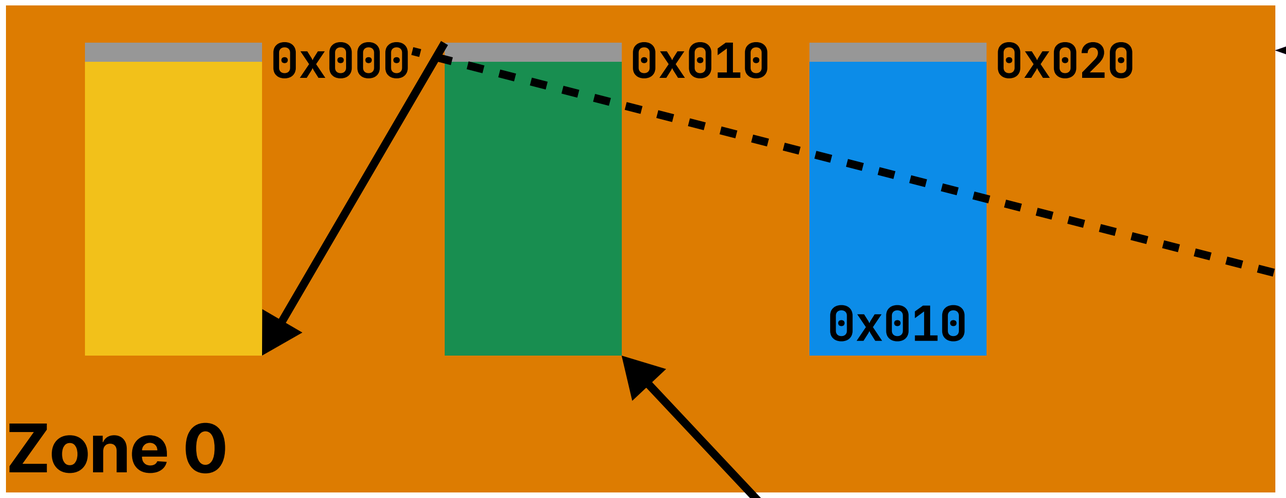
1 target handler id

1 initial zone

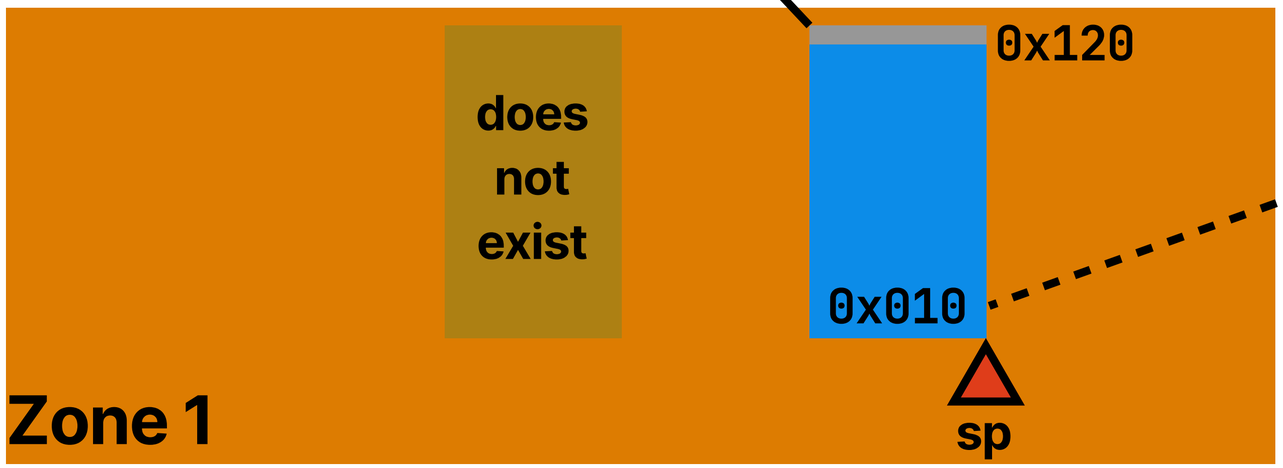
0x000

0x 10 Update the zone number in the address

Zone Walk



zone walker



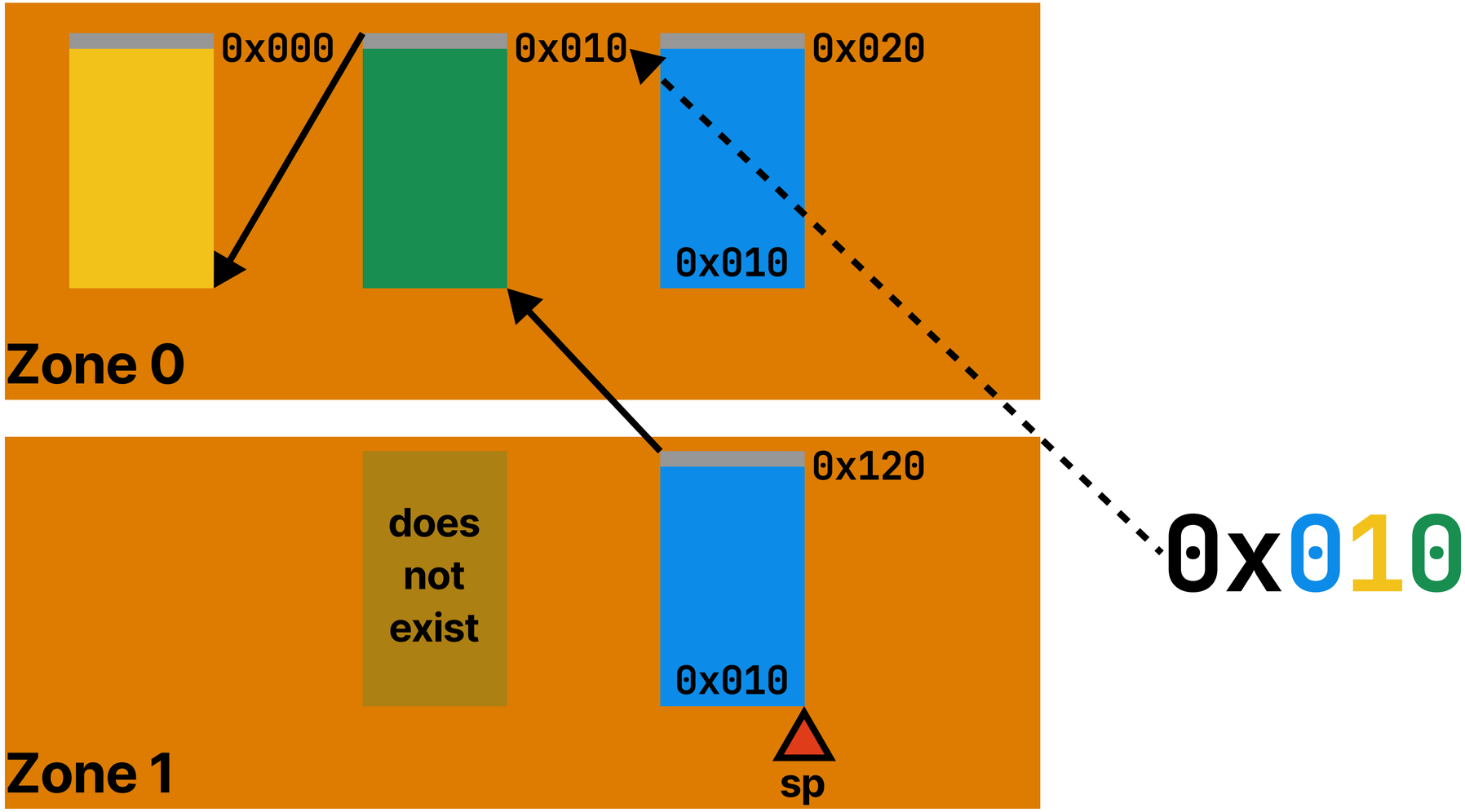
1 target handler id

1 initial zone

0x000

0x010 Update the zone number in the address

Zone Walk



1 target handler id

1 initial zone

Performance

Zone walker carries out one comparison at each zone.

Discussion

Discussion

Performance

Zone walker carries out one comparison at each zone.
The zone depth is a constant in common programs.

Discussion

Performance

Zone walker carries out one comparison at each zone.
The zone depth is a constant in common programs.
Each zone walk incurs a cost of O(1).

Discussion

Performance

Zone walker carries out one comparison at each zone.
The zone depth is a constant in common programs.
Each zone walk incurs a cost of O(1).

Correctness of Zone Walk

Performance

Zone walker carries out one comparison at each zone.
The zone depth is a constant in common programs.
Each zone walk incurs a cost of O(1).

Discussion

Correctness of Zone Walk

The runtime allocates stacks with monotonically increasing id.

Performance

Zone walker carries out one comparison at each zone.
The zone depth is a constant in common programs.
Each zone walk incurs a cost of O(1).

Discussion

Correctness of Zone Walk

The runtime allocates stacks with monotonically increasing id.
The lexical scoping of handler capabilities preserves the monotonicity.

Performance

Zone walker carries out one comparison at each zone.
The zone depth is a constant in common programs.
Each zone walk incurs a cost of O(1).

Discussion

Correctness of Zone Walk

The runtime allocates stacks with monotonically increasing id.
The lexical scoping of handler capabilities preserves the monotonicity.
Thanks to this monotonicity, a single comparison determines the presence of a handler.

Discussion

Performance

Zone walker carries out one comparison at each zone.
The zone depth is a constant in common programs.
Each zone walk incurs a cost of O(1).

Correctness of Zone Walk

The runtime allocates stacks with monotonically increasing id.
The lexical scoping of handler capabilities preserves the monotonicity.
Thanks to this monotonicity, a single comparison determines the presence of a handler.

Parallelism

Virtualizing Continuations

Cong Ma, Max Jung, Yizhou Zhang
University of Waterloo

Thank You