Return Barrier

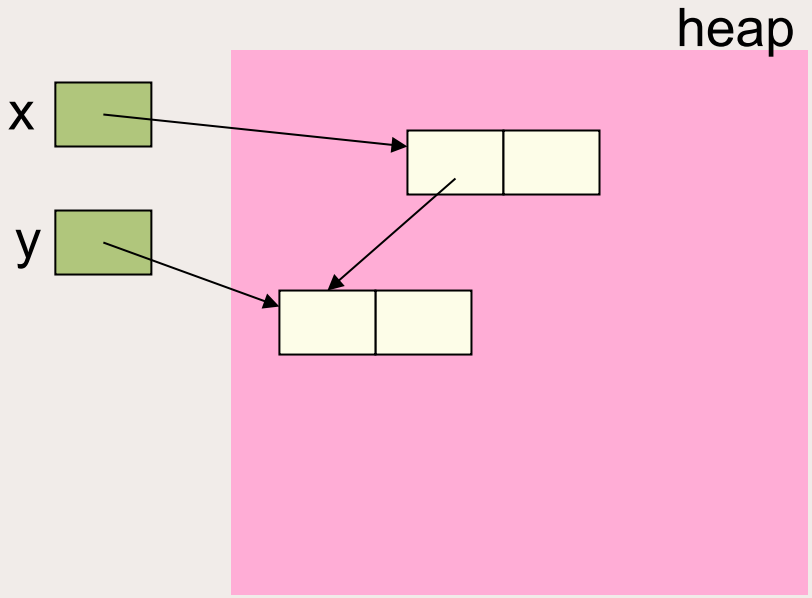
Incremental Stack Scanning for Snapshot Real-time Garbage Collection

Taiichi Yuasa
Kyoto University

Dynamic Data Allocation

- don't know how many data objects are necessary
  - allocate an object when required, i.e., dynamically

```
var x,y: ^node;
new(x);
new(y);
x^.left := y;
```



Freeing Allocated Objects

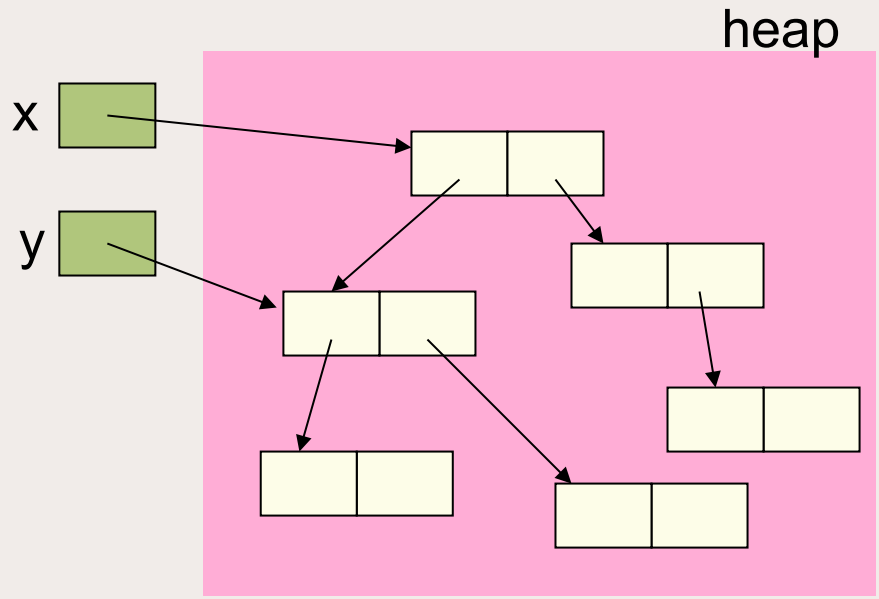
- objects may become useless
  - memory space is limited
  - free an object to reuse its memory space
- before

```
y->left := null;
```

- do

```
free(y->left);
```

- not recommended
  - forget freeing
  - error-prone



Freeing Allocated Objects

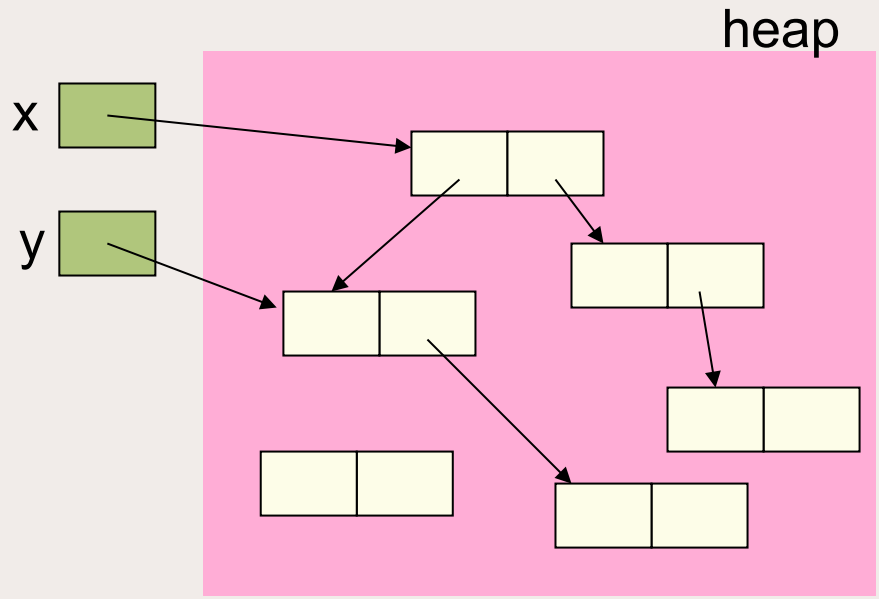
- objects may become useless
  - memory space is limited
  - free an object to reuse its memory space
- before

```
y->left := null;
```

- do

```
free(y->left);
```

- not recommended
  - forget freeing
  - error-prone



Freeing Allocated Objects

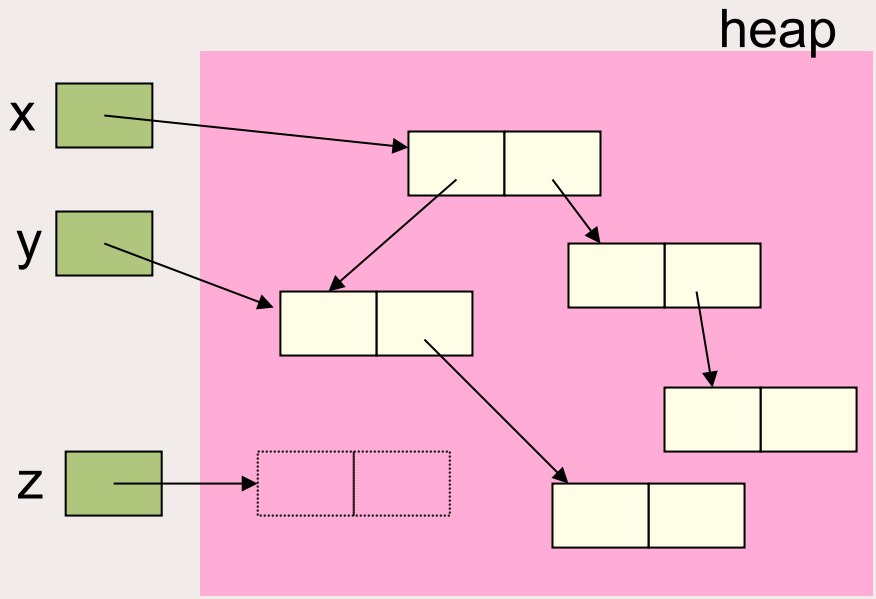
- objects may become useless
  - memory space is limited
  - free an object to reuse its memory space
- before

```
y->left := null;
```

- do

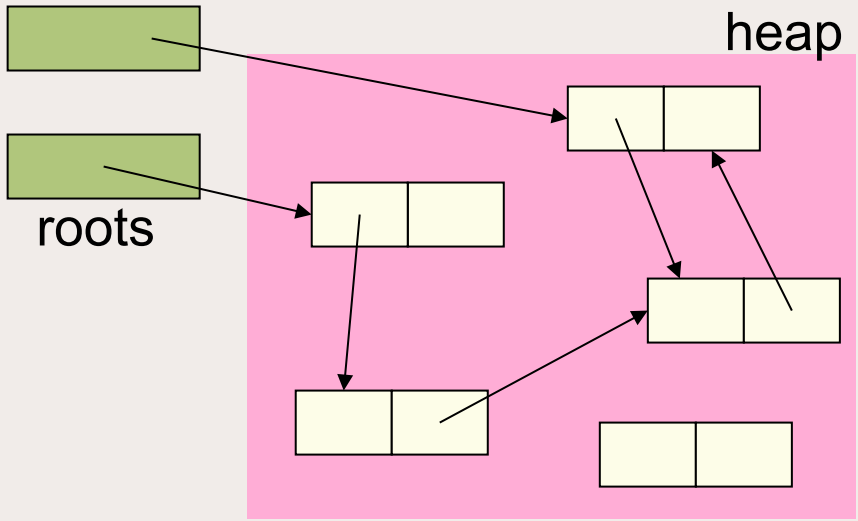
```
free(y->left);
```

- not recommended
  - forget freeing
  - error-prone



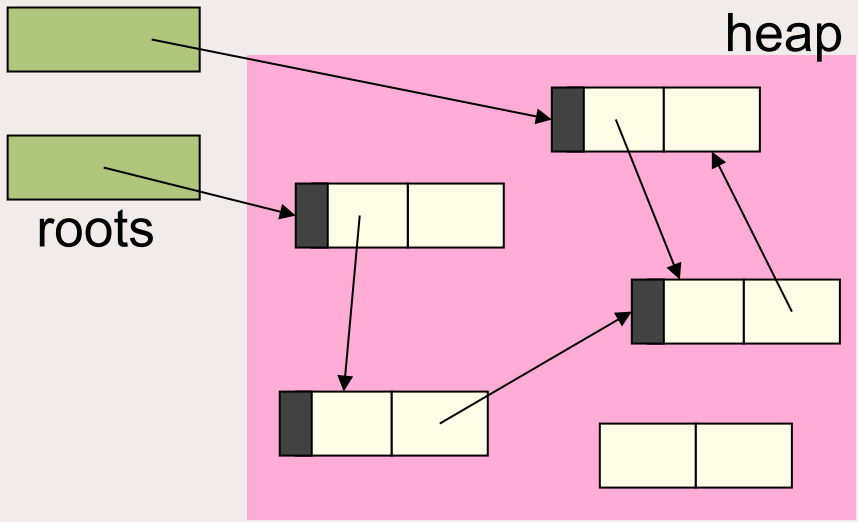
Automatic Garbage Collection

- Lisp, Prolog, C++, Java, C#, …
- 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, …



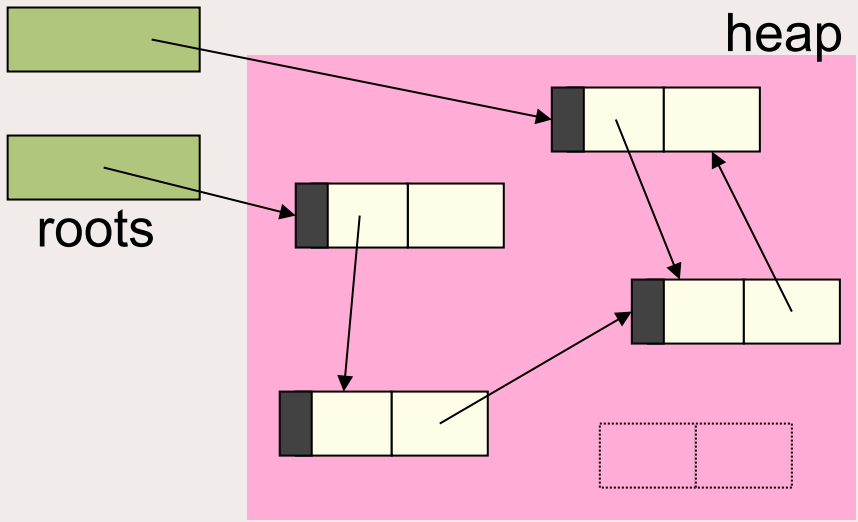
Mark & Sweep GC

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



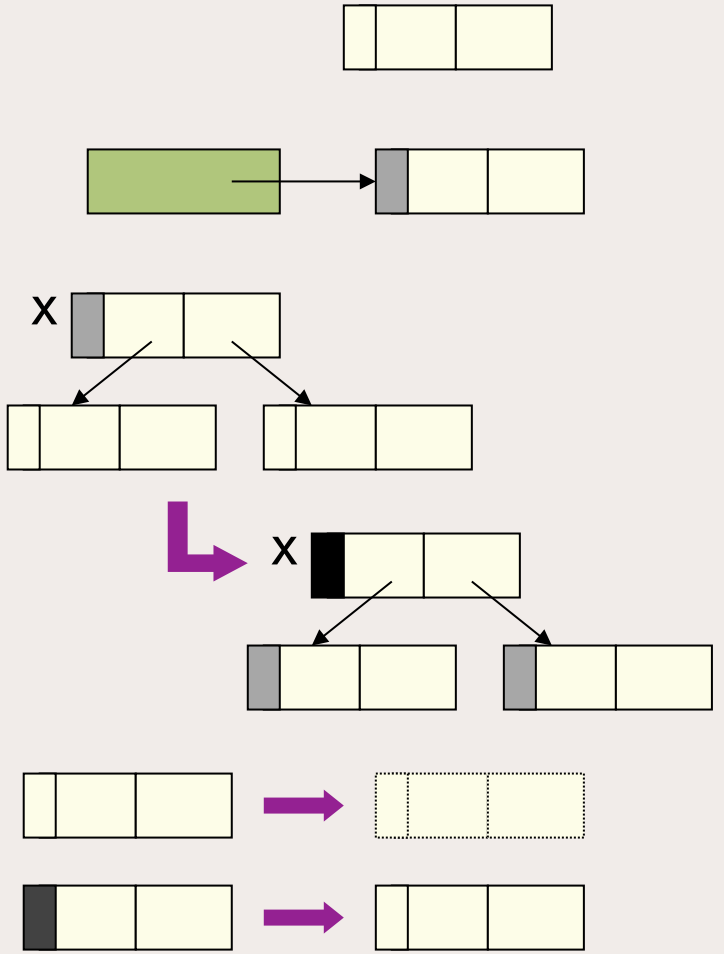
Mark & Sweep GC

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



Tri-colour Algorithm

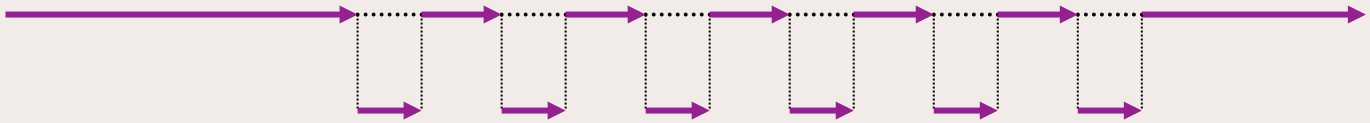
- all objects are initially white
- for each root,
  - make gray the pointed object
- while grays remain,
  - choose a gray object X
  - make X black
  - make gray all white objects pointed to from X
- for each object in the heap
  - if white, free it
  - if black, make it white



Real-time GC

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

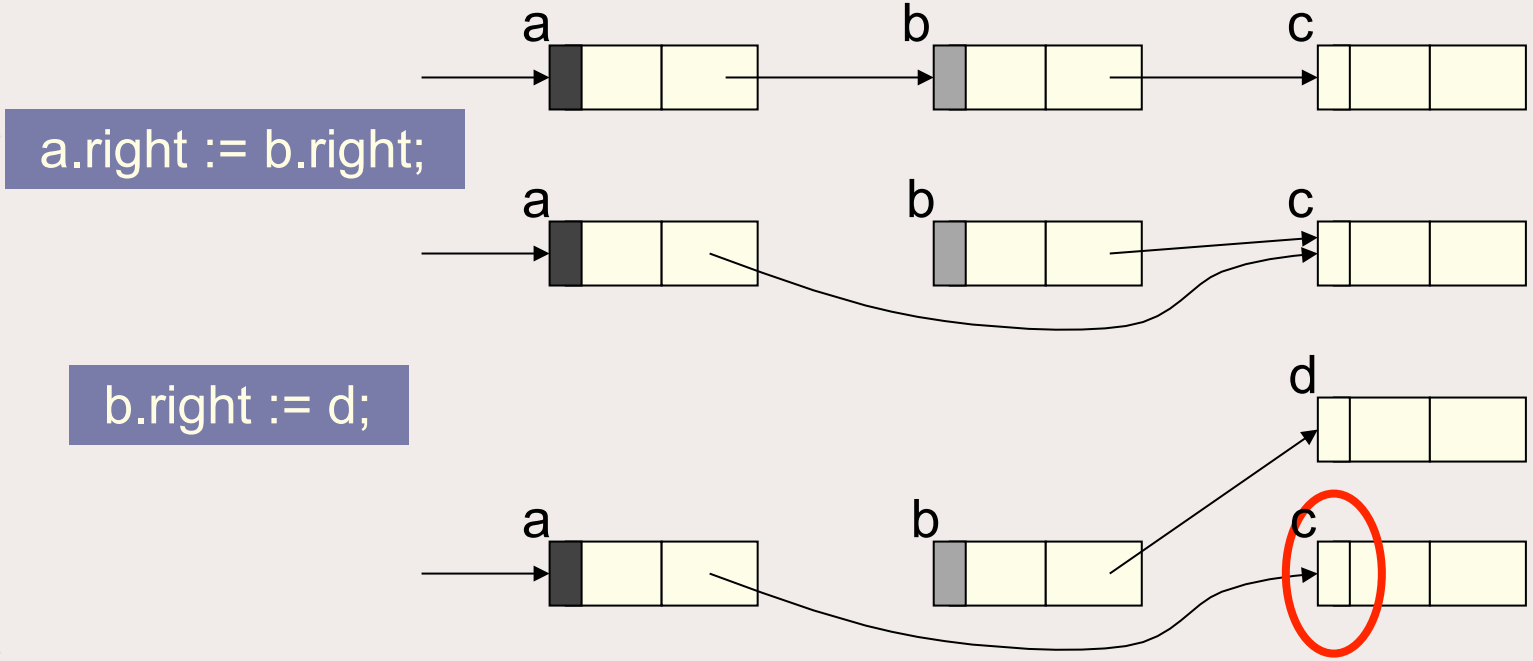
- one GC chunk (mark/sweep N objects) at a time



- 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

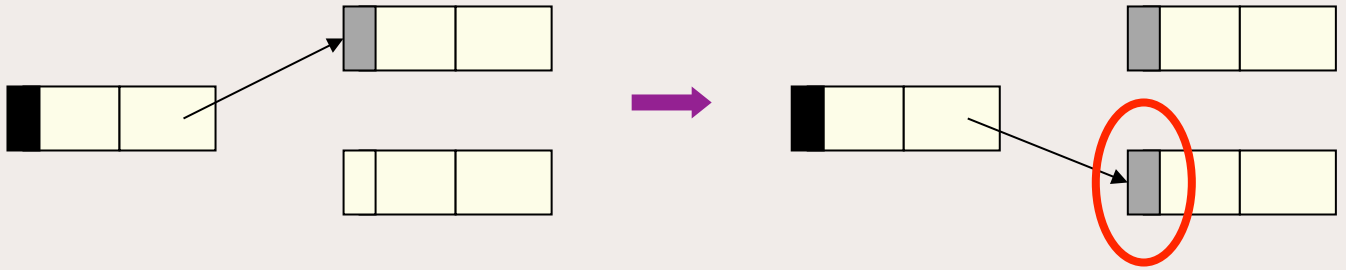


Solution 1 (Dijkstra)

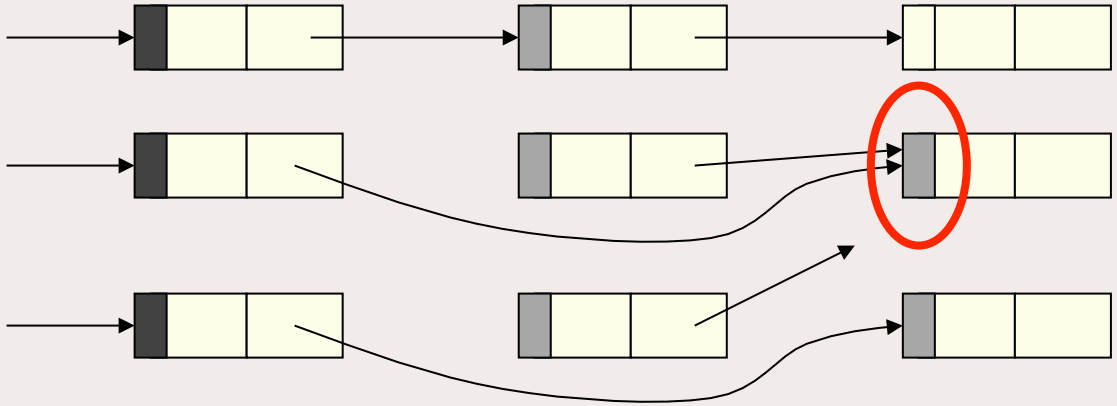
write barrier:

make gray the object newly pointed to,

when a pointer is replaced by another



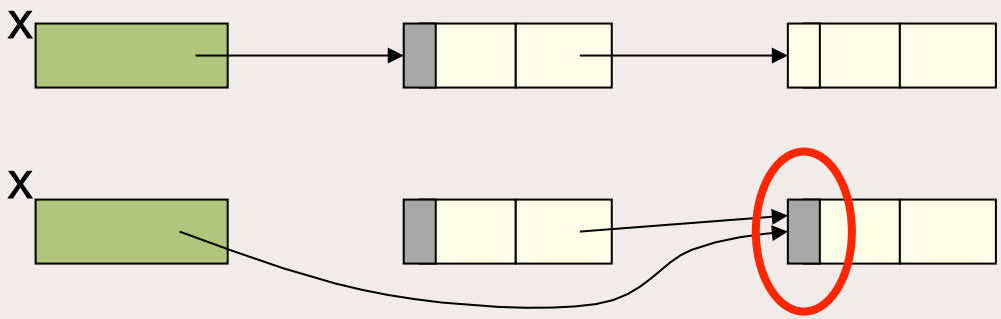
for the example:



Solution 1 (Dijkstra)

- write barrier necessary not only for objects, but also for roots (variables and registers)
- assignment to variables are more frequent than assignment to objects
- register loading is much more frequent
- inefficient for real applications
  - never used !

```
x := x->right;
```

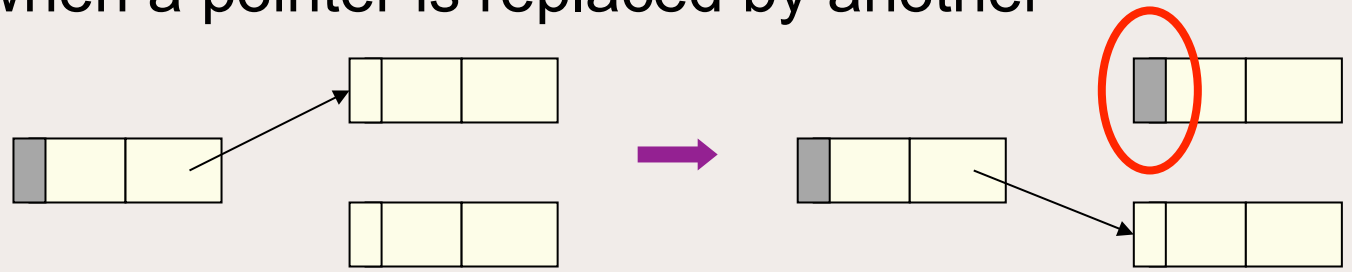


Solution 2 (Yuasa)

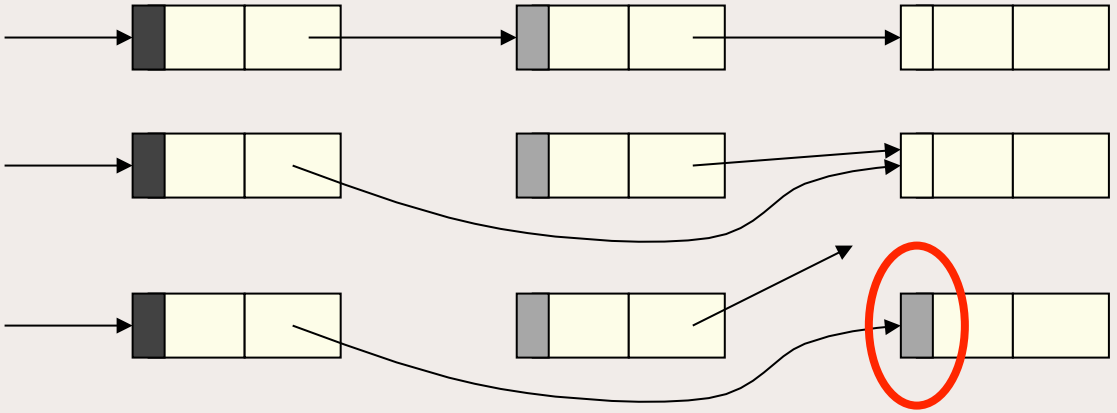
write barrier:

make gray the object previously pointed to,

when a pointer is replaced by another



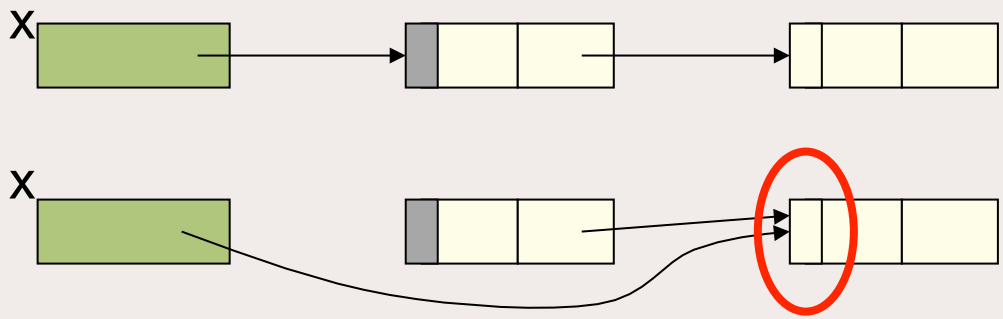
for the example:



Solution 2 (Yuasa)

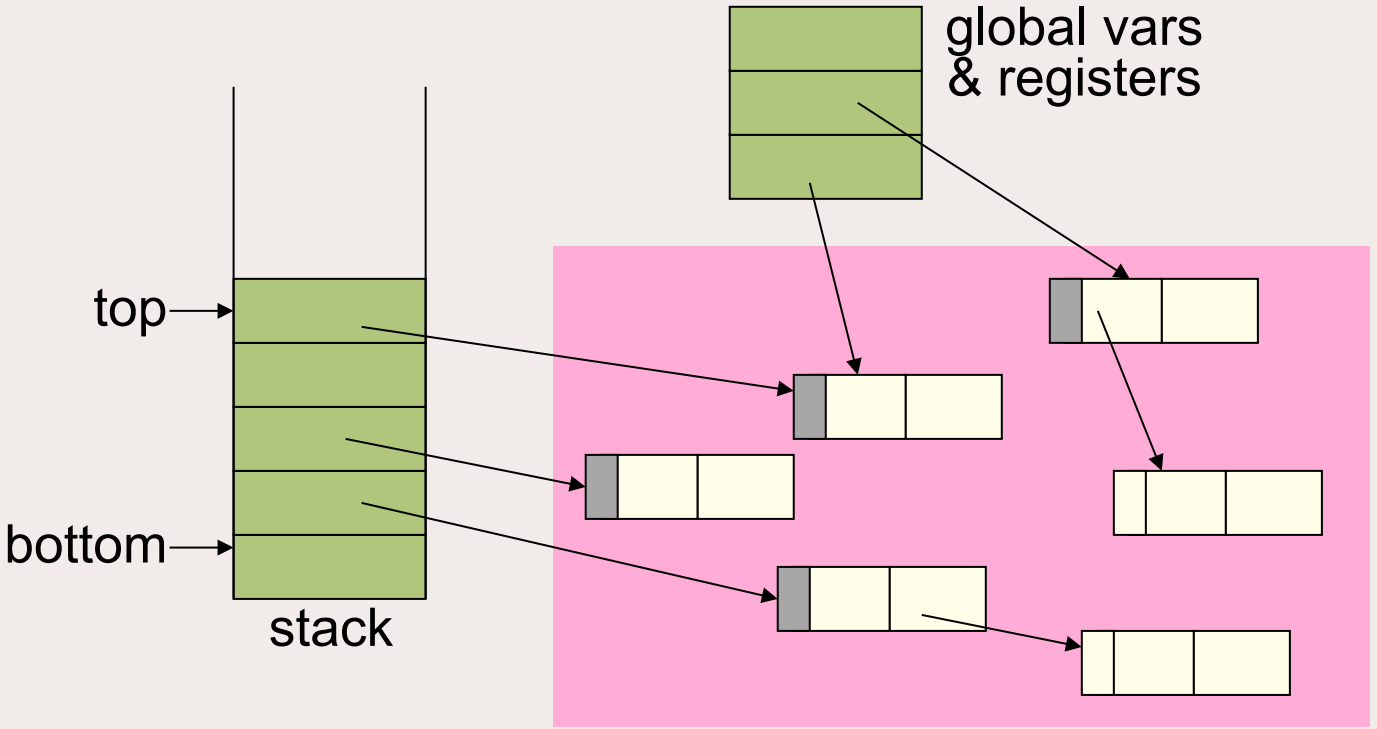
- all objects in use at the beginning of a GC
  are guaranteed to become black eventually
- at the beginning of a GC,
  make gray all objects directly pointed to from roots
- no write barrier necessary for roots
  – previous object eventually becomes black
- being in use in many systems

```
x := x->right;
```

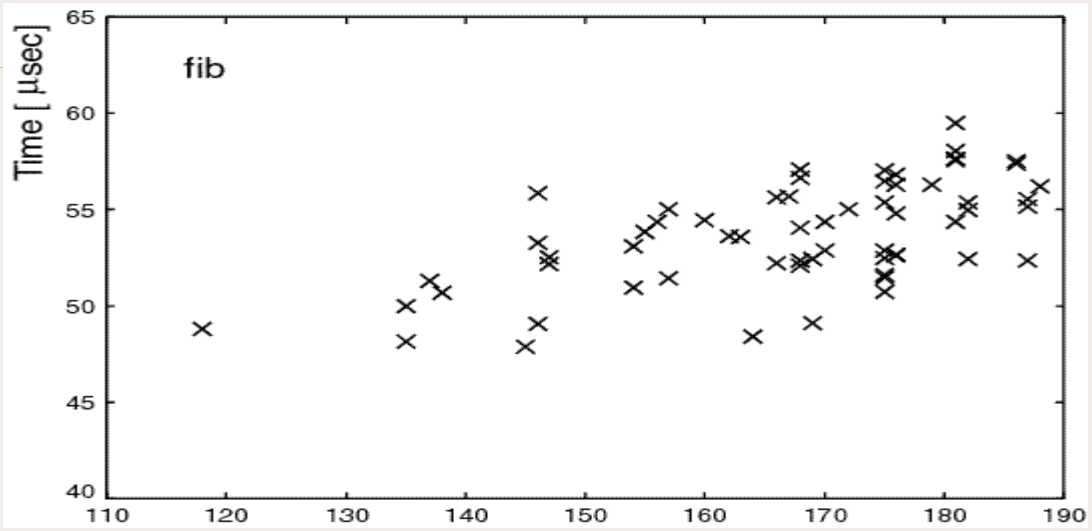


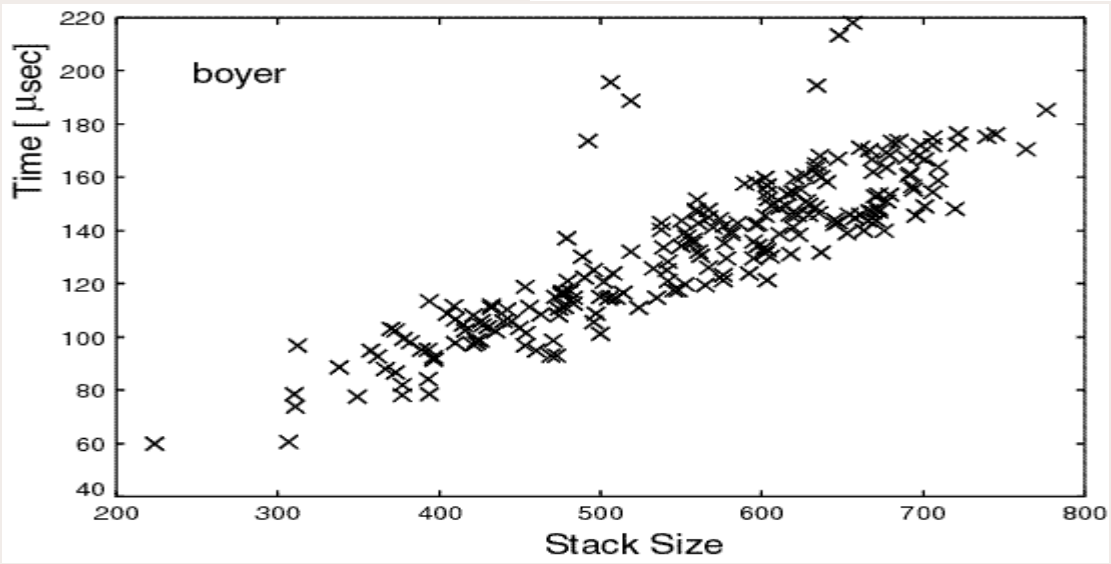
Root Scanning

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



Stack Size & Suspension Time





reduce
suspension time
＜ 100 µsec

Why 100μsec 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 100μs.

... though 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100μs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、100μsといった単位は必須なのですか？

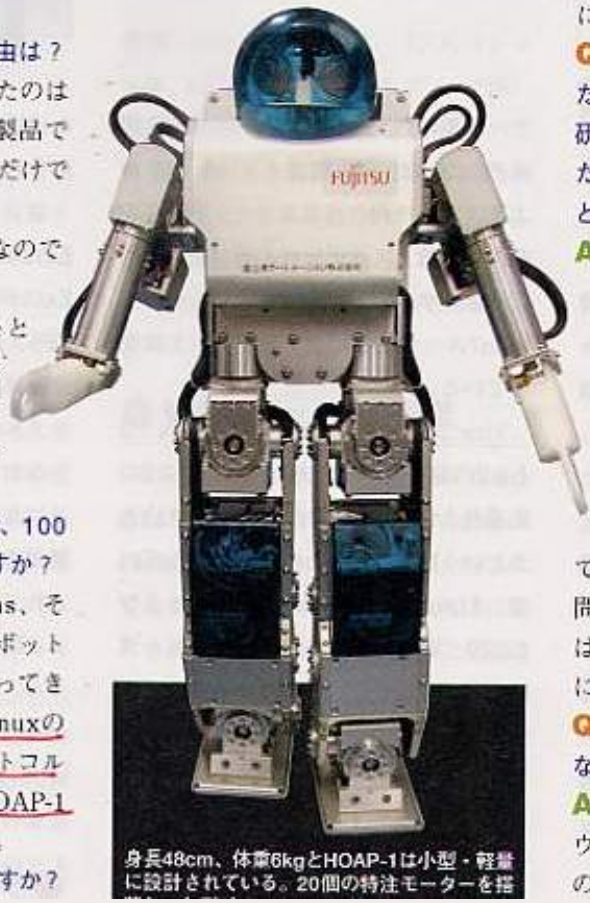
A 設定を10msから5ms、2ms、そして1msとしていくだけで、ロボットの動きはどんどんスムーズになってきます。ただ、1msを超えるとLinuxの限界ではなく、USBの通信プロトコルの制約を受けてしまうため、HOAP-1では1msで制御を行っています。

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。

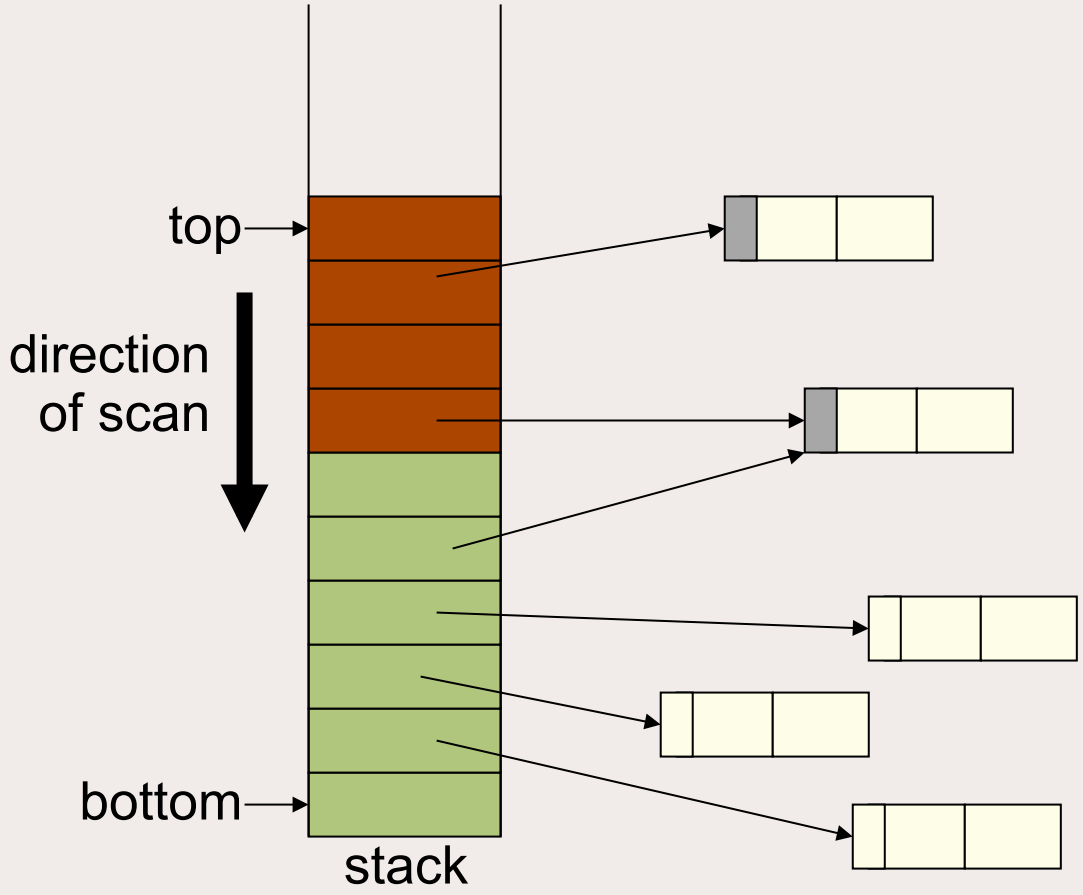
ロボットの場合、壊してしまうと修理に費用がかかるので、新しいアルゴリズムを試す場合、まずシミュレータ



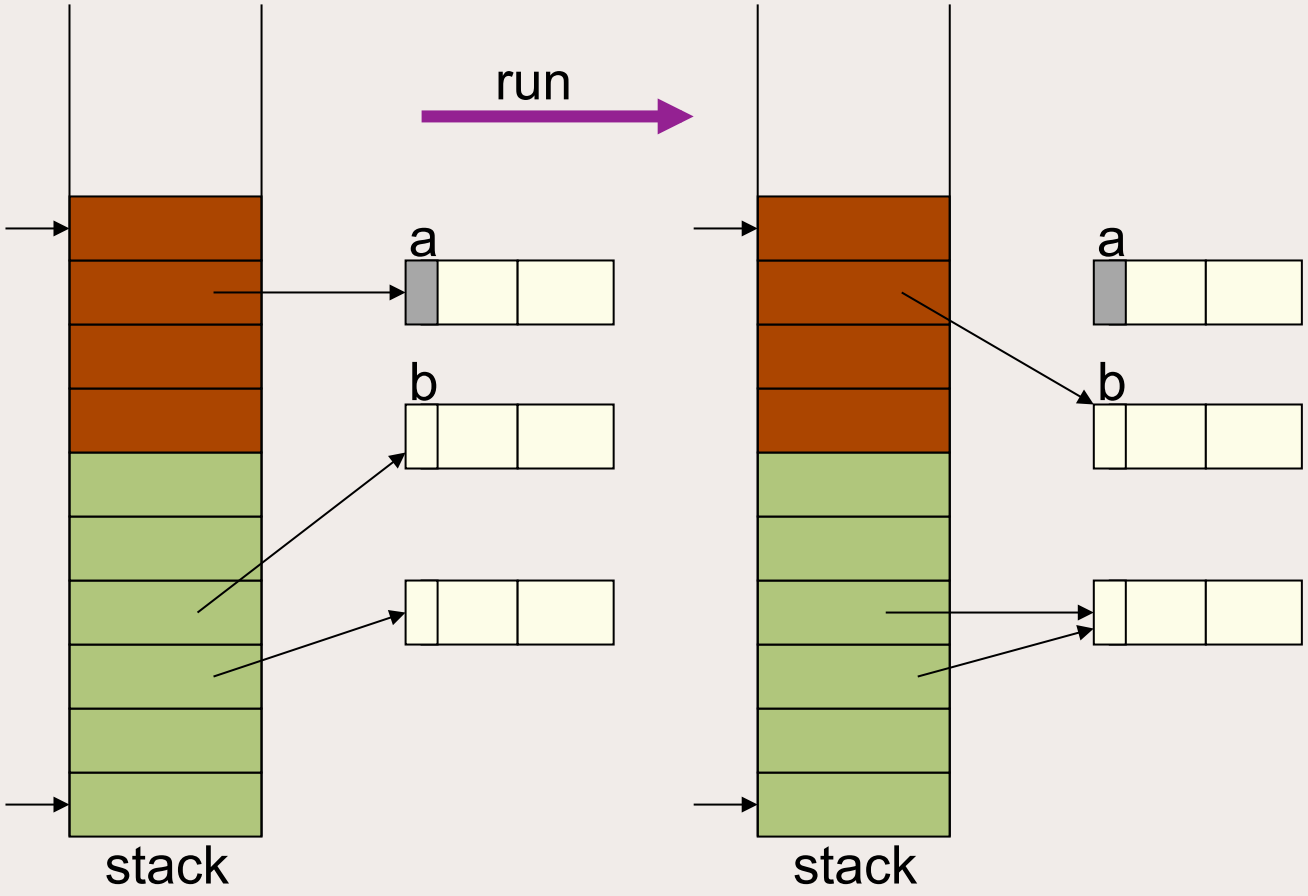
身長48cm、体重6kgとHOAP-1は小型・軽量に設計されている。20個の特注モーターを搭

Incremental Stack Scanning

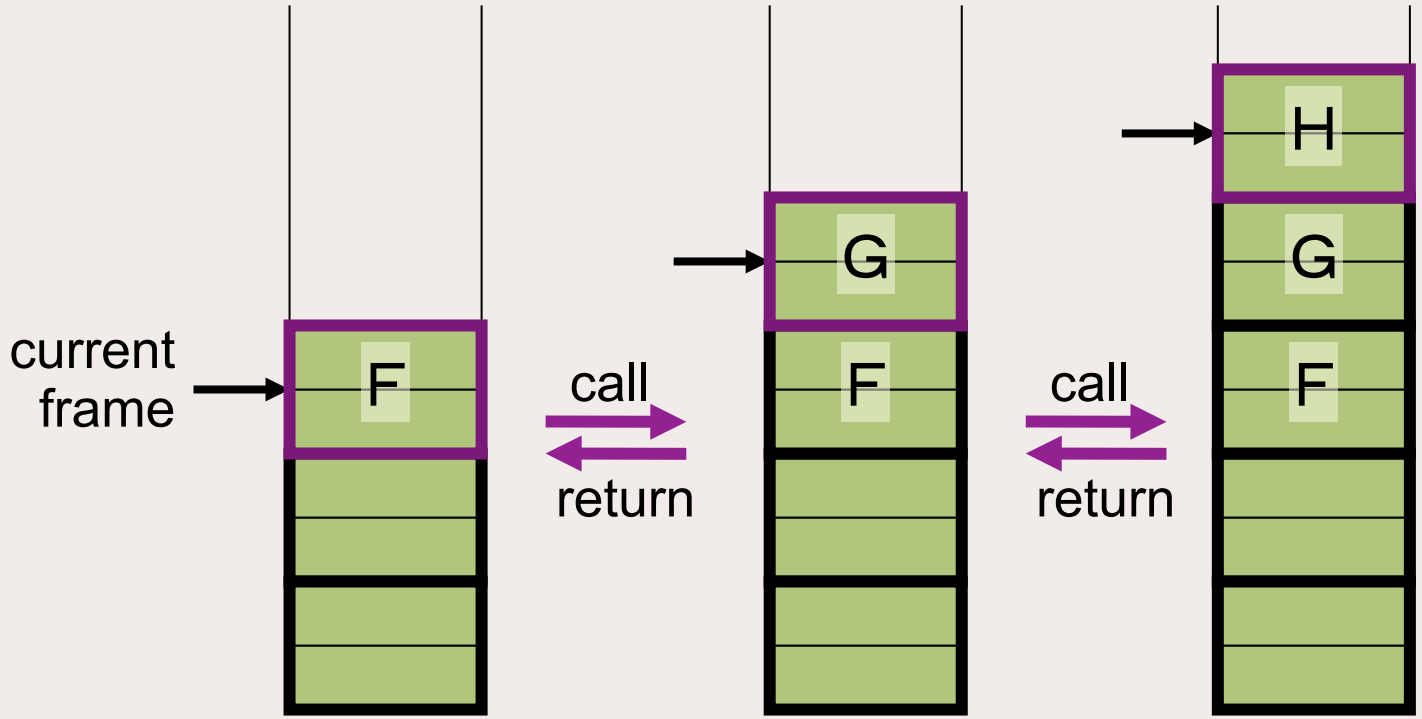
- scan the stack little by little



Problem

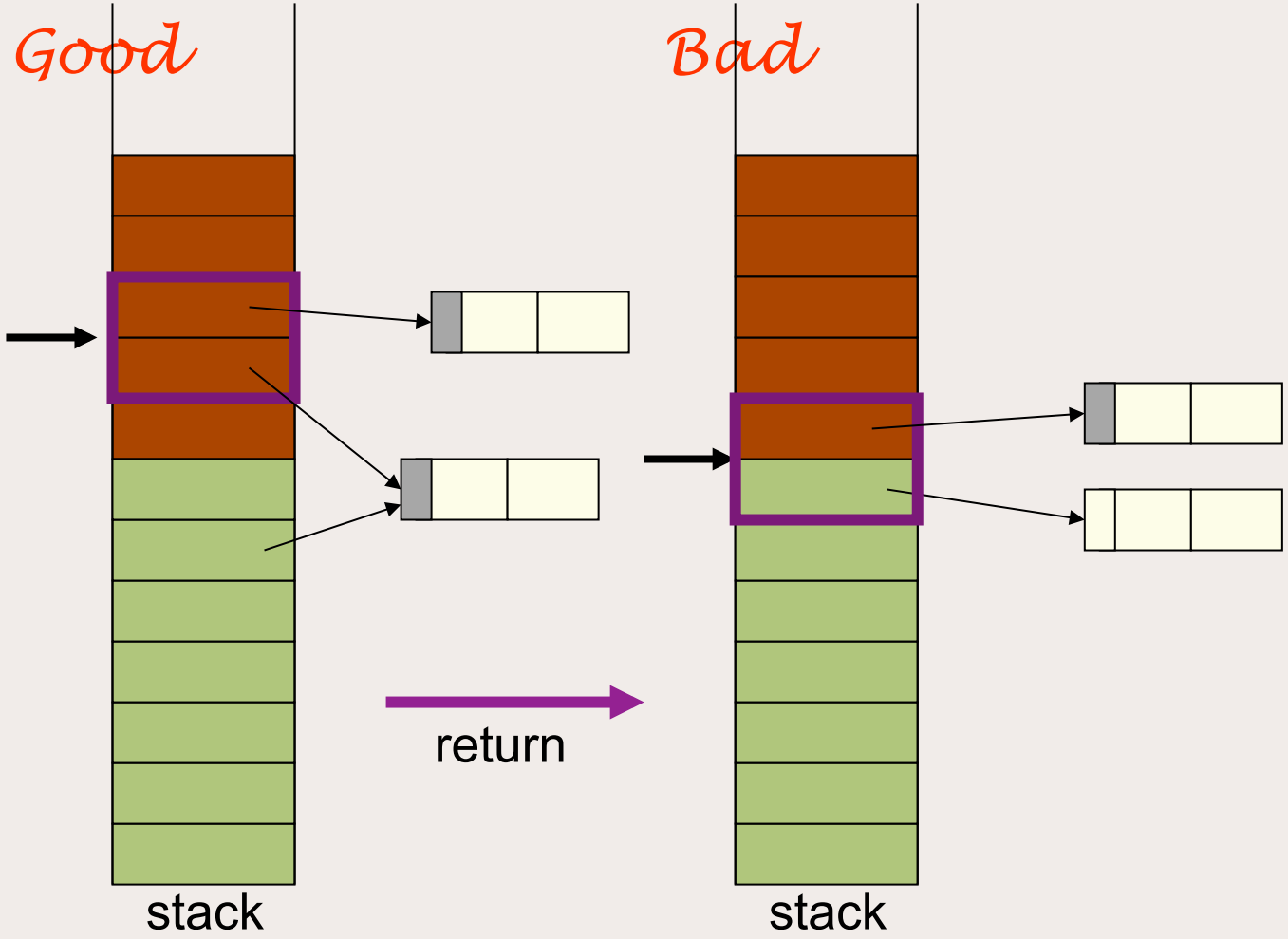


Function Frames

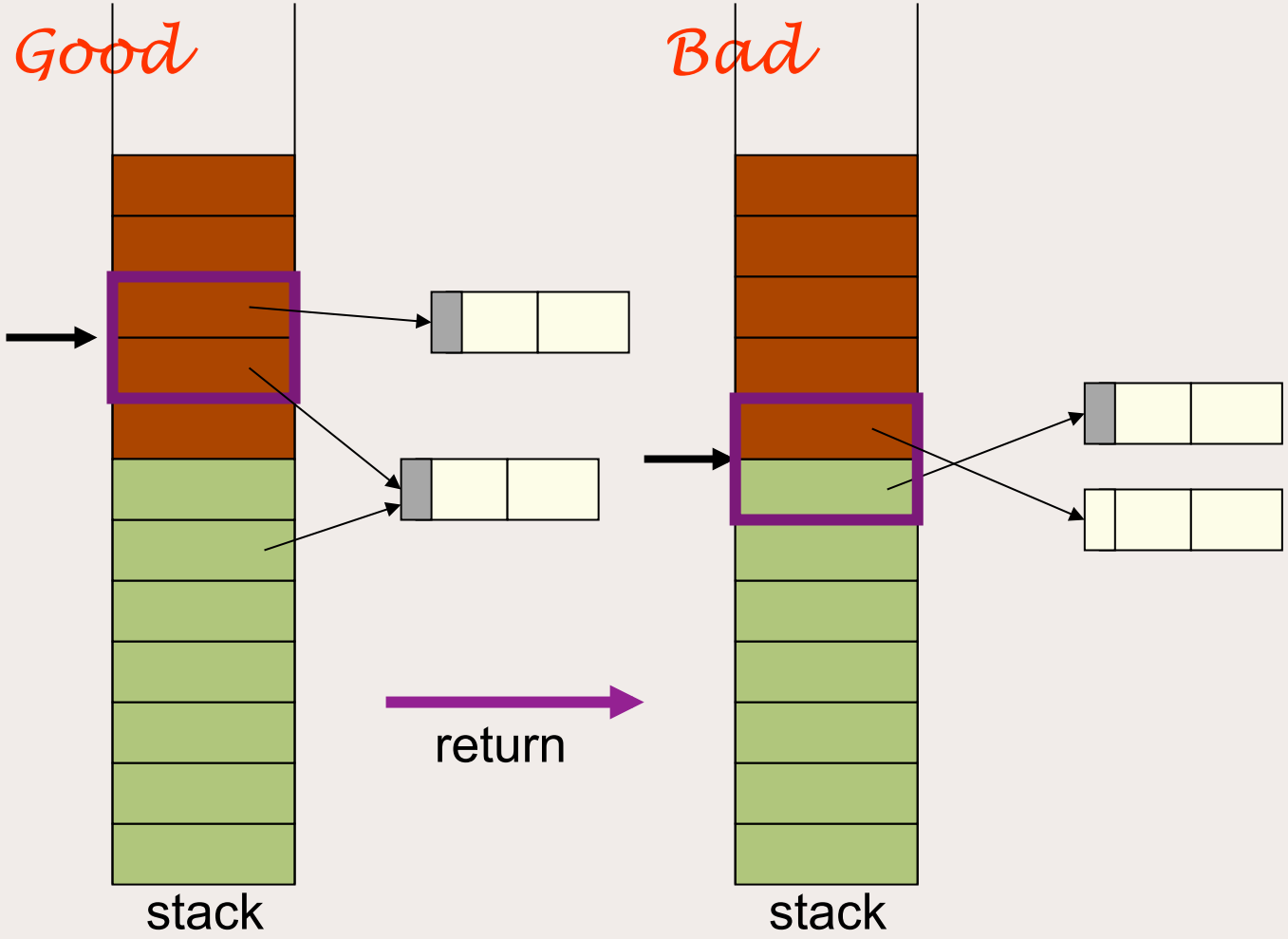


Only variables in the current frame can be accessed.

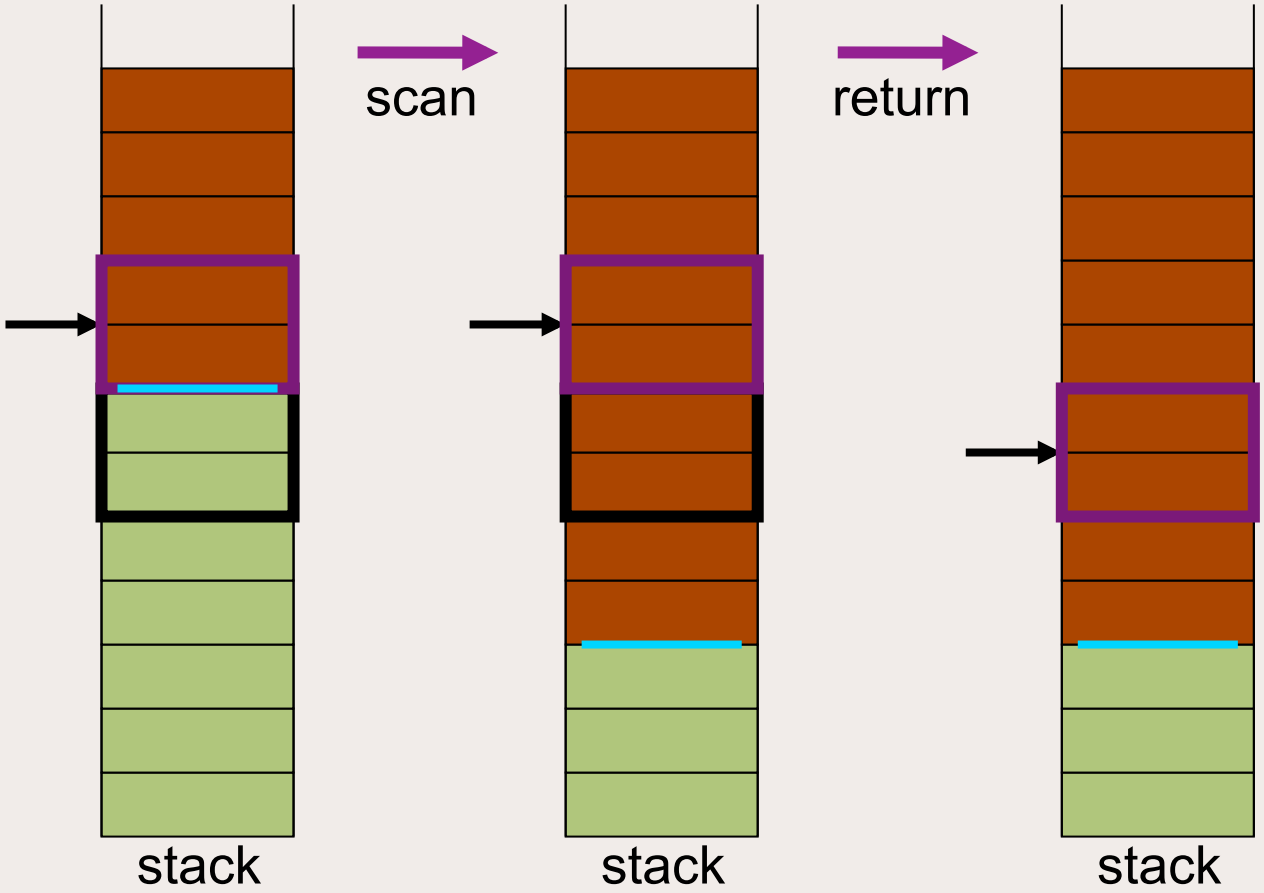
Scanning *vs* Return



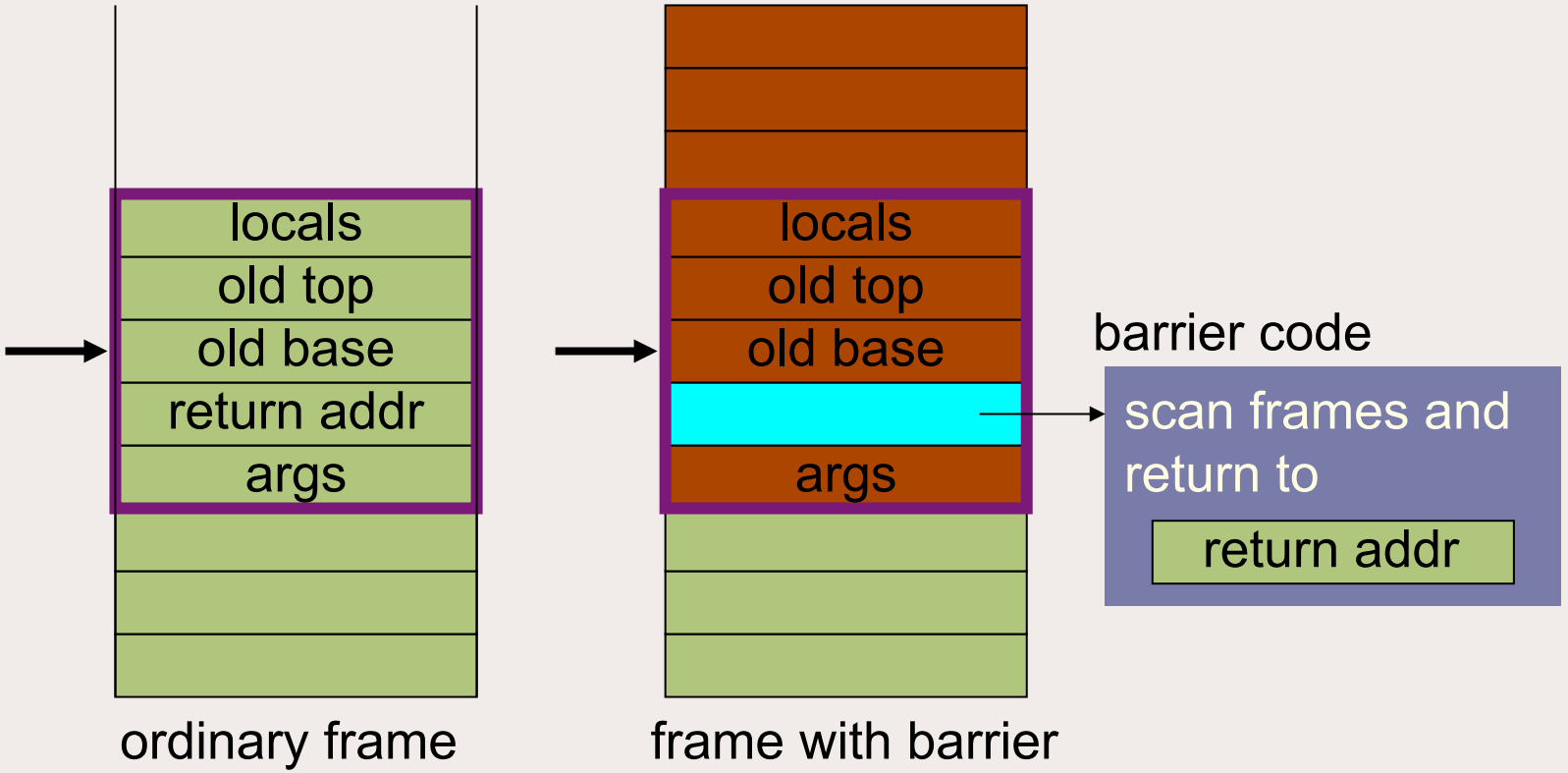
Scanning *vs* Return



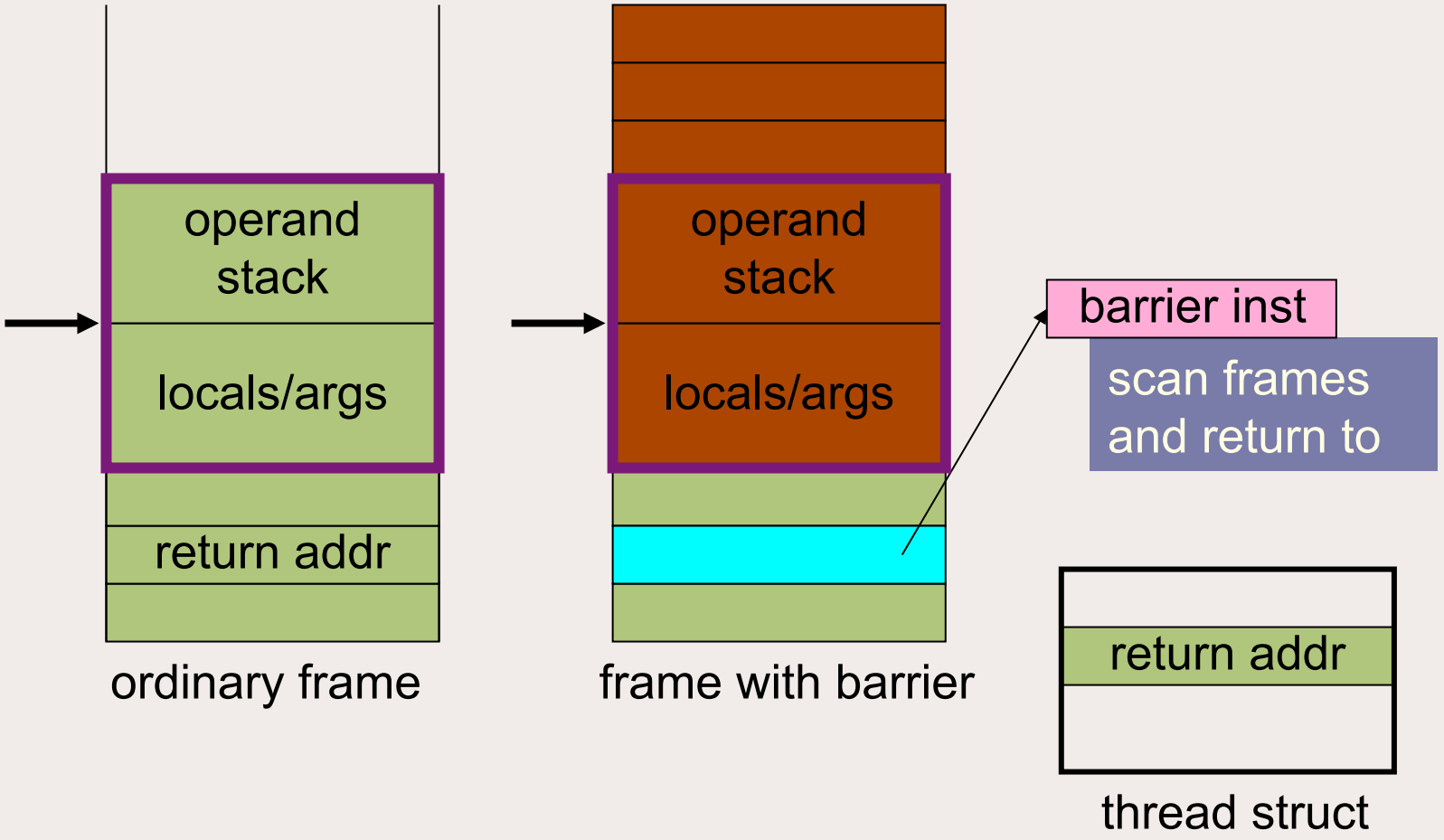
Return Barrier



It’s free (case C)

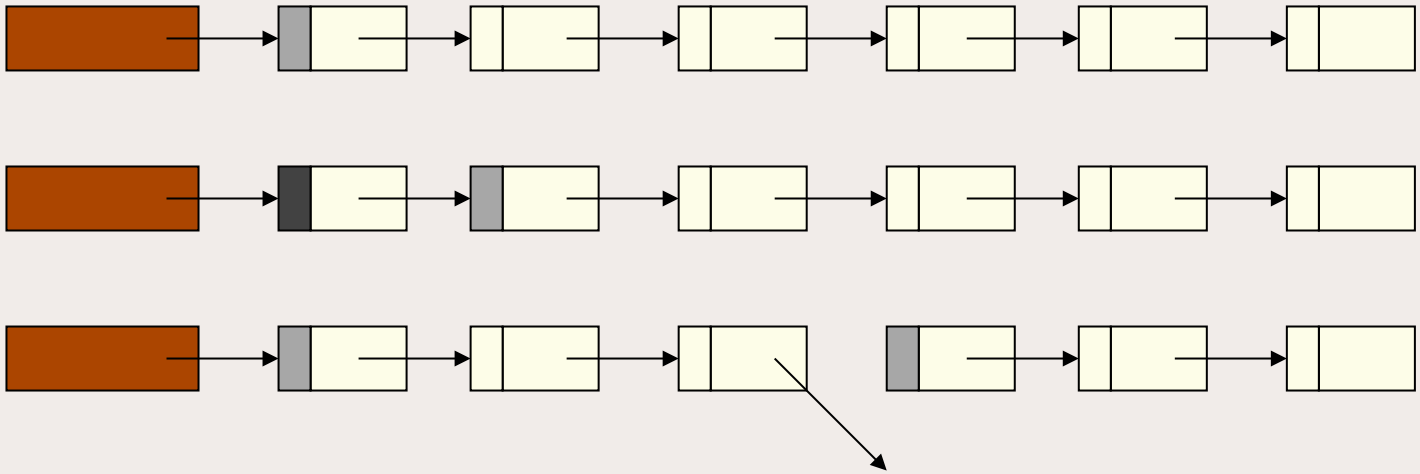


It’s free (case Java)



Correctness

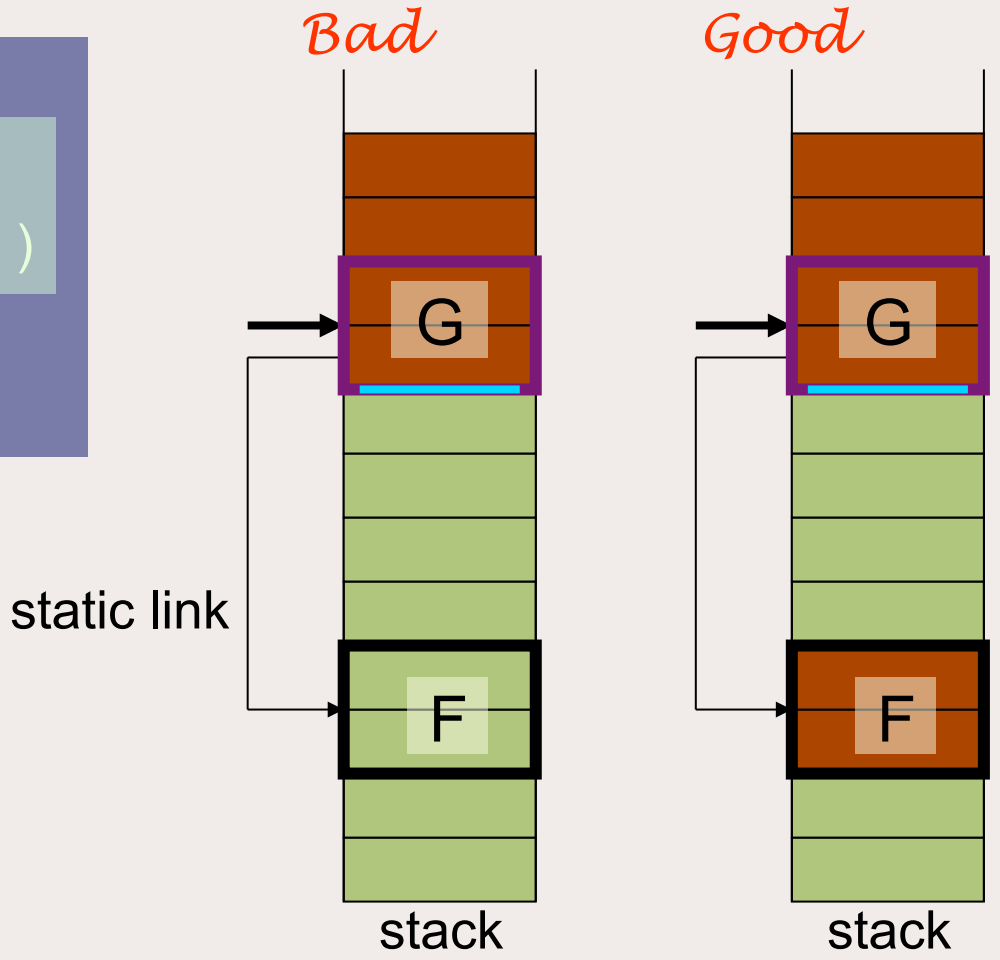
original snapshot:
Each cell reachable from a root at GC start is either marked (black) or markable.



snapshot with return barrier:
Each cell reachable from a root at GC start is either marked (black) or markable, if the root has been scanned.

Remark (local functions)

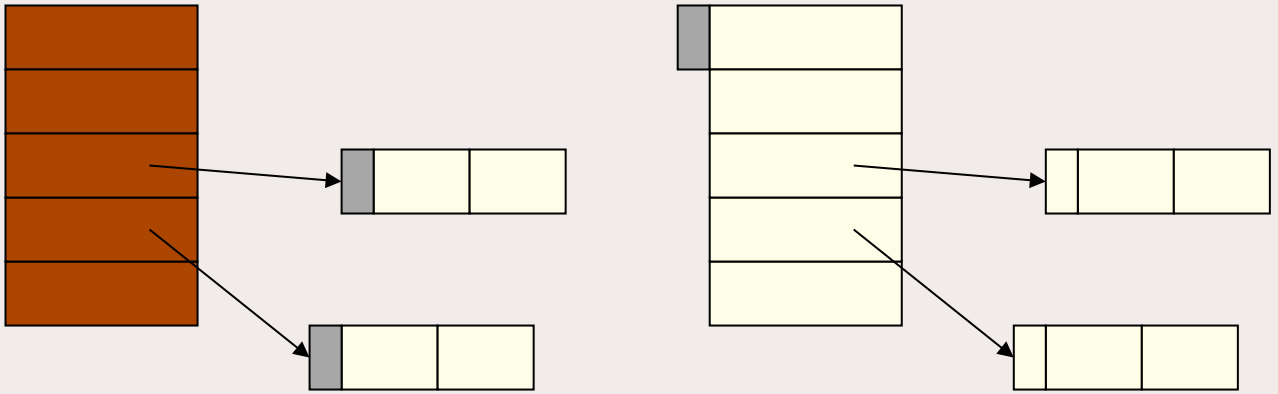
```
(define (F x)
  (define (G y z)
    … (G z x)… )
  (G x x)
)
```



Remark (constant roots)

regarded as a mark-reserved vector

⇒ need not scan at GC start



scan only variable roots at GC start.

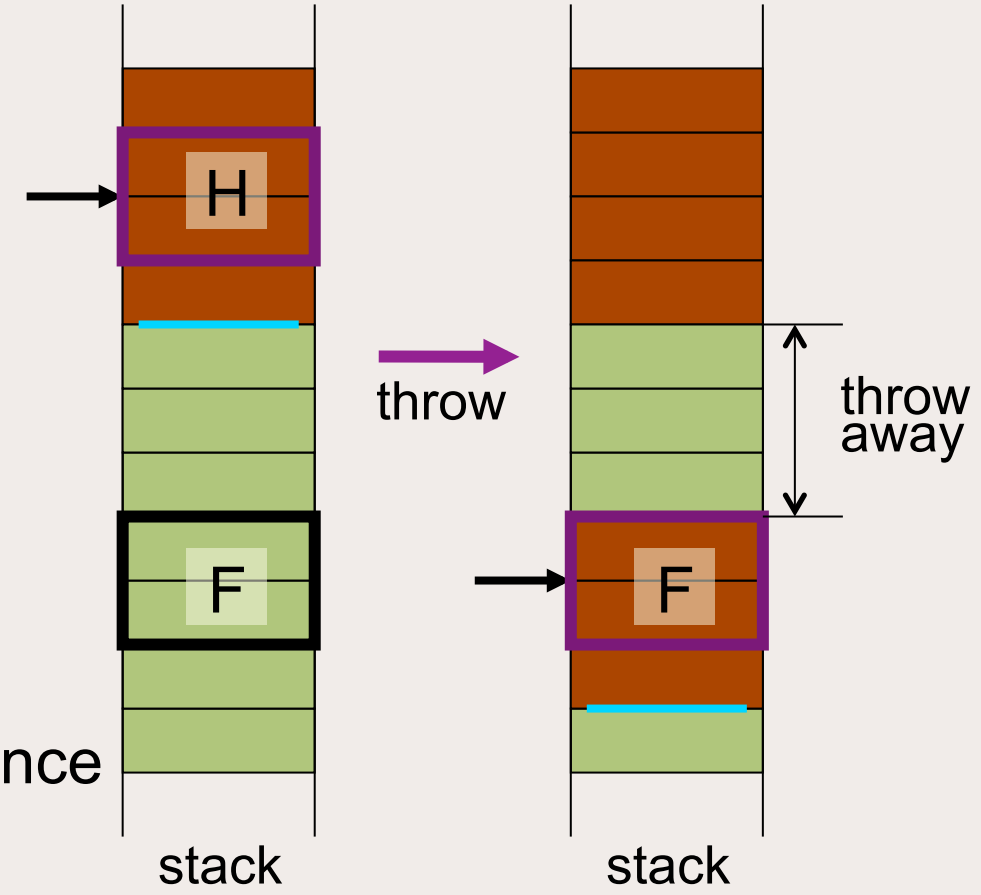
Remark (catch & throw)

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

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

not a true snapshot!

⇨ better performance



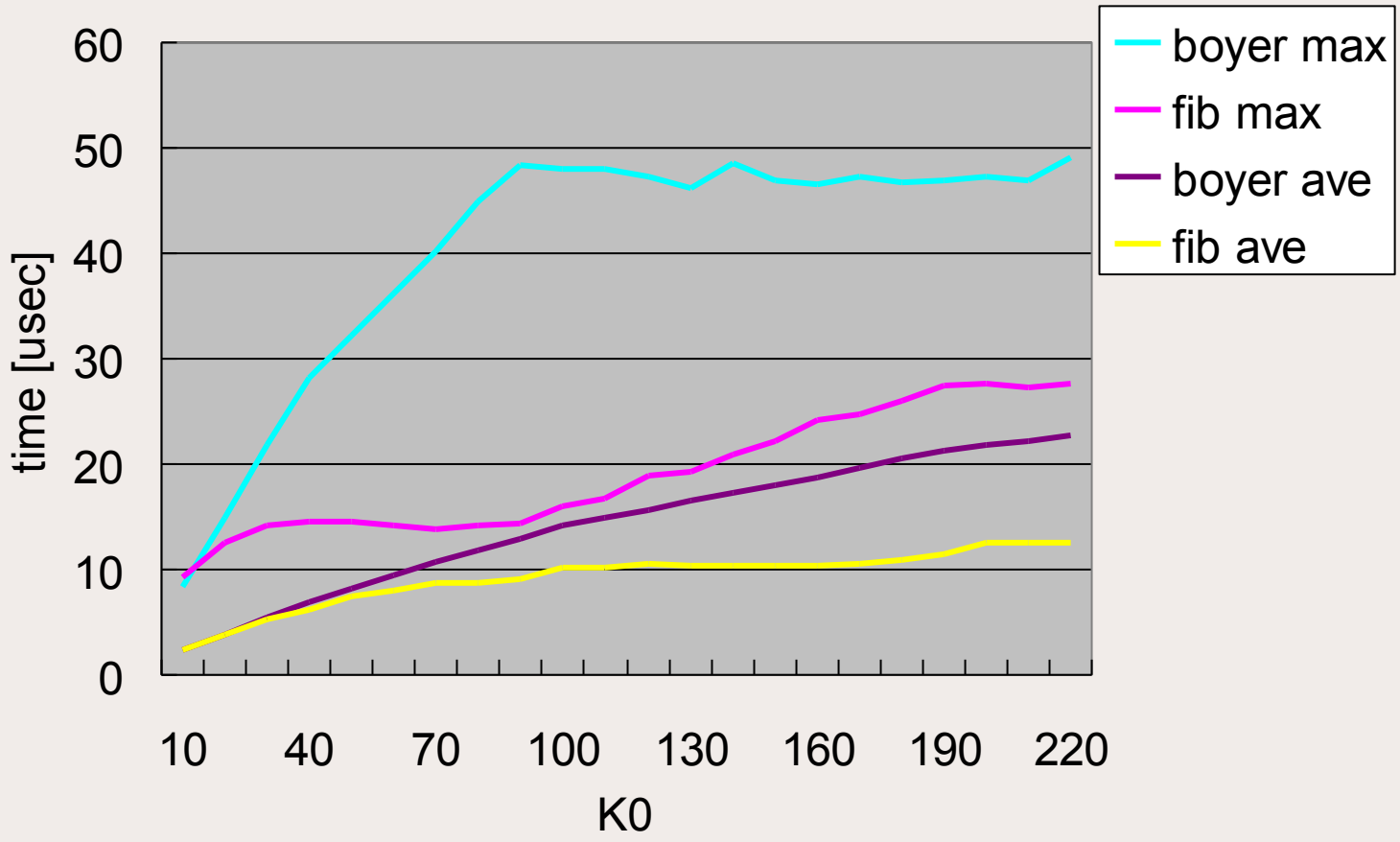
Implementation for KCL（Kyoto Common Lisp）

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

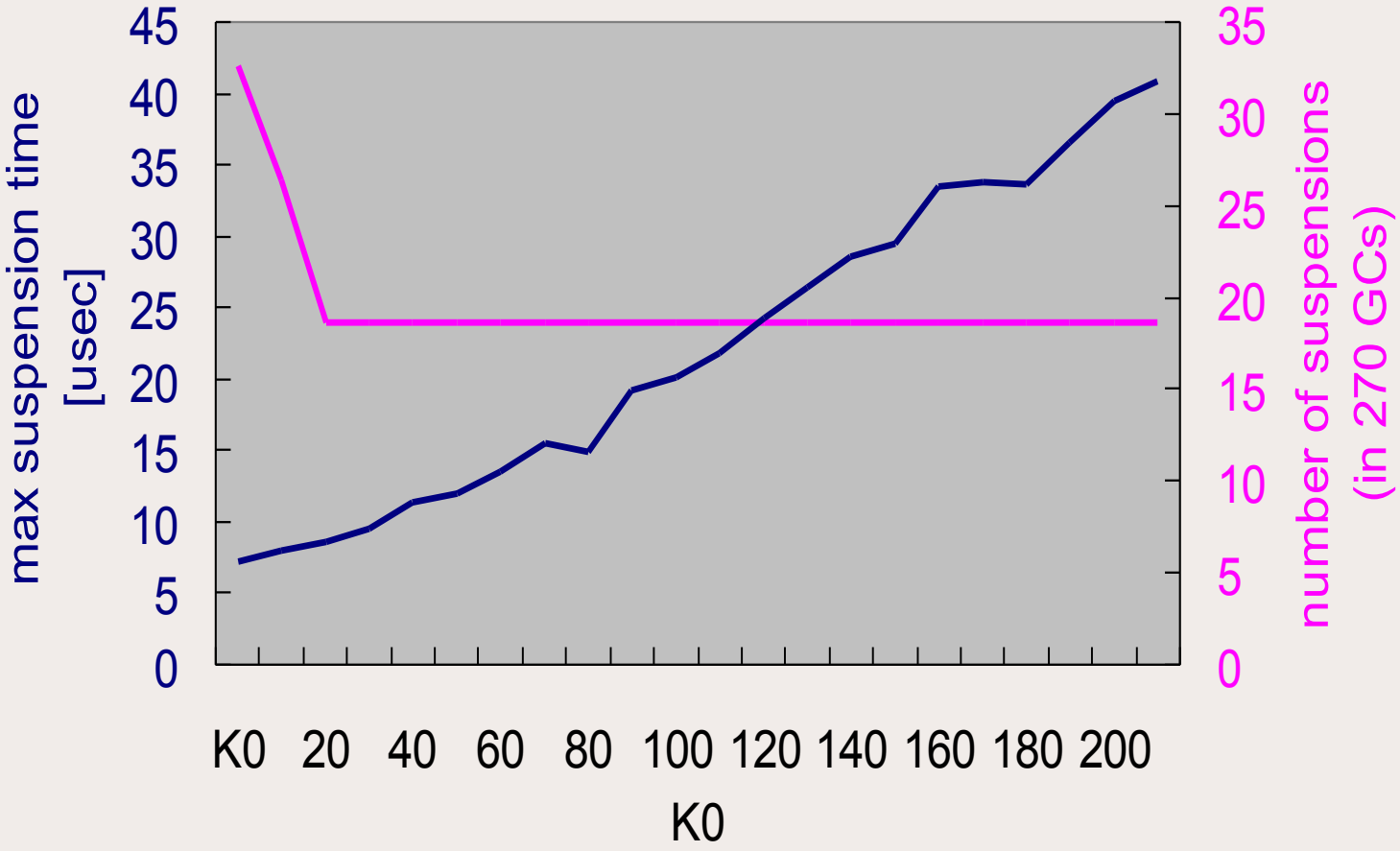
return addresses are pushed on this stack
⇒cannot handle return addresses
⇒needs explicit barrier checking on function returns

- system variables
  - only 18 variables

Suspension Times



Suspension Times by Return Barrier



Runtime Overhead

fib	return barrier			all at once	
	K0=10	K0=40	K0=160	w/check	conv.
runtime [sec]	12.27	12.03	11.98	11.92	10.83
# GC	65	65	65	65	65

boyer	return barrier			all at once	
	K0=10	K0=40	K0=160	w/check	conv.
runtime [sec]	38.38	28.71	27.38	26.68	26.11
# GC	304	226	212	208	208

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

Benchmarks aload, new&dup

aload: lots of aload

```
A a = new A();
for (int i = 0; i < N; i++)
    foo(a, a, a);
```

new&dup: not affected by the GC method

```
for (int i = 0; i < N; i++)
    // new & dup
    foo(new A(), new A(), new A());
```

Benchmark fib

aload
getfield
getstatic

```
public class I {
  Integer i;
  public I(int n) { i = new Integer(n); }
  public boolean lessthan(I n) {
    return this.i.intValue() < n.i.intValue();
  }
  public I plus(I n) {
    return new I(this.i.intValue() + n.i.intValue());
  }
}
```

```
static I one = new I(1);
static I two = new I(2);
static I fib(I n) {
    if (n.lessthan(two)) return n;
    else
      return fib(n.minus(one)).plus(fib(n.minus(two)));
}
```

Benchmark Heapsort

```
static Object[] vec = new Object[N + 1];
public static void shift_down(Object itm, int pos, int tail) {
    int posl = pos << 1; int posr = posl + 1;
    if (posr <= tail) {
      Object l = vec[posl]; Object r = vec[posr];
      if (ncmp(itm, l)) {
                if (ncmp(itm, r)) vec[pos] = itm;
                else { vec[pos] = r; shift_down(itm, posr, tail); }
      } else {
                if (ncmp(l, r)) {vec[pos] = l; shift_down(itm, posl, tail); }
                else { vec[pos] = r; shift_down(itm, posr, tail); }
      }
    } else if (posl <= tail) {
      Object l = vec[posl];
      if (ncmp(itm, l)) vec[pos] = itm;
      else { vec[pos] = l; shift_down(itm, posl, tail); }
    } else vec[pos] = itm;
  }
  public static boolean ncmp(Object a, Object b) {
    return ((Integer) a).intValue() > ((Integer) b).intValue();
  }
```

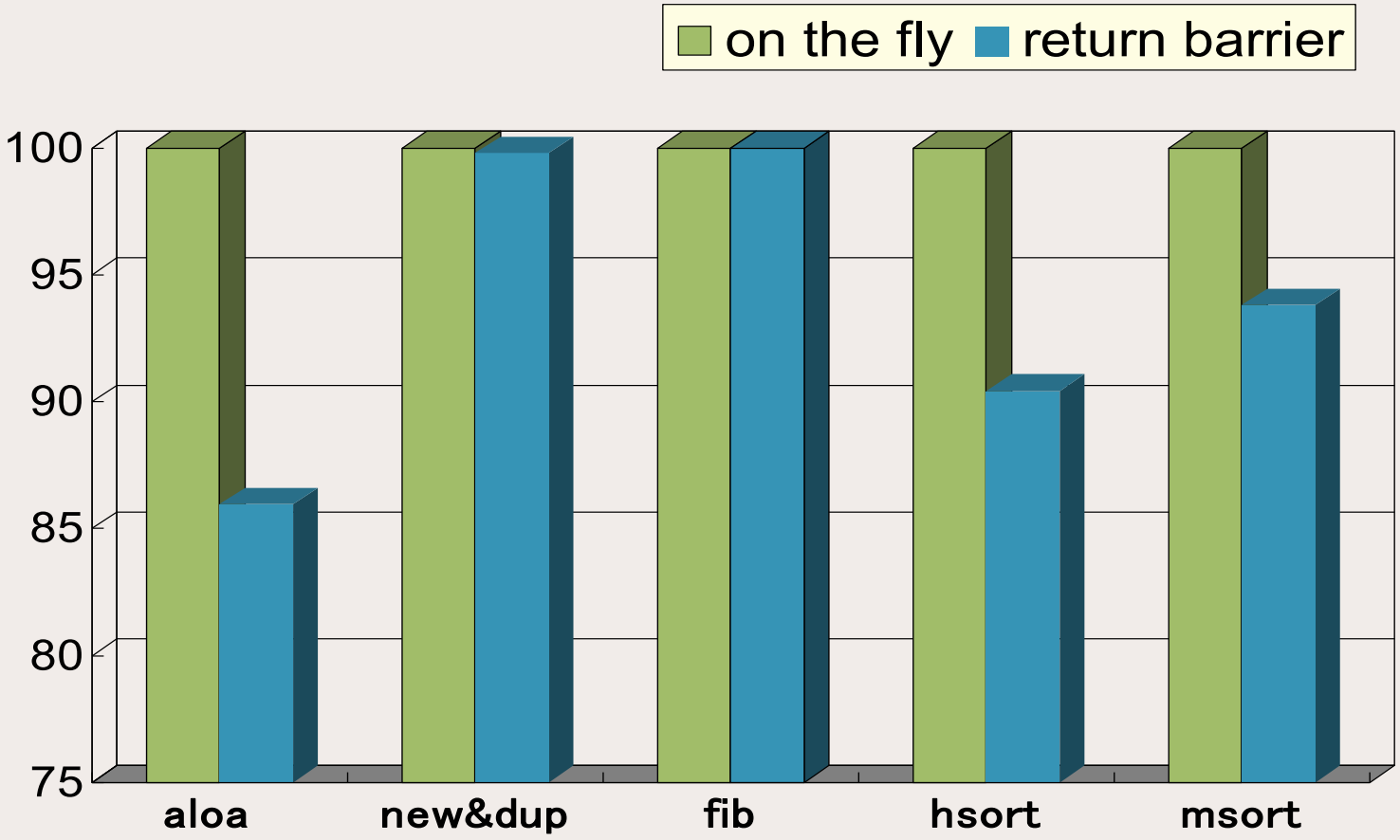
aload
aaload
getstatic

Benchmark Mergesort

```
static Pair merge (Pair lst) {
    if (lst == null || lst.cdr == null) return lst;
    else return Pair.cons(merge1((Pair) lst.car,
                                 (Pair) ((Pair) lst.cdr).car),
                          merge((Pair) ((Pair) lst.cdr).cdr));
}
static Pair merge1 (Pair la, Pair lb) {
    if (la == null) return lb;
    if (lb == null) return la;
    if (cmp(la.car, lb.car))
        return Pair.cons(la.car, merge1((Pair) la.cdr, lb));
    return Pair.cons(lb.car, merge1(la, (Pair) lb.cdr));
}
static boolean cmp(Object a, Object b) {
    return ((Integer) a).intValue() < ((Integer) b).intValue();
}
public Pair (Object ca, Object cd) {
    this.car = ca; this.cdr = cd;
}
public static Pair cons (Object ca, Object cd) {
    return new Pair(ca, cd);
}
```

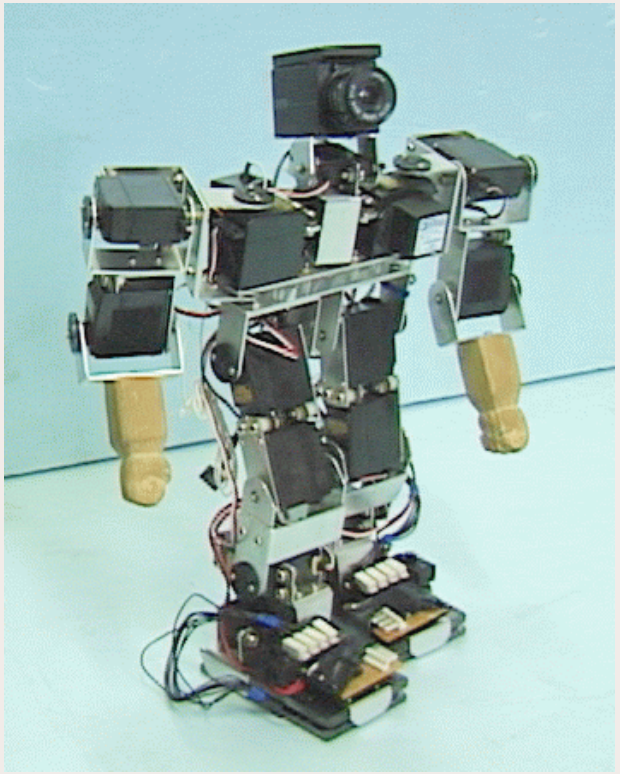
```
aload
getfield
```

Benchmark Results



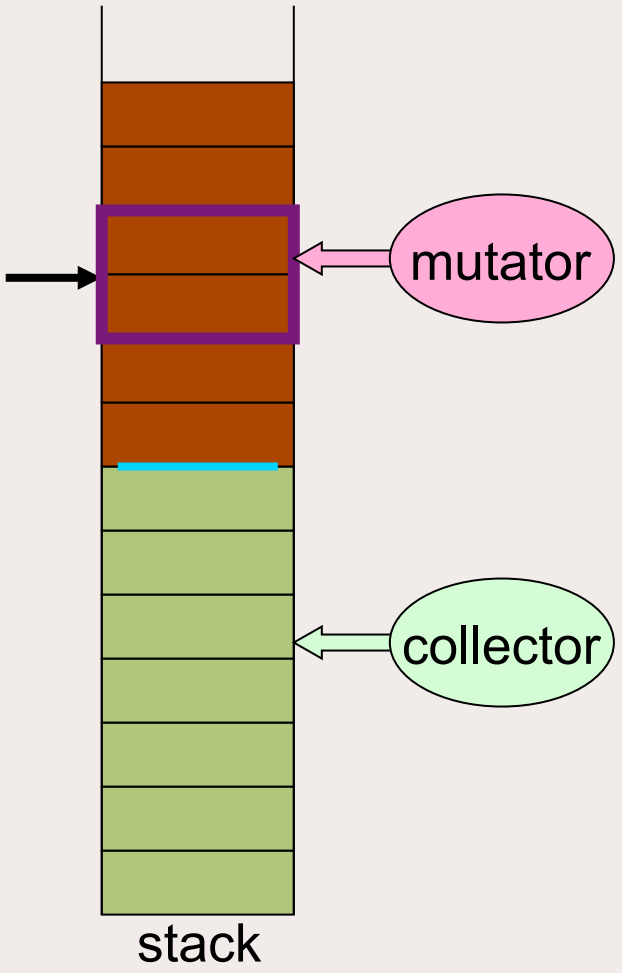
Implementation for EusLisp

- 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 one
- we are implementing return barrier for the system
- prototype implementation is running



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.



Problem