Return Barrier

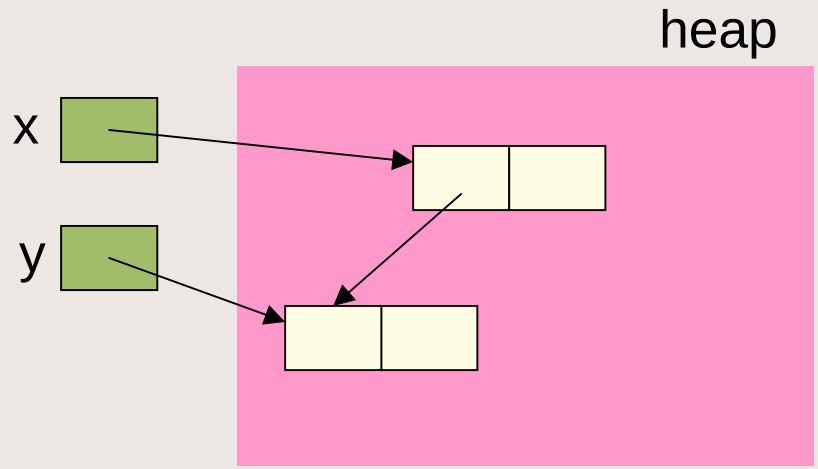
Incremental Stack Scanning for
Snapshot Real-time Garbage Collection

Taiichi Yuasa
Kyoto University

Dynamic Data Allocation

- Lisp, Prolog, C++, Java, C#, …, even BASIC
- allocate an object when required, i.e., dynamically

```
Node x, y;
x = new Node();
y = new Node();
x.left := y;
```

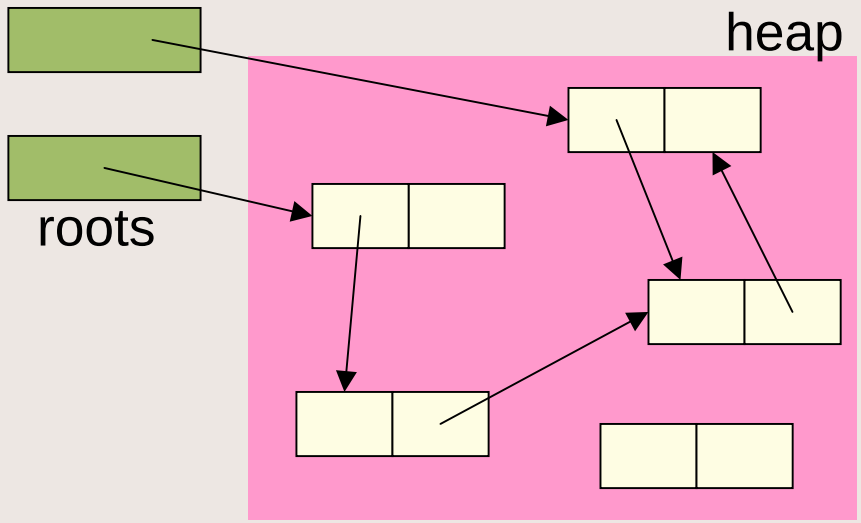


- objects may become useless
  - memory space is limited
  - reclaim unused objects so that they can be reused for further computation

Automatic Garbage Collection

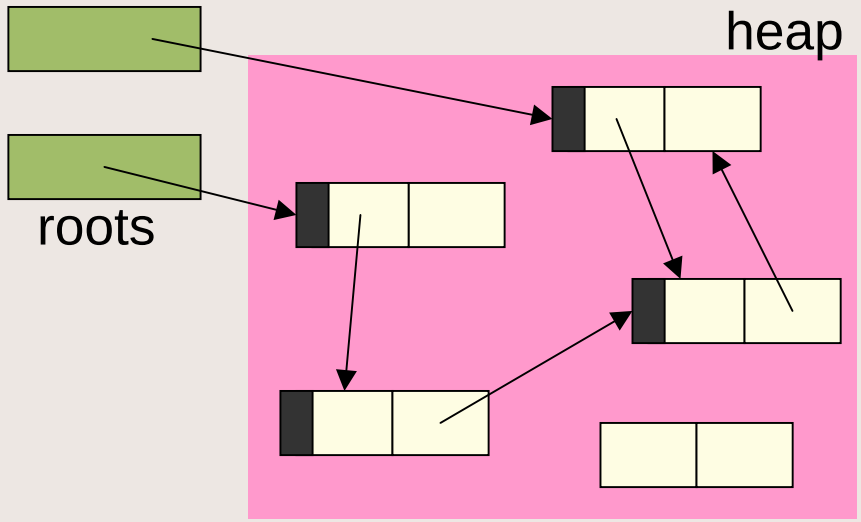
garbage: data objects that can never be accessed

- i.e., those that are not reachable from the roots
- roots: locations that the program can access directly
  e.g., global/local variables, registers, …



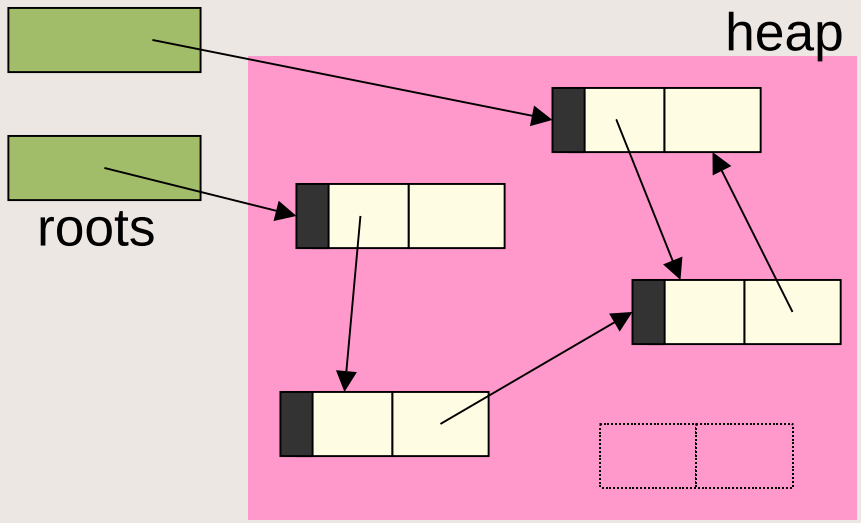
Mark & Sweep GC

- suspend the application program
- mark all objects reachable from the roots
- sweep the entire heap and reclaim all unmarked objects
- resume the application program



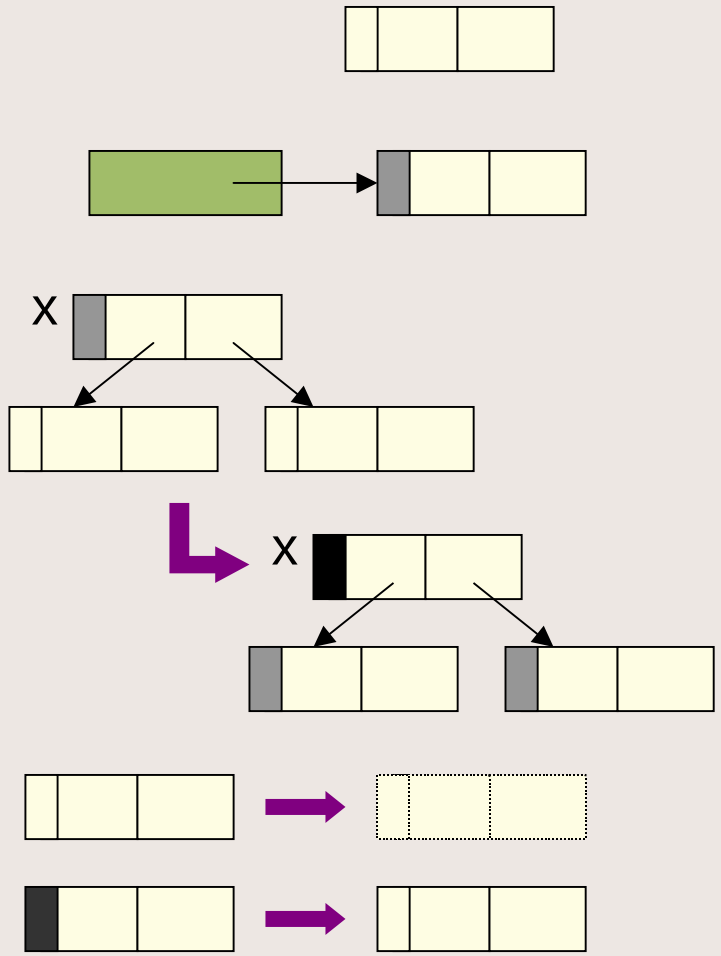
Mark & Sweep GC

- suspend the application program
- mark all objects reachable from the roots
- sweep the entire heap and reclaim all unmarked objects
- resume the application program



Tri-colour Algorithm

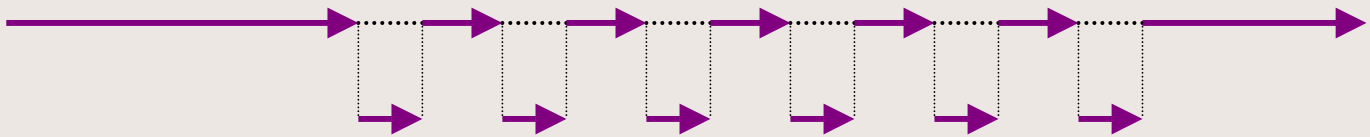
- all objects are initially white
- for each root,
  - make gray the pointed object
- while grays remain,
  - choose a gray object X
  - make X black
  - make gray all white objects pointed to from X
- for each object in the heap
  - if white, free it
  - if black, make it white



Incremental (Real-time) GC

- application program is suspended during GC
- each GC typically takes seconds to minutes
- not suitable for real-time applications

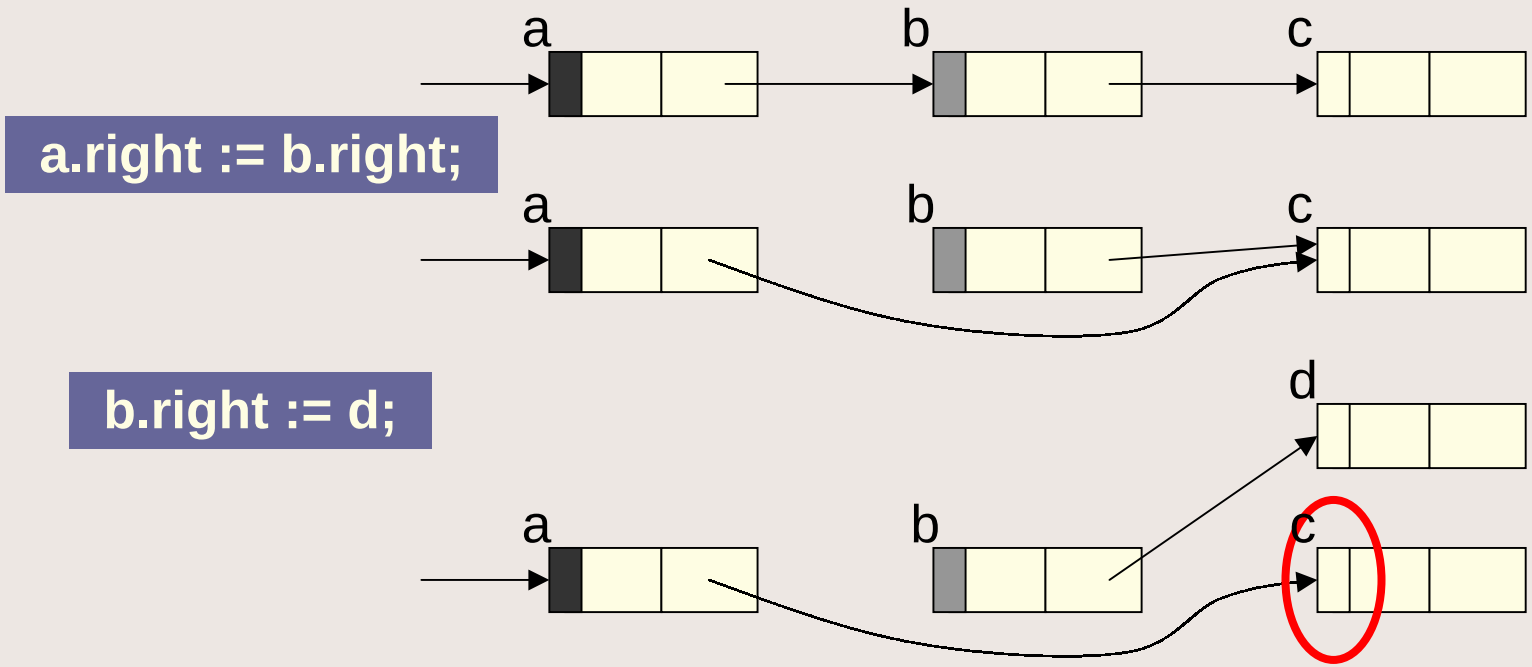
- one GC chunk (mark/sweep N objects) at a time



- each time a new object is created

Problem

- application keeps running during GC
- reference relations may change during GC
- may fail to mark some objects in use

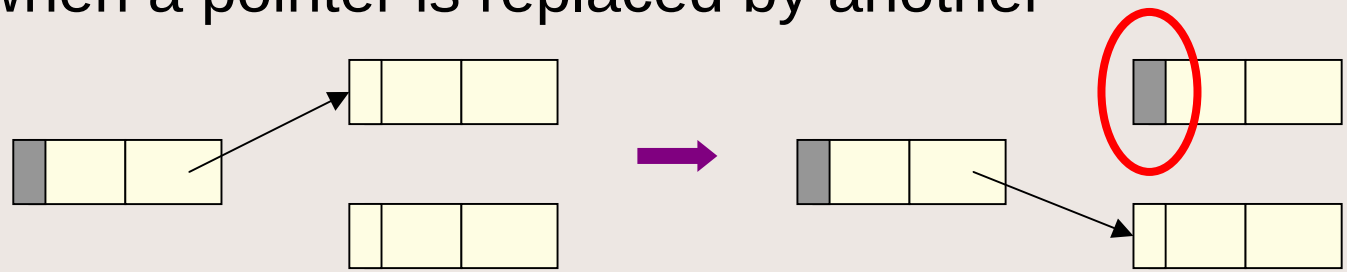


Snapshot Real-time GC
by Yuasa 1990

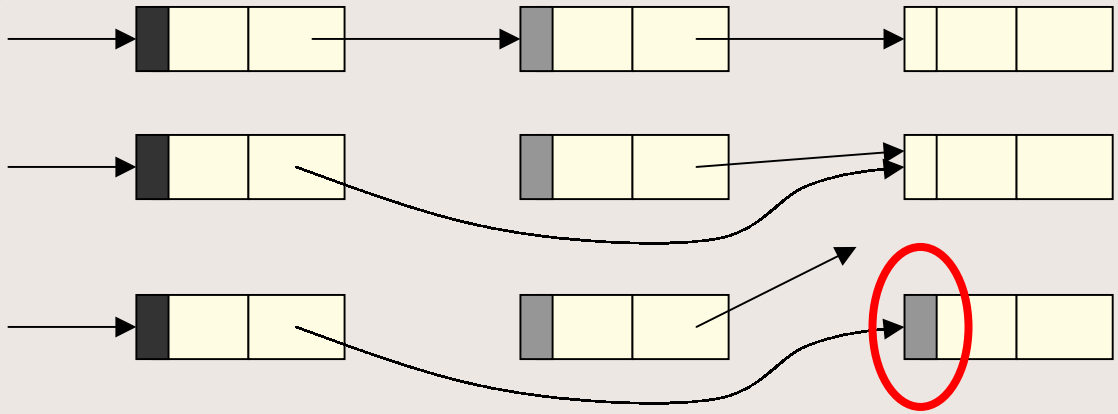
write barrier:

make gray the object previously pointed to,

when a pointer is replaced by another



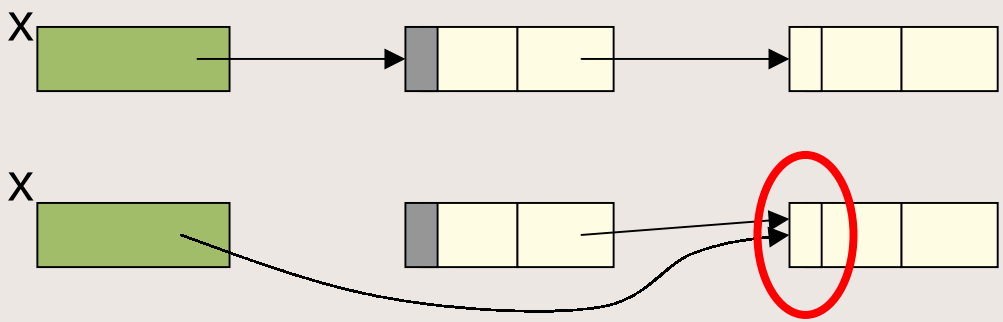
for the example:



Snapshot Real-time GC
by Yuasa 1990

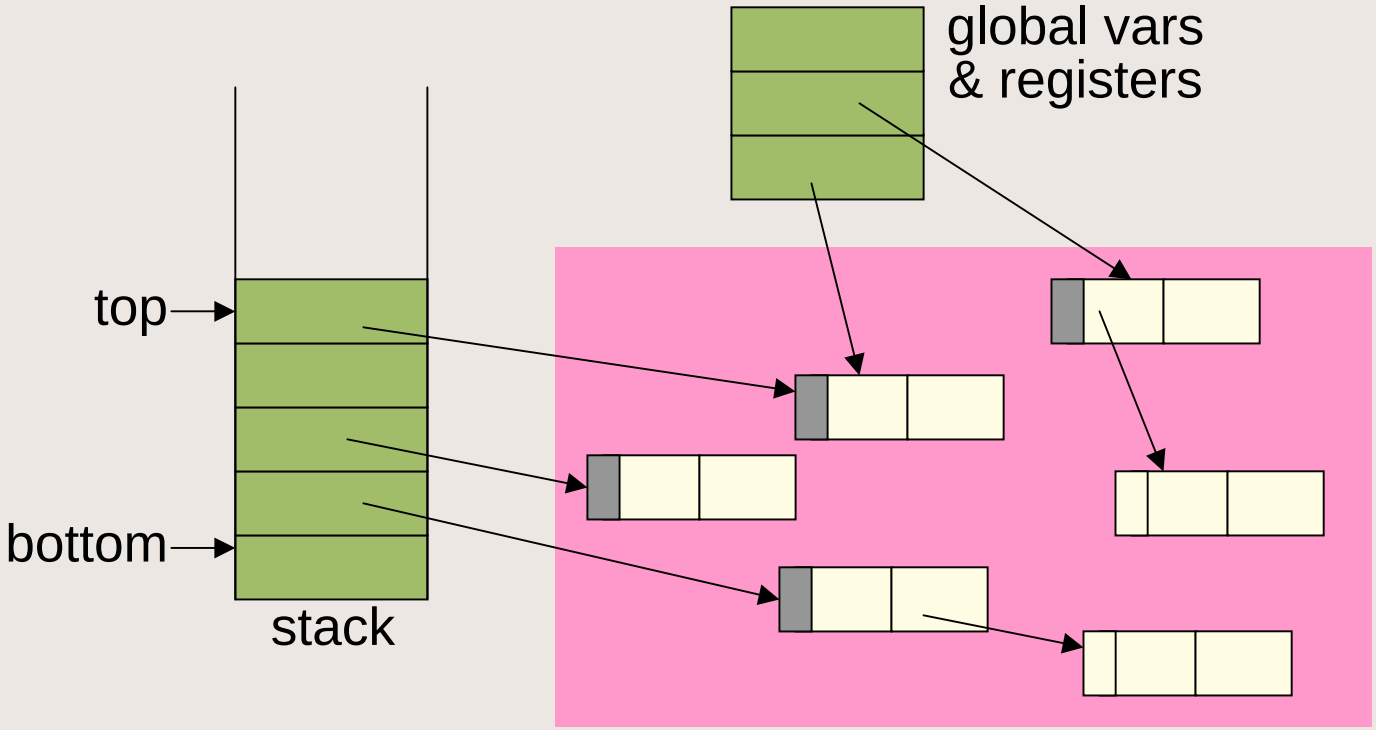
- all objects in use at the beginning of a GC
  are guaranteed to become black eventually
- at the beginning of a GC,
  make gray all objects directly pointed to from roots
- no write barrier necessary for roots
  – previous object eventually becomes black
- efficient => used in many systems

x := x.right;

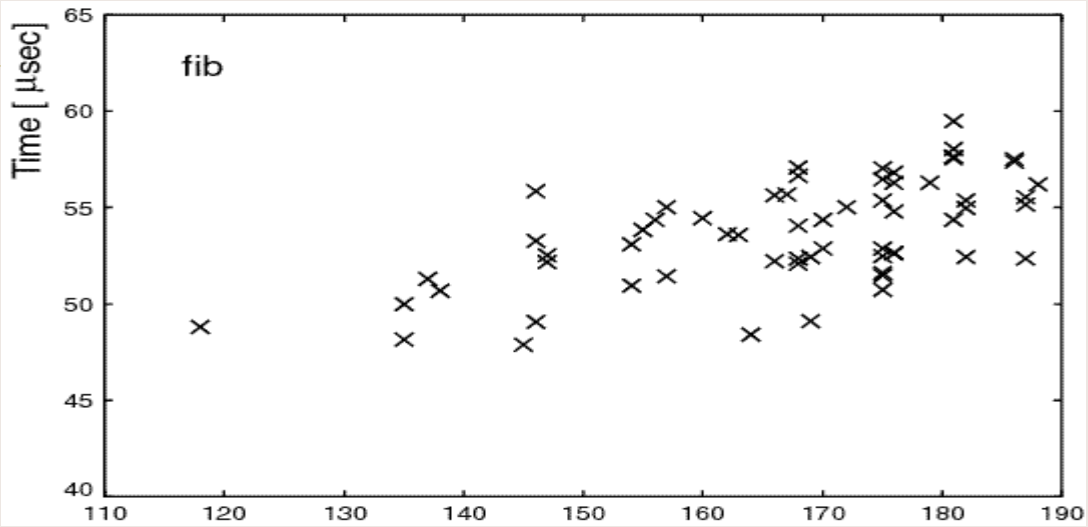


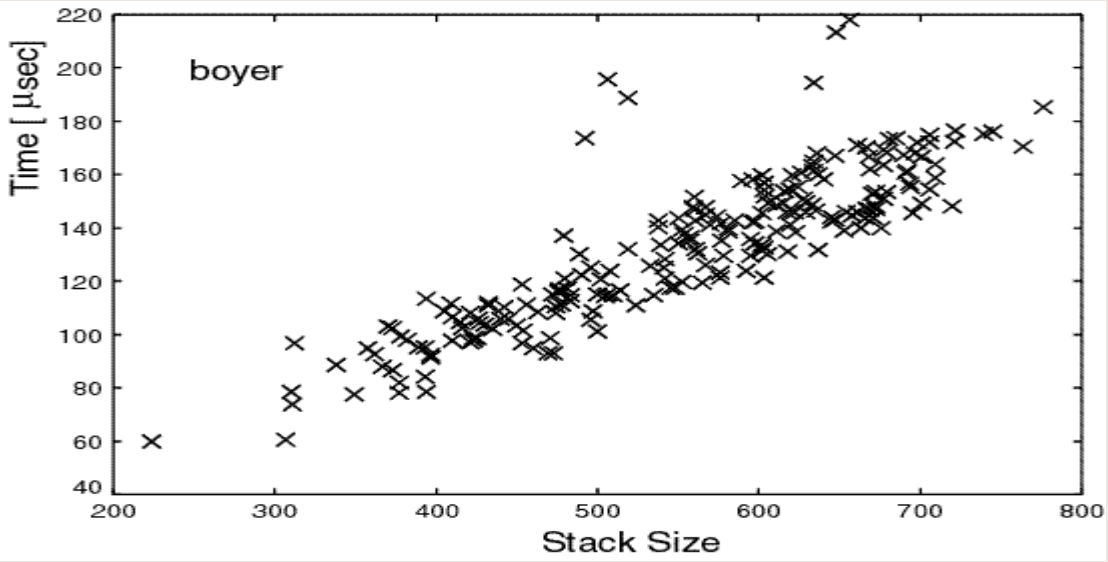
Root Scanning

- make gray all objects directly pointed to from roots
- local variables are stored in the stack
- stack size changes dynamically



Stack Size & Suspension Time





would like to reduce suspension time
< 100 µsec

Why 100 µsec Suspension?

特別企画

Business Linux

■ヒューマノイドロボット HOAP-1

UNIX USER 2002/6

We chose realtime Linux because it allows control in 100 µs.

... However, this model is controled in 1 ms, because of the constraint of USB communication protocol.

ですとリアルタイム性が保証されていませんが、リアルタイムLinuxなら100 µsまでの制御が可能になります。

リアルタイムLinuxとして有名なものにRT-LinuxやART-Linuxがあります。HOAP-1ではRT-Linuxを使っています。

Q RT-Linuxを選択された理由は？

A RT-Linuxの前に使っていたのはRTXというWindowsベースの製品です。こちらですと、ライセンスだけで数百万かかってしまいます。

RT-Linuxはオープンソースなので研究者の利用には最適です。

あとLinuxを採用したメリットとして、TRONと比べると研究者にとって開発が容易であることと、USBが使いやすいといった点があります。

Q ロボットを制御するうえで、100 µsといった単位は必須なのですか？

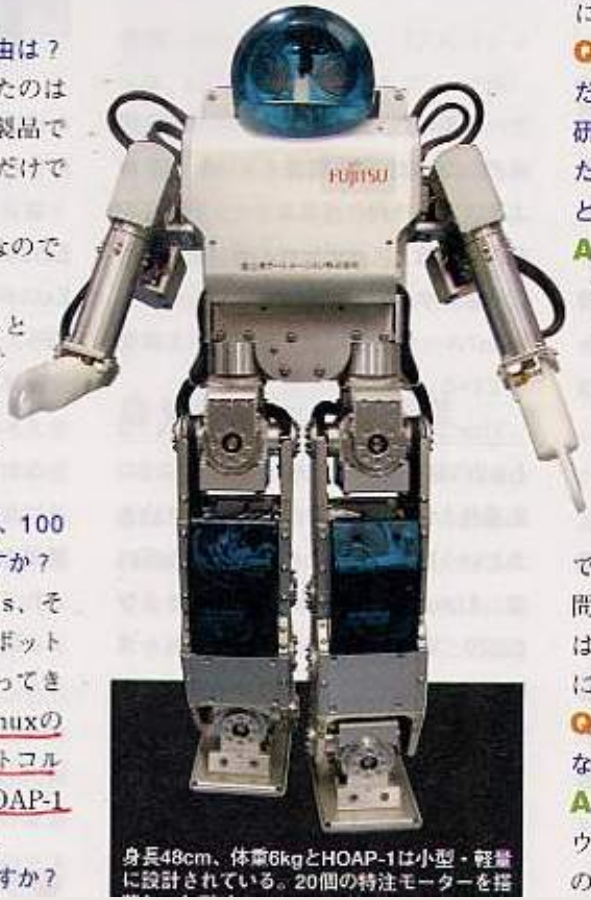
A 設定を10msから5ms、2ms、そして1msとしていくだけで、ロボットの動きはどんどんスムーズになってきます。ただ、1msを超えるとLinuxの限界ではなく、USBの通信プロトコルの制約を受けてしまうため、HOAP-1では1msで制御を行っています。

Q 大学での導入事例はいかがですか？

見えてきたという感じでしょうか？

A いや、まださまざまなアルゴリズムを研究している段階です。

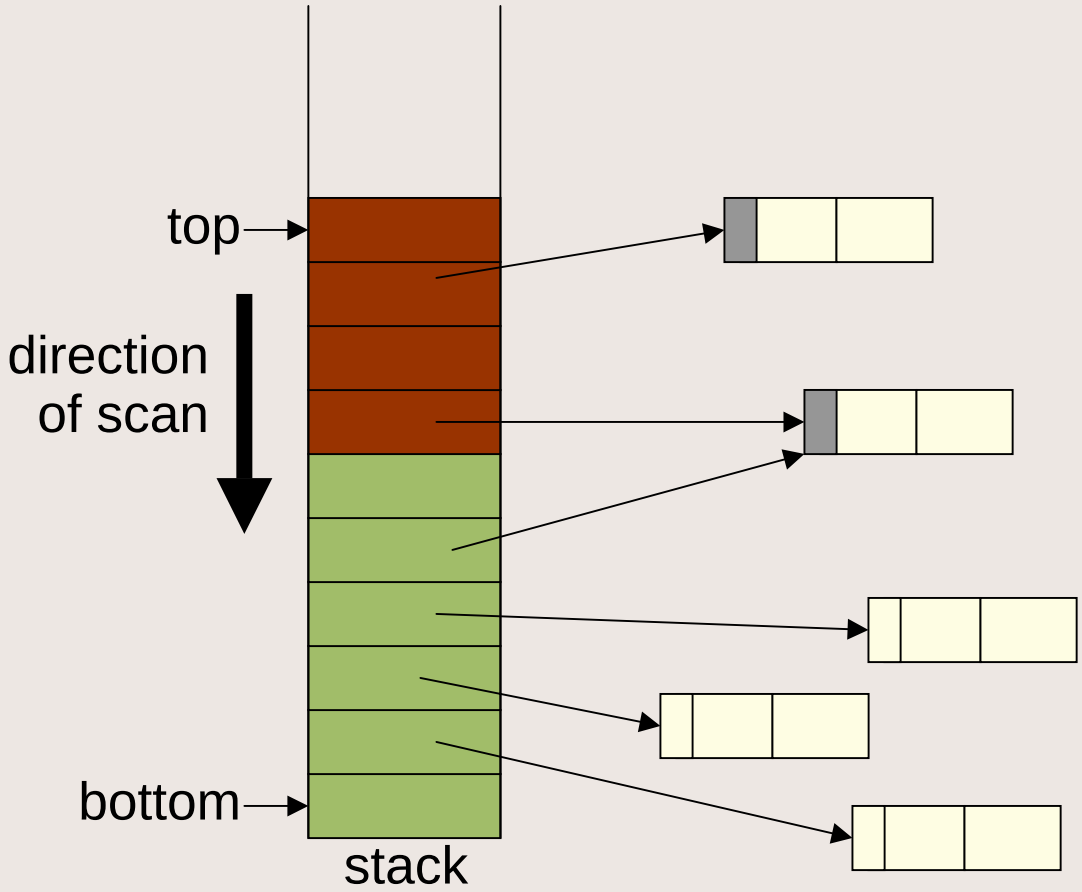
ロボットの場合、壊してしまうと修理に費用がかかるので、新しいアルゴリズムを試す場合、まずシミュレータ



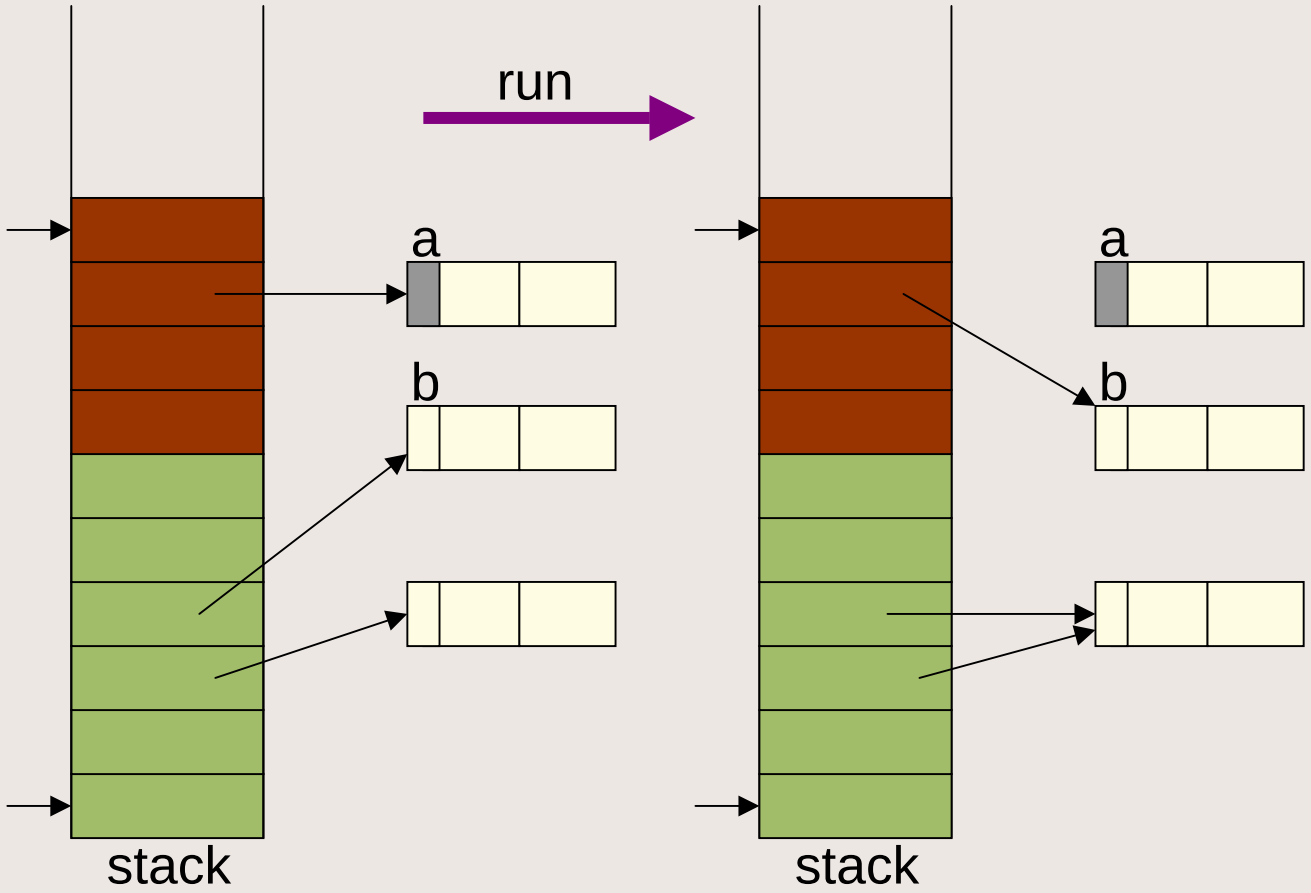
身長48cm、体重6kgとHOAP-1は小型・軽量に設計されている。20個の特注モーターを搭

Incremental Stack Scanning

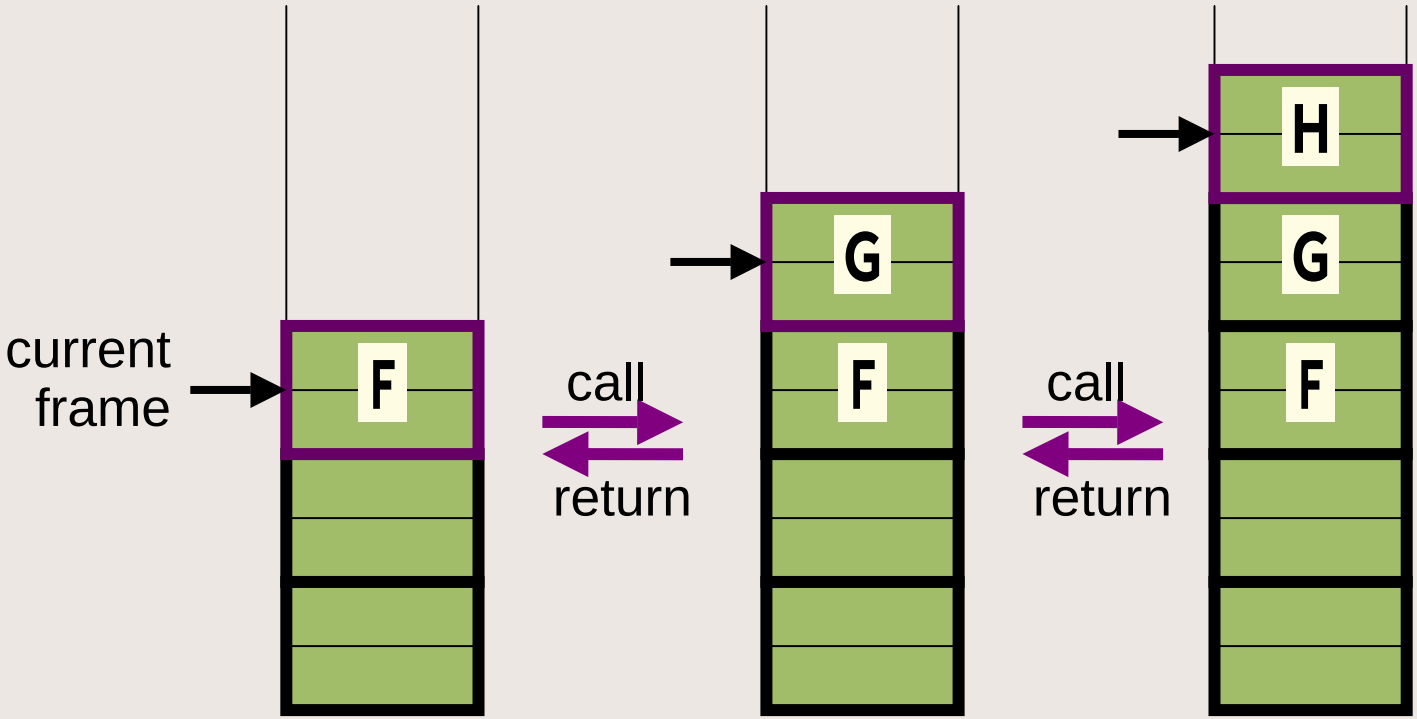
- scan the stack little by little



Problem

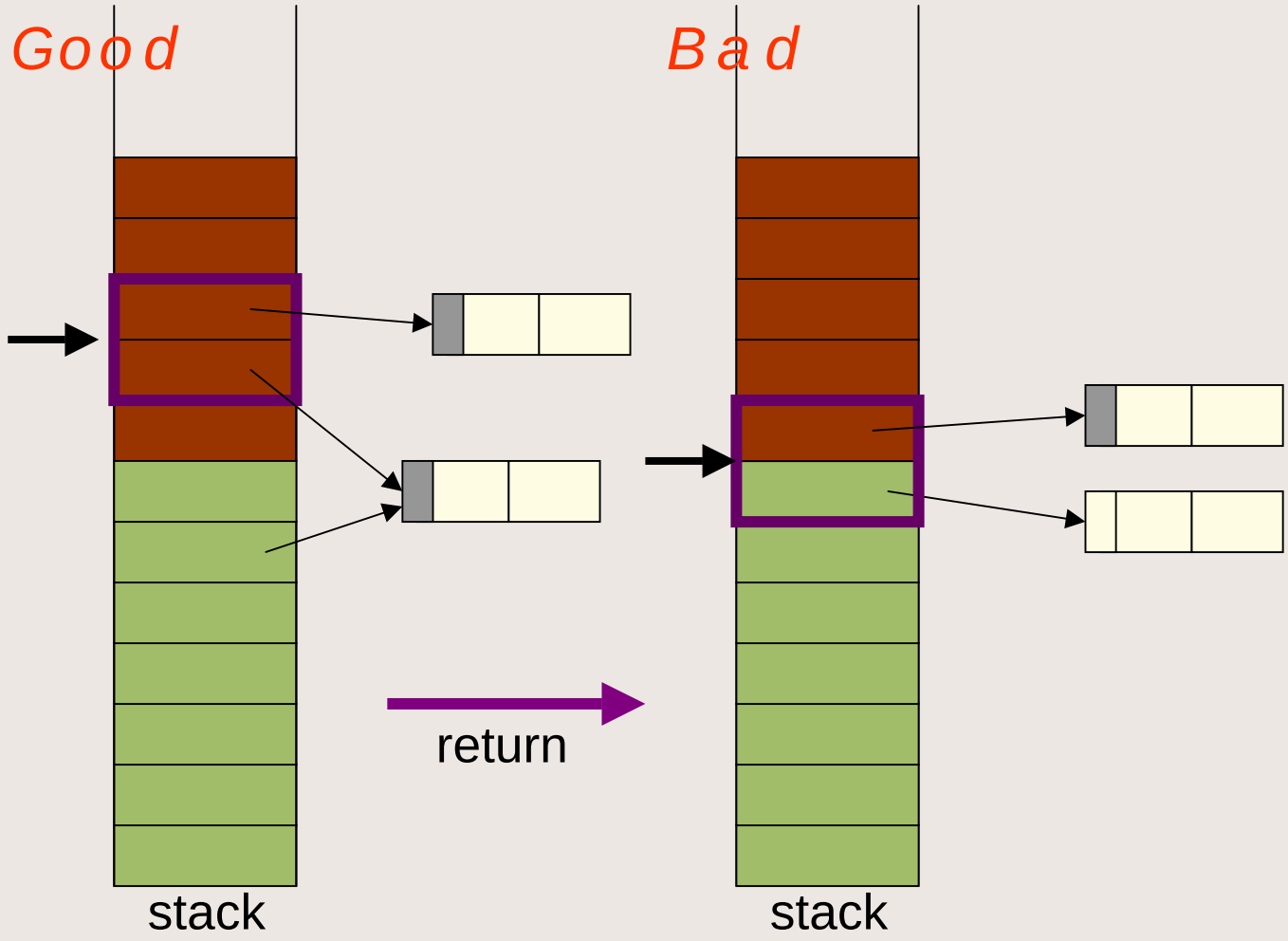


Function Frames

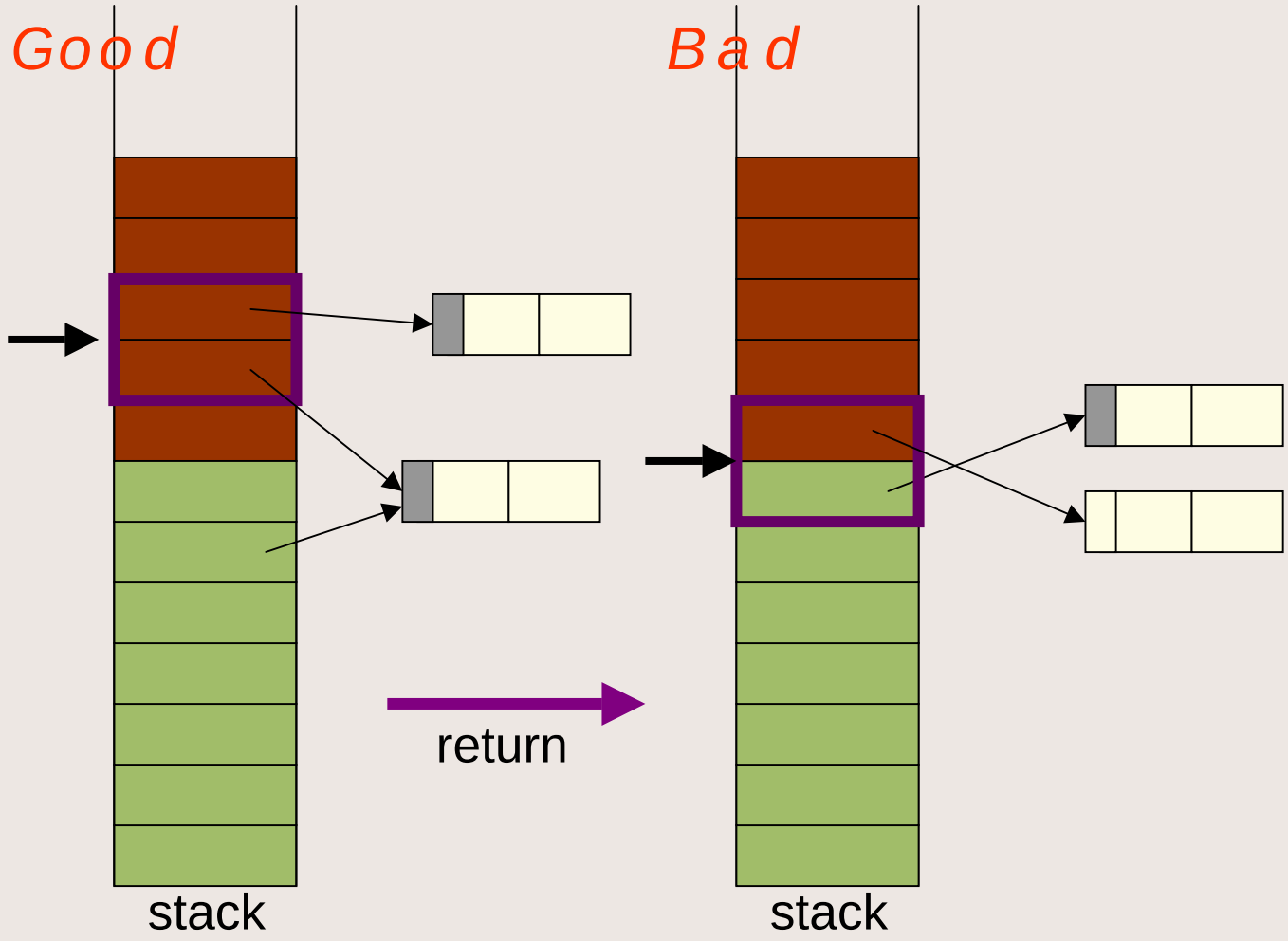


Only variables in the current frame can be accessed.

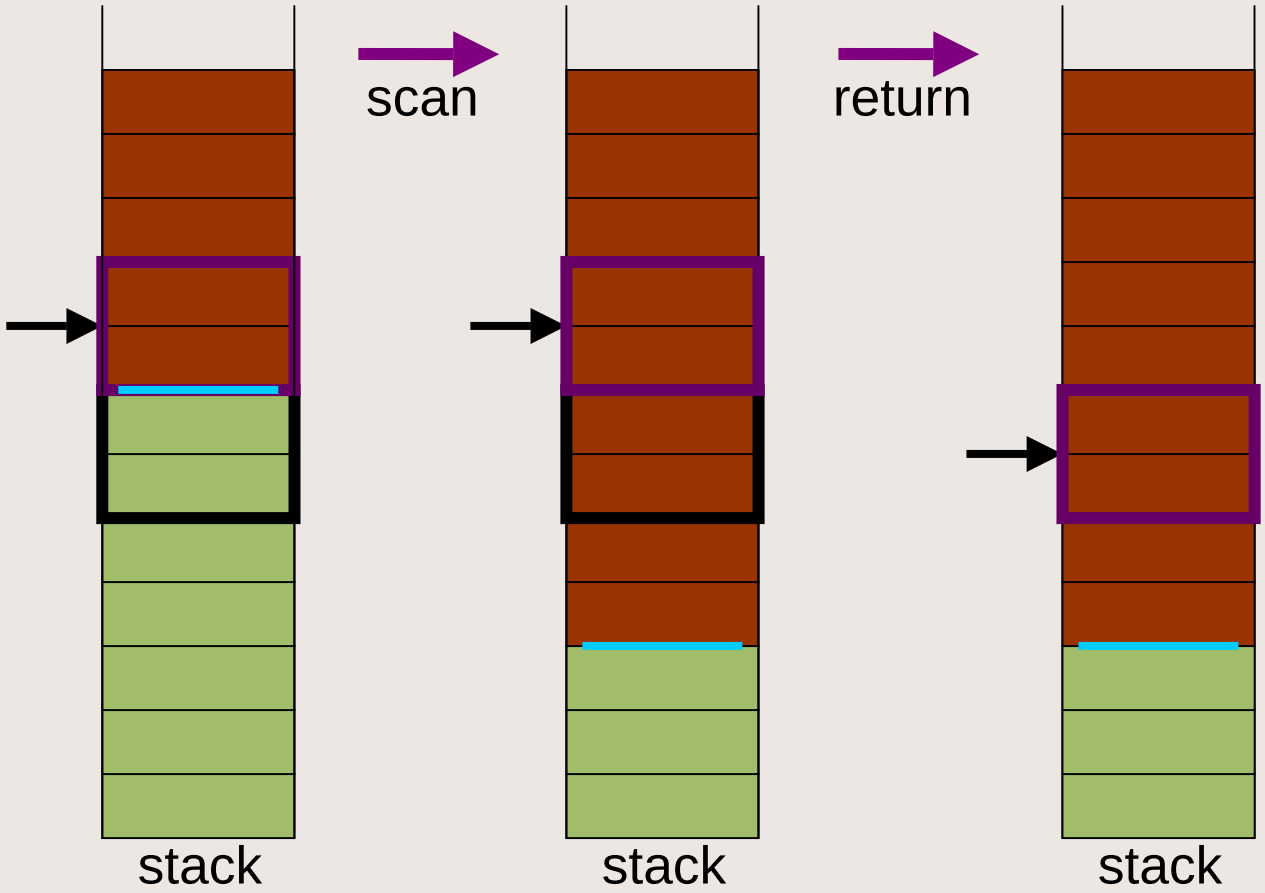
Scanning *vs* Return



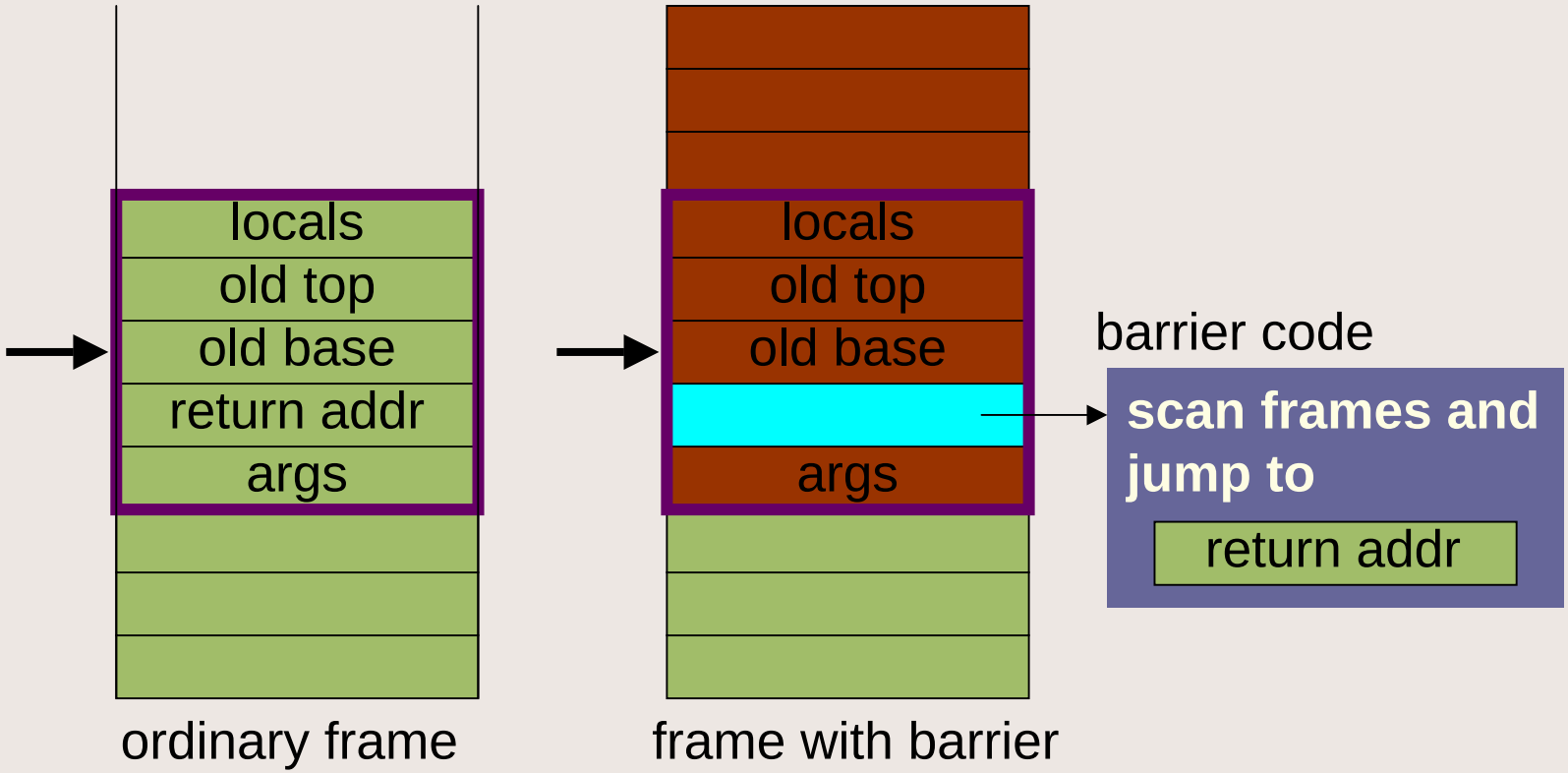
Scanning *vs* Return



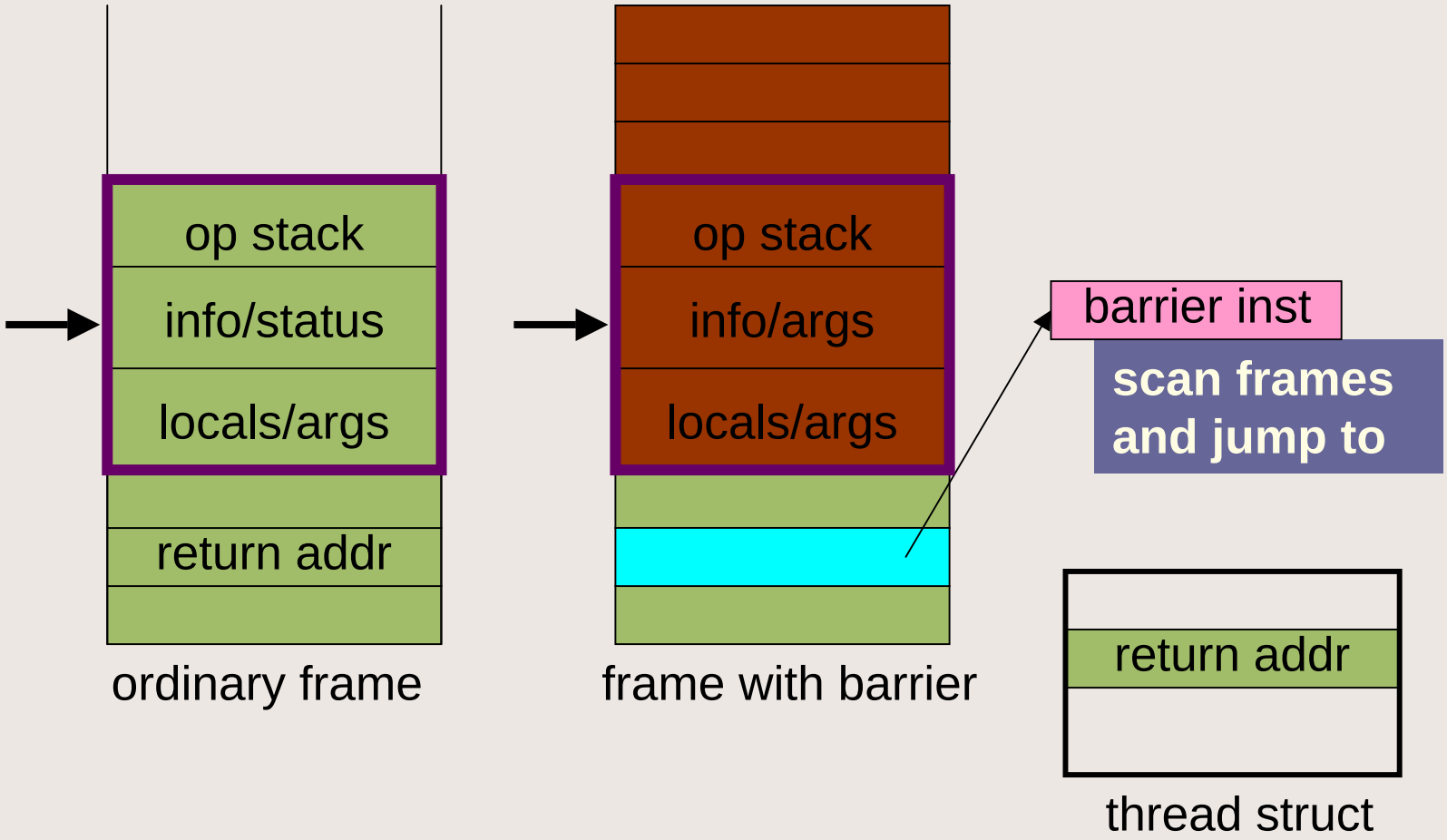
Return Barrier



It’s free (for C-like frames)

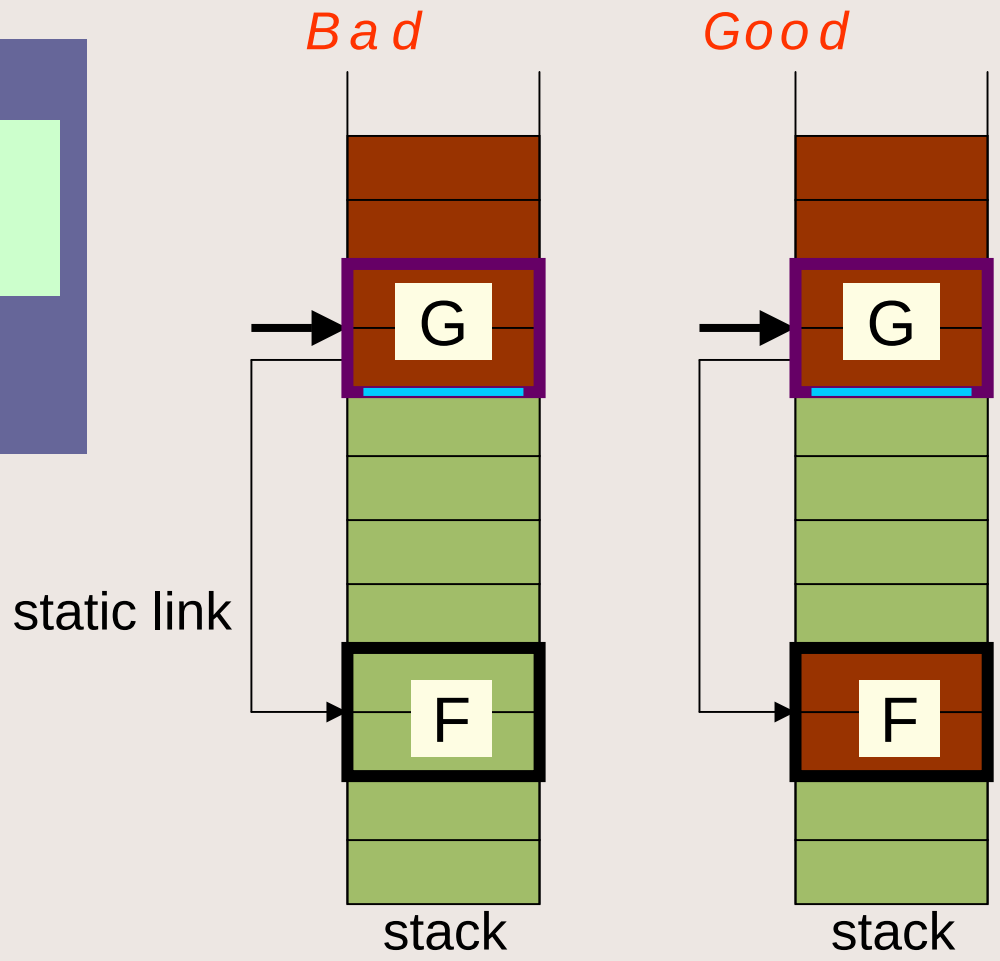


It’s free (case Java)



Remark (local functions)

```
(define (F x)
  
  (G x x)
)
```



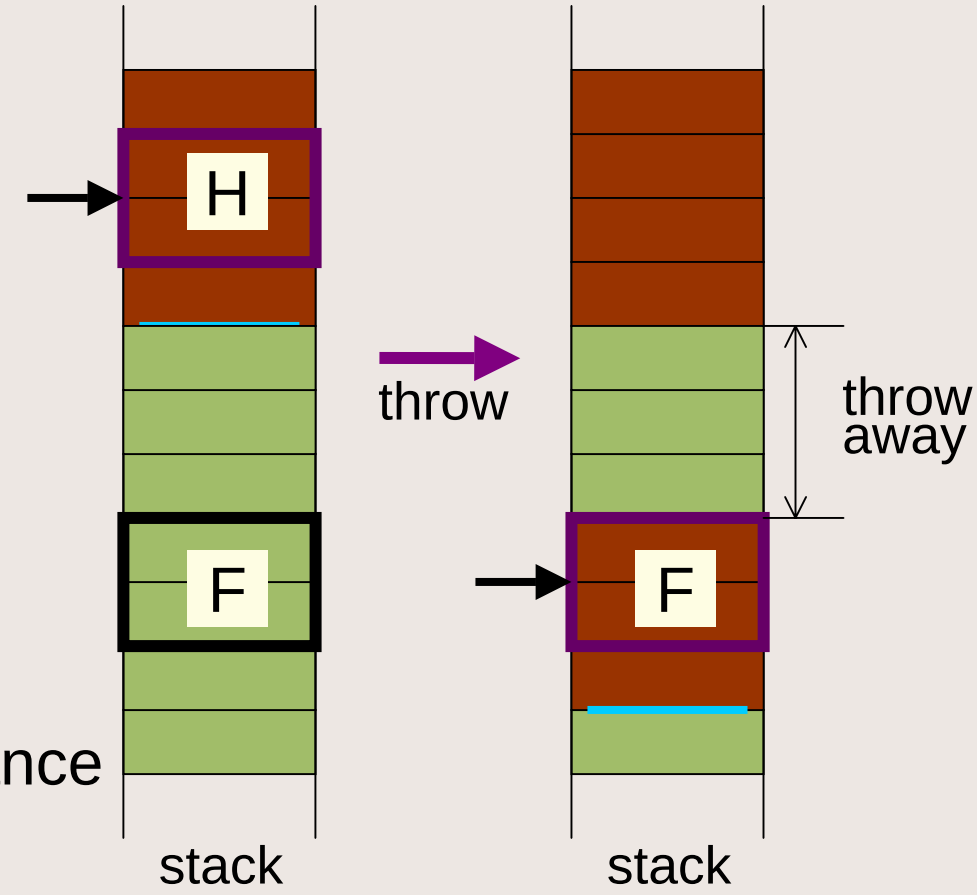
Remark (catch & throw)

```
(define (F x)
    (catch 'A (G))
)
```

```
(define (H)
    (throw 'A ...)
)
```

not a true snapshot!

⇨ better performance



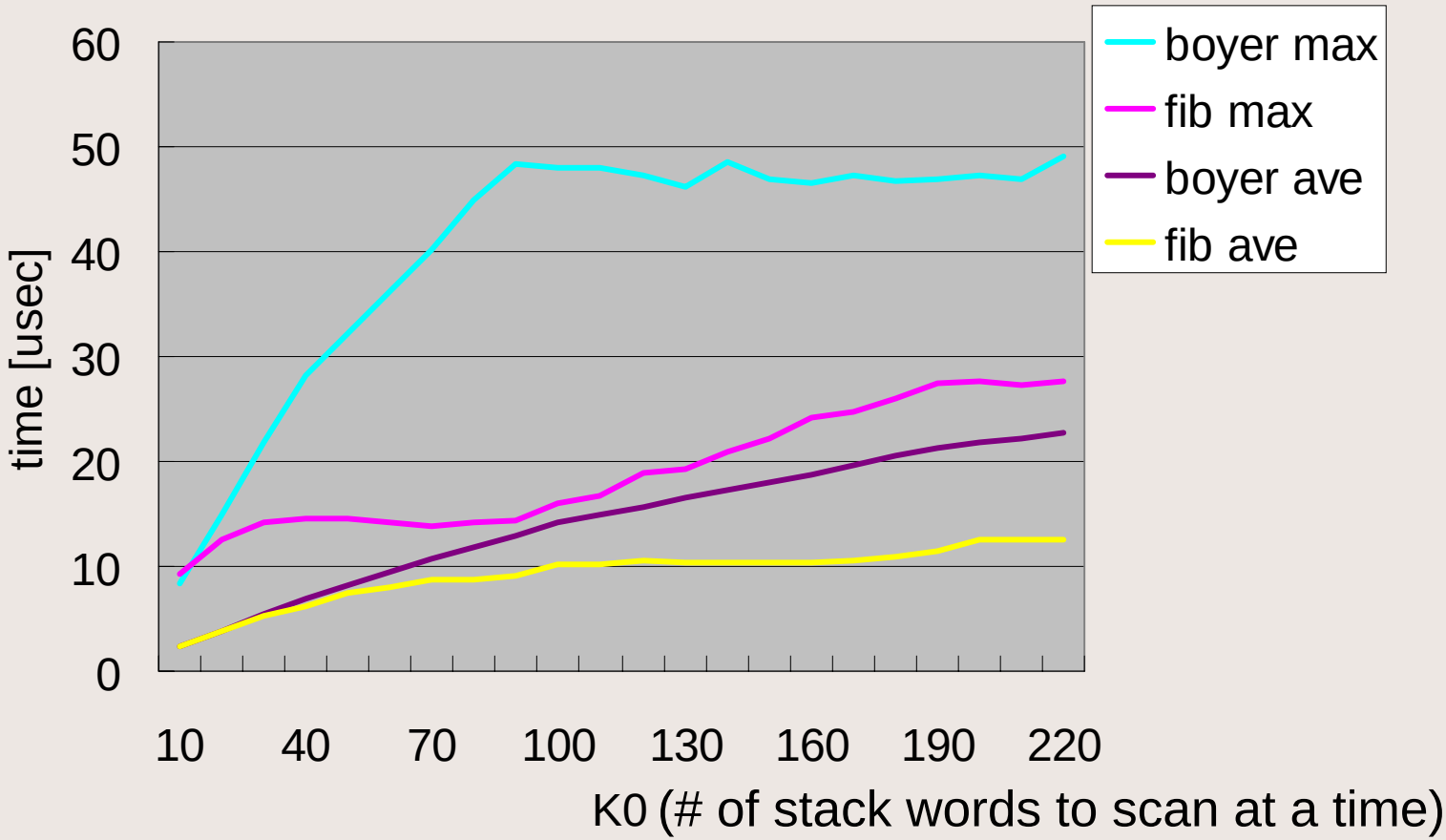
Implementation for KCL (Kyoto Common Lisp)

- stacks
  - value stack
  - bind stack
  - frame stack
  - invocation history stack
  - C language stack (KCL does not access this)

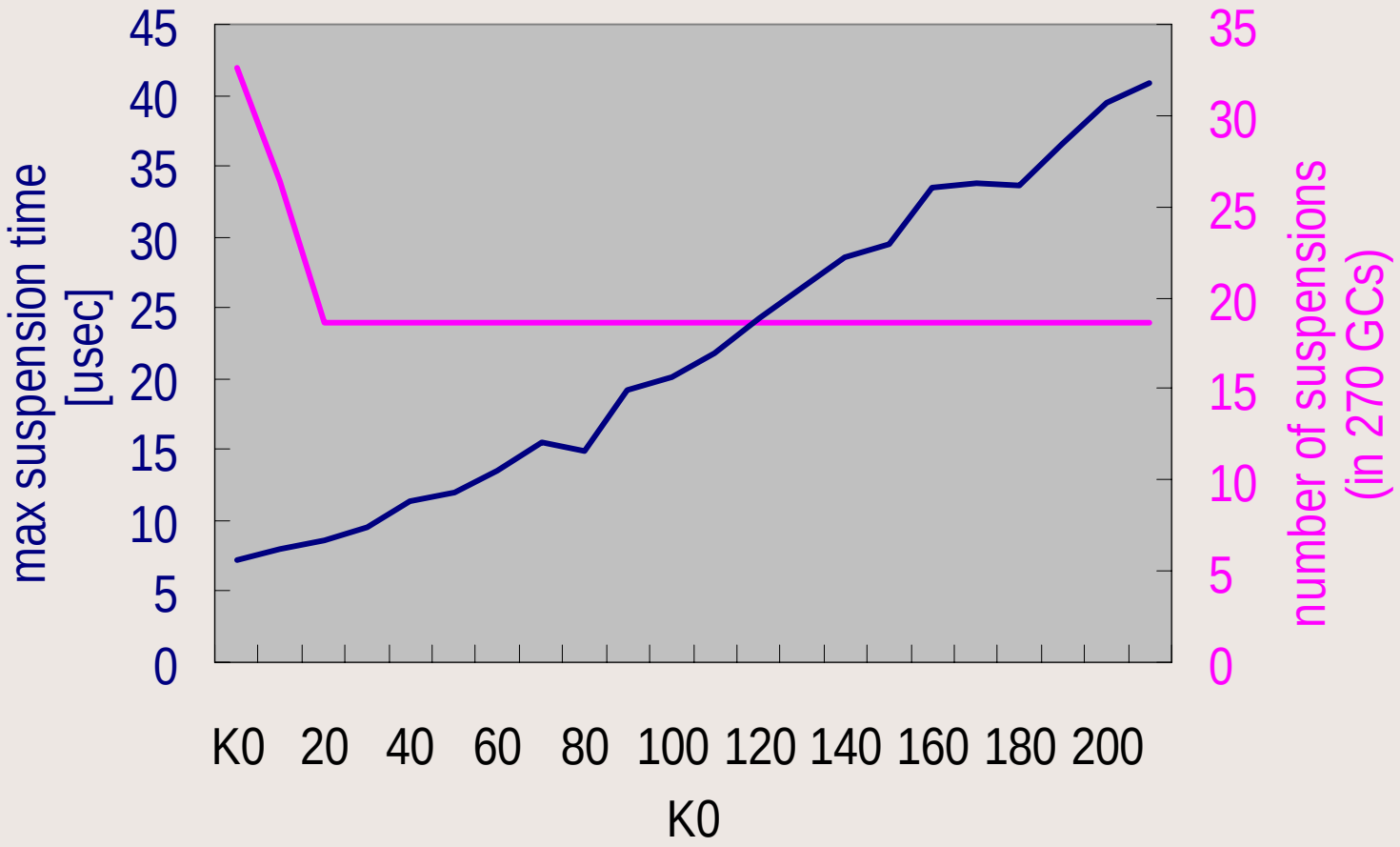
return addresses are pushed on the C stack
⇒cannot handle return addresses
⇒needs explicit barrier checking on function returns

- system variables
  - only 18 variables

Suspension Times



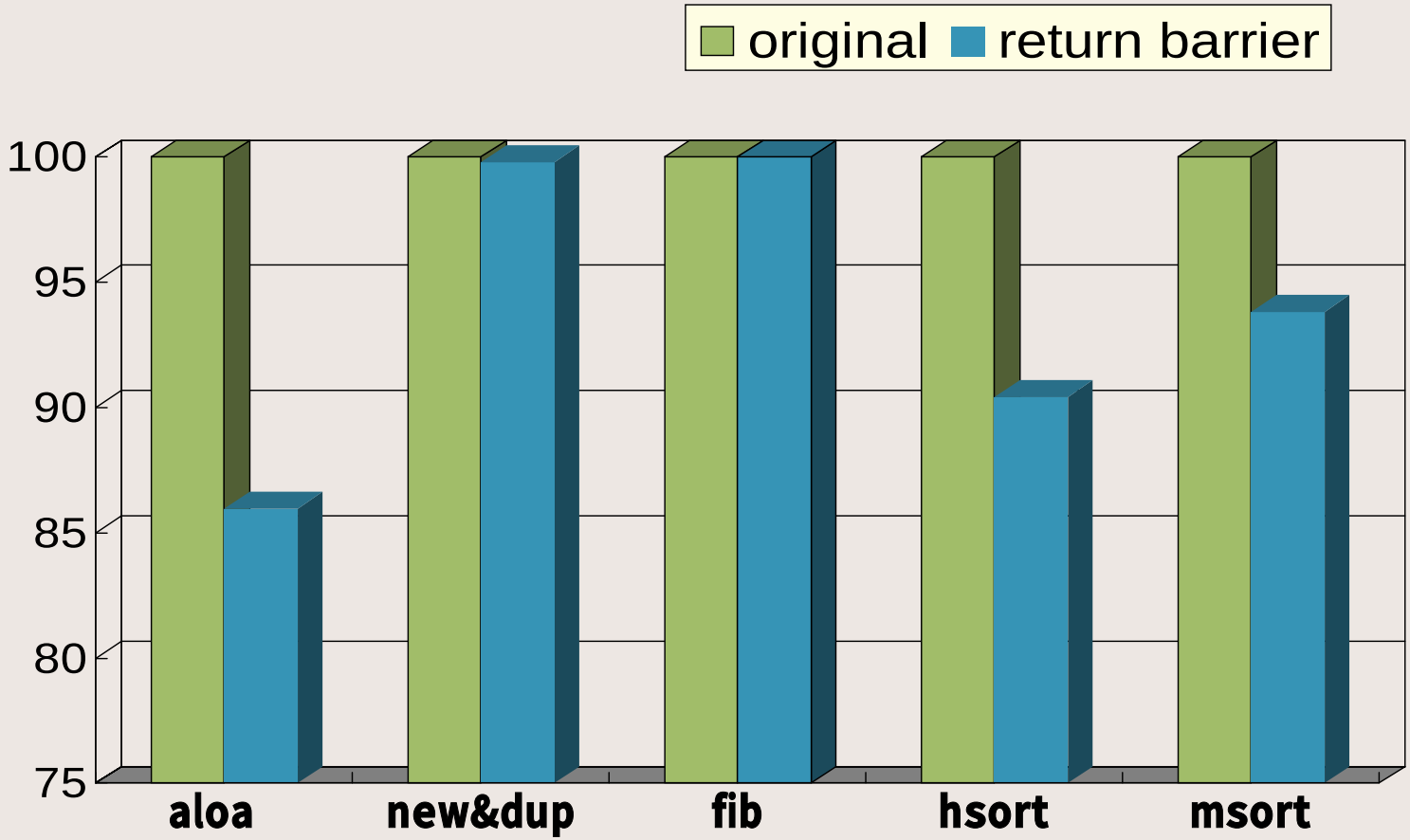
Suspension Times by Return Barrier



Implementation for JeRTy

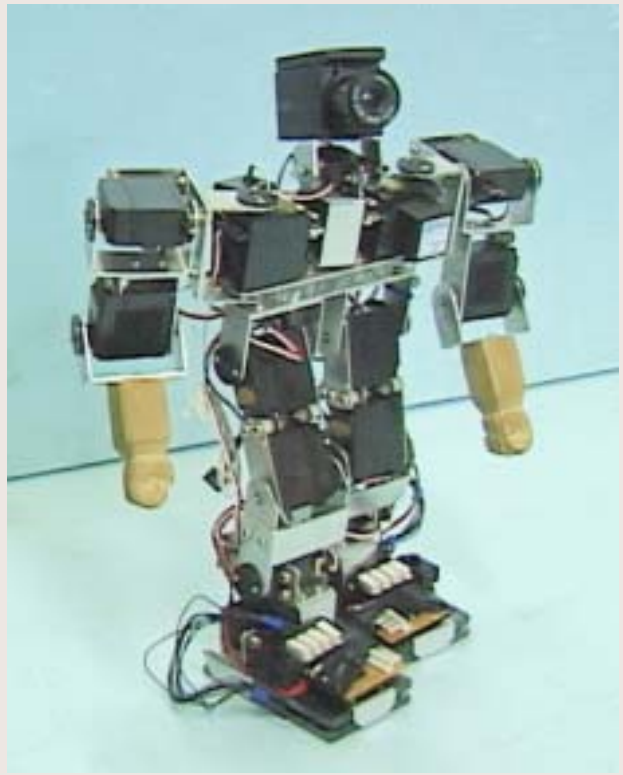
- Java runtime system of Omron Inc.,
- for process control
- snapshot real-time GC
  - with write barrier for roots
- we implemented return barrier and removed write barrier for roots

Benchmark Results



Implementation for EusLisp

- multi-threaded Lisp system of Tokyo Univ.
- to control robots (humanoids)
- badly needs a real-time GC
- they implemented snapshot GC ...
- but stack scanning is done at once
- we have implemented return barrier for the system
- They are rewriting C code into Lisp!



Parallel GC

- The mutator accesses above the return barrier.
- The collector accesses below the return barrier.
- No lock is necessary to access the stack.
- The return barrier need to be locked when moved.