CSE304
Compiler Design
Runtime Storage Management

TONY MIONE

Lecture Outline

Scope

Run-time Storage Management

- Static Allocation
- Stack Allocation
- Heap Allocation
  - Programmer-driven allocation/deallocation
  - Garbage Collection

Static Allocation

- Typically exist from start to end of execution (but may have smaller scope)

  Allocated in a block of memory specifically for data
  - Read Only data segment
  - Read/write data segment
- Include:
  - Global variables
  - Program’s machine code [read only]
  - Local function variables that retain value from one call to the next
  - Literals and constants

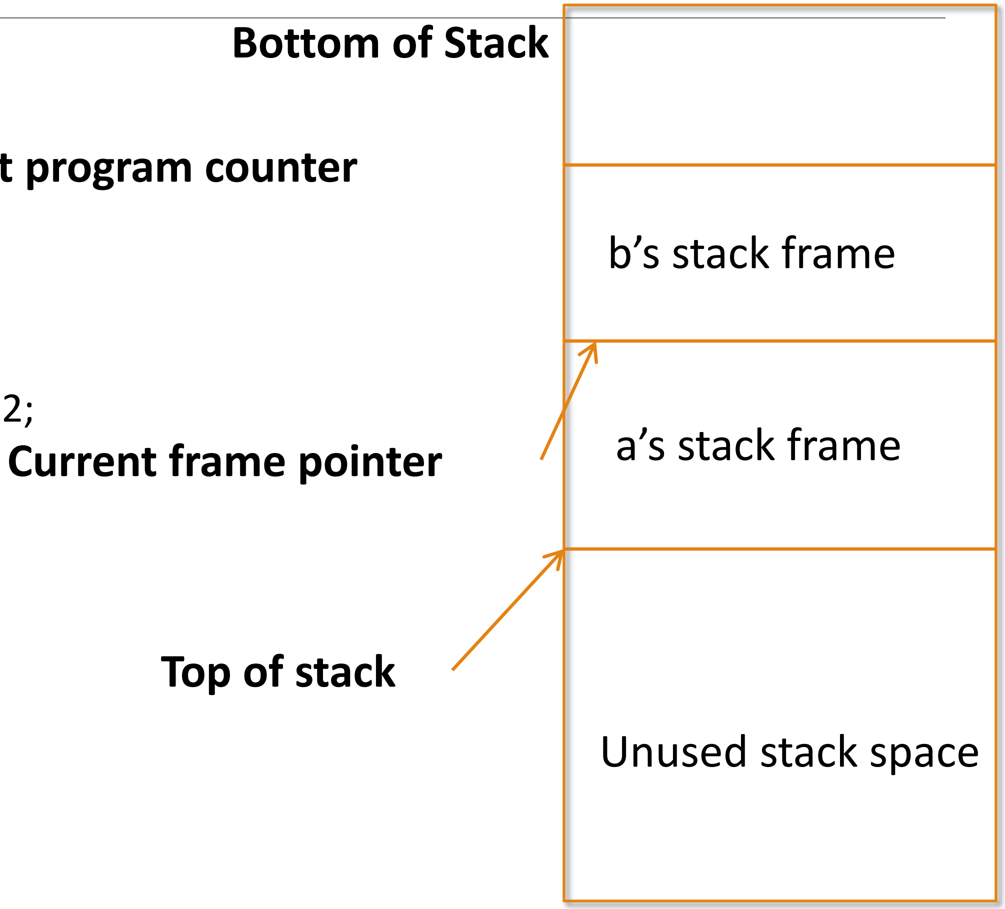
Dynamic Allocation

- Typically allocated on a ‘stack’
  - Stack grows downward from higher addresses
  - Each function call creates an area called a *frame* or *activation record*
  - Variables do not have same address from one call to next
  - Referenced with a constant offset to the stack frame’s base
- Include:
  - Function parameters
  - Function’s local variables

Dynamic Allocation

```
void a(int invalue)
{
  int c,d;
  …
}
void b(int val1, int val2)
{
  float res1 = (float) val1 / (float) val2;
  a(res1);
}
…
int main(int argc, char **argv)
{
  int v1 = 1;
  int v2 = 2;
  b(v1, v2);
  ….
}
```

Current program counter



Bottom of Stack

b’s stack frame

Current frame pointer

a’s stack frame

Top of stack

Unused stack space

Heap Allocation

- Allocation occurs as a result of specific programmer actions
  - Explicit allocation (i.e. *malloc()* call from C run time library)
  - May occur on object creation (i.e. *new* operator in C++)

Managing the Heap

- Language may require allocated space to be freed by developer
  - *free()* in C
  - *delete in C++*
- Some languages provide automatic space reclamation (when object is not referenced by any variable)
- Either must have facilities to track heap usage

Managing the Heap

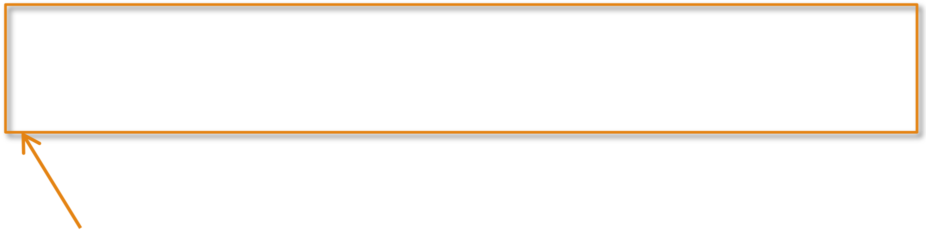
- Heap starts as a single block
- Free space usually managed by one or more linked lists
- Allocation and de-allocation of different size memory blocks tend to leave ‘holes’. This is called *Fragmentation*.

Fragmentation

- *Fragmentation*:
  - Non-contiguous blocks of memory
  - Results from alloc/dealloc of memory over time
- Types:
  - *Internal* – Wasted space within allocated memory blocks
    - object uses less space than what was allocated
      - ➔ Alignment requirements force the use of padding that is not needed by the object
  - *External* – Wasted space outside allocated blocks
    - small unusable chunks of free memory (not large enough for any request)
    - Severity of fragmentation varies based on
      - Memory allocation strategy [first fit, best fit, worst fit, etc]
      - Specific program behavior

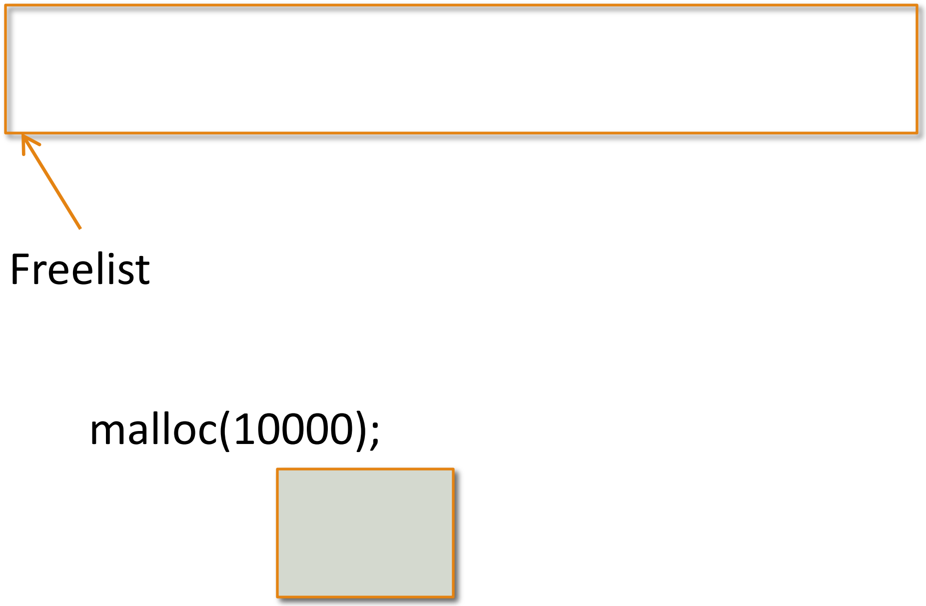
Fragmentation

Heap at start of program



Freelist

Fragmentation



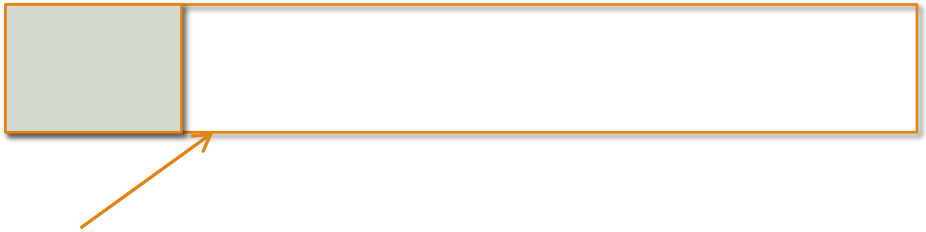
Heap at start of program

Freelist

malloc(10000);

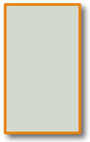
Fragmentation

Heap after 1st allocation request

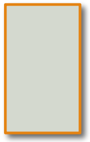


Freelist

malloc(2000);

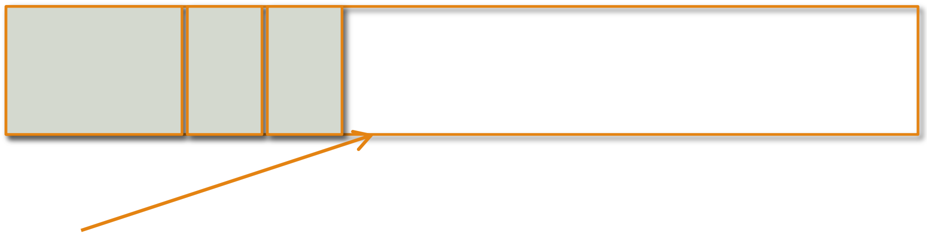


malloc(2000);



Fragmentation

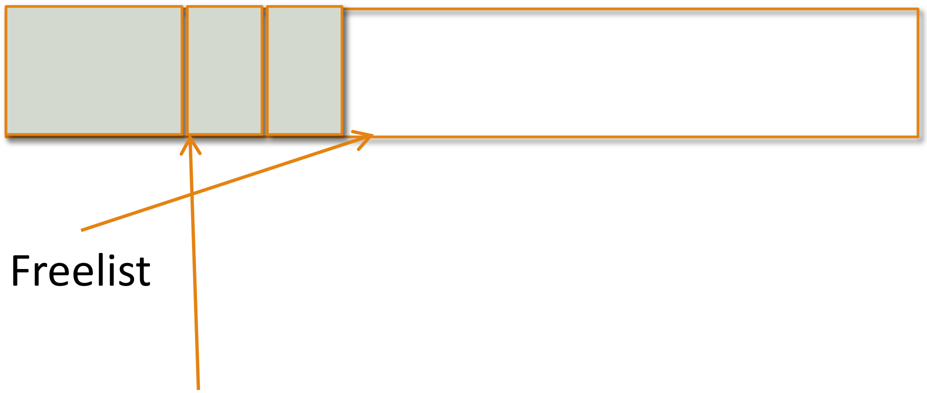
Heap after 3 allocation requests



Freelist

Fragmentation

Heap after 3 allocation requests

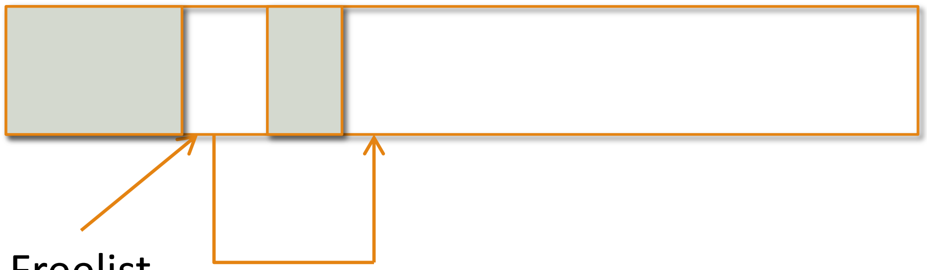


Freelist

free(addr);

Fragmentation

Heap after allocation requests and a free() call



Freelist

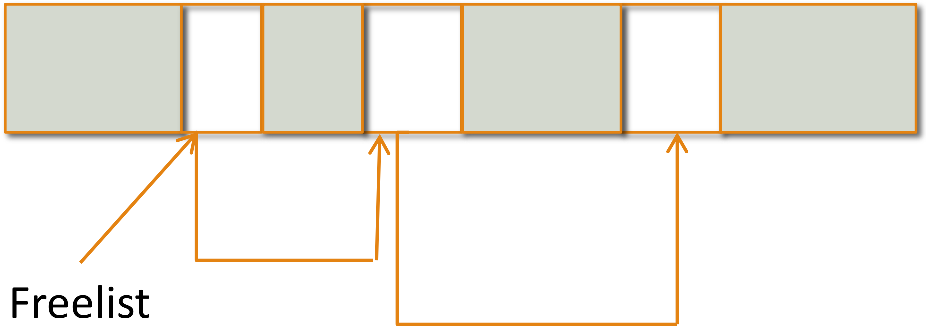
Fragmentation

malloc(10000);

???

FAIL!

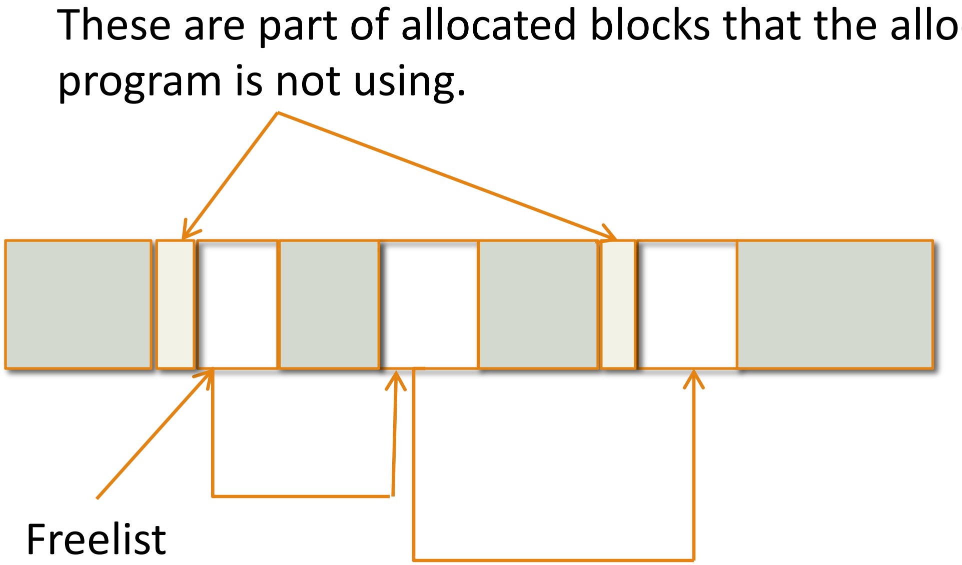
Heap after numerous allocations and de-allocations



Freelist

This is an outcome of *External Fragmentation*

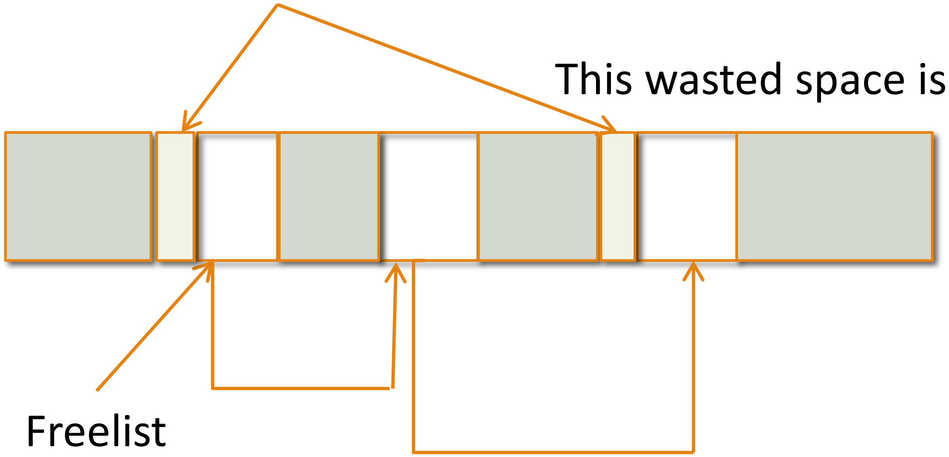
Fragmentation



These are part of allocated blocks that the allocating program is not using.

Freelist

Fragmentation



These are part of allocated blocks that the allocating program is not using.

This wasted space is due to *Internal Fragmentation*

Freelist

Garbage Collection

- Languages may provide a *garbage collector* as part of their runtime system
- Process usually triggered
  - When free memory drops below some threshold
  - On the first failed memory allocation request
- Purpose:
  - Find variable space no longer referenced
  - Automatically de-allocate and reclaim space
- Does not usually include *Compaction*
  - More difficult than just freeing blocks since pointers must be corrected

Garbage Collection

- Advantages:
  - Less error prone than relying on developer to properly allocate/deallocate space
  - Helps assure sufficient space will be available for dynamic object creation
- Disadvantages:
  - Algorithms add complexity to language system
  - Costly and it slows run-time performance

Garbage Collection

- Language tracks heap usage
  - Understands language constructs (pointers, structures, objects, etc.)
- Tracking methods
  - *reference counts* to allocated blocks
  - *tracing* a collection of objects

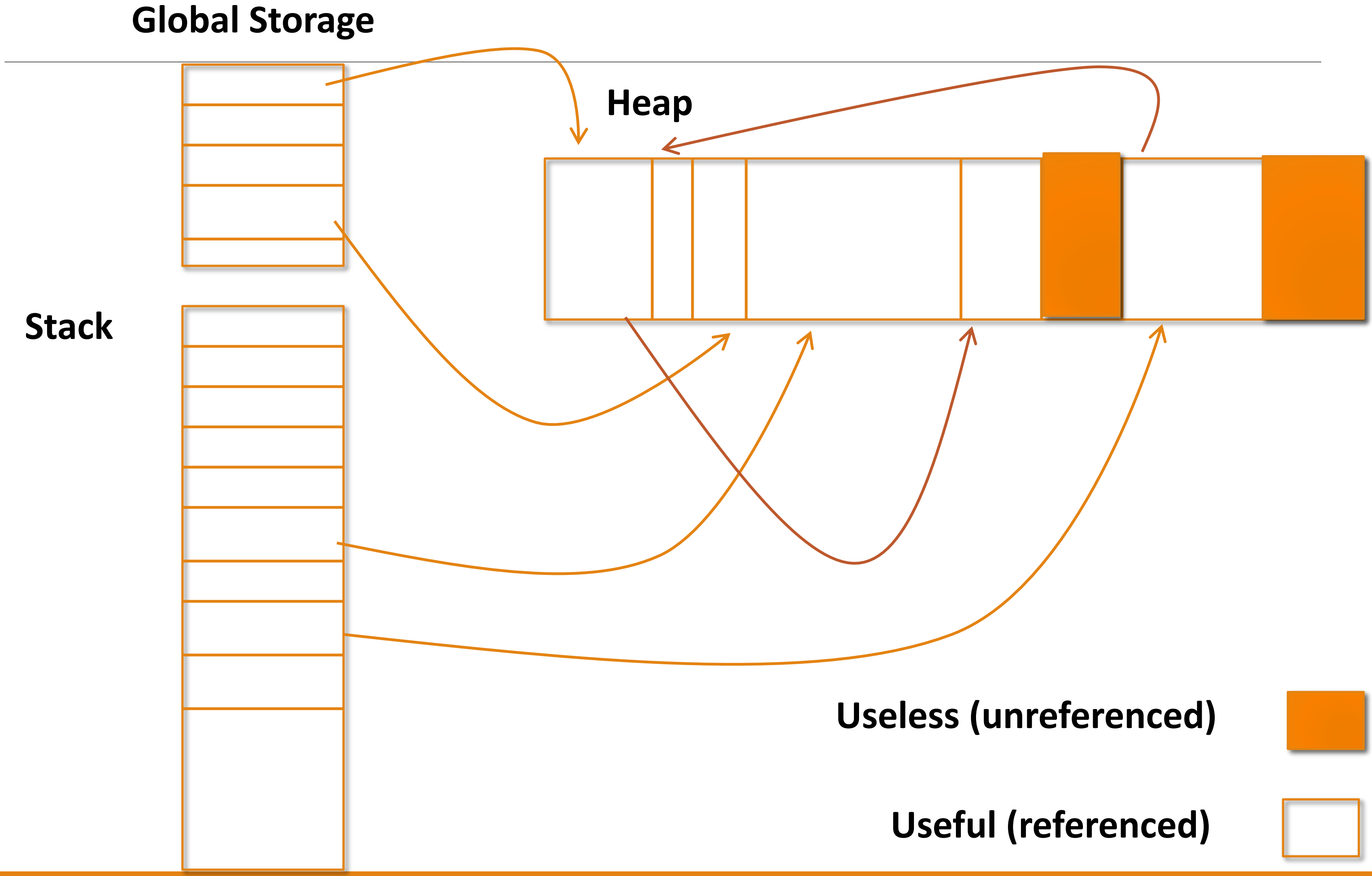
Garbage Collection Algorithms

- Mark-and-sweep
- Pointer reversal
- Stop-and-copy
- Generational collection
- Conservative collection

Mark-and-sweep

1. Mark all blocks useless
2. For each pointer outside the heap (local or global program variables):
   - Follow pointers to heap memory
   - Mark block ‘useful’
   - Recursively follow pointers from that structure/object to others
3. Walk through heap sequentially, moving ‘useless’ blocks to the free list
4. Disadvantage: Costs stack space due to recursion

Mark and Sweep



Global Storage

Heap

Stack

Useless (unreferenced) 🟧

Useful (referenced) ☐

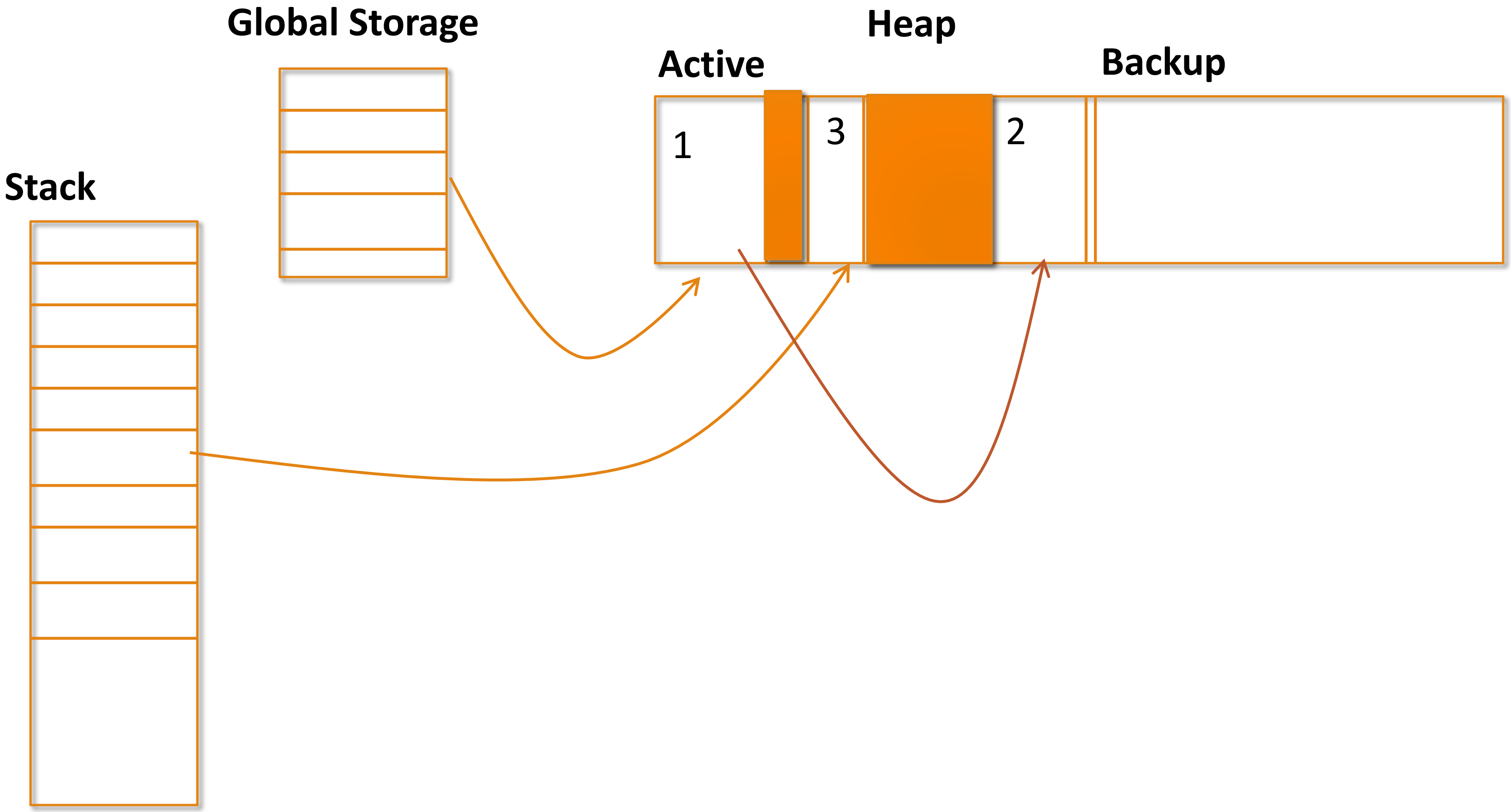
Pointer reversal

- As blocks are explored (step 2)
  - Save a pointer to explore
  - Use field to point back to last explored block
- Advantage:
  - Avoids excessive stack space usage (no recursion)

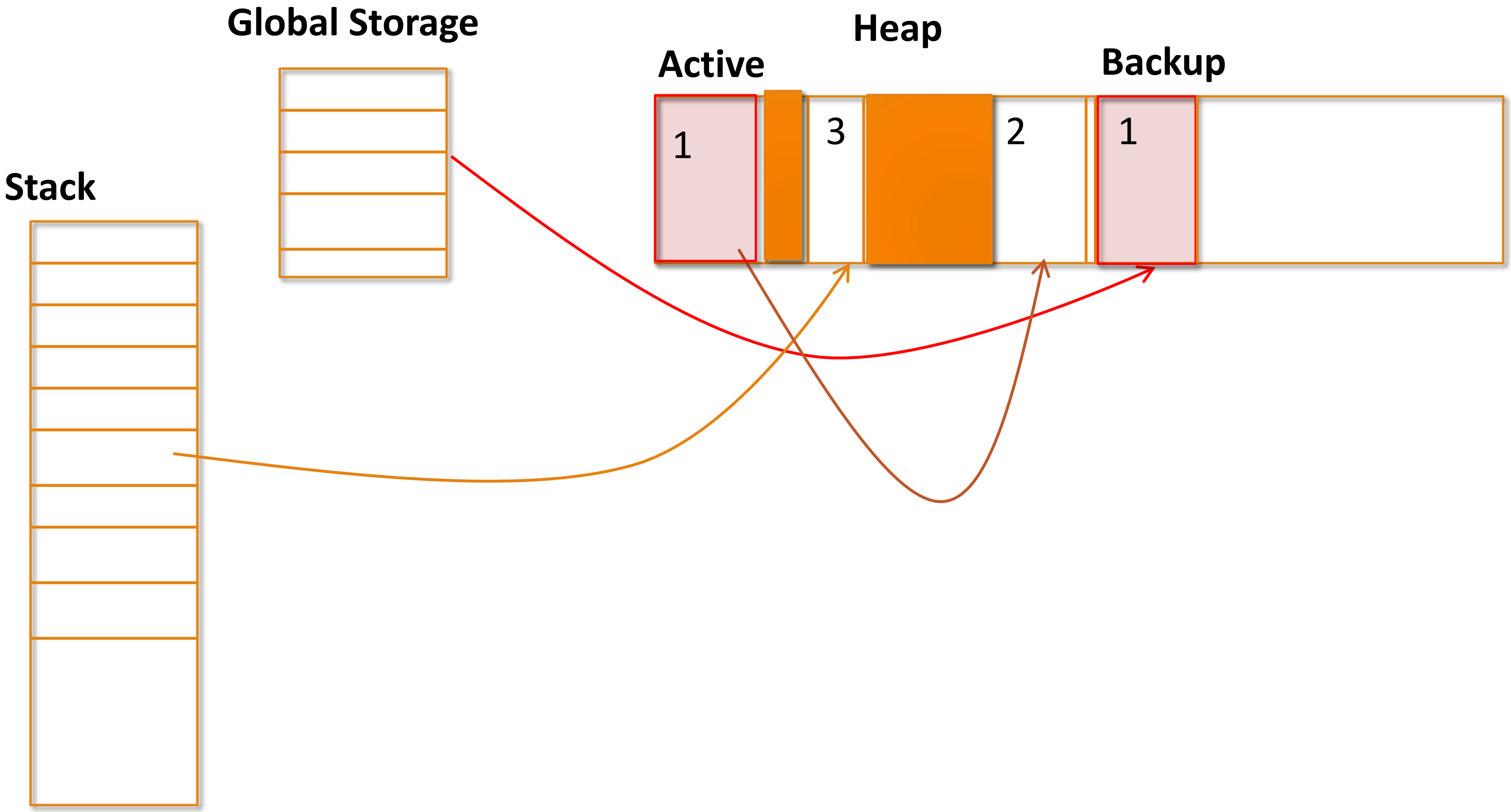
Stop and Copy

- Divide heap in two
- Use only one of the two parts
- At collection time:
  - For each visited block
    - Move to 'other' heap
    - Mark first block 'useful' with a forwarding pointer
    - If block was already marked useful, update non-heap pointer
  - When done, switch the 'current' heap to the newly created one (all pointers in code have been updated)

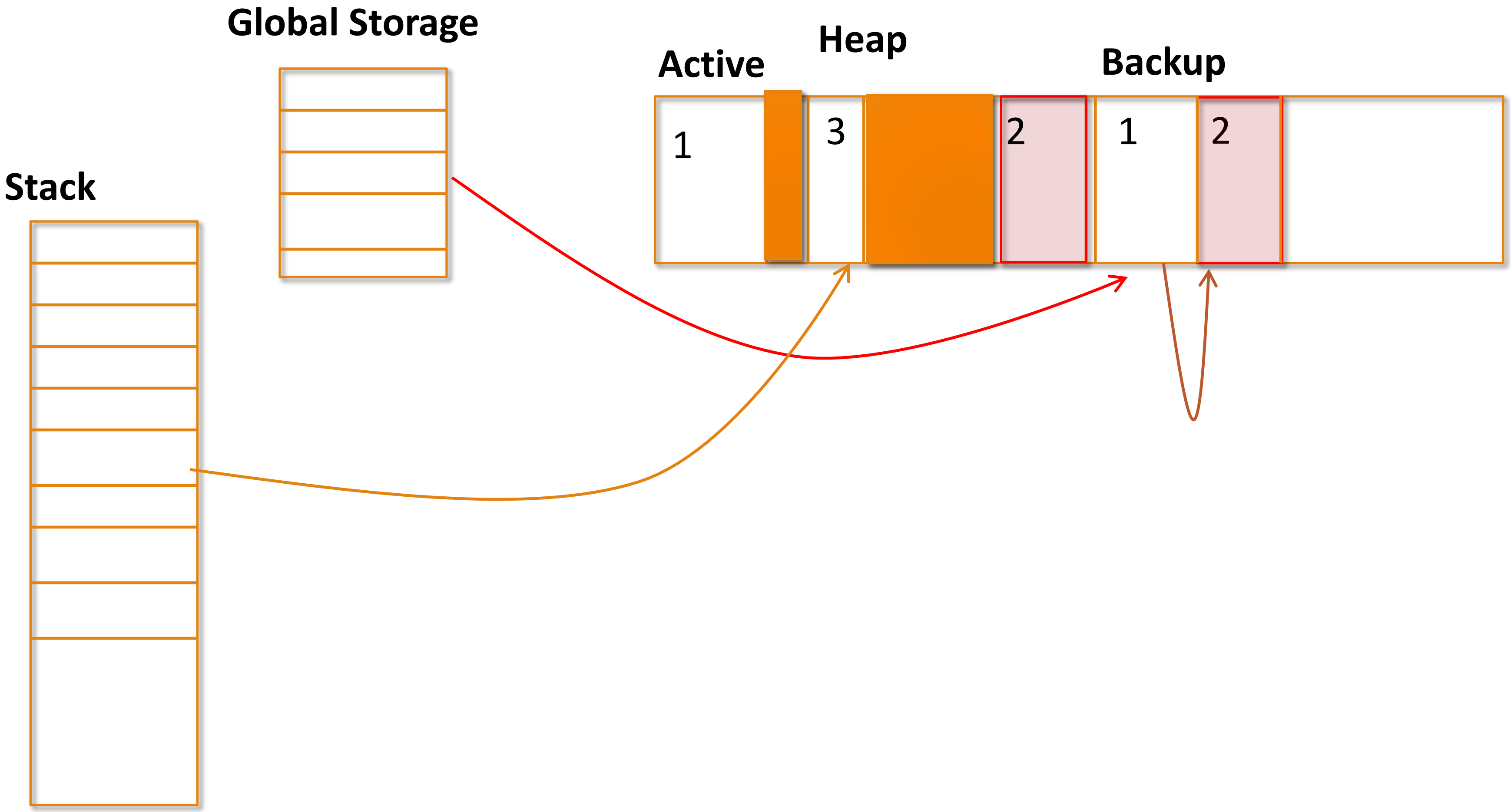
Stop and Copy



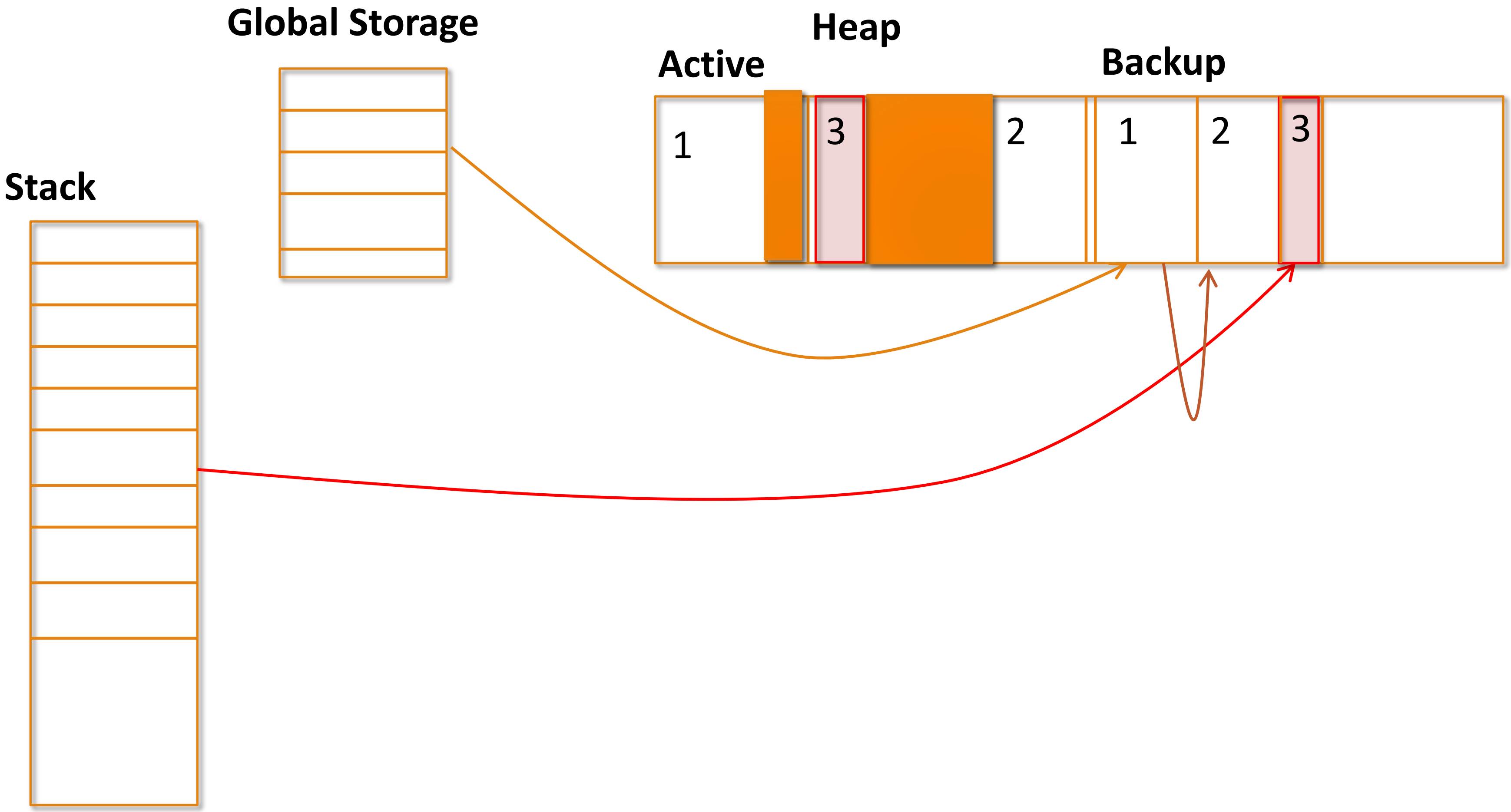
Stop and Copy



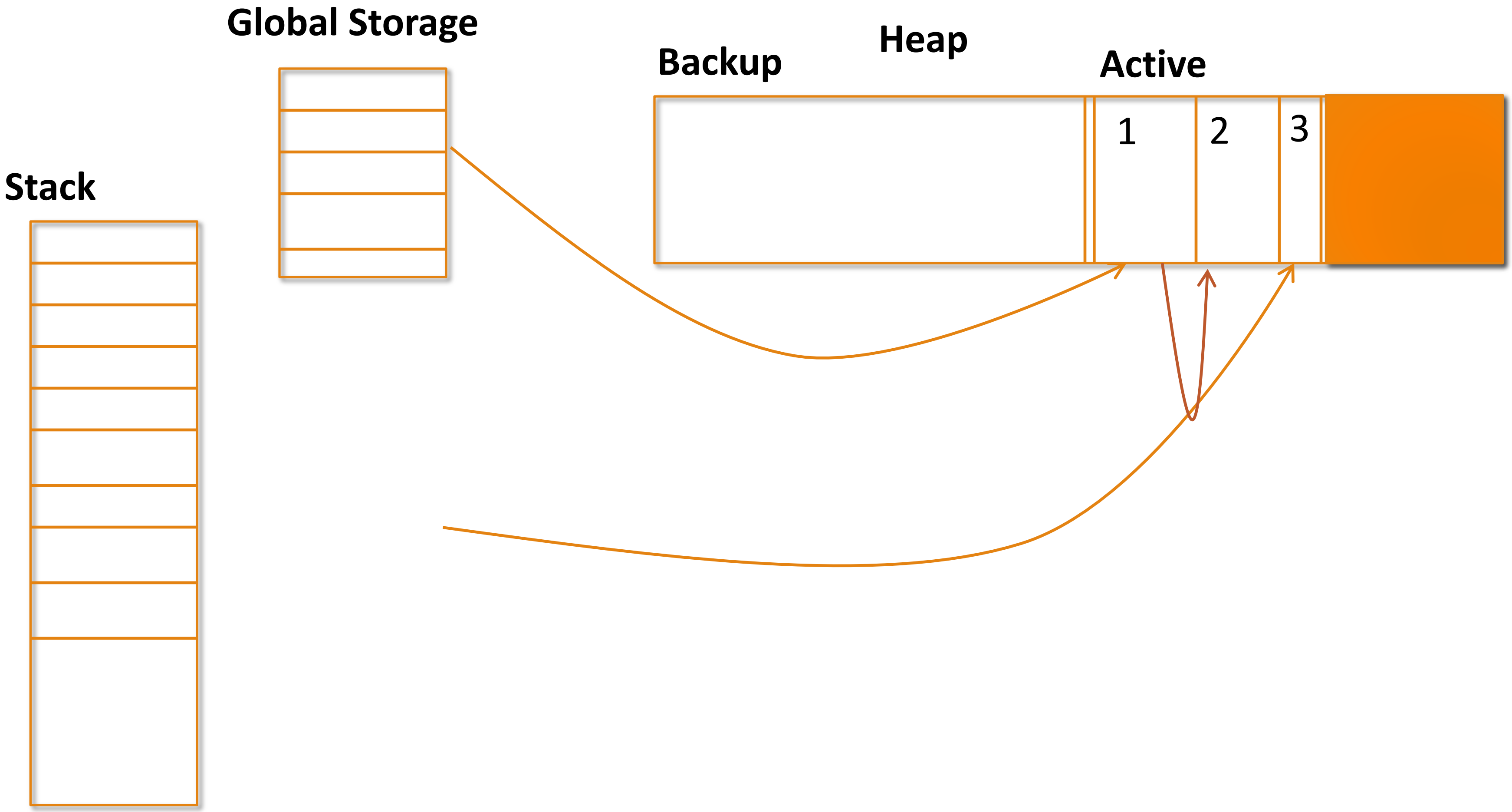
Stop and Copy



Stop and Copy



Stop and Copy



Stop and Copy

- Advantages:
  - Eliminates external fragmentation (uses compaction)
  - Run-time proportional to ‘useful’ blocks not all blocks
- Disadvantage:
  - Can only use ½ of the heap at one time

Generational Collection

- Idea: most allocations are short-lived
- Heap is divided into several sections
  - Usually 2 but can be more
  - Current allocations are 'youngest'
  - Collector typically only runs in 'youngest' generation area of heap
- Each time a block survives collection, it moves to an 'older' generation area (ala Stop-and-Copy)

Generational Collector

- Garbage collector must be prepared to collect in all heap generations
- Usually, cost is proportional to youngest heap region size

Conservative Collection

- Used when no detailed structure knowledge is available
- Similar to Mark-and-sweep
- Procedure:
  - Scan stack and global space
  - Identify pointers by checking bit pattern against heap address range
  - Mark block ‘useful’ and scan block for other words that ‘look like’ pointers

Observation

- Note that we are marking blocks based on a bit pattern so it is only a ‘guess’ but is conservative so may miss opportunities to reclaim space

Questions