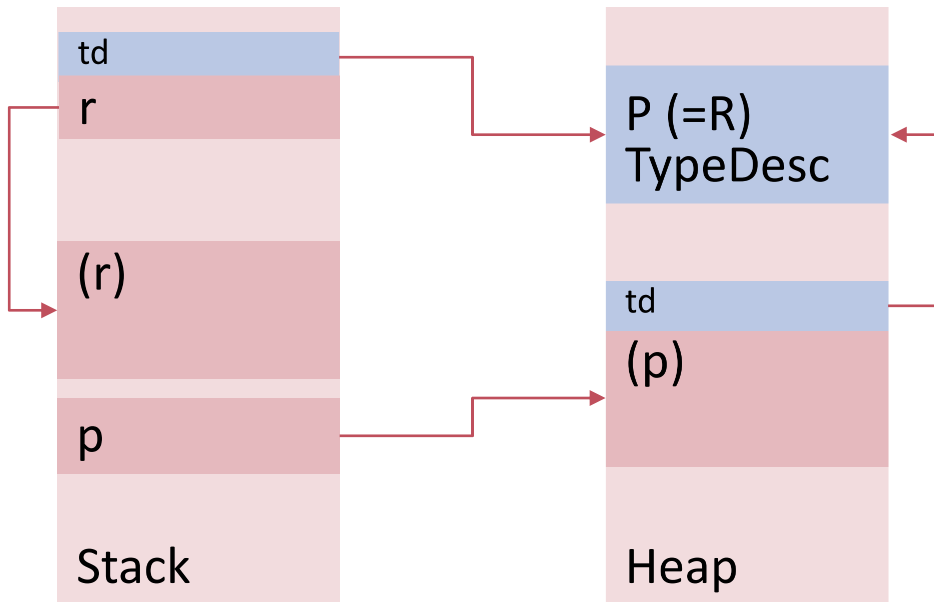
2.4. HEAP MANAGEMENT

252

Heap Blocks

Prerequisites

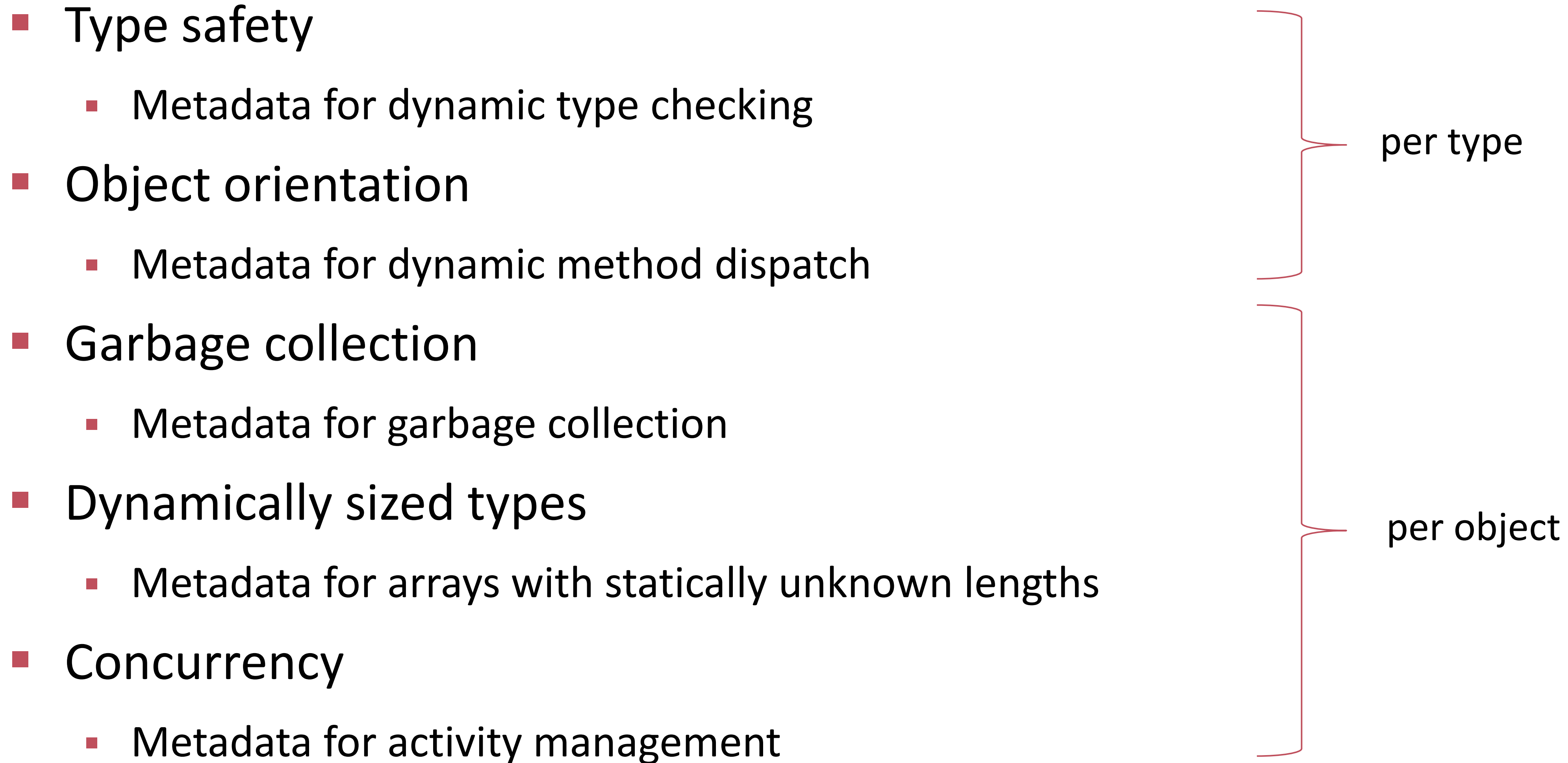
- Type safety
  - Metadata for dynamic type checking or dynamic method dispatch
  - Where?




```
TYPE R = RECORD END;
TYPE RE = RECORD(R) e: LONGINT END;
TYPE P = POINTER TO R;
TYPE PE = POINTER TO RE;

PROCEDURE T1(p: P);
BEGIN
   WITH p: PE DO
      p.e := 10
   END;
END T1;

PROCEDURE T2(VAR r: R);
BEGIN
   WITH r: RE DO
      r.e := 10
   END;
END T1;
```

Heap Blocks

Prerequisites



Heap Blocks

Further Requirements

- *Efficient* Access to type tags, method table, array bounds / index lengths, lock data structres
  - immediate access to type tags for each block
  - constant lookup time in type tag table
  - constant lookup time in method table
  - immediate access to array geometry descriptor
  - immediate access to lock- / await queues
- *Maintainability* of the Garbage Collector and Heap Allocation
  - good documentation and/or readable code
  - e.g.: data structures in high level language

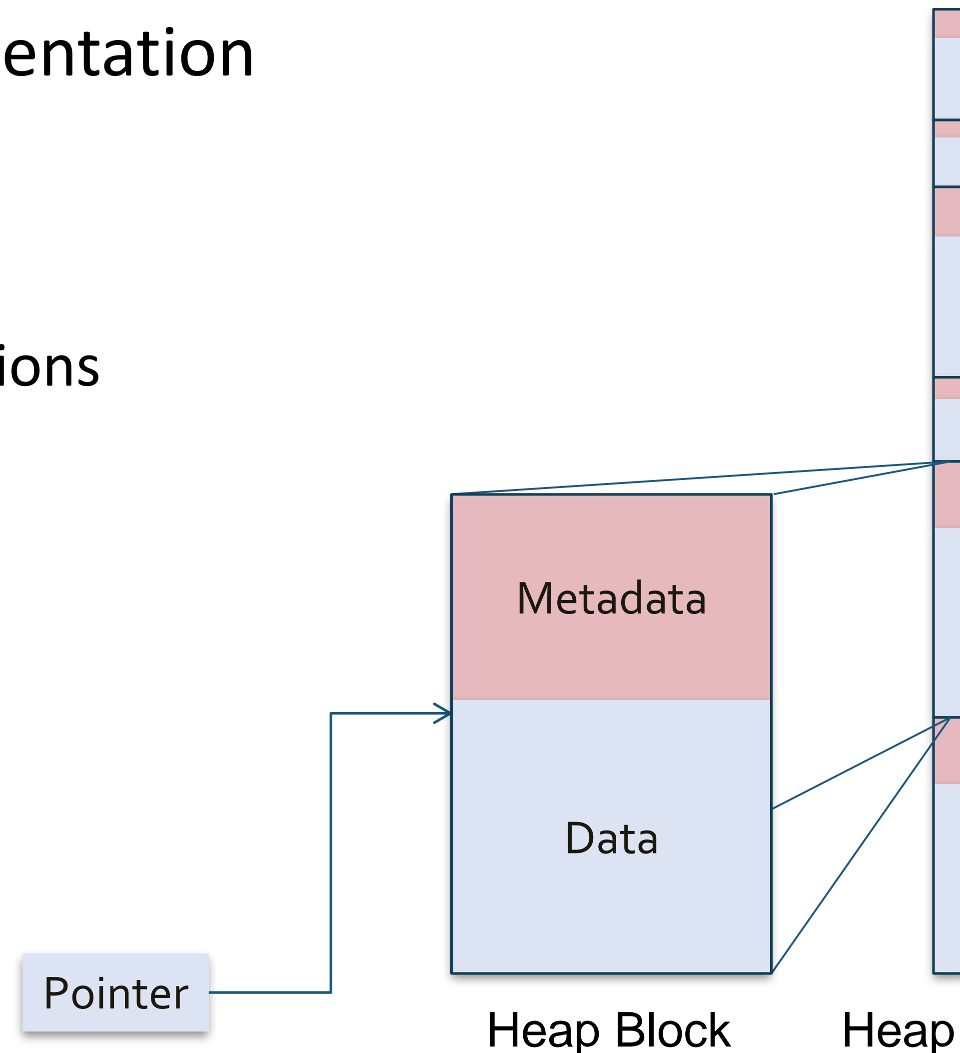
Heap Block Types

- Free Blocks
- Record
  - without locks / active body
- Protected Object
  - record with locks / active body
- Array
  - static array / dynamic array
  - array with / without pointers
- (Math Arrays: separate array descriptors with increments)

```
A= RECORD … END;
P= POINTER TO A;
O= OBJECT … END O;
A= ARRAY c OF T;
PA = POINTER TO ARRAY OF T;
MA = ARRAY [*,*] OF T;
MT = ARRAY [?] OF T;
```

Further Prerequisite: Consistency

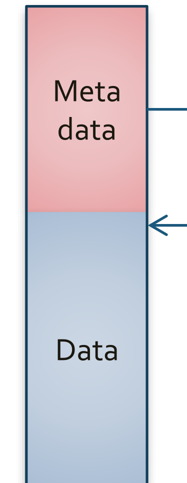
- Compiler must not have to discriminate between data on stack/ heap: consistency = uniformity of representation
- Possible solutions
  - metadata everywhere (stack / heap), possibly with indirections
  - metadata at negative offsets
    - adopted in A2



Heap Block Structure

- Idea: use *type system* to discriminate heap blocks
  - (Fully) expressed in high-level language
  - instead of introducing cumbersome additional tag fields / flags or even rely on heap block alignment
- Heap blocks are typed
  - Basic heap block:

```
HeapBlock* = POINTER TO HeapBlockDesc;
HeapBlockDesc* = RECORD
   mark: LONGINT;
   dataAdr-: SYSTEM.ADDRESS;    } meta data
   size-: SYSTEM.SIZE
END;
```



Type tests possible:
VAR block: HeapBlock;
...
IF block IS ProtRecBlock THEN ... END

In principle OO approach also viable

- Extended heap blocks:
  - FreeBlock, SystemBlock, RecordBlock, ProtRecBlock, ArrayBlock and StaticTypeBlock

Runtime Data Structure

- Unused blocks: *FreeBlock*

```
FreeBlock* = POINTER TO FreeBlockDesc;
FreeBlockDesc* = RECORD (HeapBlockDesc)
END;
```

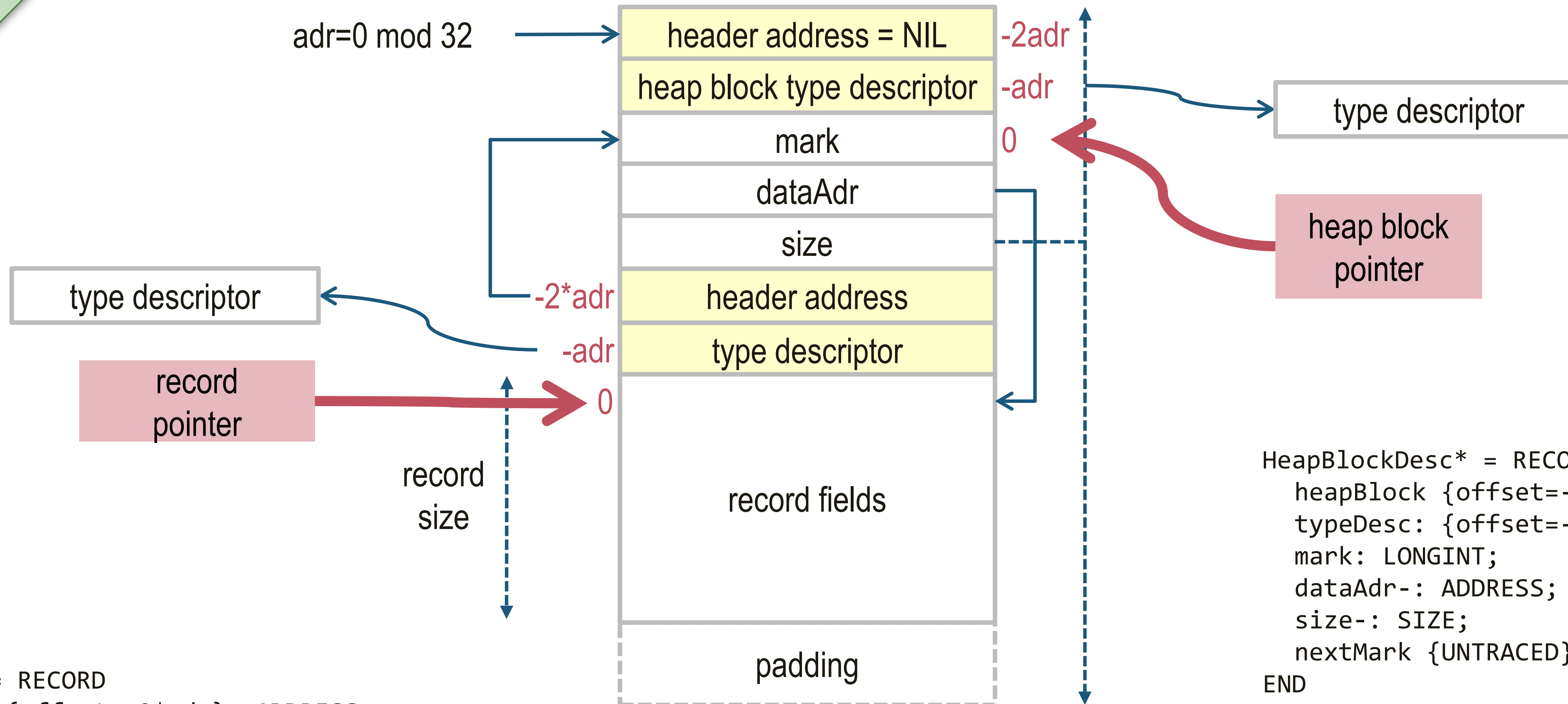
- Used blocks: Object graph consisting of
  - Objects and Arrays
  - Abstract root object (base type)
    - Method *FindRoots*
  - Concrete root objects (extended types)
    - Module descriptor
      (roots = global pointers)
    - Process descriptor
      (roots = pointers on stack)

```
RootObject = OBJECT (HeapBlock)
     PROCEDURE FindRoots()
     …
Module  = OBJECT(RootObject)…
Process  = OBJECT(RootObject)…
```

(Non-Shared) Object Blocks

compiler and code perspective

runtime perspective



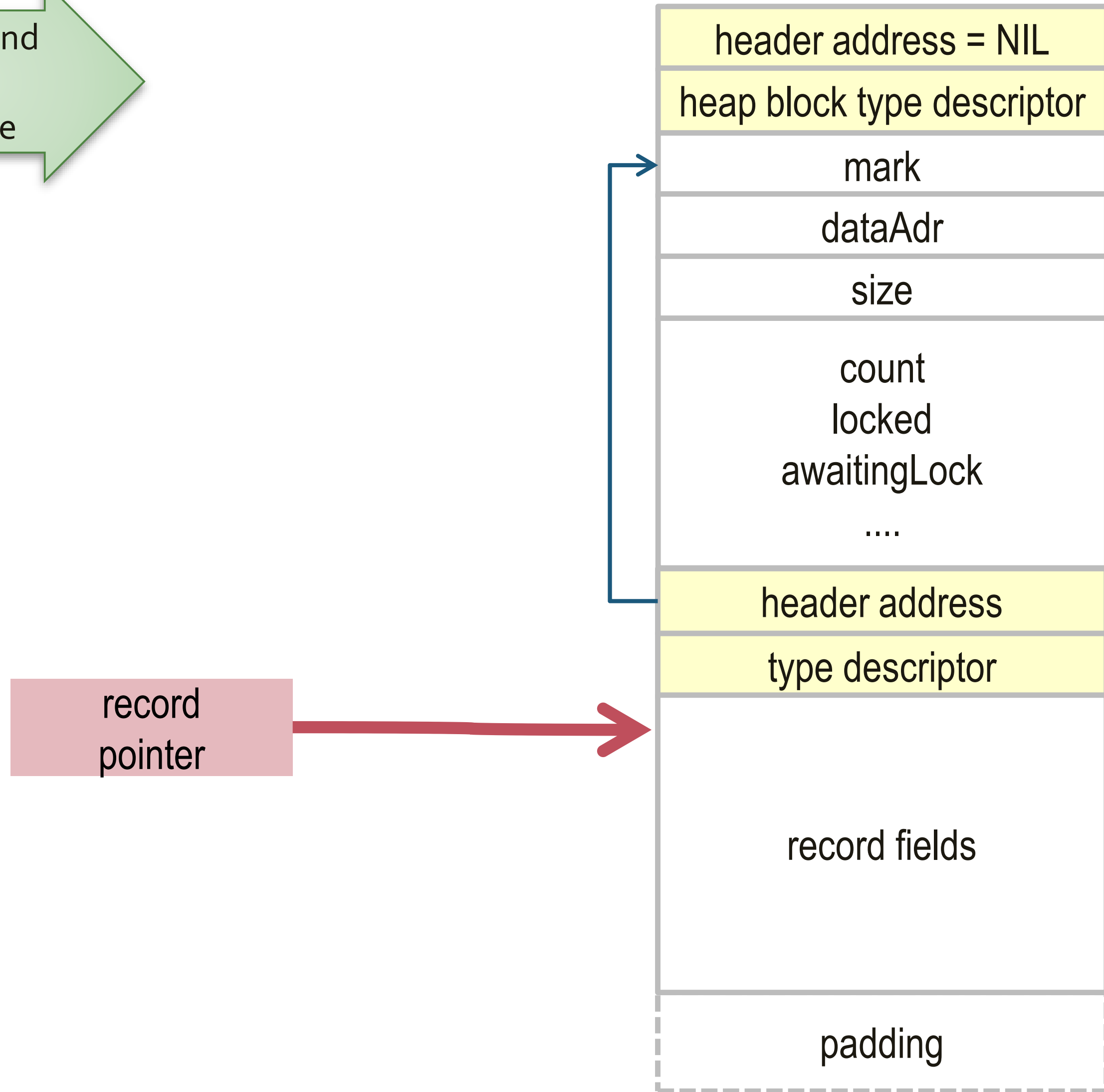

```
HeapBlockDesc* = RECORD
  heapBlock {offset=-2*adr}: ADDRESS;
  typeDesc: {offset=-adr}: StaticType;
  mark: LONGINT;
  dataAdr-: ADDRESS;
  size-: SIZE;
  nextMark {UNTRACED}: HeapBlock;
END

RecordBlockDesc = RECORD(HeapBlockDesc)
END;
```

```
DataBlock* = RECORD
  heapBlock {offset=-2*adr}: ADDRESS;
  typeDesc: {offset=-adr}: StaticType;
END;
```

(Shared) Object Blocks

compiler and code perspective

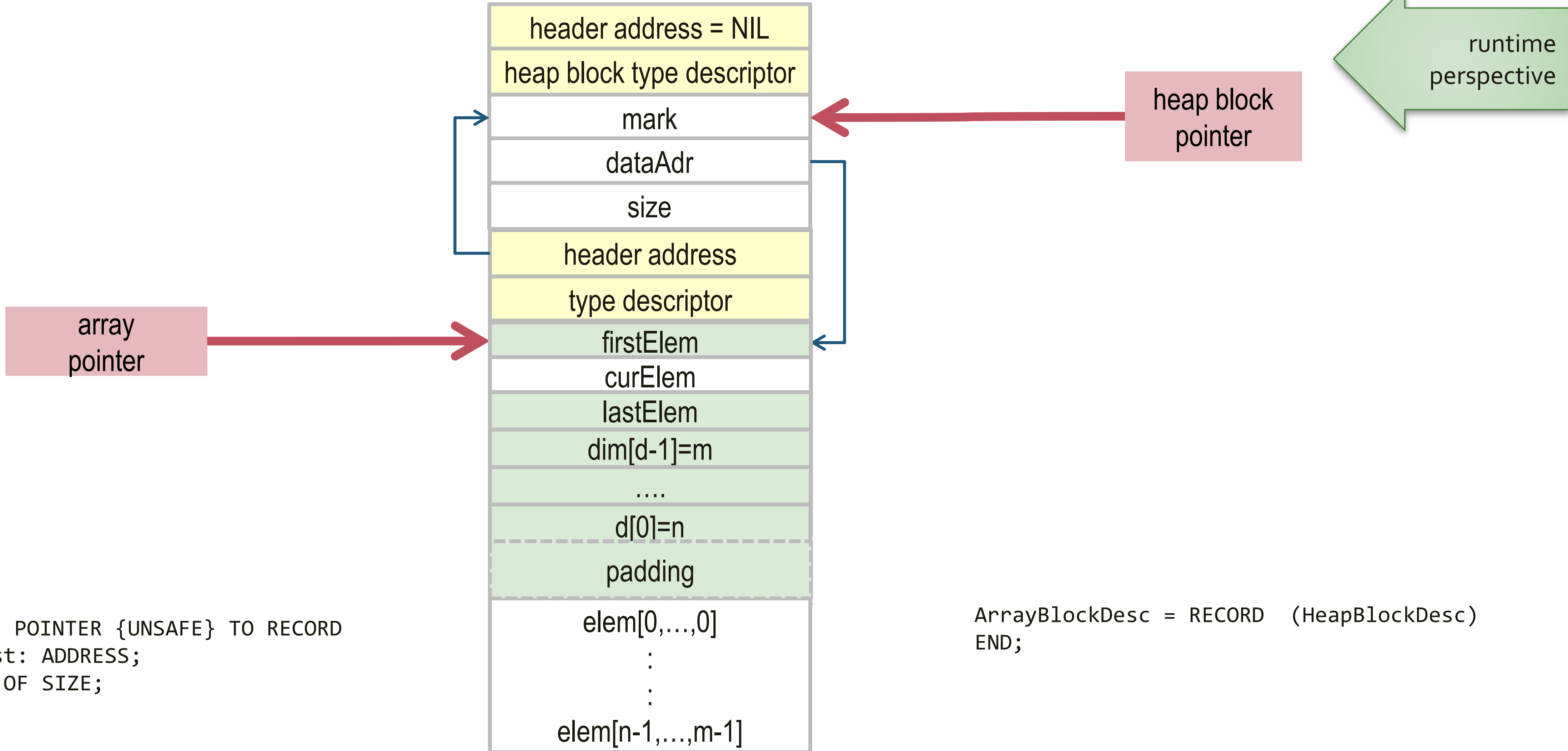
runtime perspective



heap block pointer

```
ProtRecBlockDesc* = RECORD  (RecordBlockDesc)
  count*: LONGINT;
  locked*: BOOLEAN;
  awaitingLock*, awaitingCond*: ProcessQueue;
  lockedBy*: ANY;
  waitingPriorities*: ARRAY NumPriorities OF LONGINT;
END;
```

Array Blocks



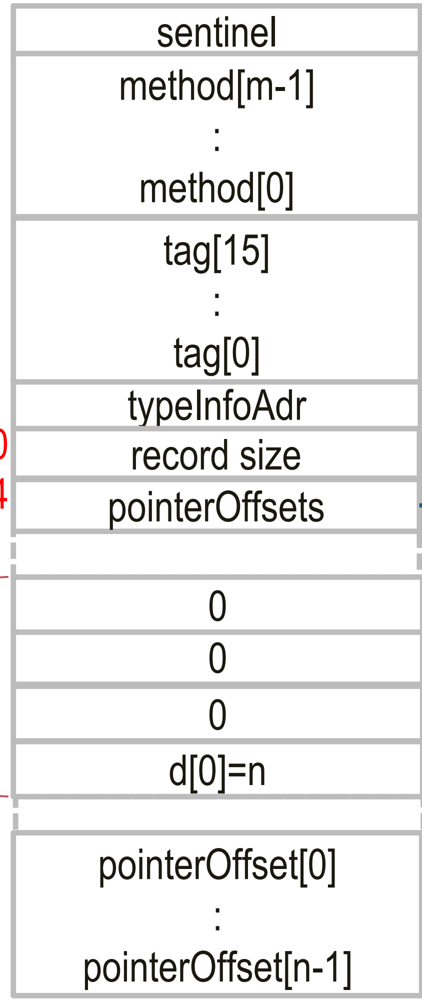

```
ArrayDescriptor*= POINTER {UNSAFE} TO RECORD
  first, cur, last: ADDRESS;
  lens*: ARRAY 8 OF SIZE;
END;
```

```
ArrayBlockDesc = RECORD  (HeapBlockDesc)
END;
```

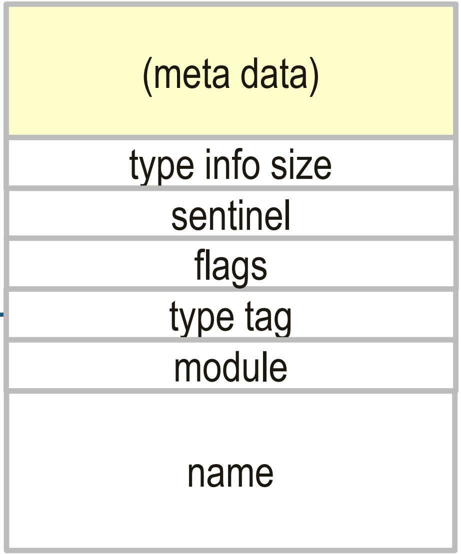
Type Descriptor Blocks

Integrated in module block

type tag reference



(array Header)

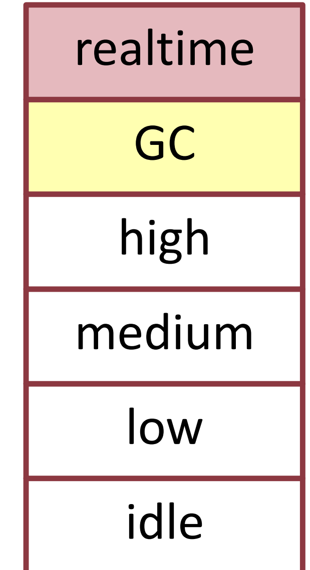



```
StaticTypeBlock*= POINTER TO
StaticTypeDesc;
StaticTypeDesc = RECORD
  recSize: SYSTEM.SIZE;
  pointerOffsets*
{UNTRACED}:PointerOffsets;
END;
```

```
TypeInfo* = POINTER TO RECORD
    descSize: LONGINT;
    sentinel: LONGINT;  (* = MPO-4 *)
    tag*: SYSTEM.ADDRESS;
    flags*: SET;
    mod*: Module;
    name*: Name;
END;
```

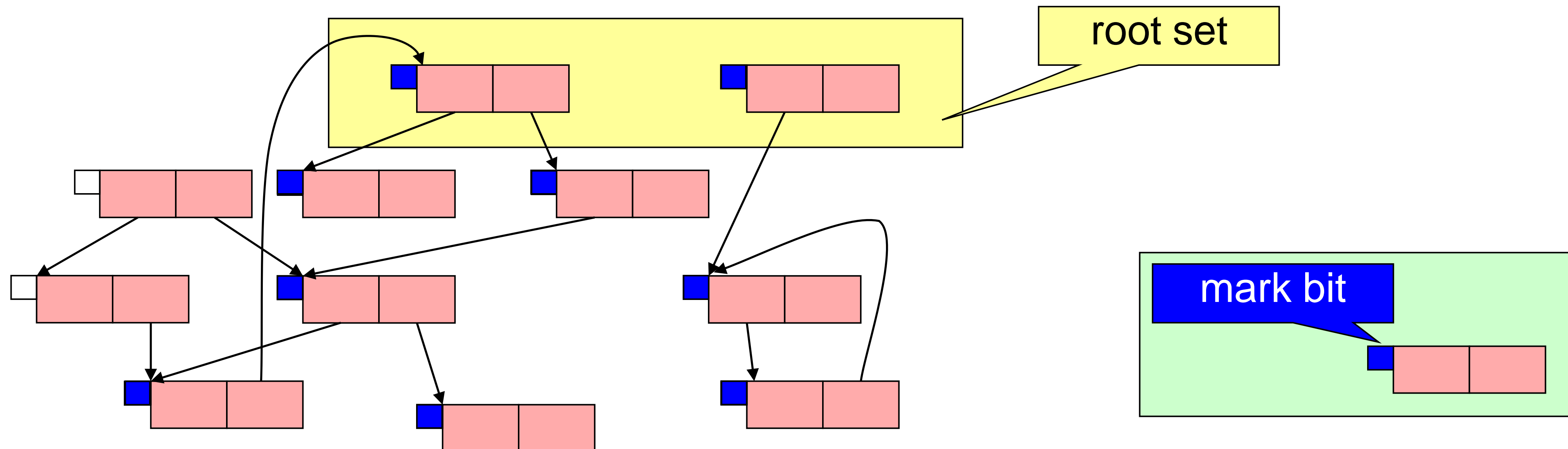
Garbage Collection

- Stop & Go Method
  - Mark phase in mutual exclusion with all mutators
- Garbage Collector has a higher priority than all normal processes
  - Once the GC is running, no normal process may be running (and scheduled)
- However, there are realtime processes with even higher priority that may preempt the GC at any time.
  - ⇨ Realtime processes do not mutate the collectable heap
  - ⇨ The garbage collector is implemented as one possible process. Consequence: special scheduling policy when the GC task becomes / is active.



Mark Phase (Mark reachable blocks)

- Determine root set
  - global pointers in modules
  - local pointers on the stacks
  - (temporary pointers in registers)
- Traverse object graph starting from root set and mark all reachable objects



Root-Set Determination

- Module Heap
  - compiler generates pointer offsets in modules
- Stacks
  - Compiler generates meta-data
    - pointer offsets per procedure

  or

  - speculatively ("heuristics")
    - plausibility tests first (pointer to allocatable heap?)
    - check for existence of heap block at address
    - conservative approach

Root Set Traversal

```
PROCEDURE CollectGarbage*(root : ANY);
VAR obj: RootObject;
BEGIN
 rootList := NIL;
 INC(currentMarkValue);
 Mark(root);
 REPEAT
   WHILE rootList # NIL DO
      obj := rootList;
      rootList := rootList.nextRoot;
      obj.FindRoots;
   END
   CheckFinalizedObjects;
 UNTIL rootList = NIL;
END CollectGarbage
```

called in Objects:
Heaps.CollectGarbage(Modules.root)

```
RootObject* = OBJECT
   VAR nextRoot: RootObject;
   PROCEDURE FindRoots*;
END RootObject;
```

Most important root objects:

- Modules
- Processes

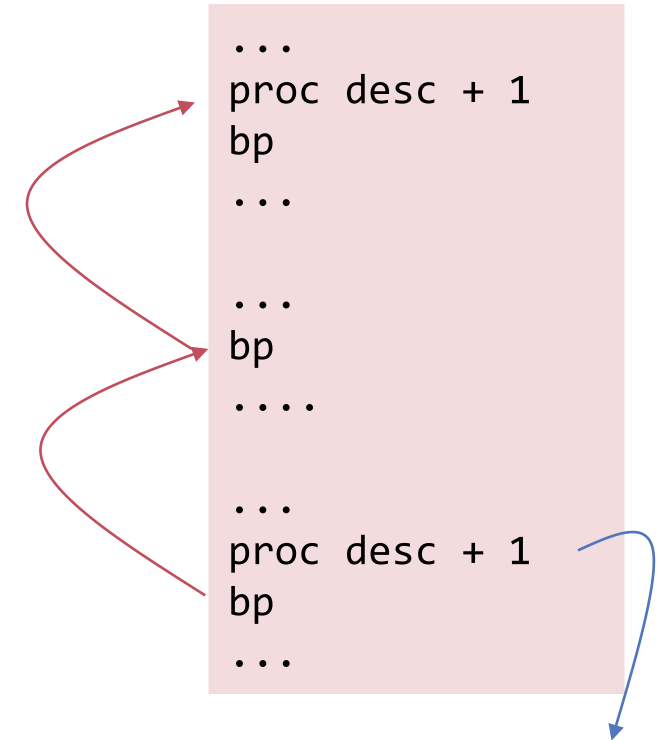
All reachable via Modules.root
➔strictly speaking Modules.root constitutes the root set

Modules.Module.FindRoots

```
TYPE
Module = OBJECT
...

    PROCEDURE FindRoots;  (* override *)
    VAR i: LONGINT; ptr: ANY; false: BOOLEAN;
    BEGIN
       IF published THEN  (* mark global pointers *)
          FOR i := 0 TO LEN(ptrAdr) - 1 DO
             SYSTEM.GET (ptrAdr[i], ptr);
             IF ptr # NIL THEN Heaps.Mark(ptr) END
          END;
          Heaps.AddRootObject(next);
       END;
    END FindRoots;


END Module
```

Objects.Process.FindRoots

```
Process.CollectGarbage:
...

WHILE (bp # Heaps.NilVal) & (bp < stackBottom)  DO
   SYSTEM.GET(bp, n);
   IF ODD(n) THEN (* procedure descriptor at bp *)
      desc := SYSTEM.VAL(Modules.ProcedureDescPointer, n-1);
      IF desc # NIL THEN
         FOR i := 0 TO LEN(desc.offsets)-1 DO
            adr := bp + desc.offsets[i]; (* pointer at offset *)
            SYSTEM.GET(adr, p); (* load pointer *)
            Heaps.Mark(p);
         END;
      END;
      SYSTEM.GET(bp + SIZEOF(ADDRESS), bp);
   ELSE (* classical / foreign stack frame *)
      bp := n;
   END;
END;
```




```
ProcedureDesc*= RECORD
  pcFrom-, pcLimit-: ADDRESS;
  offsets- {UNTRACED}: POINTER TO ARRAY OF ADDRESS;
END;
```

Mark(p)

```
Inspect(p,currentGeneration);
orgHeapBlock := ExtractFromMarkList();
WHILE orgHeapBlock # NIL DO
	orgBlock := orgHeapBlock.dataAdr; meta := orgBlock; staticTypeBlock := meta.staticTypeBlock;
	IF ~(orgHeapBlock IS ArrayBlock) THEN
		FOR i := 0 TO LEN(staticTypeBlock.pointerOffsets) - 1 DO
			b := orgBlock + staticTypeBlock.pointerOffsets[i];
			Inspect(b.p,currentGeneration)
		END
	ELSE
		currentArrayElemAdr := meta.first;
		lastArrayElemAdr := meta.first + meta.last * staticTypeBlock.recSize;
		WHILE currentArrayElemAdr < lastArrayElemAdr DO
			...
		END
	END;
	IF orgHeapBlock IS ProtRecBlock THEN
		protected := orgHeapBlock(ProtRecBlock);
		Inspect(protected.awaitingLock.head, currentGeneration);
		...
	END;
	orgHeapBlock := ExtractFromMarkList();
END;
```

GC Code Example: Inspect (called by Mark)

```
PROCEDURE Inspect(block {UNTRACED}: ANY; generation: LONGINT);
VAR
   heapBlock {UNTRACED}: HeapBlock;
   rootObj {UNTRACED}: RootObject;
   blockMeta : Block;
BEGIN
   blockMeta := block;
   heapBlock := blockMeta.heapBlock;
   IF (heapBlock = NIL) OR (heapBlock.mark >= currentMarkValue) THEN RETURN END;

   heapBlock.mark := currentMarkValue;
   IF (heapBlock IS RecordBlock) OR (heapBlock IS ProtRecBlock) OR (heapBlock IS ArrayBlock) THEN
        IF block IS RootObject THEN
             rootObj := block(RootObject);
             rootObj.nextRoot := rootList; rootList := rootObj;(* link root list *)
        END;
        IF (LEN(blockMeta.typeBlock.pointerOffsets) > 0) OR (heapBlock IS ProtRecBlock) THEN
             AppendToMarkList(heapBlock);
        END
   END
END Inspect;
```

Usefulness of Abstraction

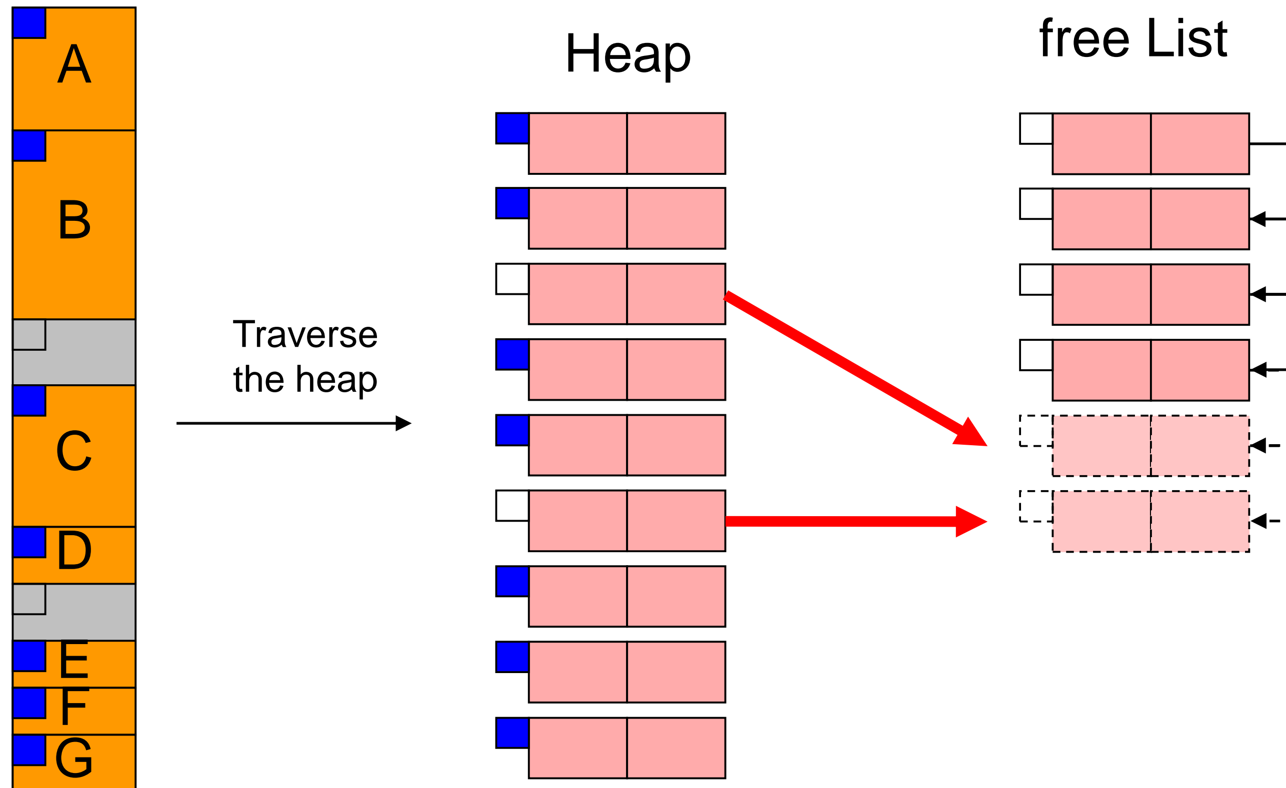
```
IF ~(SubObjBit IN SYSTEM.VAL(SET, refBlock)) THEN
    IF ~(ProtObjBit IN SYSTEM.VAL(SET, refBlock)) THEN (* RecBlk, ArrBlk *)
                   refPTagAdr := refBlock-4
    ELSE (* ProtRecBlk *)
         refPTagAdr := refBlock-(ProtOfs+4)
    END
ELSE (* SysBlk, SysBlkArray, TypeDesc *)
    SYSTEM.GET(refBlock-4, refPTagAdr);
    refPTagAdr := SYSTEM.VAL(LONGINT, SYSTEM.VAL(SET, refPTagAdr) - {ArrayBit}) - 4;
END;
SYSTEM.GET(refPTagAdr, refTag);
IF ~(MarkBit IN SYSTEM.VAL(SET, refTag)) THEN
    SYSTEM.PUT(refPTagAdr, SYSTEM.VAL(SET, refTag) + {MarkBit});
    IF ~(SubObjBit IN SYSTEM.VAL(SET, refBlock)) THEN
        SYSTEM.GET(refBlock-4, refPTDescAdr);
        refPTDescAdr := SYSTEM.VAL(LONGINT, SYSTEM.VAL(SET, refPTDescAdr) - {MarkBit,
                   ArrayBit});
        SYSTEM.GET(refPTDescAdr-4, refTdIndirectTag);
        refTdIndirectTag := SYSTEM.VAL(LONGINT, SYSTEM.VAL(SET, refTdIndirectTag) -
                   {MarkBit}) - 4;
                       SYSTEM.GET(refTdIndirectTag, refTdEffectiveTag);
        SYSTEM.PUT(refTdIndirectTag, SYSTEM.VAL(SET, refTdEffectiveTag) + {MarkBit});
        IF (SubObjBit IN SYSTEM.VAL(SET, refPTDescAdr)) &
                (SYSTEM.GET32(refPTDescAdr-8) = SYSTEM.TYPECODE(RootObject)) THEN
                ......
```

(with typed record blocks)

```
IF (heapBlock # NIL) & (heapBlock.mark < currentMarkValue) THEN
    heapBlock.mark := currentMarkValue;
    IF (heapBlock IS RecordBlock) OR (heapBlock IS ProtRecBlock)
              OR (heapBlock IS ArrayBlock) THEN
        IF block IS RootObject THEN
              .....
```

Sweep Phase (Return unmarked blocks)

- Trvaerse the heap in a linear fashion. Merge free blocks and enter into free lists
- Can be combined with alloc ("lazy sweep")



Mark and Lazy Sweep

- GC implemented in two phases
  - Mark
    - Traverse objects in heap starting from root objects
    - Metadata are used to identify pointers on stack
  - Lazy Sweep
    - Get free block from heap on allocation request
    - Does not necessarily run directly after mark phase

Lazy Sweep

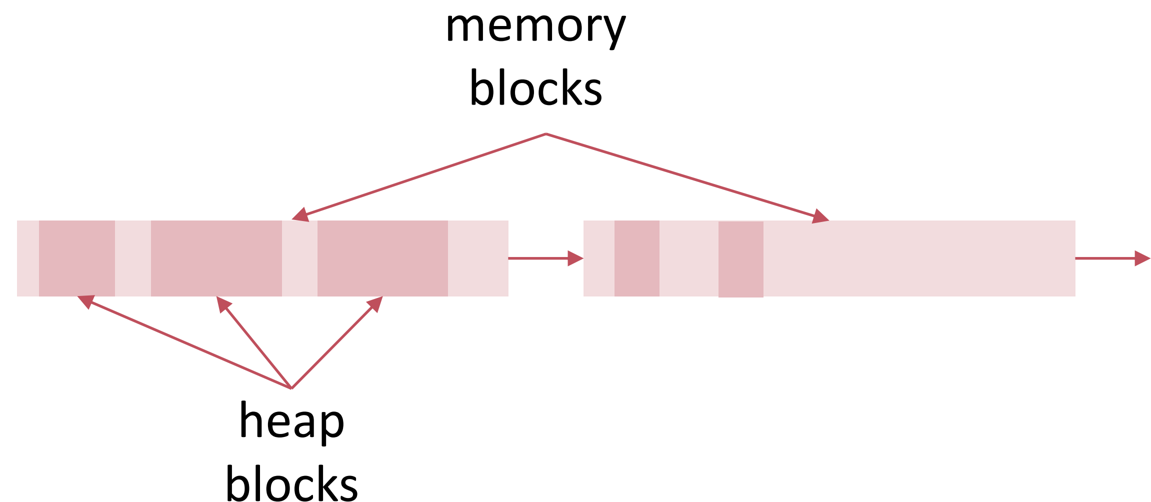
- `PROCEDURE LazySweep(size: SIZE, VAR p: FreeBlock)`
  - Traverse heap and identify free blocks
    starting at most recent sweep position (nextFit algorithm)
  - Merge subsequent free blocks
  - Return first found large enough free block
  - No free lists necessary
  - Full Sweep available via LazySweep(MAX(SIZE),p)

Two levels of heap management

- Memory Block Management
  - Module Machine
  - Provides hardware abstraction and permits handling discontigous memory
  - Interface
    - FreeMemBlock*(memBlock: MemoryBlock);
    - ExpandHeap*(try: LONGINT; size: SYSTEM.SIZE; VAR memBlock: MemoryBlock; VAR beginBlockAdr, endBlockAdr: SYSTEM.ADDRESS);

```
MemoryBlock* = POINTER TO MemoryBlockDesc;
MemoryBlockDesc* = RECORD
    next- {UNTRACED}: MemoryBlock;
    startAdr-: SYSTEM.ADDRESS;
    size-: SYSTEM.SIZE;
    beginBlockAdr-, endBlockAdr-: SYSTEM.ADDRESS
END;
```

- Heap Block Management
  - Module Heaps
  - Memory allocation for NEW constructs
  - Garbage collection



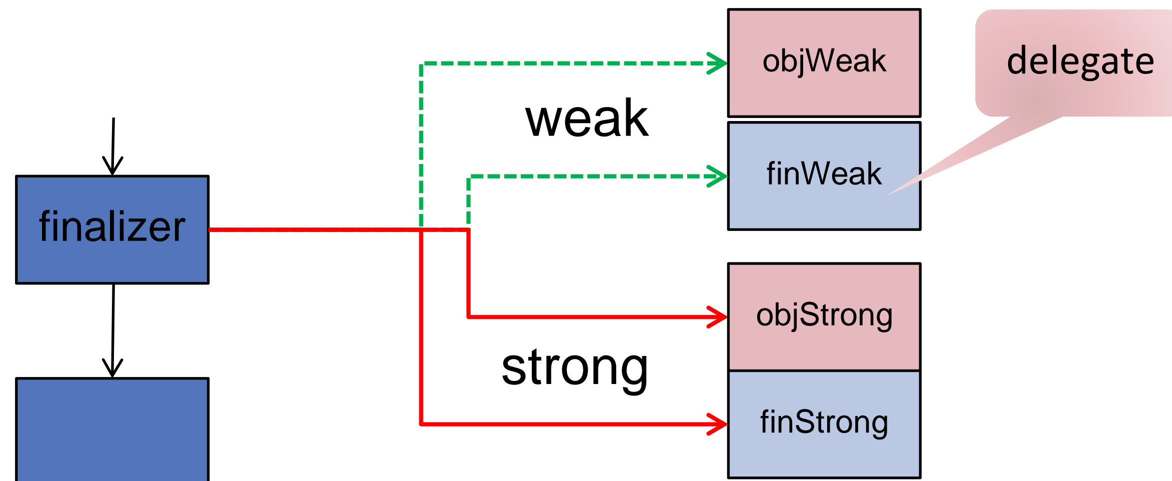
Finalizers

- Finalizers are special methods / delegates

- Are called when objects are released
  - Consistency: save caches such as (file) buffers
  - Return resources (files), release connections

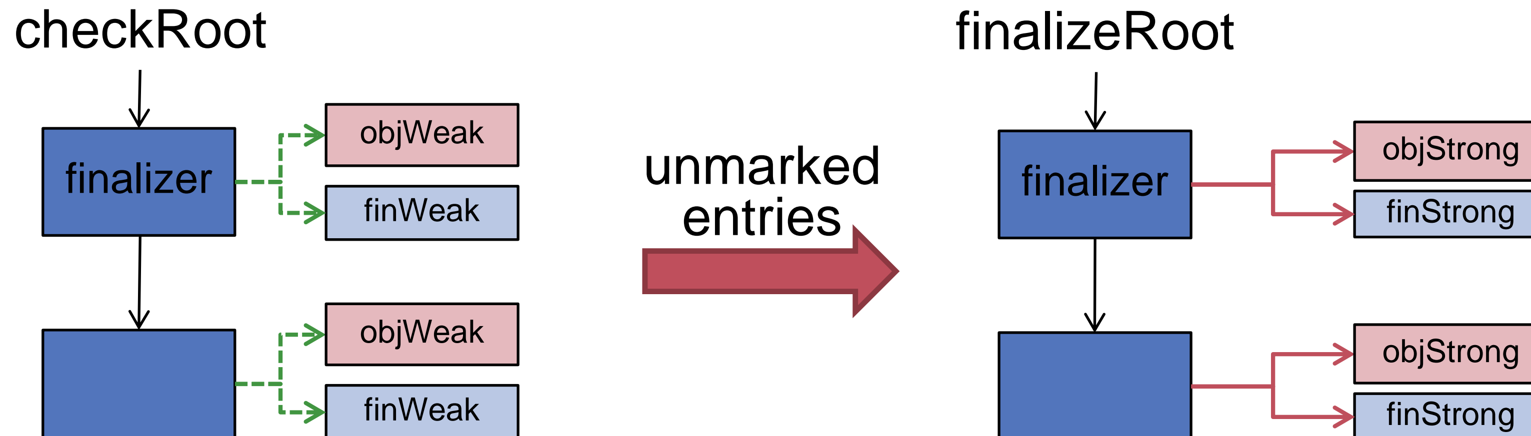
Finalizers

- Weak pointer: Pointer invisible to the GC
  - p { UNTRACED }: AnyObjectType
- Finalizer: associated delegate to be executed after object has been recognized as garbage
- Finalizer list

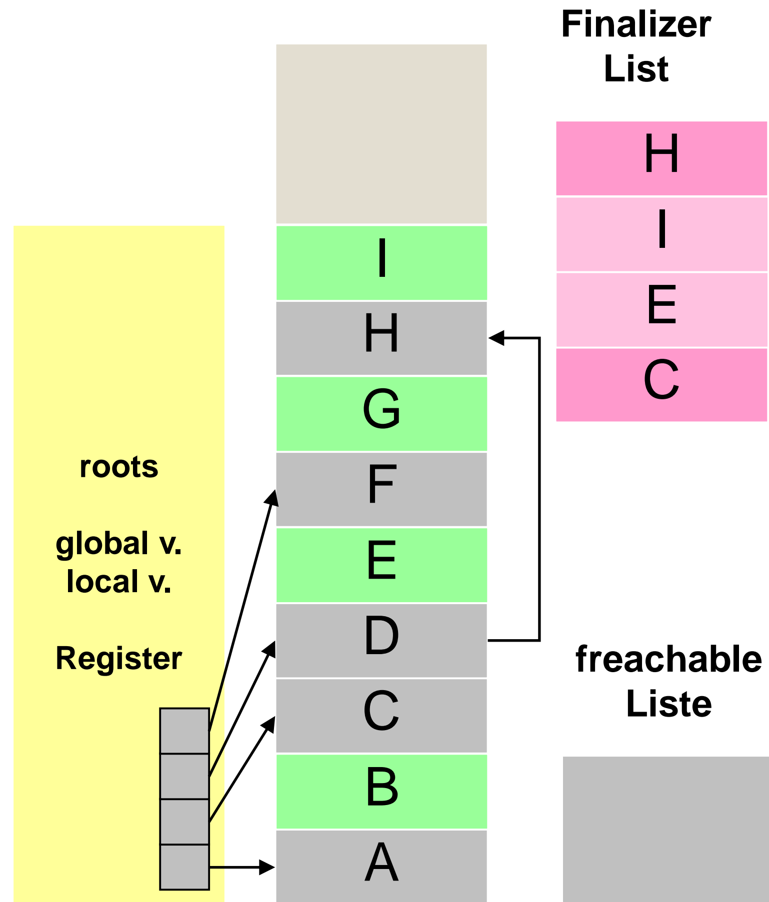


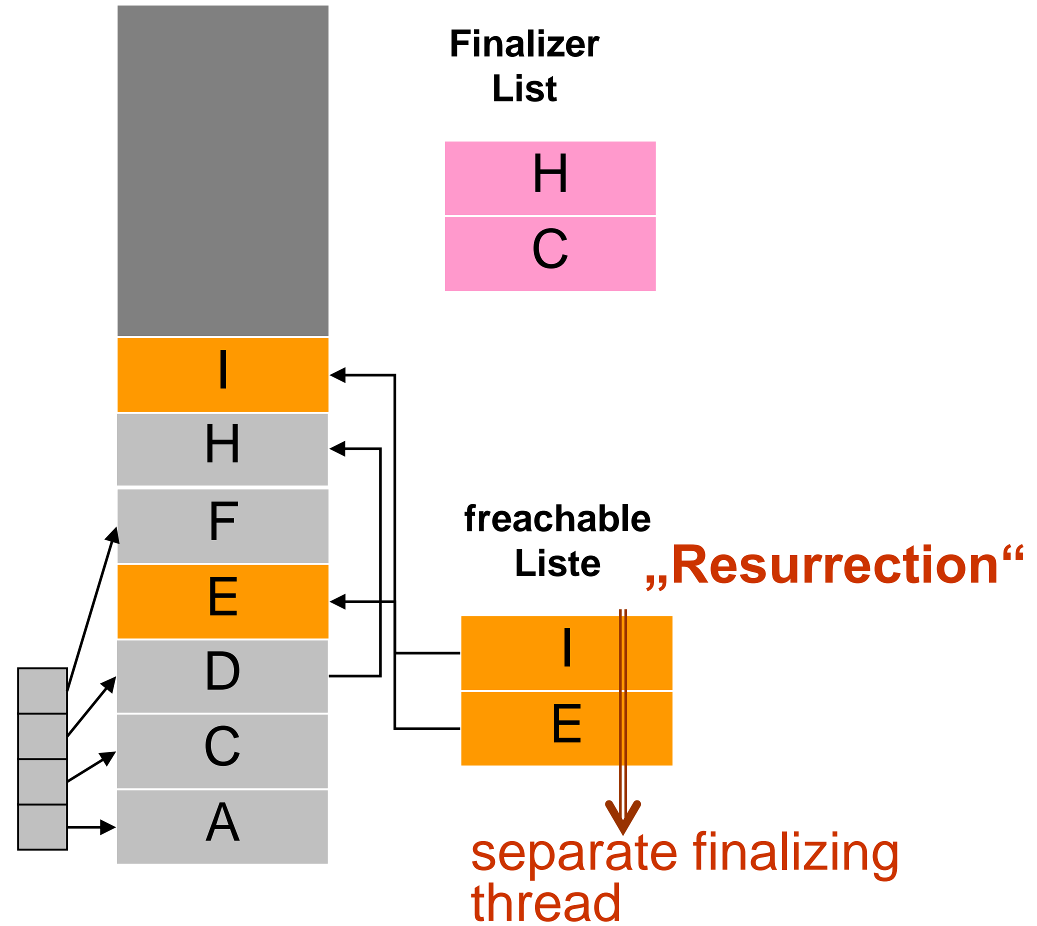
Check Finalized Objects

- `PROCEDURE CheckFinalizedObjects`
  - Traverse list of all objects registered for finalization
  - Mark unmarked objects and move them to the finalizer list for later finalization by separate process
  - Mark weak delegates of marked finalizers



Finalizing





before GC

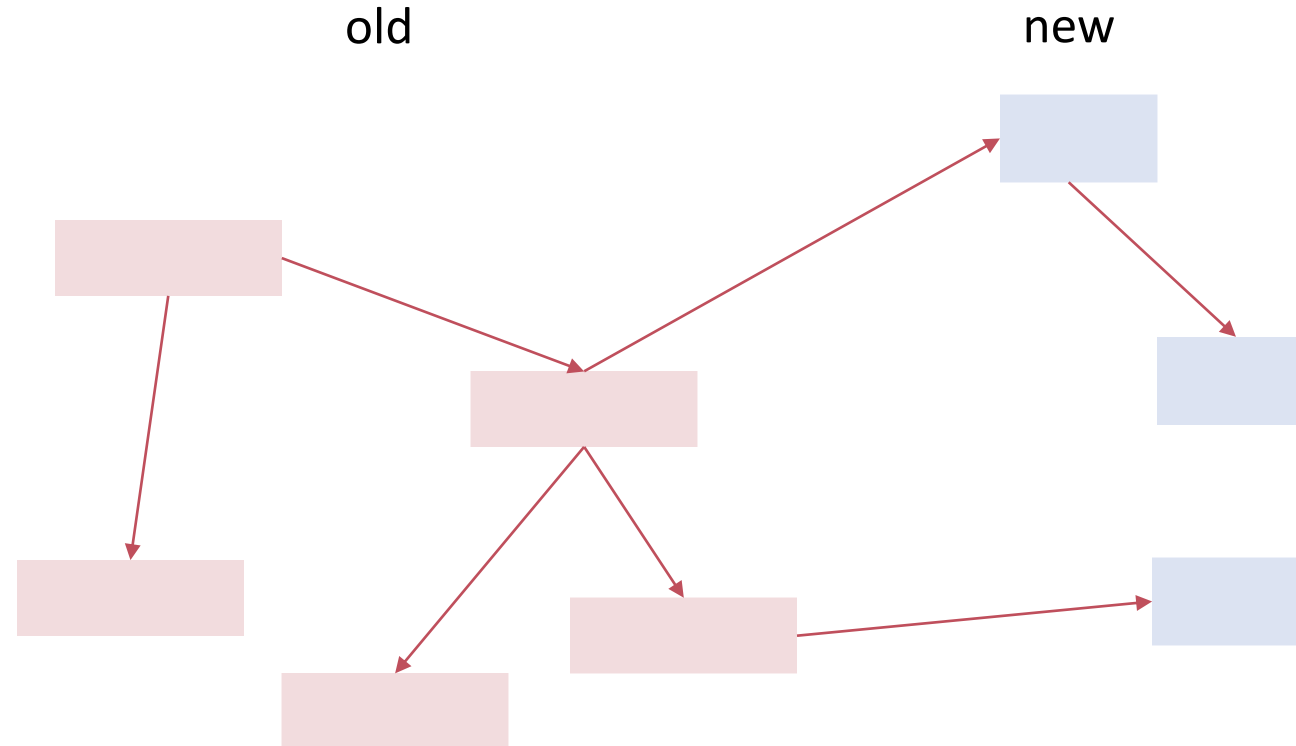
after GC

Generational Garbage Collection

- Observation 1: many short-lived objects
- Observation 2: garbage collection cycle time depends on #objects traversed

- Compromise: often check new / short lived, less often check old objects
- Problem: how to only traverse new (live) objects?

Generational Garbage Collection



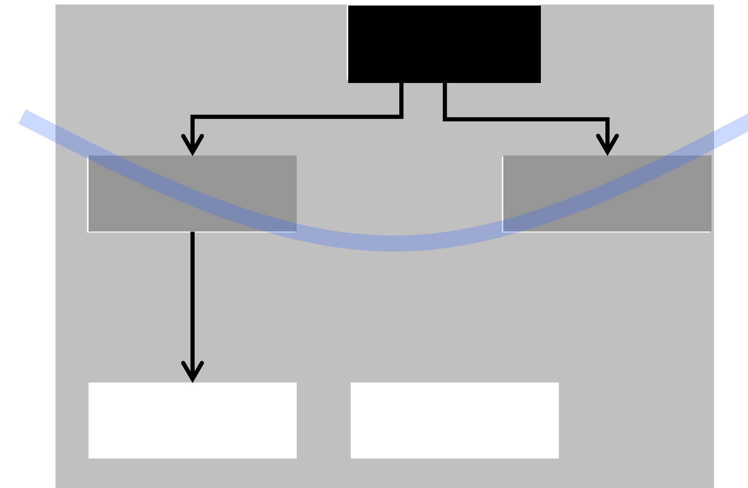
how to distinguish old from new ?
how to incorporate old -> new pointers?

Generational Garbage Collection

- how to distinguish old from new ?
  - Mark values (mark MOD #generations = #generation)
- how to incorporate old → new pointers ?

Three Color Abstraction (wavefront model)

State	Color
already traversed and marked (behind wave front)	black
is being traversed / marked (on wave front) [on mark stack / in mark list]	grey
Still untouched (in front of wave front)	white



Forbidden: Pointer (only) from white to black

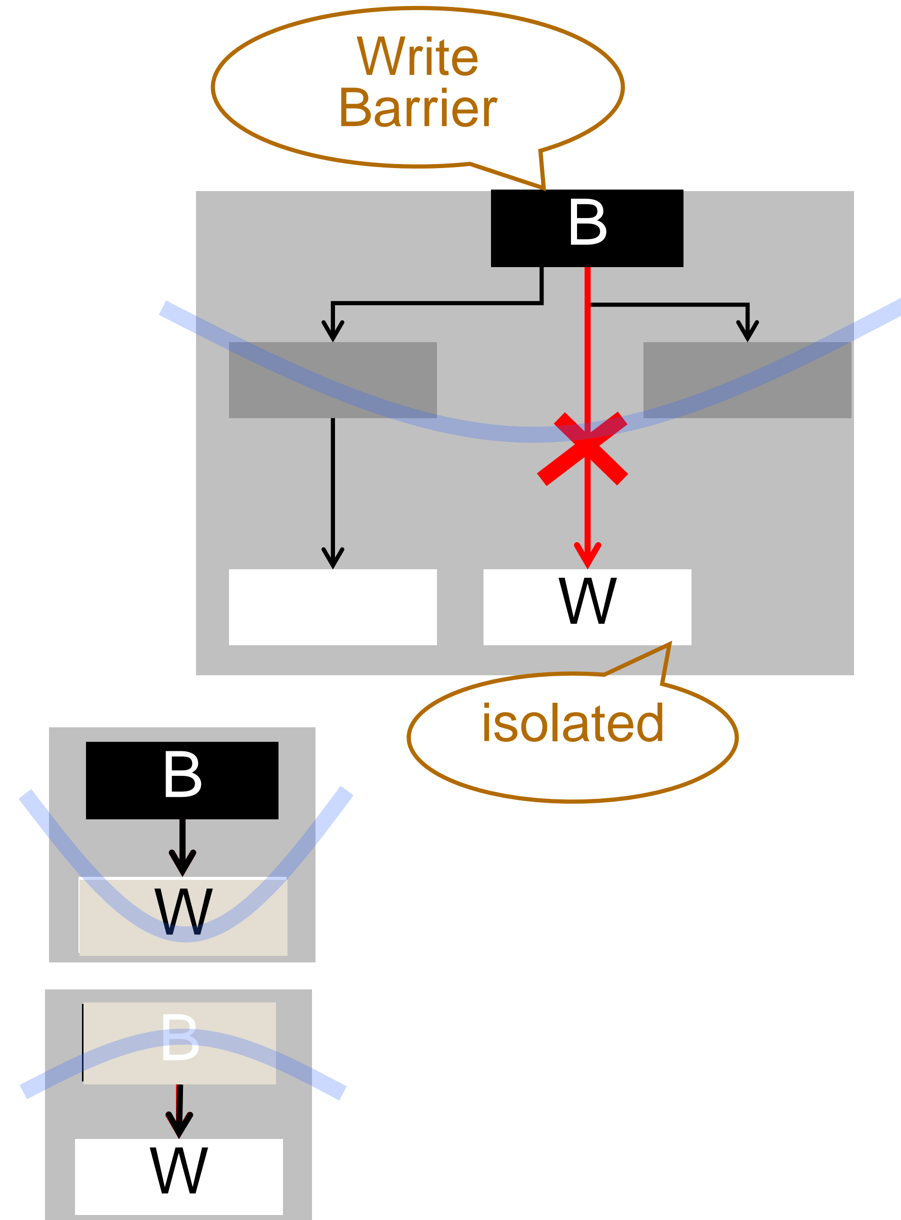
Barriers

No pointer (only) from black to white

- Critical case: new pointer black to white
- Two possibilities
  - Read Barrier

    W ← grey
  - Write Barrier

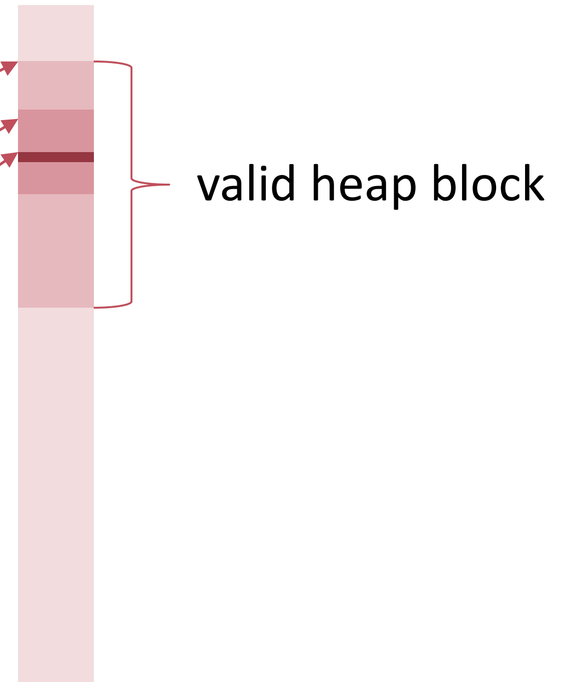
    B ← grey



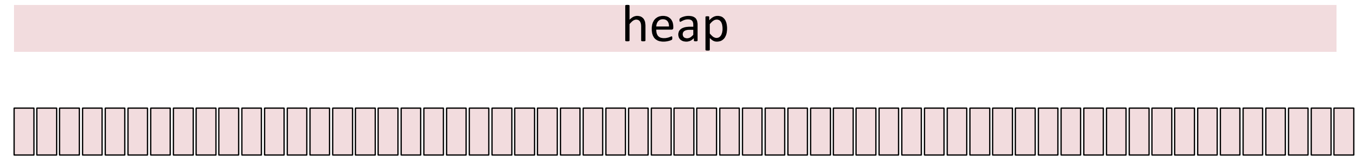
Coloring: how?

- Problem: cannot determine the destination heap block exactly. Cannot easily enter into mark list.
- Here is why:

```
PROCEDURE SomeProcedure(VAR r: MyRecord)
BEGIN
	r.field := NewPointer(); (* call write barrier *)
END SomeProcedure;

SomeProcedure(myHeapBlock.recordEmbedded);
```



Coloring: how?



"card set": bit per heap region

```
(* lock-free entry into card-set *)
PROCEDURE EnterInCardSet(adr: ADDRESS);
VAR value: SET;
BEGIN
   adr := adr DIV CardSize;
   IF  adr MOD SetSize IN CAS(cardSet[adr DIV SetSize],{},{}) THEN
      RETURN
   ELSE
      LOOP
         value := CAS (cardSet[adr DIV SetSize], {},{});
         IF CAS (cardSet[adr DIV SetSize], value, value + {adr MOD SetSize}) = value THEN EXIT END;
         (*CPU.Backoff;*)
      END;
   END;
END EnterInCardSet;
```

```
PROCEDURE CheckAssignment*(dest, src: DataBlockU);
BEGIN
   IF (src # NIL) & (src.heapBlock # NIL) & (src.heapBlock.mark MOD GenerationMask = Young) THEN
      EnterInCardSet(dest);
   END;
END CheckAssignment;
```

proper (non-static) heap object

in new generation

Collect garbage (new generation): sweep card set, enter outgoing pointers to new objects

2.5 COMPILER-SUPPORTED UNIFICATION OF STATIC AND DYNAMIC LOADING

Contents of an Object File

- Machine code and data
- Metadata, i.e. information about
  - relocation
  - reflection
  - garbage collection
  - debugging
  - exception handling
  - runtime type system
  - etc.

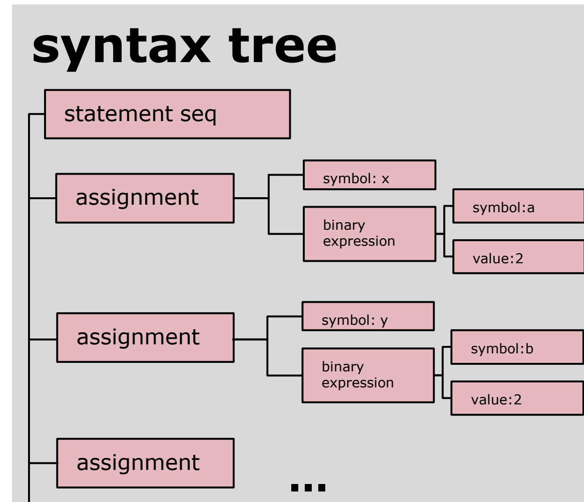
Motivation

- Modifications of toolchain and runtime support of programming language became more and more difficult over time.
- This was partially due to an increasingly complex object file format that became
  - hard to understand
  - hard to maintain
  - hard to extend
- Wanted to exploit co-design of language, compiler and runtime system to improve the situation.

Compilation Process

Frontend

syntax tree



syntax tree traversal

intermediate code

```
.module A
 ....

.code A.P
  ....
  1:  mul s32 [A.x:0], [A.a:0], 2
  2:  mul s32 [A.y:0], [A.y:0], 2
   ....
```

Compilation Process

Backend

intermediate code

```
.module A
 ....

.code A.P
  ....
  1:  mul s32 [A.x:0], [A.a:0], 2
  2:  mul s32 [A.y:0], [A.y:0], 2
  ....
```

IR code traversal

binary code

Fixups

```
fixup 2 <-- A.a (displ=0) [3+7 abs]
fixup 6 <-- A.x (displ=0) [3+7 abs]
fixup 7 <-- A.b (displ=0) [3+7 abs]
...
[  2] 30807 ; LD R1, [0]
[  3] 00401 ; MOV R0, R1
[  4] 2801F ; ROR R0, 31
[  5] 14001 ; BIC R0, 1
[  6] 34007 ; ST R0, [0]
....
```

Symbolic References

Source Code / Interface (Symbol File)

```
MODULE A;
VAR a*: LONGINT;

PROCEDURE P*;
BEGIN a := a*10 END P;
END A.
```

```
MODULE A;
    VAR a*: LONGINT;
    PROCEDURE P*;
```

```
MODULE B;
IMPORT A;
BEGIN A.a := 1; A.P() END
B.
```

```
MODULE B;
    IMPORT A;
```

Intermediate Code

```
.module A
.const A.@moduleSelf offset=0
  0:  data u32 0
.var A.a offset=-4
  0:  reserve 4
.code A.P offset=0
  0:  enter 0, 0
  1:  mul s32 [A.a:0], [A.a:0], 10
  2:  leave 0
  3:  return 0
```

```
.module B
.imports A
.const B.@moduleSelf offset=0
    0:  data u32 0
.code B.$$Body offset=0
  0:  enter 0, 0
  1:  mov s32 [A.a:0], 1
  2:  call u32 A.P:0, 0
  3:  leave 0
  4:  return 0
```

Fixups

Object File*

```
const A.@moduleSelf 651605535 8 aligned 4 0 4
    00000000
data A.a 471859334 8 aligned 4 0 4
    00000000
code A.P 2046820492 8 aligned 0 2 24
    abs 12 A.a 471859334 0 0 1 0 32
    abs 18 A.a 471859334 0 0 1 0 32
    8C0000008BA0000000F0FA50000000000985000000000009C3C
code A.$$BODY -278328333 8 aligned 0 0 6
    8C0000009C3C
```

```
const B.@moduleSelf 651605535 8 aligned 4 0 4
    00000000
code B.$$Body -345437455 8 aligned 0 2 21
    abs 6 A.a 471859334 0 0 1 0 32
    rel 15 A.P 2046820492 -4 0 1 0 32
    8C0000007C500000000010000008EDEFFFFFF9C3C
```

Fingerprint

consistency

Fingerprint (must match)

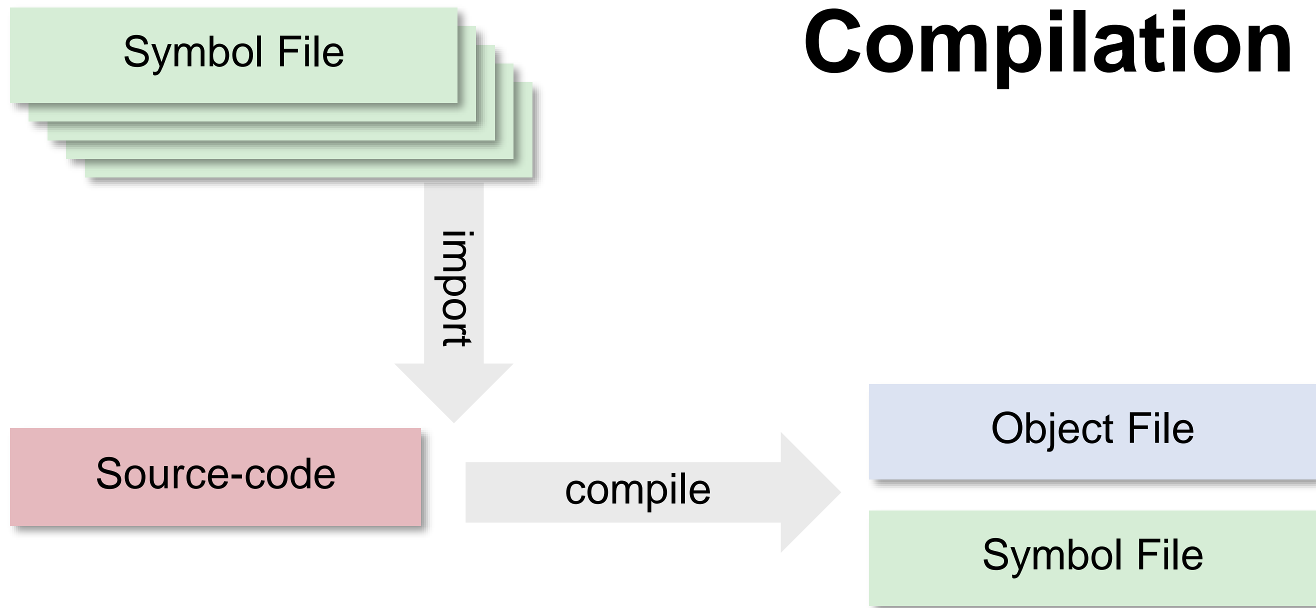
(relative) Fixup

294

*here: Intel x86

Recap: The role of the Object File

in a system with dynamic loading

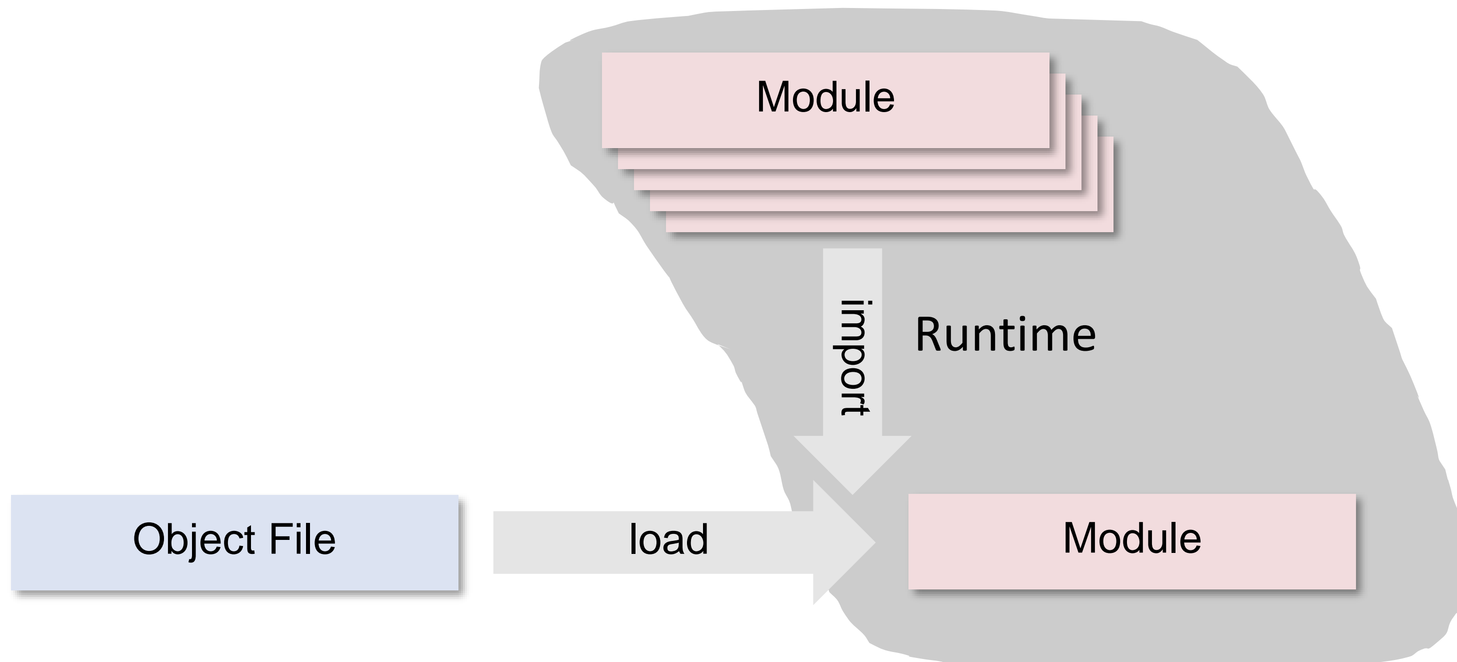


Compilation

Recap: The role of the Object File

in a system with dynamic loading

Static Linking



Dynamic Linking

Required Metadata

Language Features

- type-safe
- object oriented
- modular
- garbage collected
- exception handling
- process synchronisation
- post-mortem debugger
- …

Required Metadata

- type descriptor
- method table
- module descriptor
- pointer offsets
- exception pointers
- process queues
- debugging info
- …

The Linking Process

- Linker or loader must provide at least a mechanism to resolve references when arranging sections of an object file in memory.

- Metadata are usually generated from designated parts of the object file stored in a proprietary format.

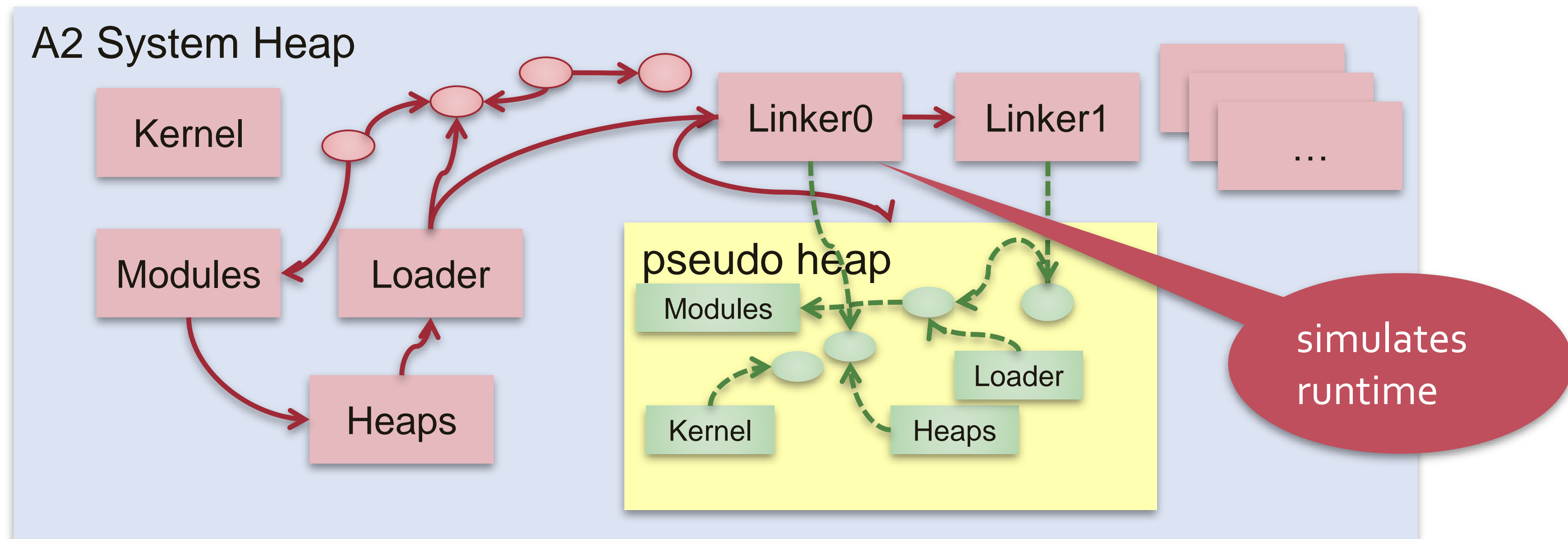
Traditional Object File in Oberon

SectionName	Meaning	Data stored
Commands	list of commands	name, argType, retType, codeOffset
Pointers	list of addresses of global variable pointers	pointerOffset
Imports	imported	module Name
Links	fixup lis	code and data offsets of fixup queues / case tables
Constants	constant section	
Exports	exported symbols	gerprint, entry , exportType
Code	code section	b
Use	references to imported modules and system calls	moduleNa gerprint
Types	description of types	name, baseModule , base ntry, methods etc.
Refs	debugging information	(long proprietary format)
ExceptionTable	list of finally sections	pcFrom, pcTo, pcHandler
PtrsInProcs	pointers on stack frames	codeOfs, beginOfs, endOfs, numberPointers list of pointers

Too complex

Traditional Oberon Linking Approach

- Allocate a pseudo-heap and use a simulated module loading to place modules and all necessary data structures in this heap.
- Then store heap as an array of bytes.
- To solve the Hen-and-Egg problem, the linker imitates behaviour of the runtime system and thus generates a 1:1 image.



Disadvantages of Traditional Approach

- Static Linker and Dynamic Loader and all data structures duplicated.
- A modification of the kernel implies a lot of work and is error-prone.
- A reduction of complexity is hardly possible.
- **Definitive Showkiller:** No crosslinking possible!

A different approach

The compiler generates all **metadata as ordinary data sections** and uses the fixup mechanism to establish the necessary links.

- For a modification of the runtime structures only the compiler and little parts of the runtime modules have to be adapted.
- Loader and linker do only need to arrange data in memory / boot image and patch the fixups
- Loader and linker are nearly identical and can use the same code base for patching fixups
- Optimizations, such as sorting and generation of hash-tables for addresses or fingerprints may still be performed by the loader

Object File Format

Object File consists of a list of sections

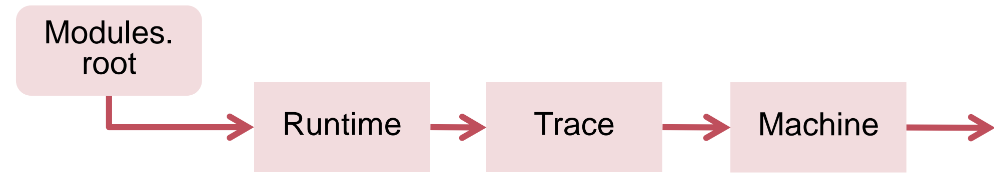
- Section Types
  - Code Sections
  - Data Sections
- Section Attributes
  - Identifier: globally unique name + fingerprint
  - Relocatability: aligned or fixed
  - Unit in bits, Size in units

Each section contains

- A list of fixups
- A sequence of bits representing code or data

Metadata Hook:

Modules.Module




```
Module* = OBJECT (Heaps.RootObject)
VAR
	next*: Module;		(** once a module is published, all fields are read-only *)
	name*: Name;
	init, published: BOOLEAN;
	refcnt*: LONGINT; (* counts loaded modules that import this module *)
	sb*: ADDRESS; (* reference address between constants and local variables *)
	entry*: POINTER TO ARRAY OF ADDRESS;
	command*: POINTER TO ARRAY OF Command;
	ptrAdr*: POINTER TO ARRAY OF ADDRESS; (* traced explicitly in FindRoots *)
	...

END Module;
```

Metadata: Modules.Module

Intermediate code

```
.const Test.@Module
0:  data u32 0    ; headerAdr
1:  data u32 0    ; typeDesc
2:  data s32 -1    ; mark: LONGINT;
3:  data u32 Test.@Module:21  ; dataAdr
4:  data u32 0    ; size-: SYSTEM.SIZE
5:  data s32 0    ; count*: LONGINT
6:  data s32 0    ; locked*: BOOLEAN
7:  data u32 0    ; awaitingLock*: ProcessQueue
8:  data u32 0
…
19: data u32 Test.@Module:2    ; HeapBlock
20: data u32 0    ; TypeDescriptor
21: data u32 0    ; nextRoot*: RootObject
22: data u32 0    ; next*: Module
23: data u8 84    ; name*: Name
24: data u8 101
….
62: data u32 Test.@CommandArray:7;
….
```

Layout of a Heap Block (lines 0–21)

Modules.Module (lines 21–62)

Object File

```
module Module;

var variable: integer;

procedure Procedure;
begin
    variable := 10;
    trace(variable);
end Procedure;

end Module.
```

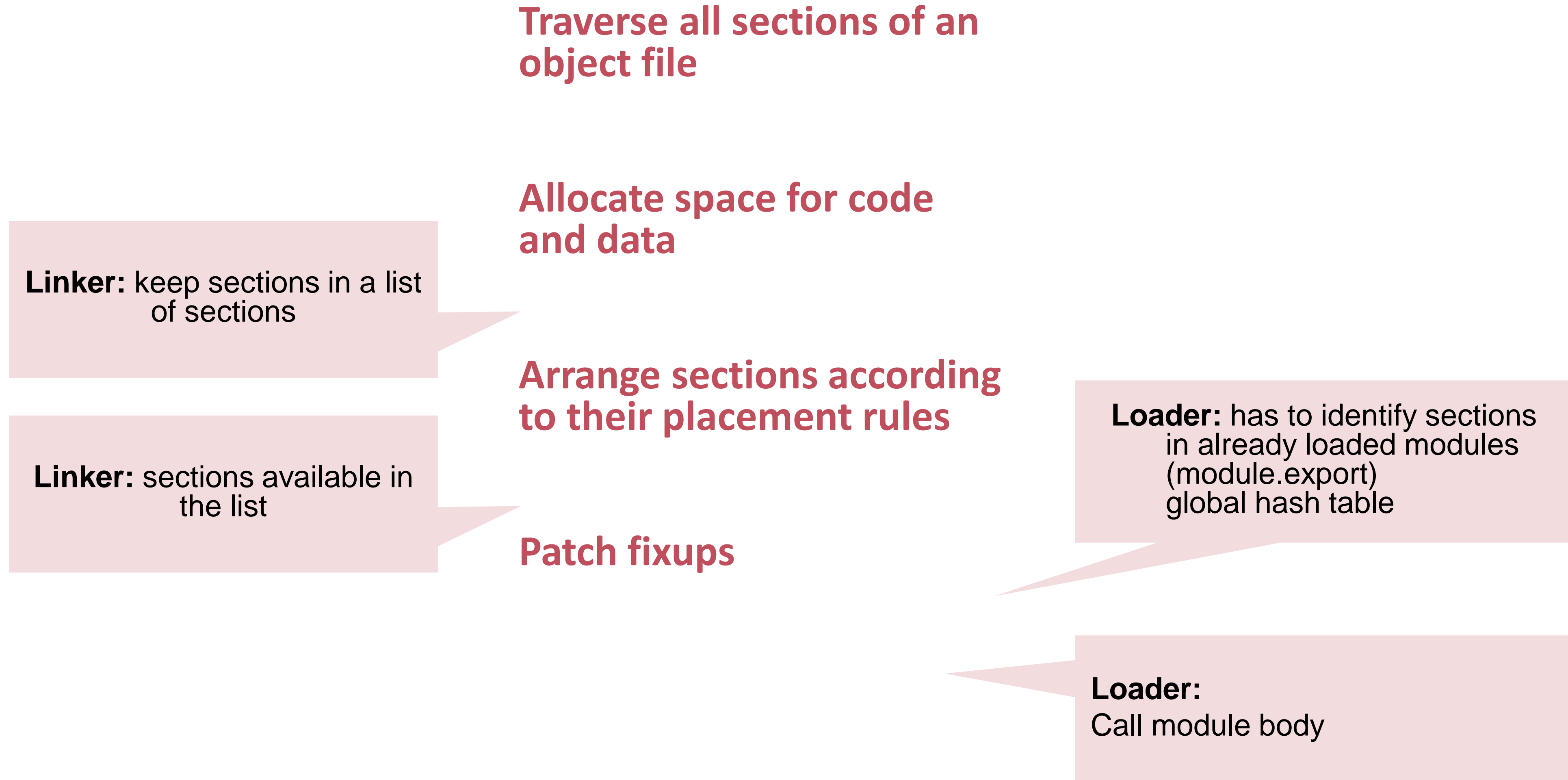
section= type name fingerprint   unit-size   alignment   #fixups size

```
...
data Module.variable 1053397876 8 aligned 2 0 2


code Module.Procedure -1838848940 8 aligned 1 6 60
```

fixup = name fingerprint   #patches {type displacement shift #patterns {offsets bits} {offsets} }

```
Module.variable 1053397876 1 abs 0 0 1 0 32 2 6 27
Module.@const0 -1762689000 1 abs 0 0 1 0 32 1 15
KernelLog.String 1370406993 1 rel -4 0 1 0 32 2 20 47
KernelLog.Int 828103137 1 rel -4 0 1 0 32 1 35
Module.@const1 846277559 1 abs 0 0 1 0 32 1 42
KernelLog.Ln -1372435551 1 rel -4 0 1 0 32 1 52
55985E667C5000000000A000A66186000000008E8E
FFFFFFF0FBD30000000075A6108E9DFFFFFFA6308
6000000008EDCFFFFFF8E8CFFFFFF98CED53C
....
```

Object File Loading / Linking



Some numbers

Traditional object file format

Module	Lines of code	Number Characters	Code size
Linker0	1554	57k	26k
Linker1	887	28k	17k
Linker	95	3k	3k
Loader	891	28k	18k
Compiler ObjFileWriter	2009	71k	37k
Sum	**5456**	**187 k**	**101 k**

Generic object file format

Module	Lines of code	Number Characters	Code size
ObjFileWriter	278	8k	6k
GenericLinker	234	8k	8k
StaticLinker*	400	16k	10k
Loader	380	12k	6k
Compiler ObjFileWriter	135	5k	6k
Sum	**1427**	**49 k**	**36 k**

*including PE32, PE64, ELF and native format