Compilation

0368-3133 2015/16a

Lecture 12

Dynamic Linking & Loading

Noam Rinetzky

- http://www.iecc.com/linker

- Legal, free preprint

Assembly ➔ Image

Source file (*e.g., utils*)

Compiler

Assembly (.s)

Assembler

Object (.o)

Source file (*e.g., main*)

Compiler

Assembly (.s)

Assembler

Object (.o)

library

Compiler

Assembly (.s)

Assembler

Object (.o)

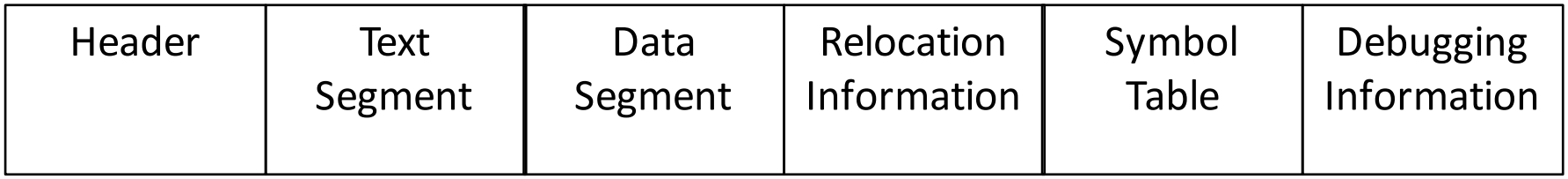
Linker

Executable (".elf")

Loader

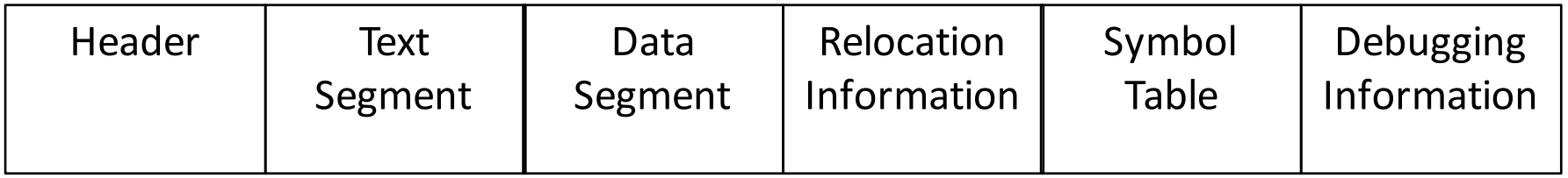
Image (in memory):

Object File Format



- Header: Admin info + “file map”
- Text seg.: machine instruction
- Data seg.: (Initialized) data in machine format
- Relocation info: instructions and data that depend on absolute addresses
- Symbol table: “exported” references + unresolved references

Object File Format

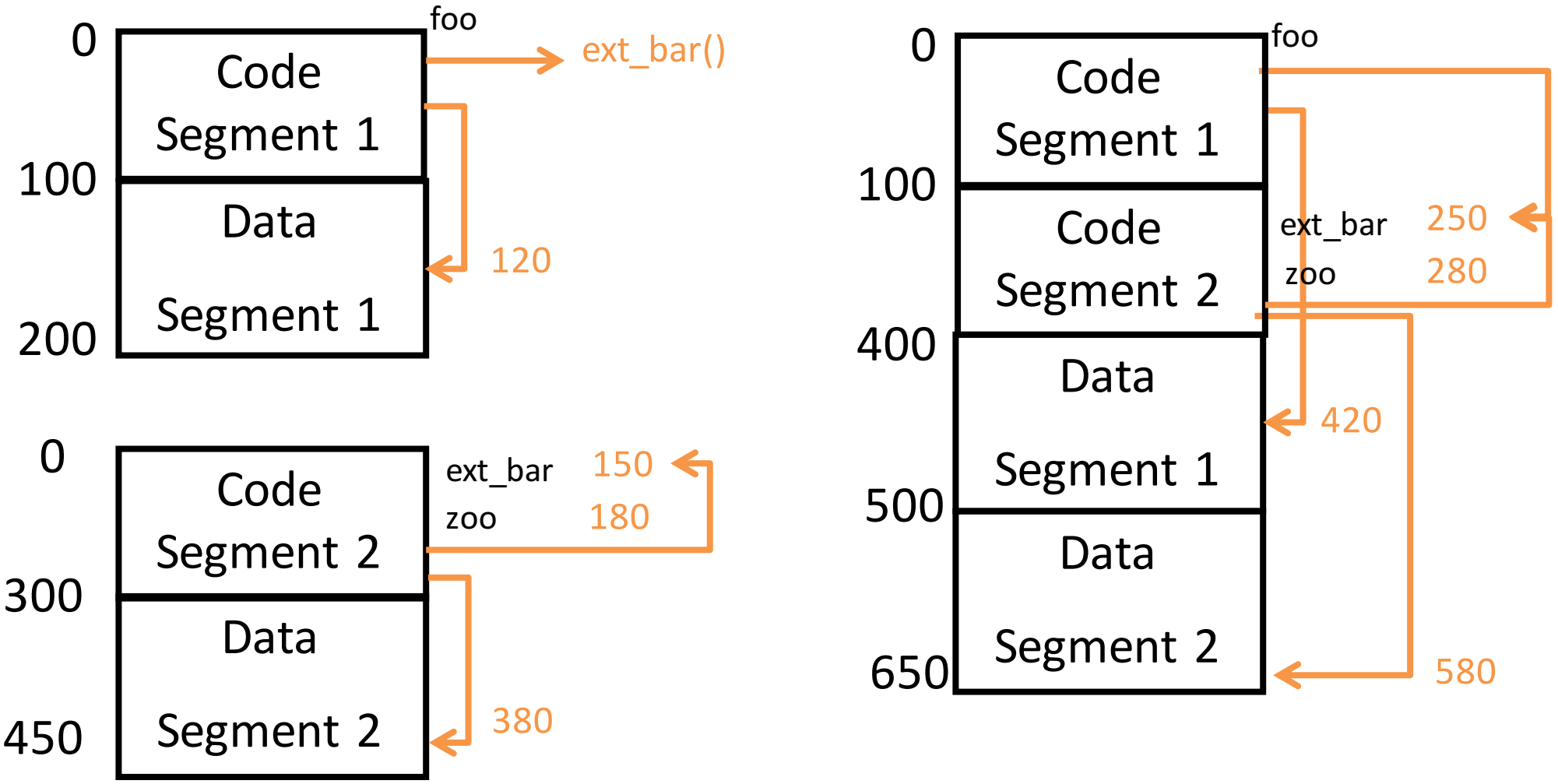


- Data code segment start at address 0

Linker

- Merges object files to an executable
  - Enables separate compilation
- Combine memory layouts of object modules
  - Recall: the data/code segments of every object file starts at address 0
  - Links program calls to library routines
    - printf(), malloc()
  - Relocates instructions by adjusting absolute references
  - Resolves references among files

Linker


0
Code
Segment 1
100
Data
Segment 1
200
foo
ext_bar()
120
0
Code
Segment 2
300
Data
Segment 2
450
ext_bar
150
zoo
180
380
0
Code
Segment 1
100
Code
Segment 2
400
Data
Segment 1
500
Data
Segment 2
650
foo
ext_bar
250
zoo
280
420
580

Relocation information

- Information needed to change addresses

- Positions in the code which contains addresses
  - Data
  - Code

- Two implementations
  - Bitmap
  - Linked-lists

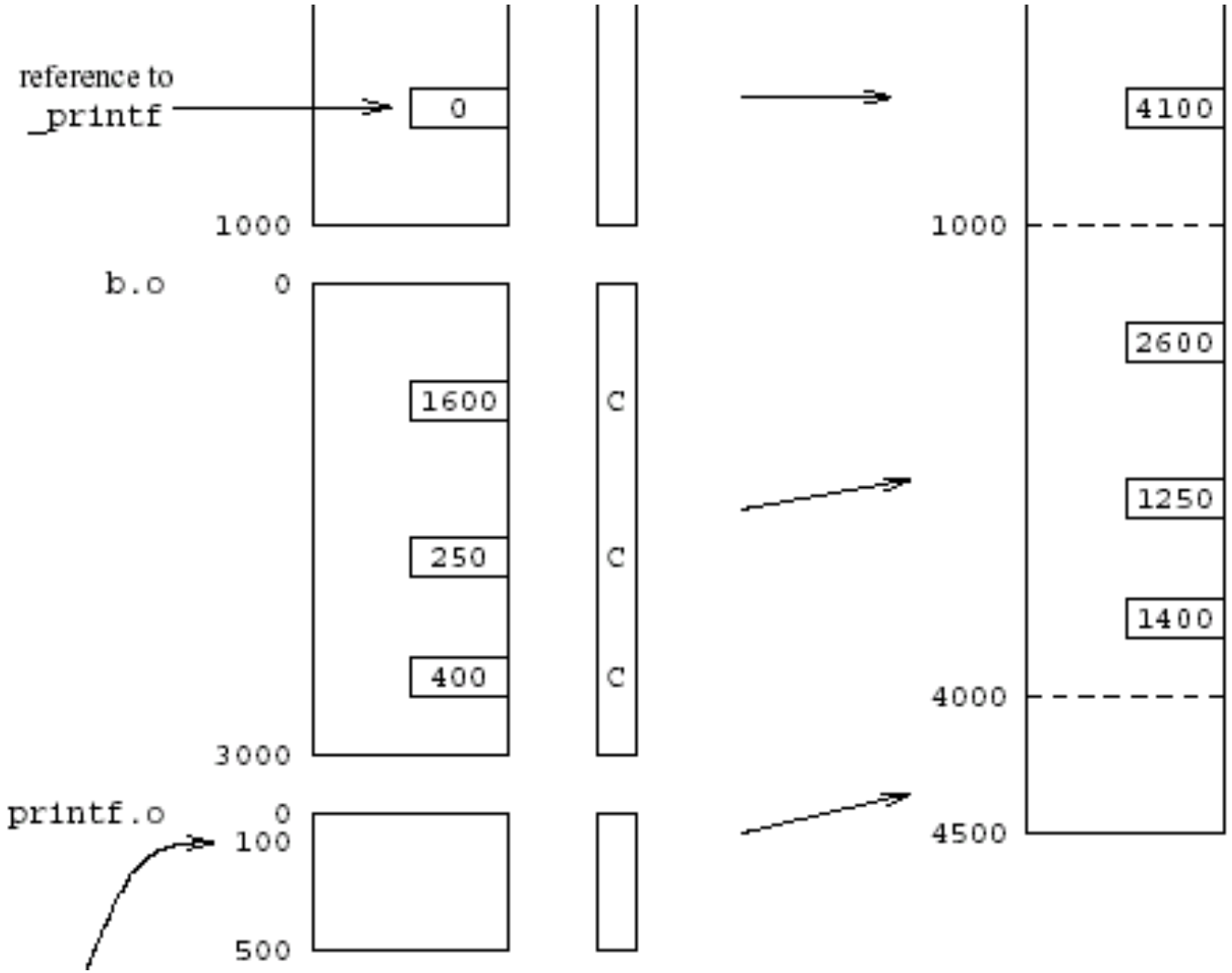
External References

- The code may include references to external names (identifiers)
  - Library calls
  - External data
- Stored in external symbol table

Example of External Symbol Table

External symbol	Type	Address
_options	entry point	50 data
__main	entry point	100 code
_printf	reference	500 code
_atoi	reference	600 code
_printf	reference	650 code
_exit	reference	700 code
_msg_list	entry point	300 data
_Out_Of_Memory	entry point	800 code
_fprintf	reference	900 code
_exit	reference	950 code
_file_list	reference	4 data

Example


reference to
_printf
0
1000
b.o
0
1600
250
400
3000
C
C
C
printf.o
0
100
500
4100
1000
2600
1250
1400
4000
4500

Linker (Summary)

- Merge several object files
  - Resolve external references
  - Relocate addresses
- User mode
- Provided by the operating system
  - But can be specific for the compiler
    - More secure code
    - Better error diagnosis

Static Linking (Recap)

- Assembler generates binary code
  - Unresolved addresses
  - Relocatable addresses
- Linker generates executable code
- Loader generates runtime states (images)

Dynamic Linking

- Why dynamic linking?
  - Shared libraries
    - Save space
    - Consistency
  - Dynamic loading
    - Load on demand

What's the challenge?

Source program

Compiler

Assembly lang. program (.s)

Assembler

Machine lang. Module (.o): program (+library) modules

Linker

"compilation" time

Executable (".exe"):

"execution" time

Loader

Image (in memory):

Libraries (.o)
(dynamic linking)

Position-Independent Code (PIC)

- Code which does not need to be changed regardless of the address in which it is loaded
  - Enable loading the same object file at different addresses
    - Thus, shared libraries and dynamic loading

- "Good" instructions for PIC: use relative addresses
  - relative jumps
  - reference to activation records

- "Bad" instructions for : use fixed addresses
  - Accessing global and static data
  - Procedure calls
    - Where are the library procedures located?

How?

“All problems in computer science can be solved by another level of indirection"

Butler Lampson

PIC: The Main Idea

- Keep the global data in a table
- Refer to all data relative to the designated register

PIC: The Main Idea

- Keep the global data in a table
- Refer to all data relative to the designated register

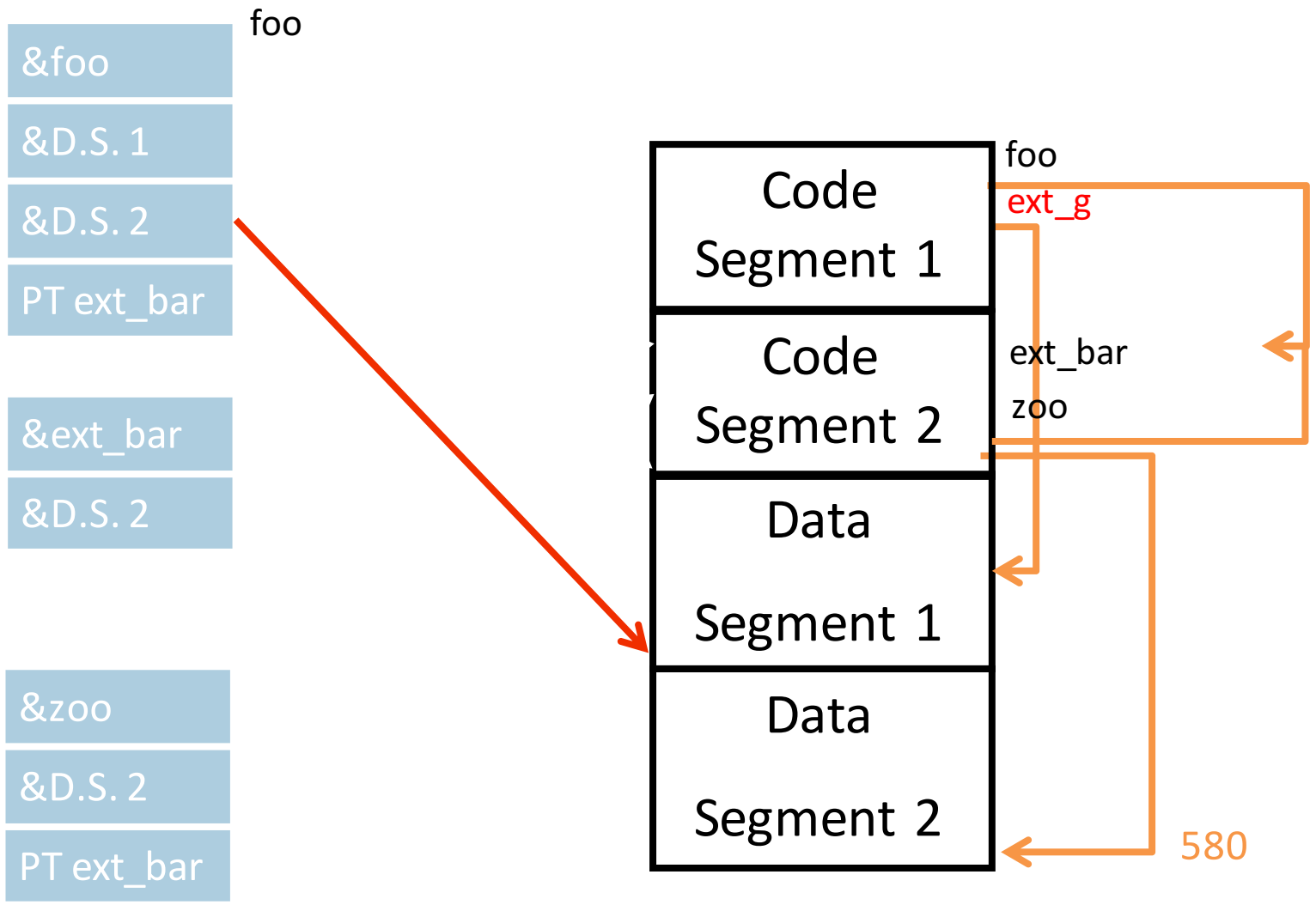
- One more indirection

Per-Routine Pointer Table

- Record for every routine in a table
- The record contains
  - Pointer to the procedure code
  - Pointers to the global data the routine uses
  - Pointer to the routine tables of immediate callees

Per-Routine Pointer Table

- Record for every routine in a table



Per-Routine Pointer Table

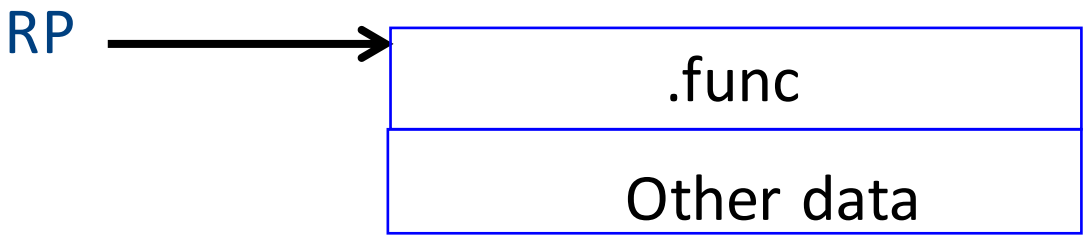
- Record for every routine in a table
- Record used as a address to procedure

Caller:

1. Load Pointer table address into RP
2. Load Code address from 0(RP) into RC
3. Call via RC

Callee:

1. RP points to pointer table
2. Table has addresses of pointer table for sub-procedures



PIC: The Main Idea

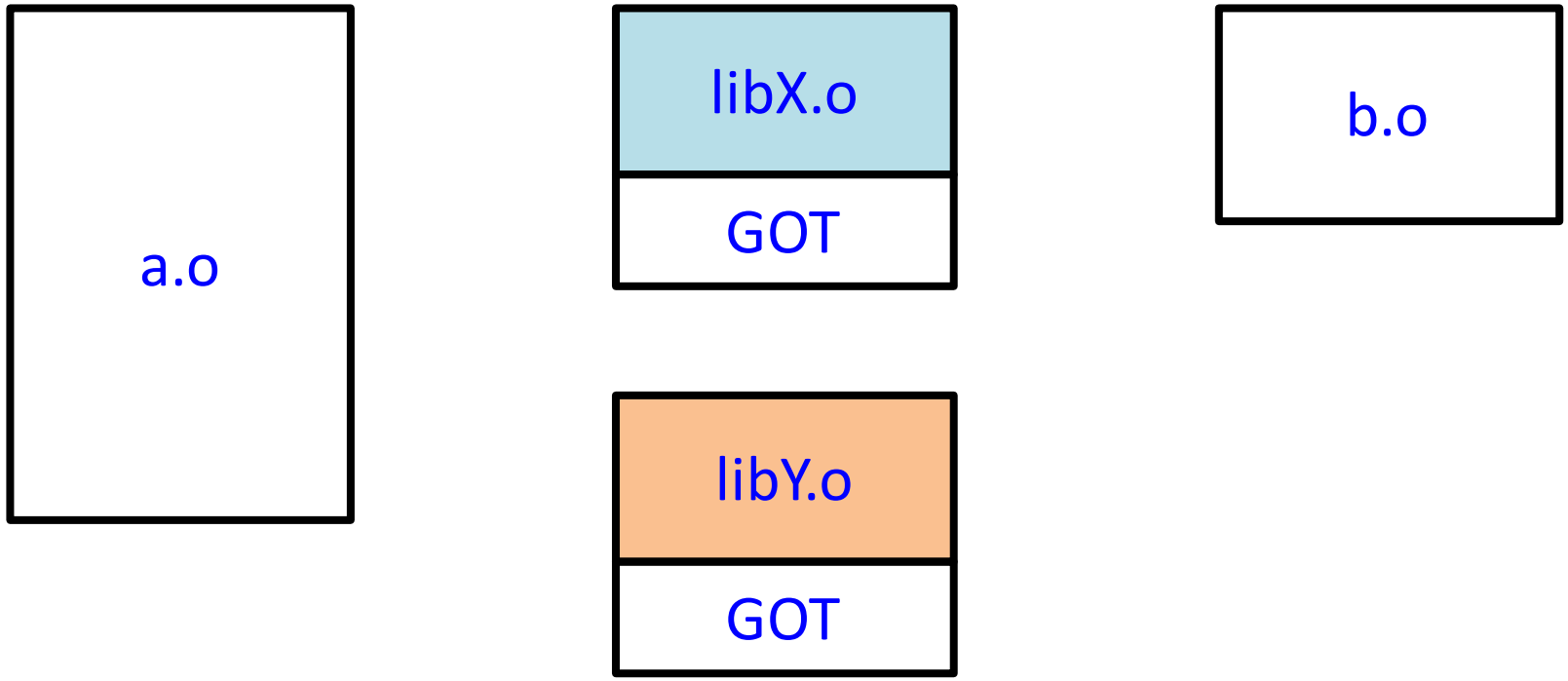
- Keep the global data in a table
- Refer to all data relative to the designated register

- Efficiency: use a register to point to the beginning of the table
  - Troublesome in CISC machines

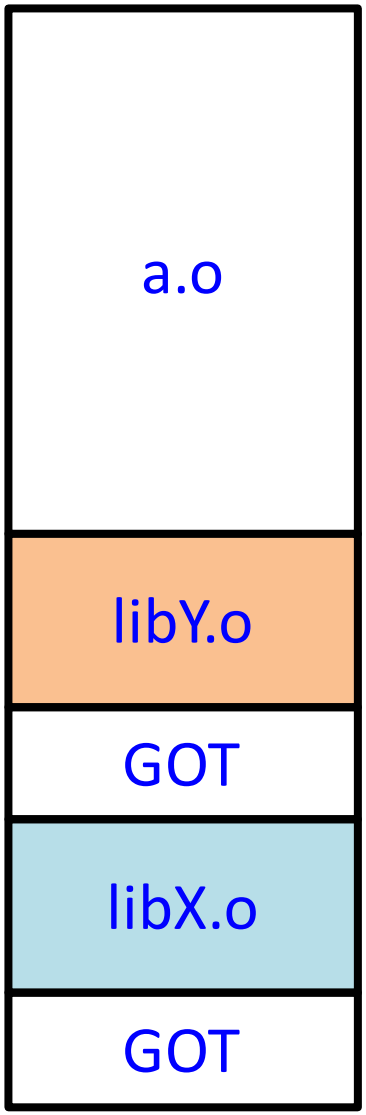
ELF: Executable and Linkable Format

- **E**xecutable and **L**inkable code **F**ormat
  - Introduced in Unix System V
- Types of ELF files
  - A **relocatable file** holds code and data suitable for linking with other object files to create an executable or a shared object file.
  - An **executable file** holds a program suitable for execution; the file specifies how exec(BA_OS) creates a program's process image.
  - A **shared object file** holds code and data suitable for linking in two contexts.
    - The link editor may process it with other relocatable and shared object files to create another object file.
    - The dynamic linker combines it with an executable file and other shared objects to create a process image.

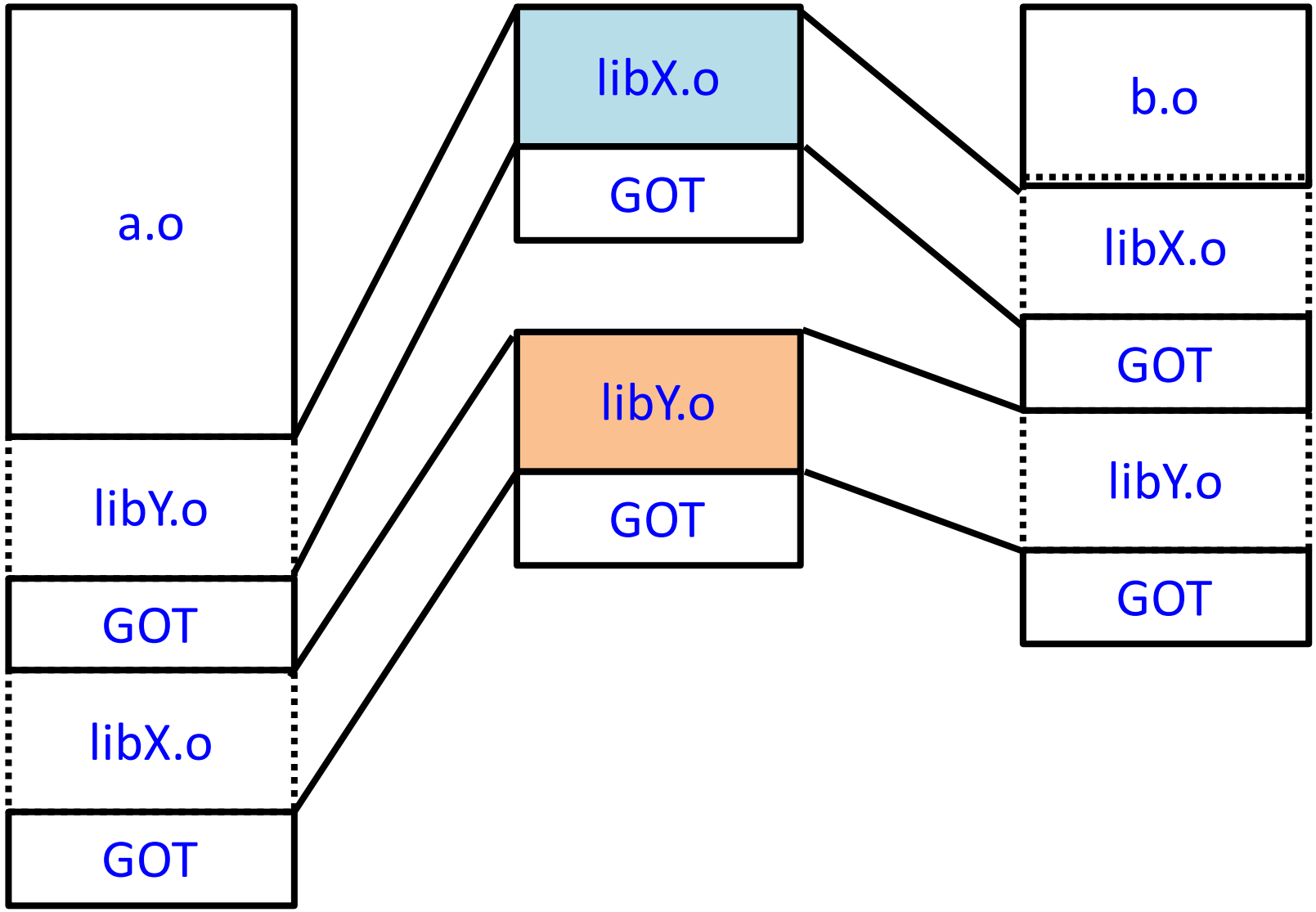
Sharing code



Sharing code

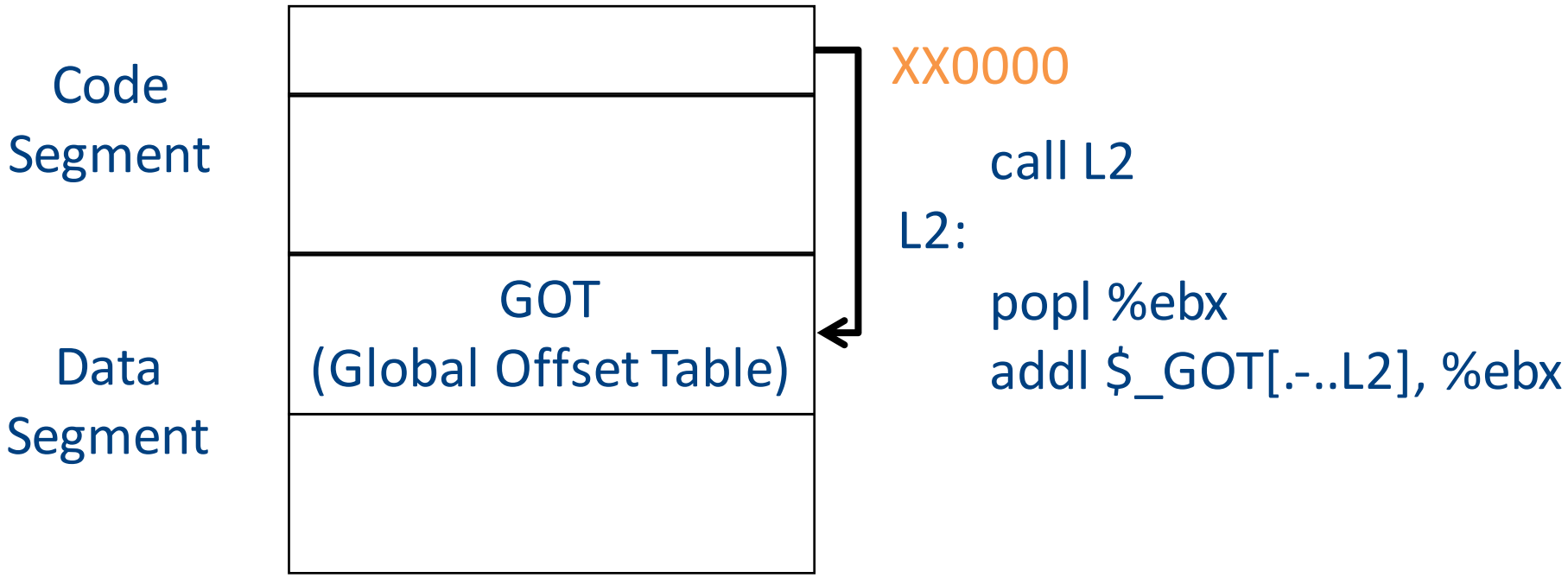


Sharing code



ELF-Position Independent Code

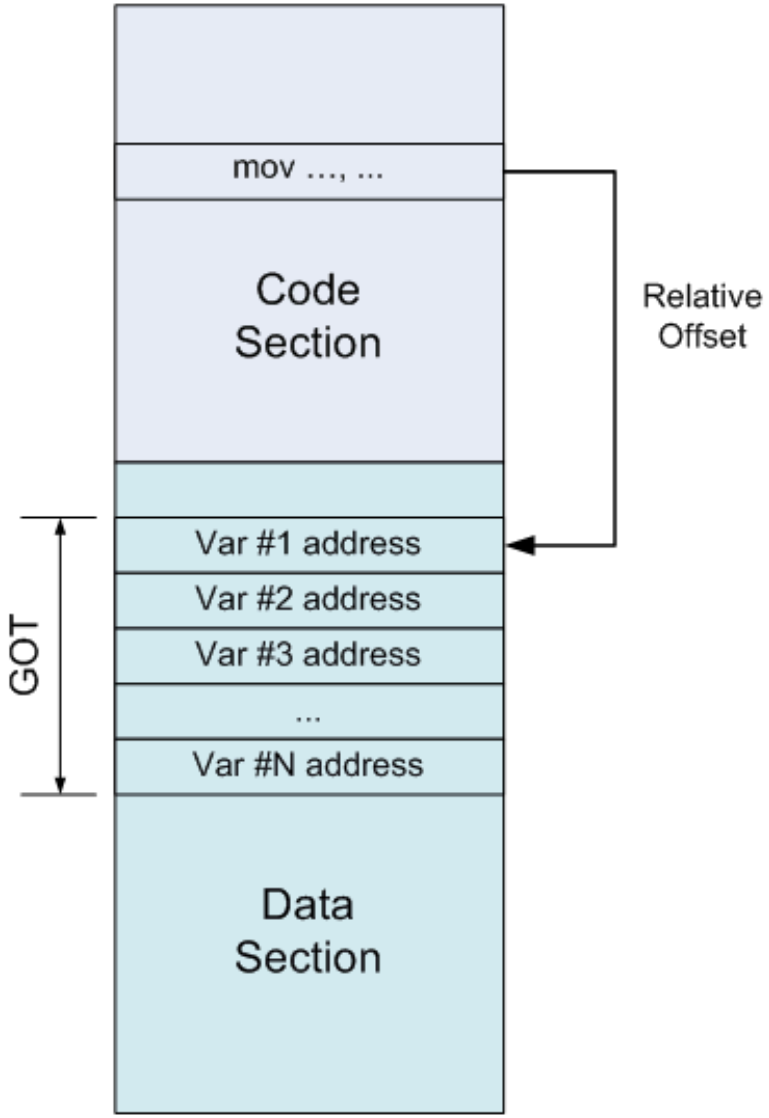
- Observation
  - Executable consists of code followed by data
  - The offset of the data from the beginning of the code is known at compile-time
  - (Every shared library has its own GOT)



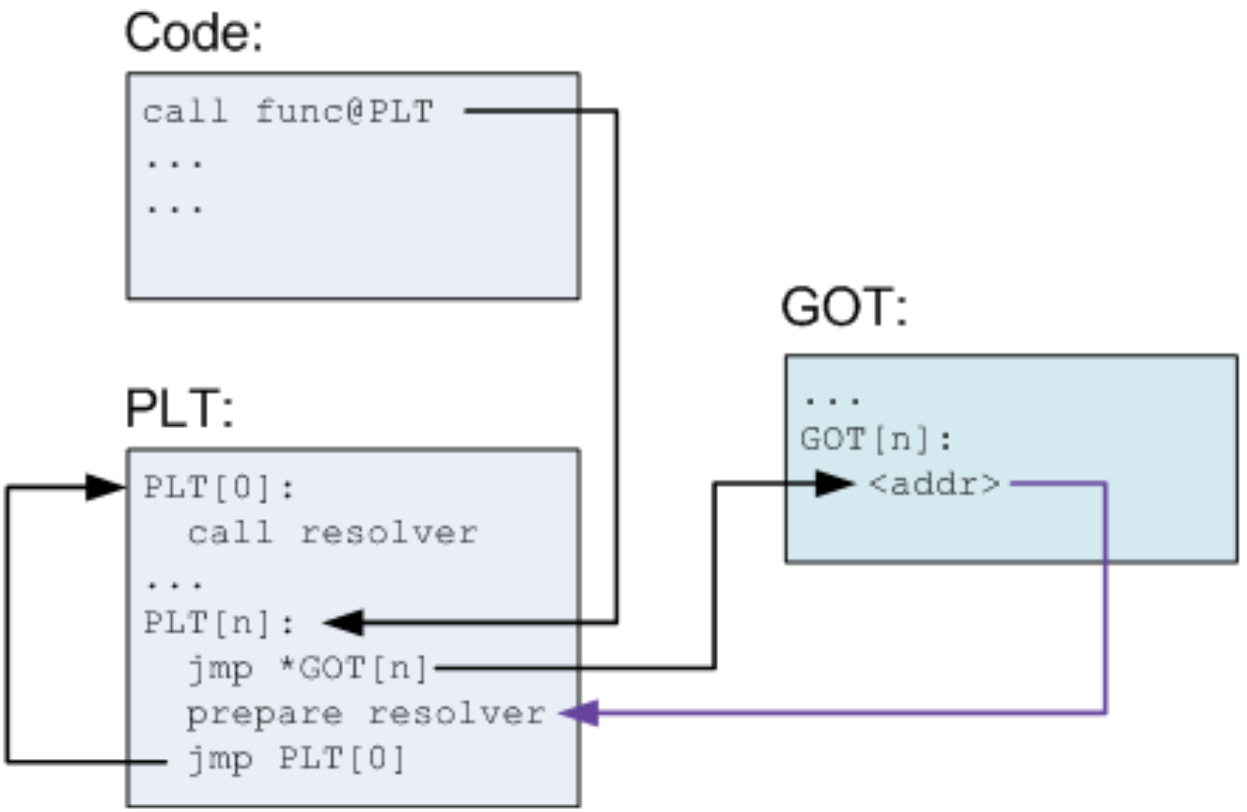
ELF: Accessing global data

Accessing global data defined is done by

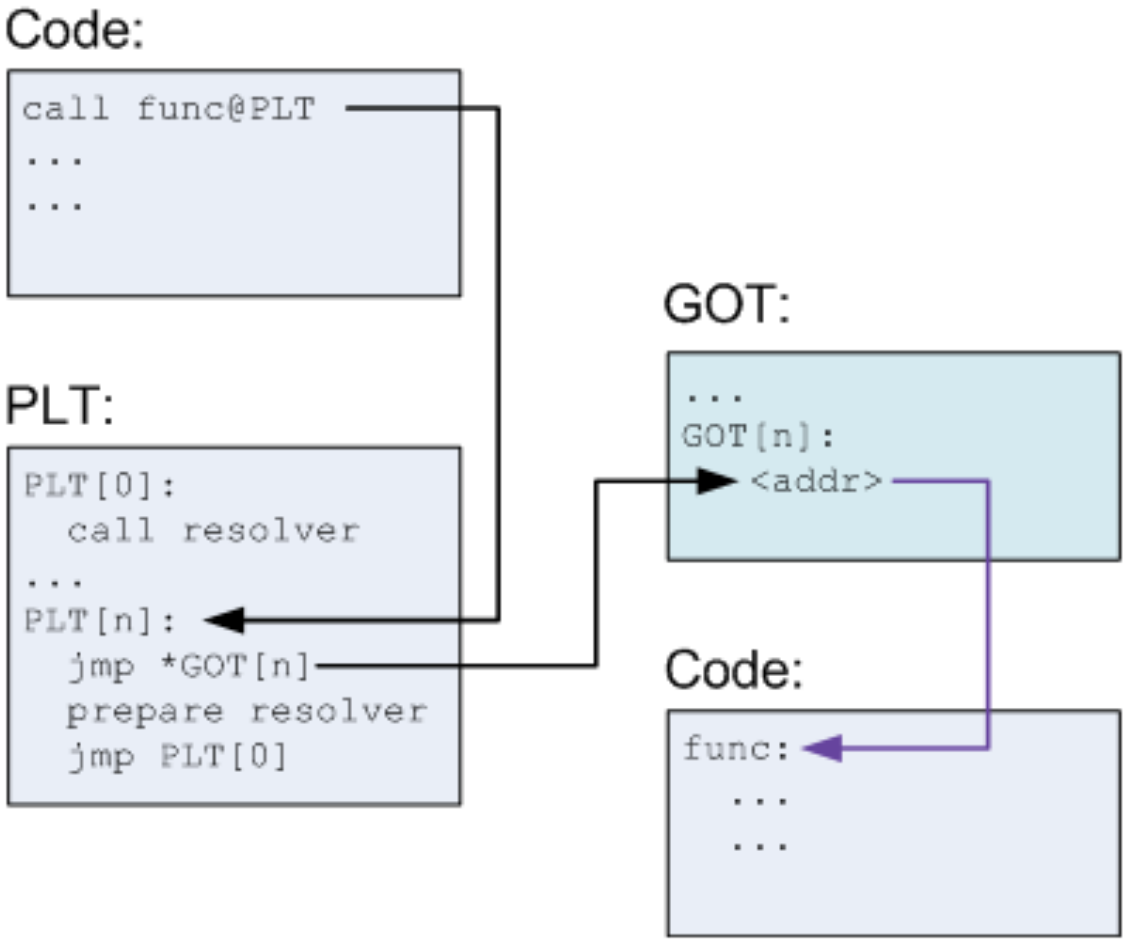
- finding its entry in the GOT table
- Dereferencing that entry



ELF: Calling Procedures
(before 1st call)



ELF: Calling Procedures
(after 1st call)



PIC benefits and costs

- Enable loading w/o relocation
- Share memory locations among processes

- Data segment may need to be reloaded
- GOT can be large
- More runtime overhead
- More space overhead

Shared Libraries

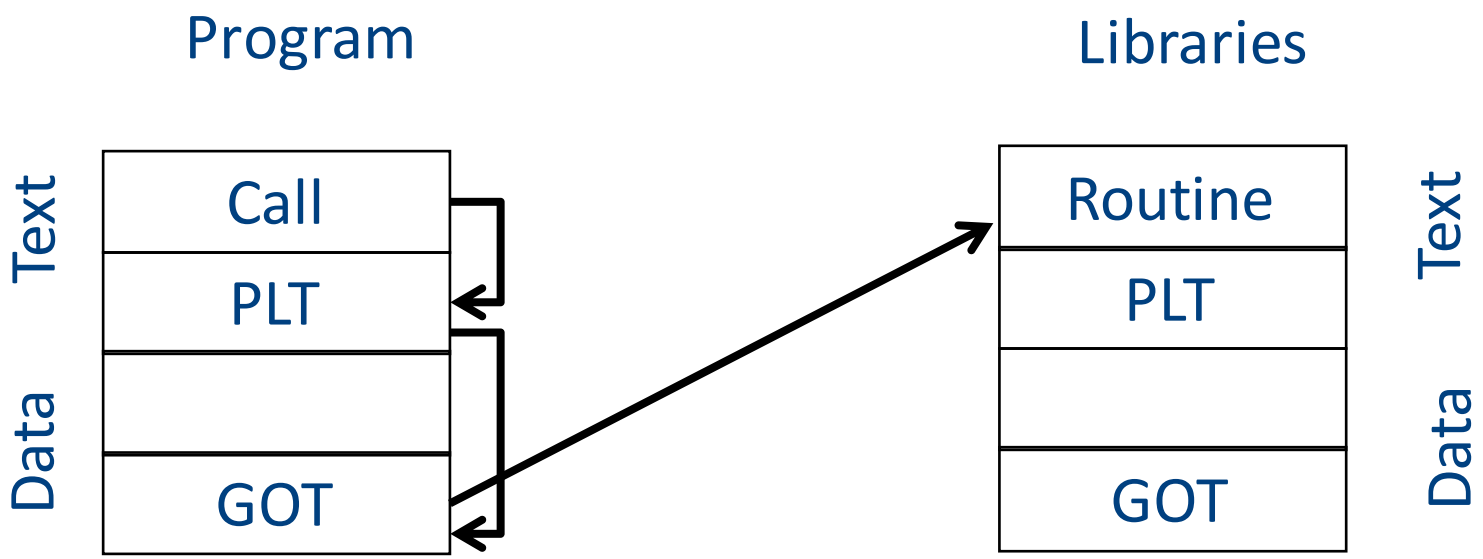
- Heavily used libraries
- Significant code space
  - 5-10 Mega for print
  - Significant disk space
  - Significant memory space
- Can be saved by sharing the same code
- Enforce consistency
- But introduces some overhead

- Can be implemented either with static or dynamic loading

Shared Libraries

- Heavily used libraries
- Significant code space
  - 5-10 Mega for print
  - Significant disk space
  - Significant memory space
- Can be saved by sharing the same code
- Enforce consistency
- But introduces some overhead

Content of ELF file



Consistency

- How to guarantee that the code/library used the “right” library version

Loading Dynamically Linked Programs

- Start the dynamic linker
- Find the libraries
- Initialization
  - Resolve symbols
  - GOT
    - Typically small
  - Library specific initialization
- Lazy procedure linkage

Microsoft Dynamic Libraries (DLL)

- Similar to ELF
- Somewhat simpler
- Require compiler support to address dynamic libraries
- Programs and DLL are Portable Executable (PE)
- Each application has it own address
- Supports lazy bindings

Dynamic Linking Approaches

- Unix/ELF uses a single name space space and MS/PE uses several name spaces
- ELF executable lists the names of symbols and libraries it needs
- PE file lists the symbols it needs and the libraries they come from
- ELF is more flexible: Executables determine the linker to use
- PE is more efficient: OS in charge of linking

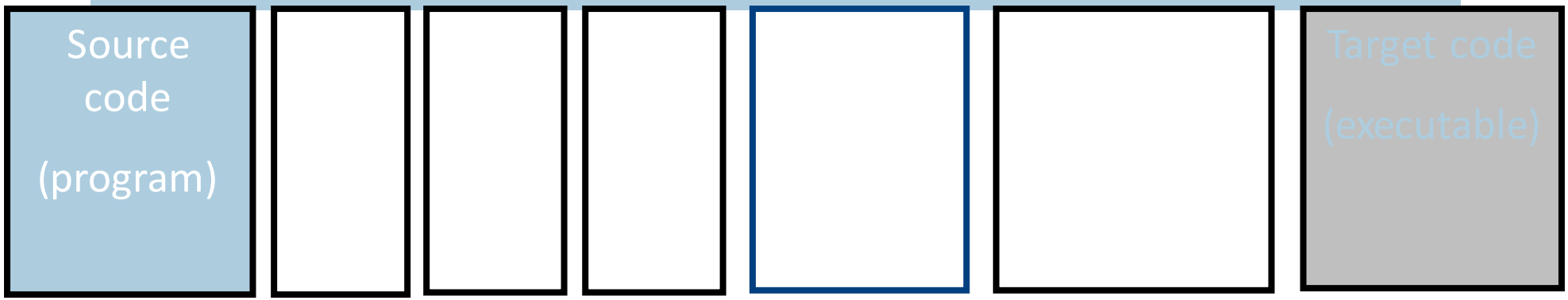
Costs of dynamic loading

- Load time relocation of libraries
- Load time resolution of libraries and executable
- Overhead from PIC prolog
- Overhead from indirect addressing
- Reserved registers

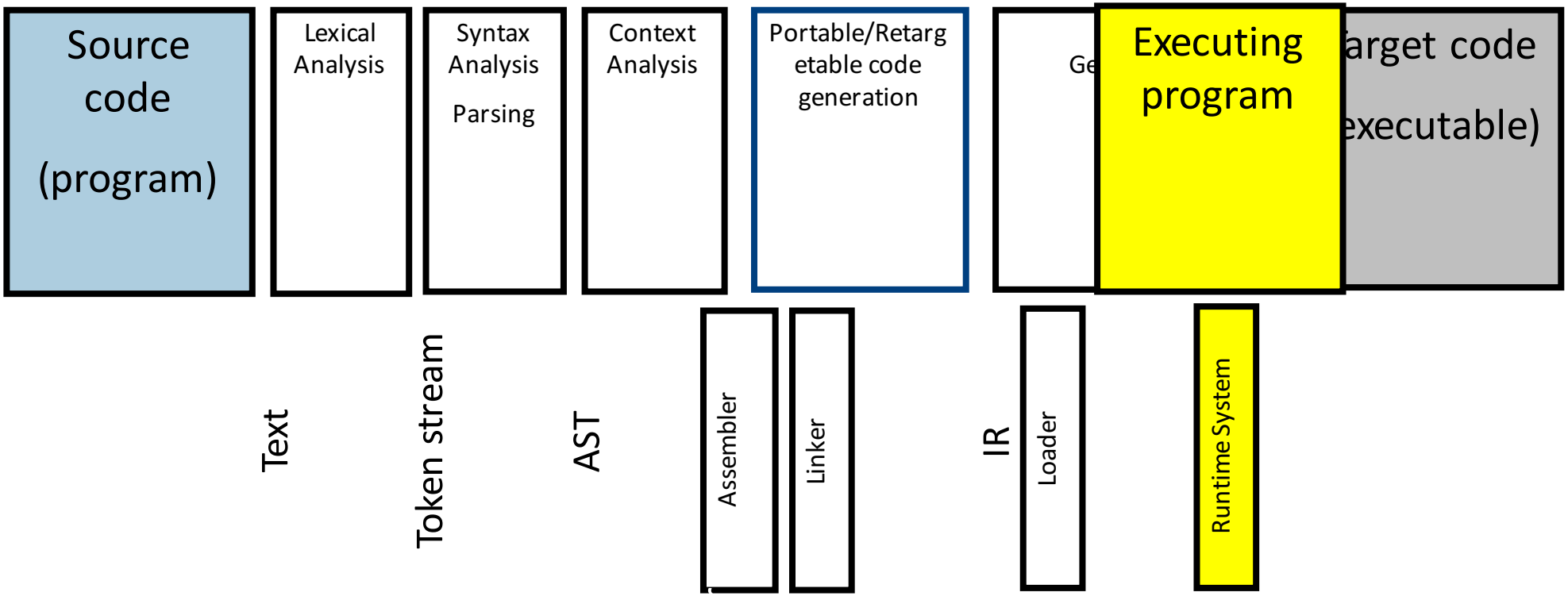
Summary

- Code generation yields code which is still far from executable
  - Delegate to existing assembler
- Assembler translates symbolic instructions into binary and creates relocation bits
- Linker creates executable from several files produced by the assembly
- Loader creates an image from executable

Stages of compilation



Compilation ➔ Execution



Runtime Environment

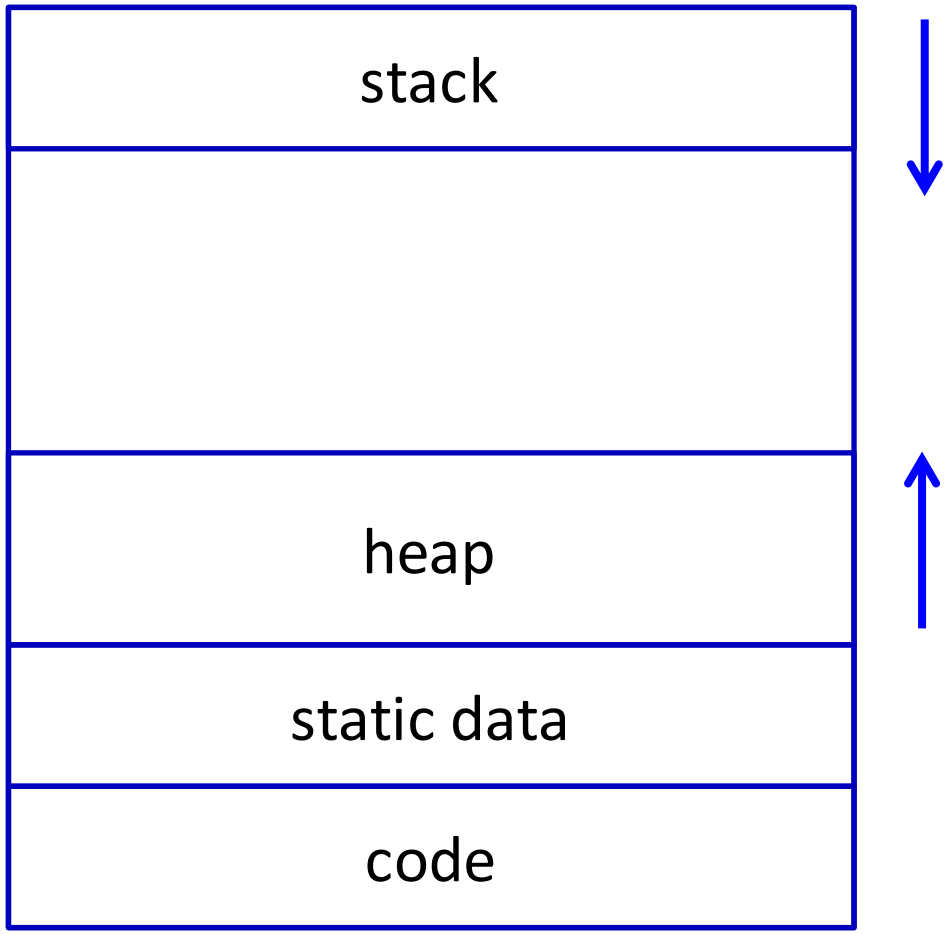
- Mediates between the OS and the programming language
- Hides details of the machine from the programmer
  - Ranges from simple support functions all the way to a full-fledged virtual machine
- Handles common tasks
  - Runtime stack (activation records)
  - Dynamic optimization
  - Debugging
  - …

Where do we allocate data?

- Activation records
  - Lifetime of allocated data limited by procedure lifetime
  - Stack frame deallocated (popped) when procedure return

- **Dynamic memory allocation on the heap**

Memory Layout



Alignment

- Typically, can only access memory at aligned addresses
  - Either 4-bytes or 8-bytes
- What happens if you allocate data of size 5 bytes?
  - **Padding** – the space until the next aligned addresses is kept empty
- (side note: x86, is more complicated, as usual, and also allows unaligned accesses, but not recommended)

Allocating memory

- In C - malloc
- `void *malloc(size_t size)`

- Why does malloc return void* ?
  - It just allocates a chunk of memory, without regard to its type
- How does malloc guarantee alignment?
  - After all, you don't know what type it is allocating for
  - It has to align for the largest primitive type
  - In practice optimized for 8 byte alignment (glibc-2.17)

Memory Management

- Manual memory management
- Automatic memory management

Manual memory management

- malloc
- free

```
a =  malloc(…) ;
// do something with a
free(a);
```

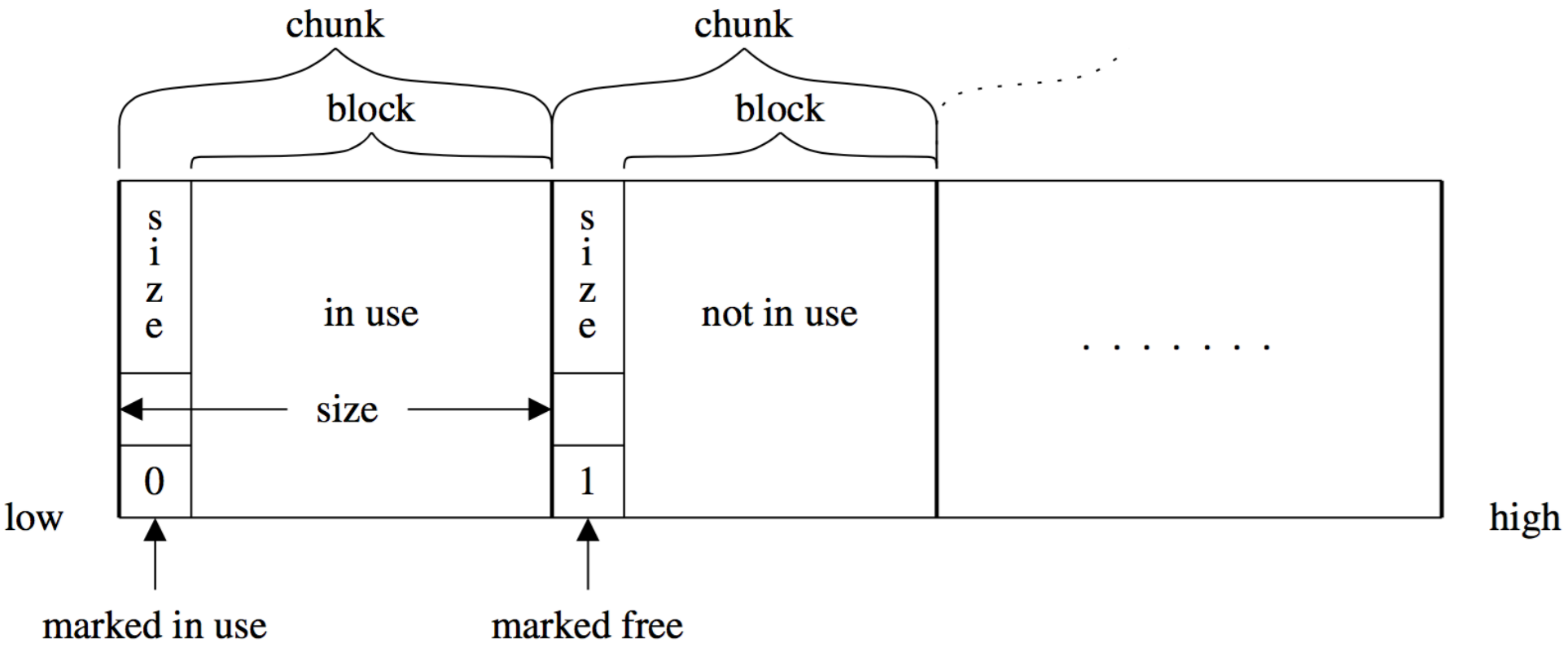
malloc

- where is malloc implemented?
- how does it work?

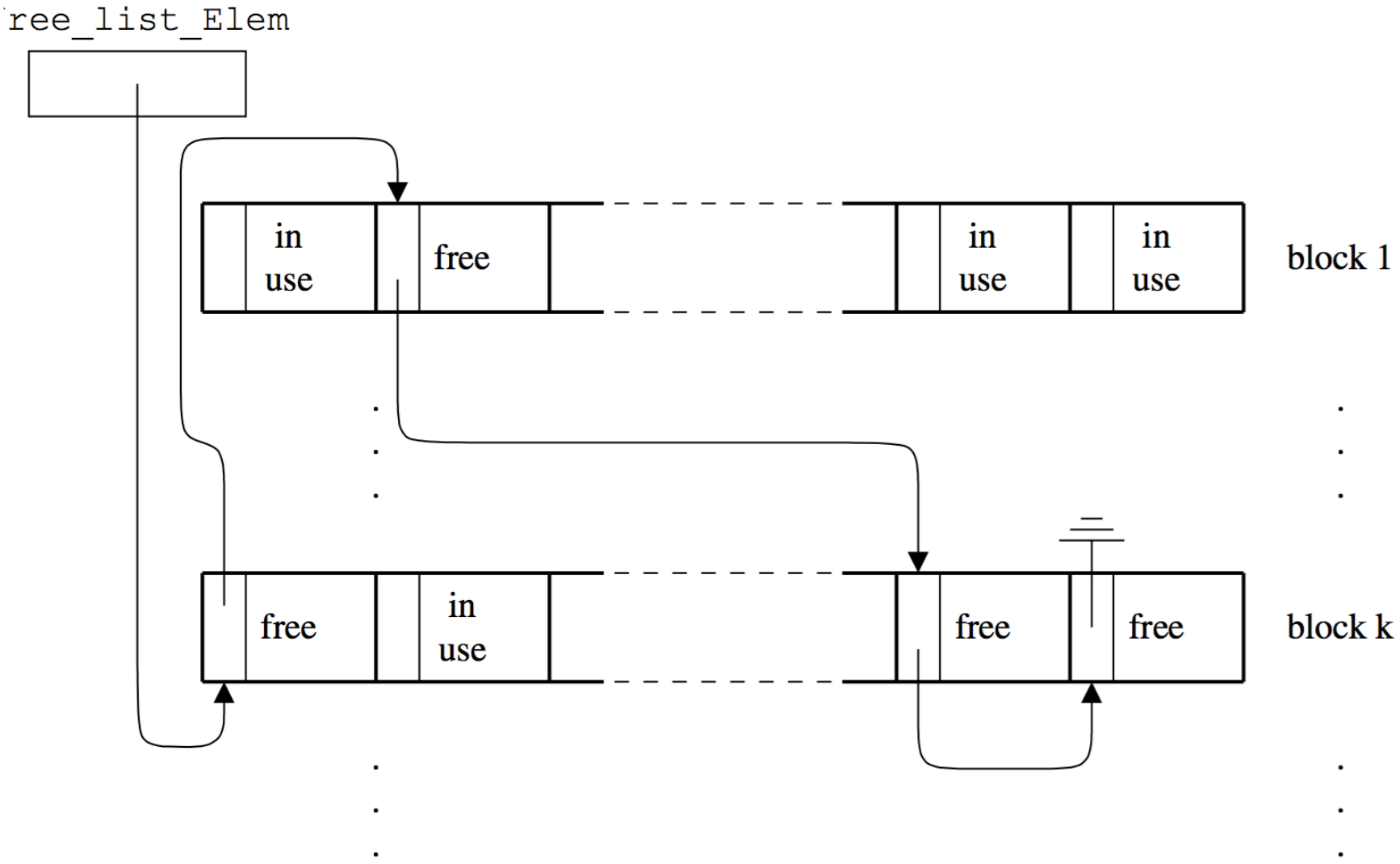
Free-list Allocation

- A data structure records the location and size of free cells of memory.
- The allocator considers each free cell in turn, and according to some policy, chooses one to allocate.
- Three basic types of free-list allocation:
  - First-fit
  - Next-fit
  - Best-fit

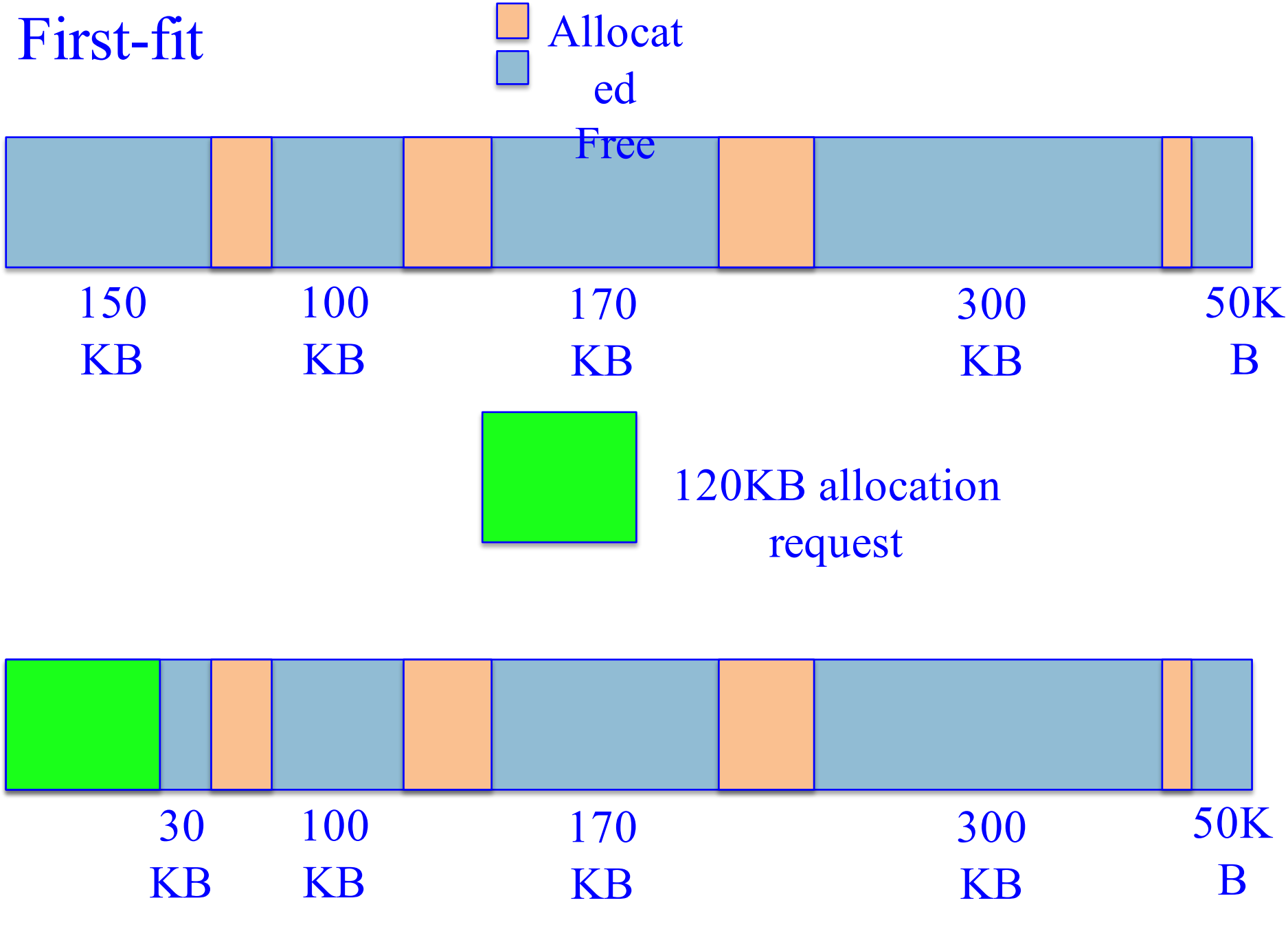
Memory chunks

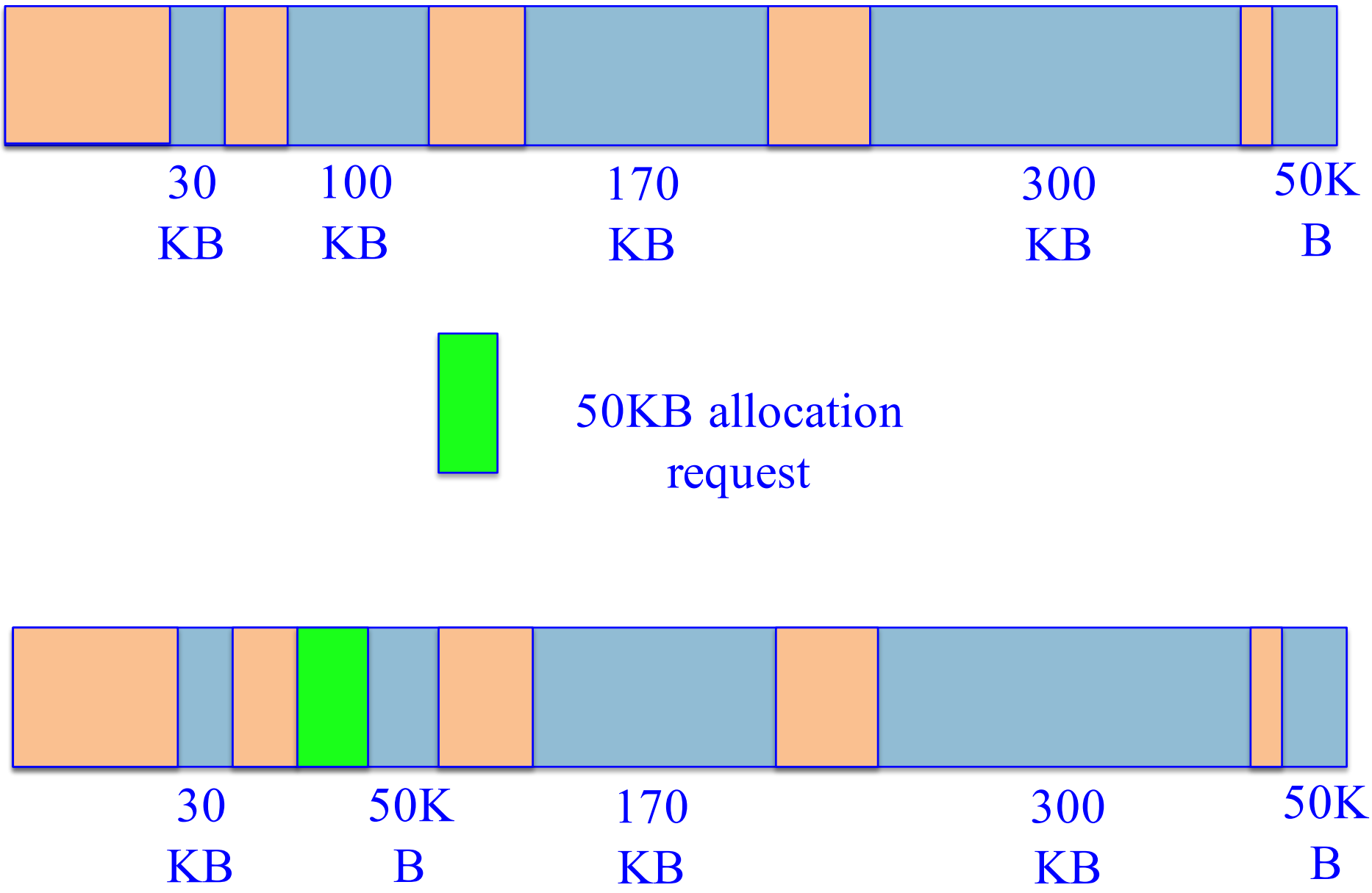


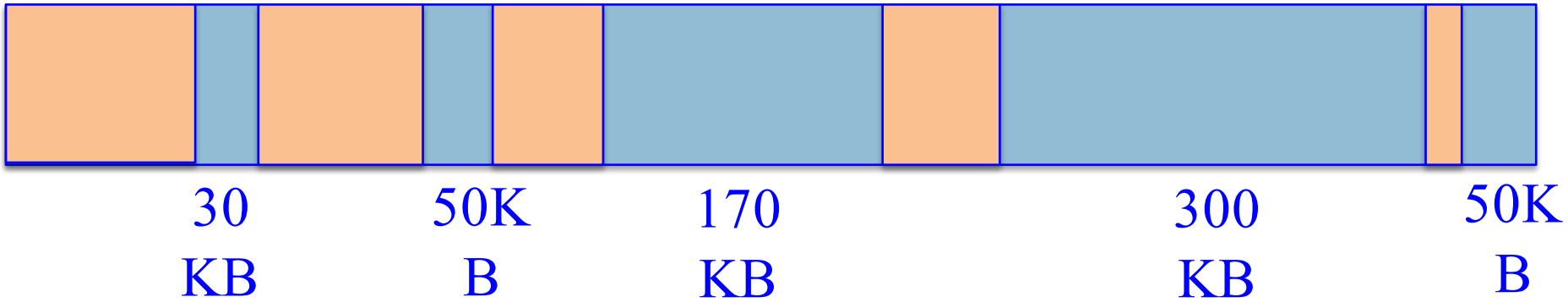
Free list


ree_list_Elem
in use
free
in use
in use
block 1
free
in use
free
free
block k

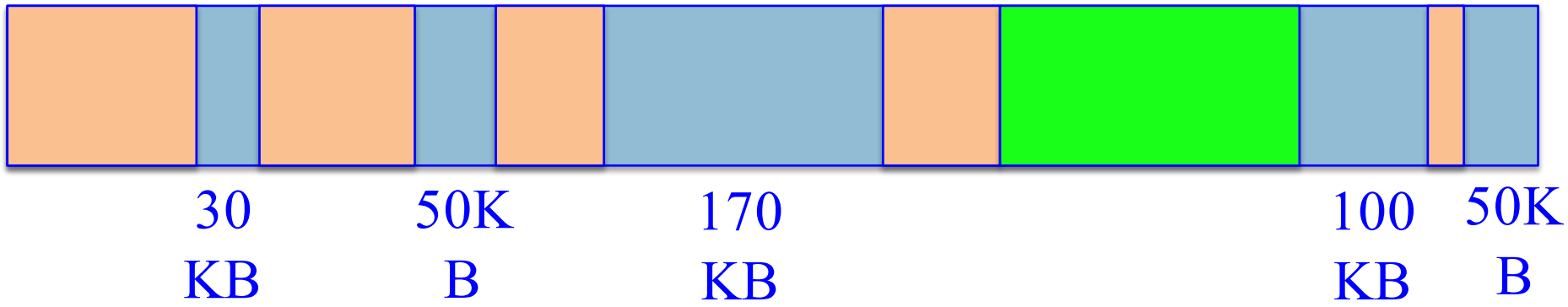
First-fit


Allocated
Free
150 KB
100 KB
170 KB
300 KB
50KB
120KB allocation request
30 KB
100 KB
170 KB
300 KB
50KB


30
KB
100
KB
170
KB
300
KB
50K
B
50KB allocation
request
30
KB
50K
B
170
KB
300
KB
50K
B



200KB allocation
request



Fragmentation

- Dispersal of free memory across a possibly large number of small free cells.
- Negative effects:
  - Can prevent allocation from succeeding
  - May cause a program to use more address space, more resident pages and more cache lines.
- Fragmentation is impractical to avoid:
  - Usually the allocator cannot know what the future request sequence will be.
  - Even given a known request sequence, doing an optimal allocation is NP-hard.
- Usually There is a trade-off between allocation speed and fragmentation.

Segregated-fits Allocation

- Idea – use multiple free-list whose members are segregated by size in order to speed allocation.
- Usually a fixed number k of size values $s_0 < s_1 < \ldots < s_{k-1}$
- $k+1$ free lists $f_0,\ldots,f_k$
- For a free cell, b, on list f_i,

$$\text{size}(b) = s_i \;\; \forall\, 1 \le i \le k-1$$

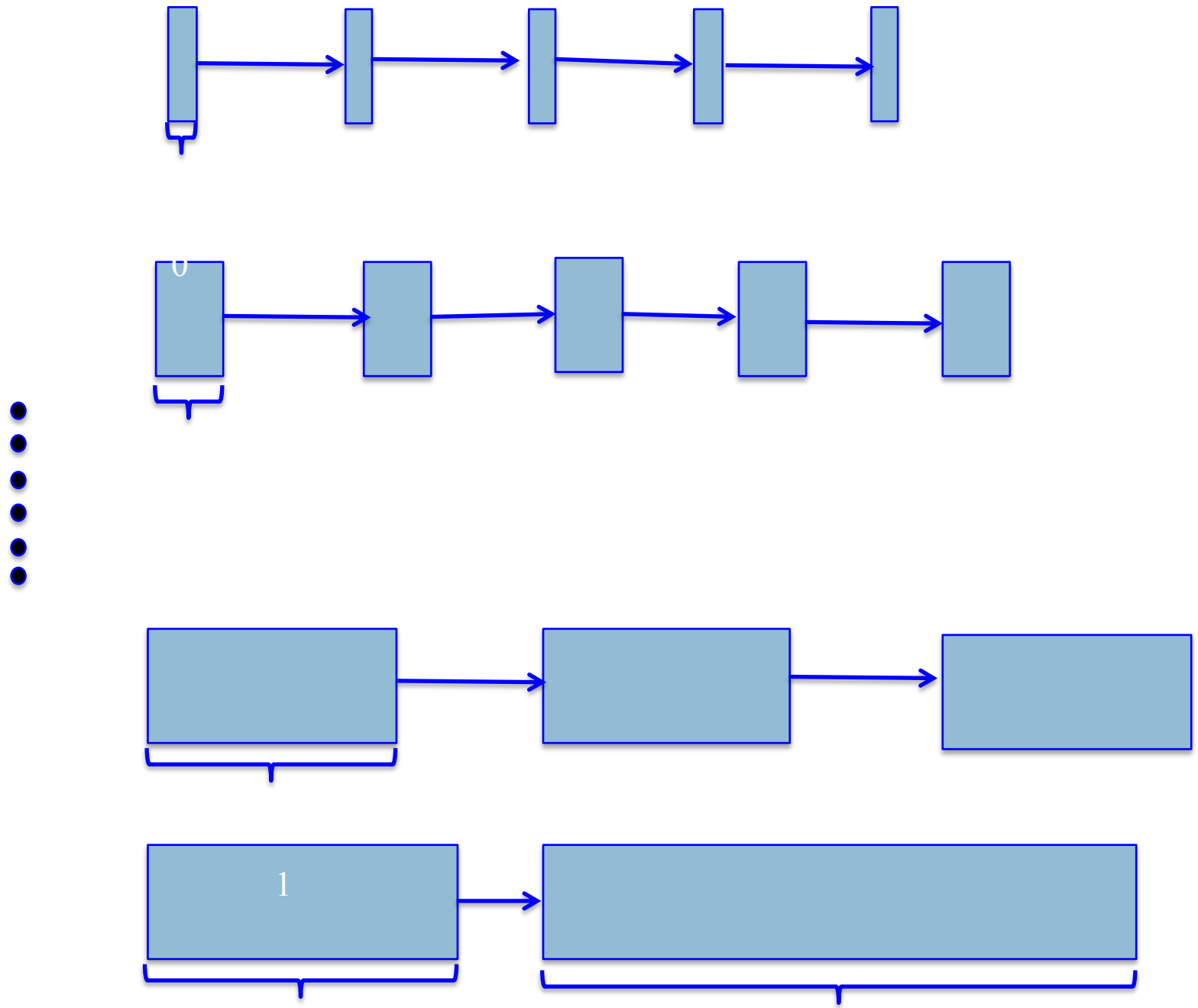
$$\text{size}(b) > s_{k-1} \text{ if } i=k$$

- When requesting a cell of size $b \le s_{k-1}$, the allocator rounds the request size up to the smallest s_i such that $b \le s_i$.
- S_i is called a **size class**

Segregated-fits Allocation

```
SegregatedFitAllocate(j):
        result ← remove(freeLists[j])
        if result = null
                large ← allocateBlock()
                if large = null
                        return null
                initialize(large, sizes[j])
                result ← remove(freeList[j])
        return result
```

- List f_k, for cells larger than s_k, is organized to use one of the basic single-list algorithm.
- Per-cell overheads for large cell are a bit higher but in total it is negligible.
- The main advantage: for size classes other than s_k, allocation typically requires constant time.


0
1

Runtime support for MM

- C's standard library provides basic memory management
  - Gets memory pages from the OS
  - Maintains inventory of free memory cells
    - mmap(), brk()

free

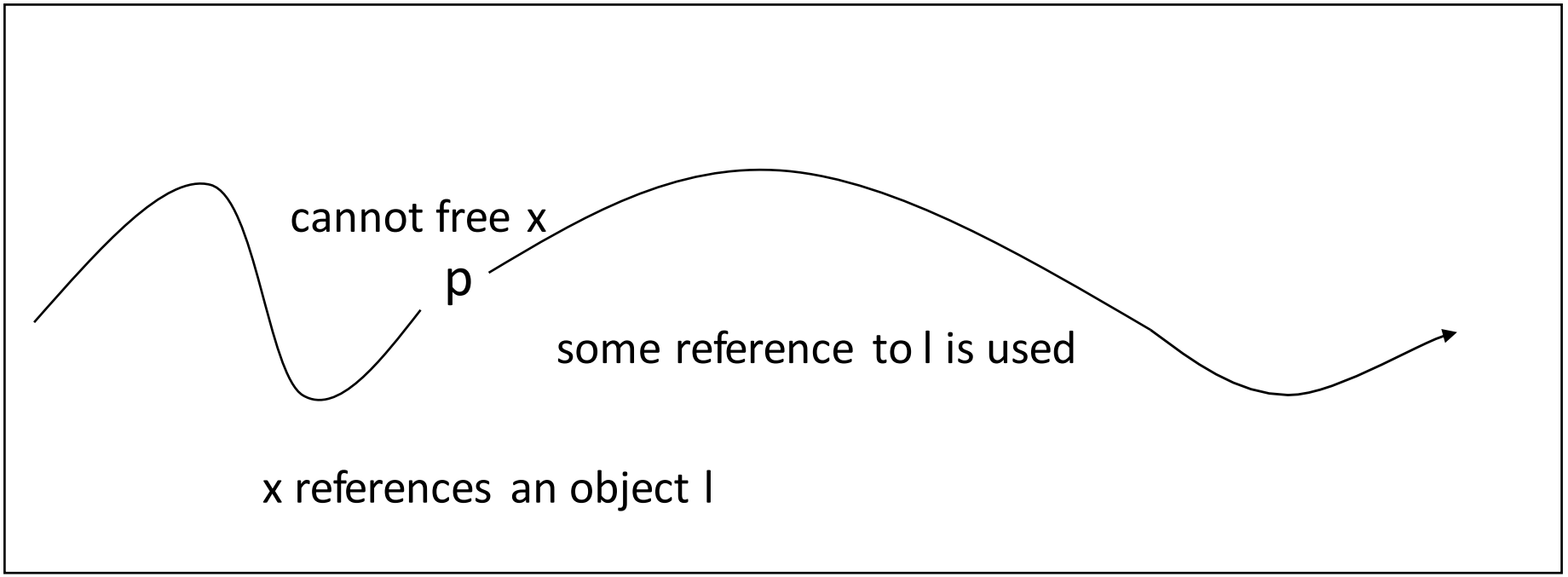
- Free too late – waste memory (memory leak)
- Free too early – dangling pointers / crashes
- Free twice – error

When can we free an object?

```
a =  malloc(…) ;
b = a;
// free (a); ?
c = malloc (…);
if  (b == c)
     printf("unexpected equality");
```

Cannot free an object if it has a reference with a future use!

When can free x be inserted after p?



On all execution paths after p there are no uses of references to the object referenced by x ➔ inserting free x after p is valid

Automatic Memory Management

- automatically free memory when it is no longer needed
- not limited to OO languages
- prevalent in OO languages such as Java
  - also in functional languages

Garbage collection

- approximate reasoning about object liveness
- use reachability to approximate liveness
- **assume reachable objects are live**
  - non-reachable objects are dead

Garbage Collection – Classical Techniques

- reference counting
- mark and sweep
- copying

GC using Reference Counting

- add a reference-count field to every object
  - how many references point to it
- when (rc==0) the object is non reachable
  - non reachable => dead
  - can be collected (deallocated)

Managing Reference Counts

- Each object has a reference count o.RC
- A newly allocated object o gets o.RC = 1
  - why?

- write-barrier for reference updates

```
update(x,old,new) {
   old.RC--;
   new.RC++;
   if (old.RC == 0) collect(old);
}
```

- collect(old) will decrement RC for all children and recursively collect objects whose RC reached 0.

Cycles!

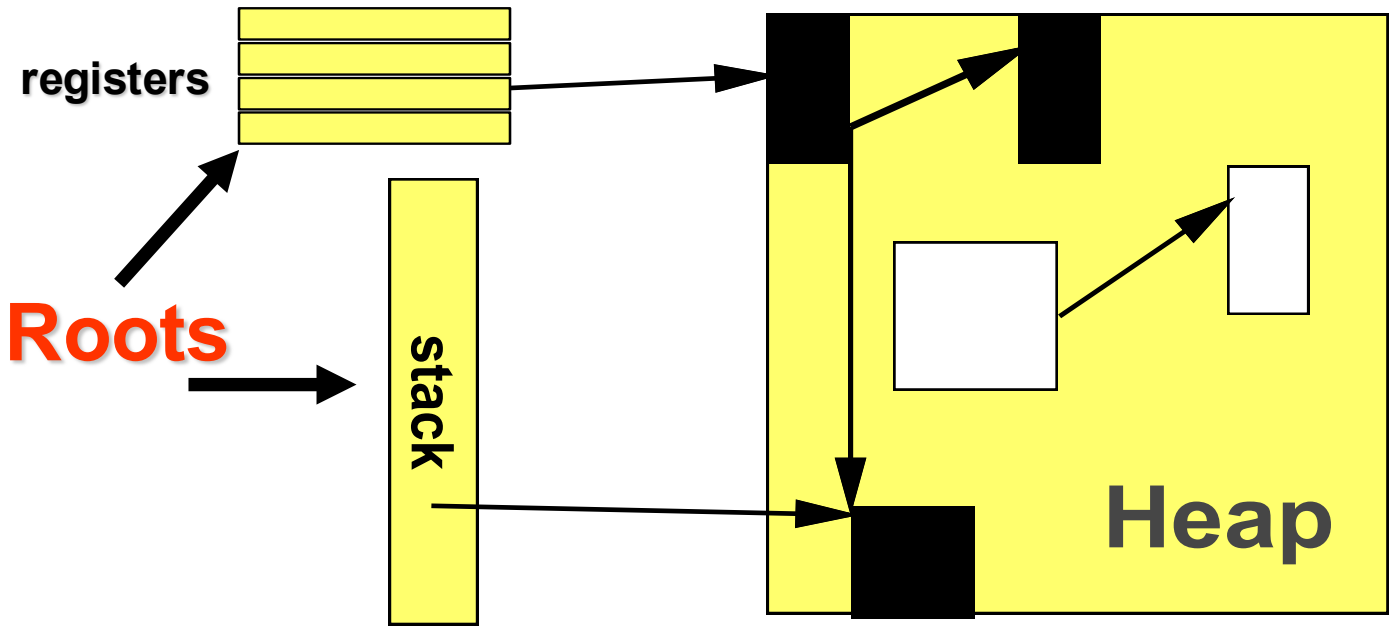
- cannot identify non-reachable cycles
  - reference counts for nodes on the cycle will never decrement to 0
- several approaches for dealing with cycles
  - ignore
  - periodically invoke a tracing algorithm to collect cycles
  - specialized algorithms for collecting cycles

The Mark-and-Sweep Algorithm [McCarthy 1960]

- Marking phase
  - mark roots
  - trace all objects transitively reachable from roots
  - mark every traversed object
- Sweep phase
  - scan all objects in the heap
  - collect all unmarked objects

The Mark-Sweep algorithm

- Traverse live objects & mark black.
- White objects can be reclaimed.



heap

Triggering

```
New(A)=
   if free_list is empty
            mark_sweep()
             if free_list is empty
                        return (“out-of-memory”)
   pointer = allocate(A)
   return (pointer)
```

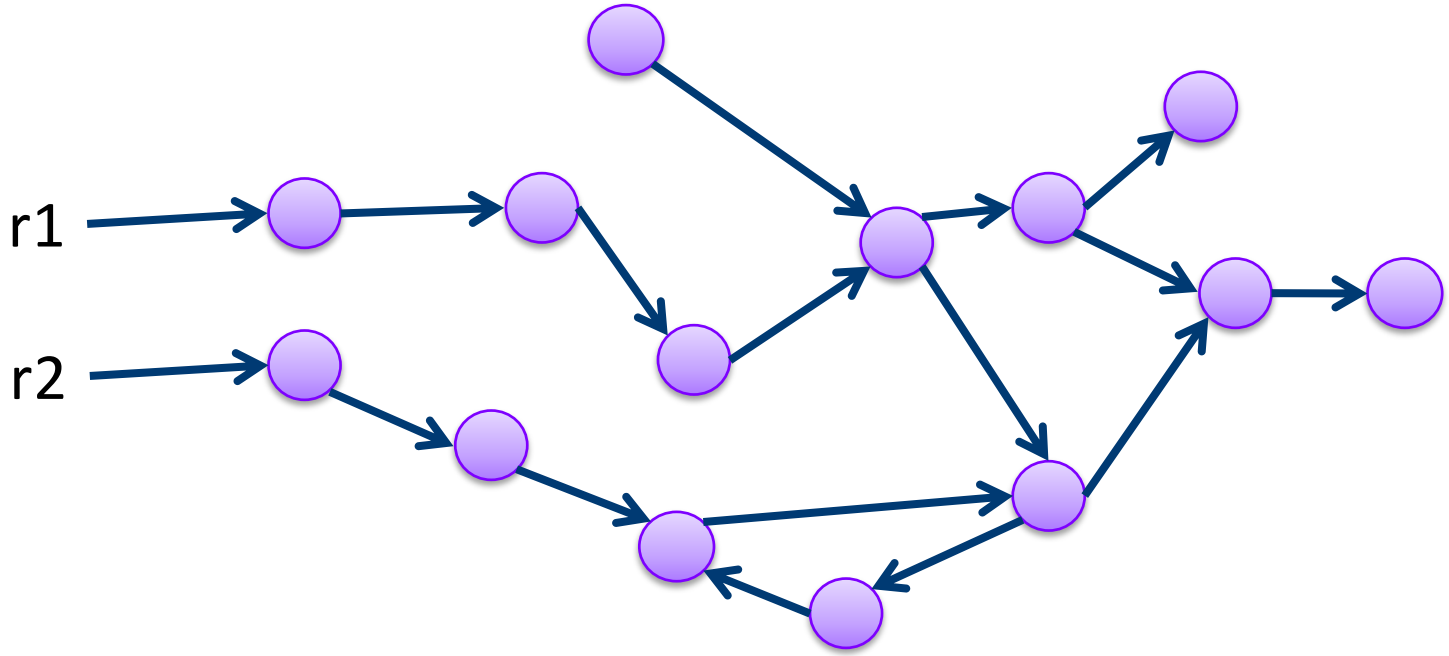
Basic Algorithm

```
mark_sweep()=
for Ptr in Roots
        mark(Ptr)
sweep()
```

```
mark(Obj)=
if mark_bit(Obj) == unmarked
        mark_bit(Obj)=marked
        for C in Children(Obj)
                mark(C)
```

```
Sweep()=
p = Heap_bottom
while (p < Heap_top)
        if (mark_bit(p) == unmarked) then free(p)
        else mark_bit(p) = unmarked;
        p=p+size(p)
```

Mark&Sweep Example



Mark&Sweep in Depth

```
mark(Obj)=
if mark_bit(Obj) == unmarked
        mark_bit(Obj)=marked
        for C in Children(Obj)
                mark(C)
```

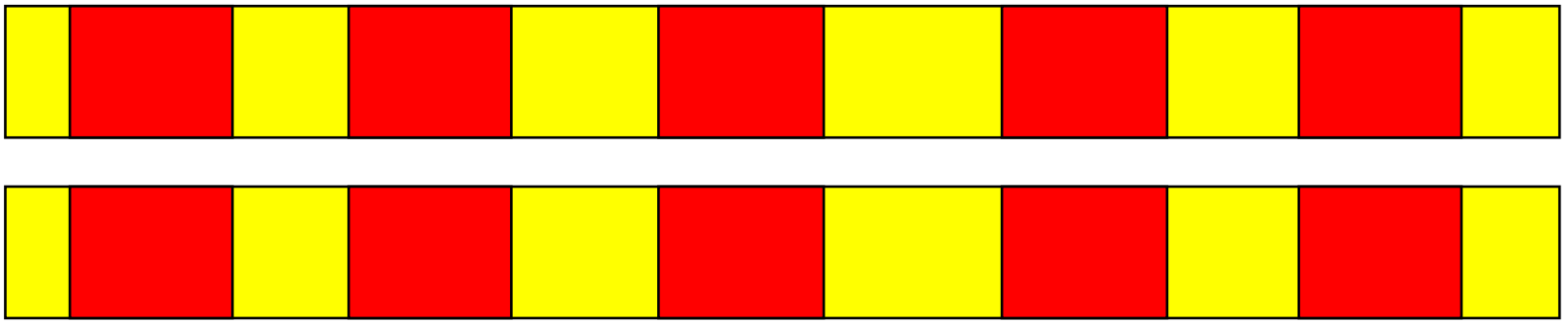
- How much memory does it consume?
  - Recursion depth?
  - Can you traverse the heap without worst-case O(n) stack?
    - Deutch-Schorr-Waite algorithm for graph marking without recursion or stack (works by reversing pointers)

Properties of Mark & Sweep

- Most popular method today
- Simple
- Does not move objects, and so **heap may fragment**
- Complexity
  - ☺ Mark phase: live objects (dominant phase)
  - ☹ Sweep phase: heap size
- Termination: each pointer traversed once
- Engineering tricks used to improve performance

Mark-Compact

- During the run objects are allocated and reclaimed
- Gradually, the heap gets fragmented
- When space is too fragmented to allocate, a compaction algorithm is used
- Move all live objects to the beginning of the heap and update all pointers to reference the new locations
- Compaction is very costly and we attempt to run it infrequently, or only partially



Mark Compact

- Important parameters of a compaction algorithm
  - Keep order of objects?
  - Use extra space for compactor data structures?
  - How many heap passes?
  - Can it run in parallel on a multi-processor?

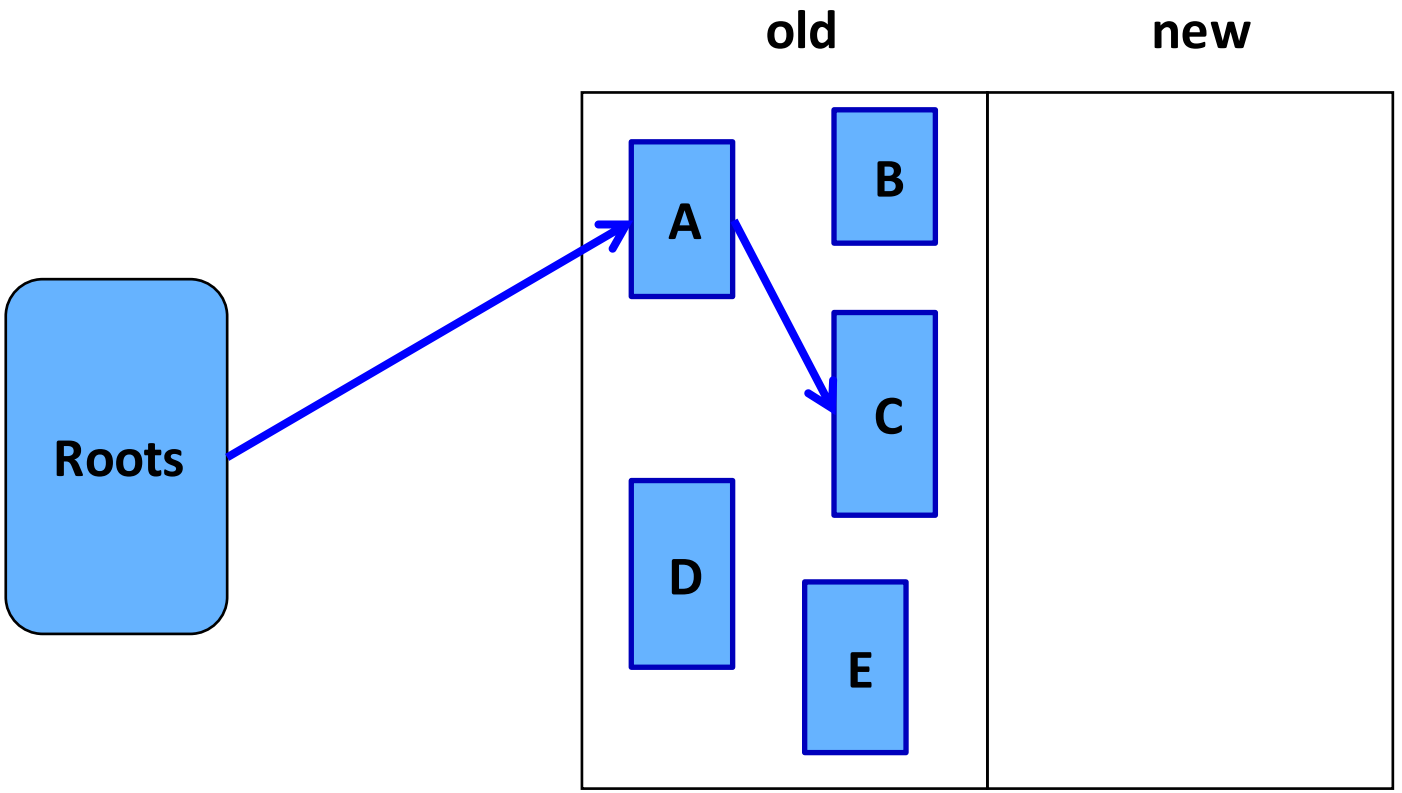
- We do not elaborate in this intro

Copying GC

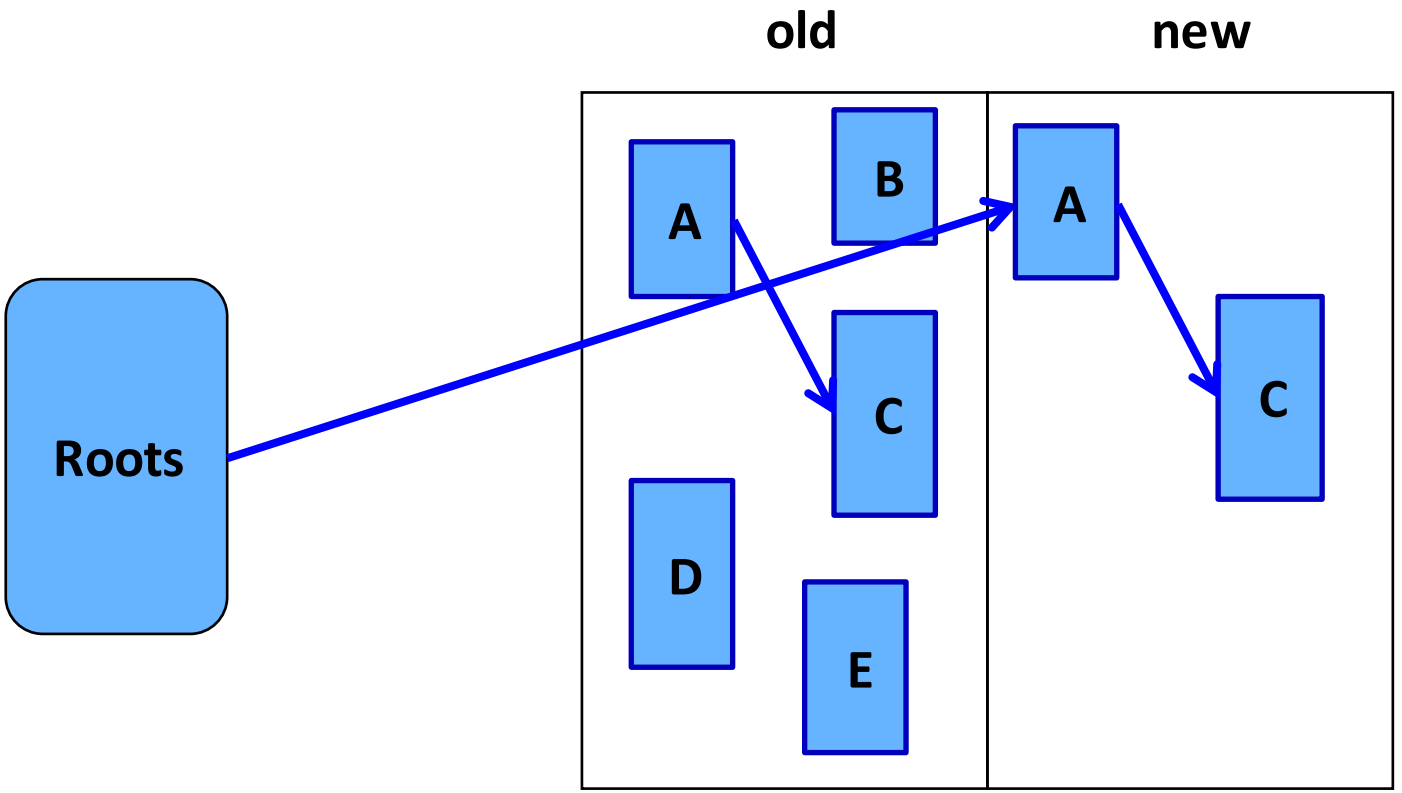
- partition the heap into two parts
  - old space
  - new space

- Copying GC algorithm
  - copy all **reachable** objects from old space to new space
  - swap roles of old/new space

Example



Example



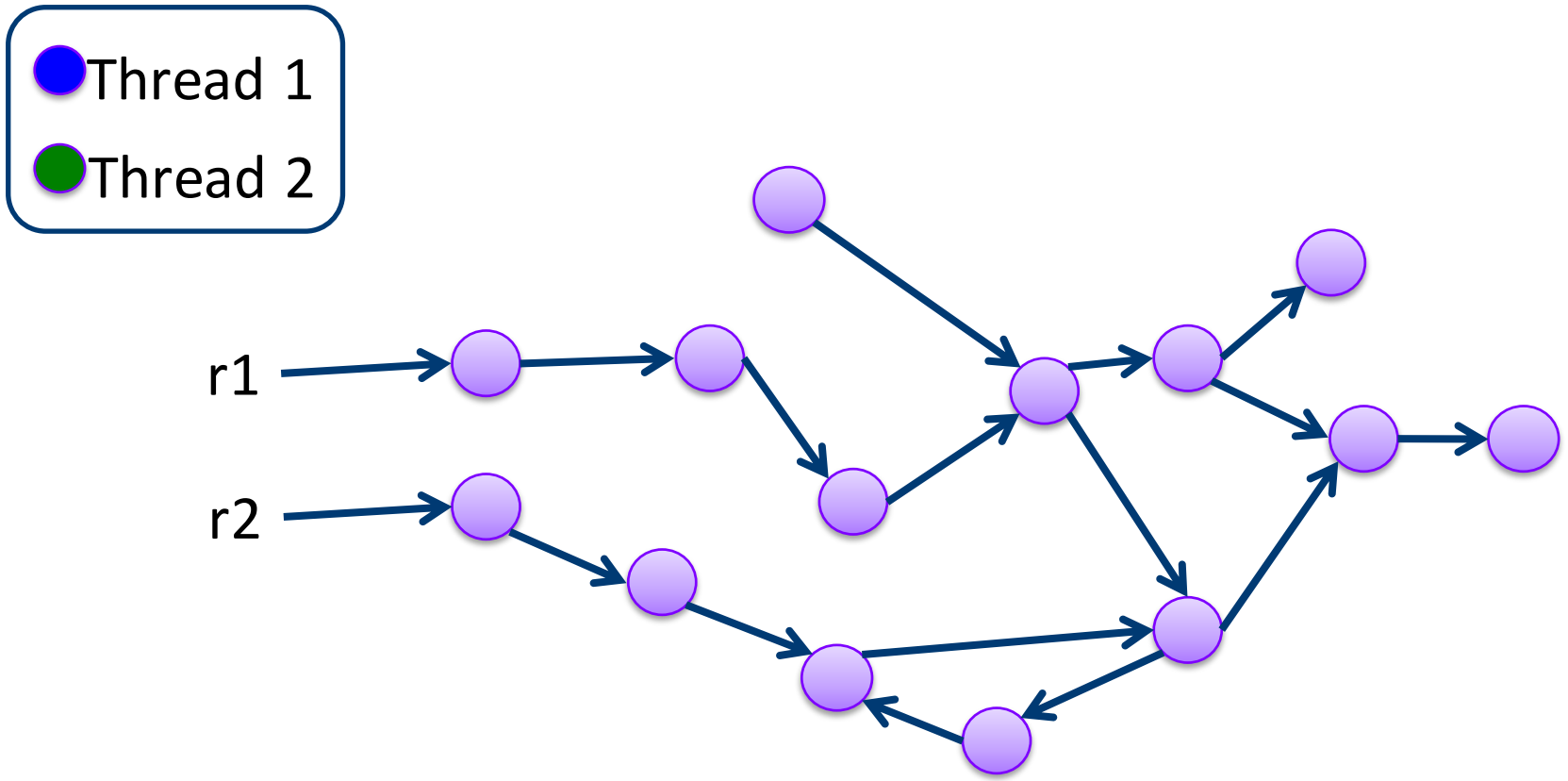
Properties of Copying Collection

- Compaction for free
- Major disadvantage: **half of the heap is not used**
- "Touch" only the live objects
  - Good when most objects are dead
  - Usually most new objects are dead
    - Some methods use a small space for young objects and collect this space using copying garbage collection

A very simplistic comparison

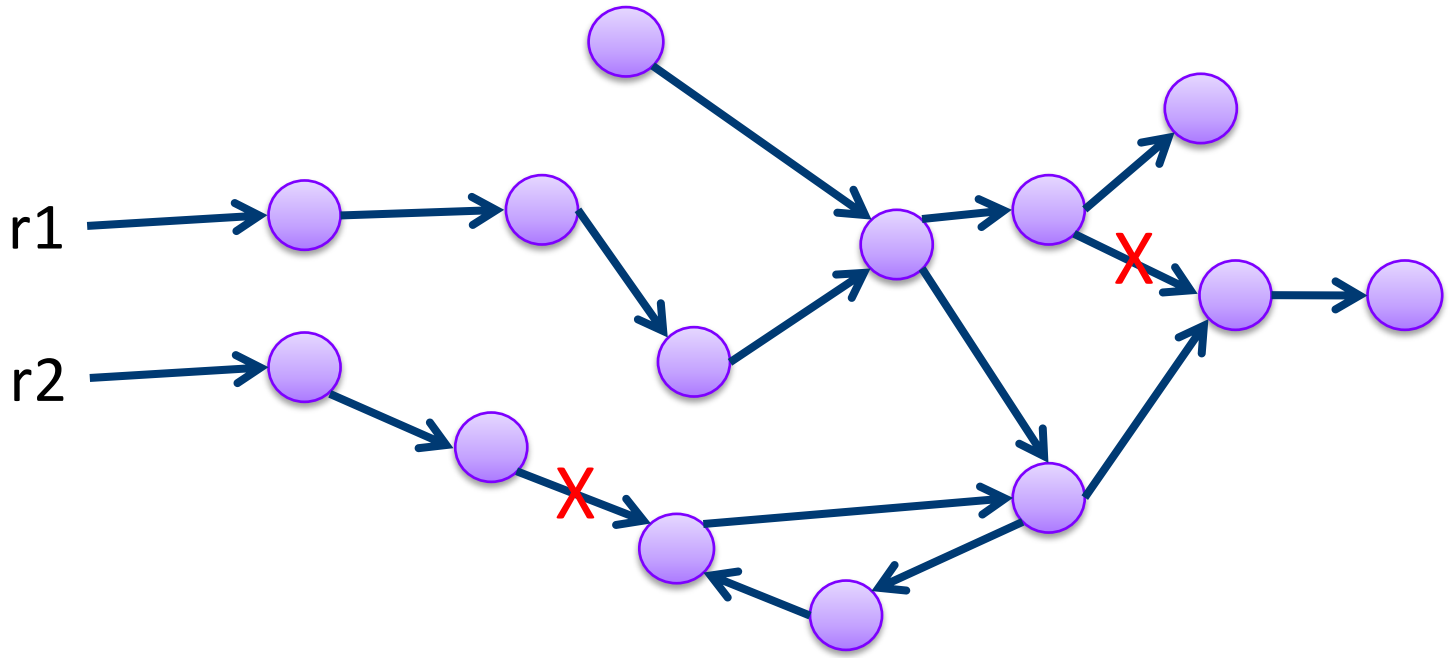
	Reference Counting	Mark & sweep	Copying
Complexity	Pointer updates + dead objects	Mark = live objects Sweep = Size of heap	Live objects
Space overhead	Count/object + stack for DFS	Bit/object + stack for DFS	Half heap wasted
Compaction	Additional work	Additional work	For free
Pause time	Mostly short	long	long
More issues	Cycle collection		

Parallel Mark&Sweep GC



Parallel GC: mutator is stopped, GC threads run in parallel

Concurrent Mark&Sweep Example



Concurrent GC: mutator and GC threads run in parallel, no need to stop mutator

Conservative GC

- How do you track pointers in languages such as C ?
  - Any value can be cast down to a pointer
- **How can you follow pointers in a structure?**

- Easy – be conservative, consider anything that can be a pointer to be a pointer
- Practical! (e.g., Boehm collector)

Conservative GC

- Can you implement a conservative **copying GC**?
- What is the problem?
- **Cannot update pointers to the new address… you don’t know whether the value is a pointer, cannot update it**

Modern Memory Management

- Considers standard program properties
- Handle parallelism
  - Stop the program and collect in parallel on all available processors
  - Run collection concurrently with the program run
- Cache consciousness
- Real-time

Terminology Recap

- Heap, objects
- Allocate, free (deallocate, delete, reclaim)
- Reachable, live, dead, unreachable
- Roots
- Reference counting, mark and sweep, copying, compaction, tracing algorithms
- Fragmentation