Functional Programming

Christopher D. Clack

FUNCTIONAL PROGRAMMING

Lecture 24

MARK-SCAN GARBAGE COLLECTION

FUNCTIONAL PROGRAMMING
MARK-SCAN GARBAGE COLLECTION

In this lecture we will explore the first of three canonical garbage collection algorithms – Mark Scan Garbage Collection

The lecture provides simple imperative pseudo-code and discusses implementation issues such as the use of a marking stack versus the use of pointer reversal, and the use of a variant of the scan phase known as "lazy scanning"

The lecture ends with a summary of good and bad characteristics of Mark-Scan Garbage Collection, and a few words about its use in Miranda

CONTENTS

FUNCTIONAL PROGRAMMING
MARK-SCAN GARBAGE COLLECTION

OVERVIEW

- Mark-Scan GC triggered by *malloc()* when free memory becomes low
- Program evaluation typically <u>pauses</u> during GC
- Garbage is detected by tracing all live pointers & marking all live blocks
- Garbage is collected / made available for re-use by scanning the whole heap
- After GC, all free blocks will be on the free list
- Live blocks don't move: fragmentation may occur
- A separate compactor can be used

Triggering the garbage collection

Mark-scan (or "mark-sweep") garbage collection is triggered by *malloc()* when the amount of available free memory is low (but not completely exhausted)

Evaluation of the program typically pauses until garbage collection is done

Identifying garbage

The mark-scan garbage collection algorithm automatically identifies garbage by following all live pointers and "marking" every block of memory that can be reached by following all live pointers

Collecting garbage

Following the "mark" phase, the mark-scan garbage collector must collect the garbage blocks and make them available for re-use. It does this in a "scan" phase that visits every block in the heap

- before the scan, a new (empty) free list is created
- during the scan, all blocks that are not marked as being "live" are added to the free list

Interaction with memory allocation

Following the scan phase, all free blocks are on the free list and are available to *malloc()*

Fragmentation

Live blocks don't move: fragmentation may persist unless a separate compactor is also used

FUNCTIONAL PROGRAMMING
MARK-SCAN GARBAGE COLLECTION

ASSUMPTIONS

- *malloc()* allocates free memory from a free list
- all live blocks are reachable by tracing all live pointers
- can distinguish pointers from other data
- all pointers point to the start of a block data area
- all live pointers can be found by tracing from a "root set" of pointers
- evaluation pauses during GC
- every block has an associated "mark bit", initialised to the value 0 (meaning "free")

Assumptions

Mark-scan garbage collection assumes:

- normally, the memory allocator uses a simple free list (other allocation methods can also be supported – though not normally pointer-increment allocation)
- all live blocks can be reached by following all live pointers
- it is always possible to distinguish between values that are pointers and those that are other data
- all pointers in the data area of a block point to the start of the data area of a block
- all live pointers can be found by following (transitively) all pointers found in one or more known live blocks that are referenced by a small set of pointers (sometimes called the "root set")
- the simple Mark-Scan method assumes that program evaluation pauses during garbage collection and resumes after the scan phase has finished
- It is assumed that every block – live or free – has an associated "mark bit" (perhaps held in the block header), and that at the start of the program all of these mark bits are set to the value 0

FUNCTIONAL PROGRAMMING
MARK-SCAN GARBAGE COLLECTION

SIMPLE IMPERATIVE PSEUDO-CODE

```
mark() =
   for each P in RootSet {  xmark(P) }

xmark(P) =
   markbit = read(P-Headersize)
   if (markbit == 0){
      write(P-Headersize,1)
      for each M in children(P) {  xmark(M)  }
   }

scan() =
   FLP=0
   xscan(HeapStartAddress)

xscan(P) =
   if (P < HeapEndAddress) {
      (markbit, datasize) = read(P-Headersize)
      if (markbit == 1) write(P-Headersize, 0)  else  free(P)
      xscan(P + datasize + Headersize)
   }
```

Simple Mark-Scan garbage collection first pauses evaluation of the program, then inspects the "root set" of pointers

Mark phase: for each pointer P in the root set call x*mark(P)*, which will:

- find the block to which P points, and check its "mark bit":
  - if it is 1, this block has already been visited by the x*mark()* function, so end this call to x*mark*() without error
- otherwise:
  - set this block's mark bit to 1
  - read the entire data area of this block, looking for pointers – for each such pointer M call the function x*mark(M)* recursively and wait for it to finish
  - when all such pointers M have been found, and calls to *xmark(M)* have finished, end this call to x*mark()* without error

Scan phase: after the mark phase has finished, create a new (empty) free list, and call *xscan(lowest address in heap memory),* which will:

- terminate if P>= end of heap
- else read the block header to find the mark bit and the block data size
- if the mark bit is 1, set it to 0
- otherwise, add this block to the free list, and call xscan(P + header size + data size)

End garbage collection and resume evaluation of the program

FUNCTIONAL PROGRAMMING
MARK-SCAN GARBAGE COLLECTION

USING A MARKING STACK

```
mark() =
   markingstack = Empty
   for each P in RootSet {
      write((P-Headersize), 1)
      push(P, markingstack)
      xmark()
   }

xmark() =
   while not_empty(markingstack) {
      N = pop(markingstack)
      for each M in children(N){
         markbit = read(M-Headersize)
         if (markbit == 0){
            write(M-Headersize, 1)
            if (children(M)!=Ø) push(M, markingstack)
   }  }  }
```

Using a marking stack

Previously we described code for *mark()* that called itself recursively, in a way that does not lend itself to tail-recursion optimisation. There will be a new stack frame on the runtime stack for each new recursive call, and excessive recursion can take both time and (precious) memory

It is more efficient for *mark()* to replace the recursive calls with an iterative loop and to use an auxiliary stack data structure to hold pointers to live blocks that have not yet been visited

This slide gives pseudo-code to illustrate how stack-based marking works (based loosely on Jones & Lins 1996, Page 78). The function *children(N)* takes the address N of a block and creates a set of all pointers contained in its data area

In pathological cases, for both the recursive and iterative versions of *mark()*, stack overflow can occur and must be managed. Overflow can for example be checked at every call to *push()*. Alternatively, a write-protected memory location could be set as a stack limit, so that a memory protection error is triggered on overflow (and handled by the garbage collector). The latter however can be expensive if triggered

Other approaches to managing the potential for overflow involve, for example, pausing *mark()* and conducting an interim *scan()*, then creating a new RootSet and resuming *mark()*

FUNCTIONAL PROGRAMMING
MARK-SCAN GARBAGE COLLECTION

POINTER REVERSAL

- First introduced in Lecture 21
- Can be used during the marking phase of Mark-Scan GC
- Can be used for variable-sized blocks with more than 2 pointers, but must be modified:
    - block header: *n-field* (number of pointers in block) and *i-field* (number of pointers processed so far)
    - n-field & i-field must have enough bits to represent the largest possible number of pointers in the largest possible block
    - pointers in a block processed in fixed order
    - *i-field* determines (i) next pointer to process and (ii) which pointer currently holds the back-pointer
    - the *i-field* can double as a mark bit

Pointer reversal during marking

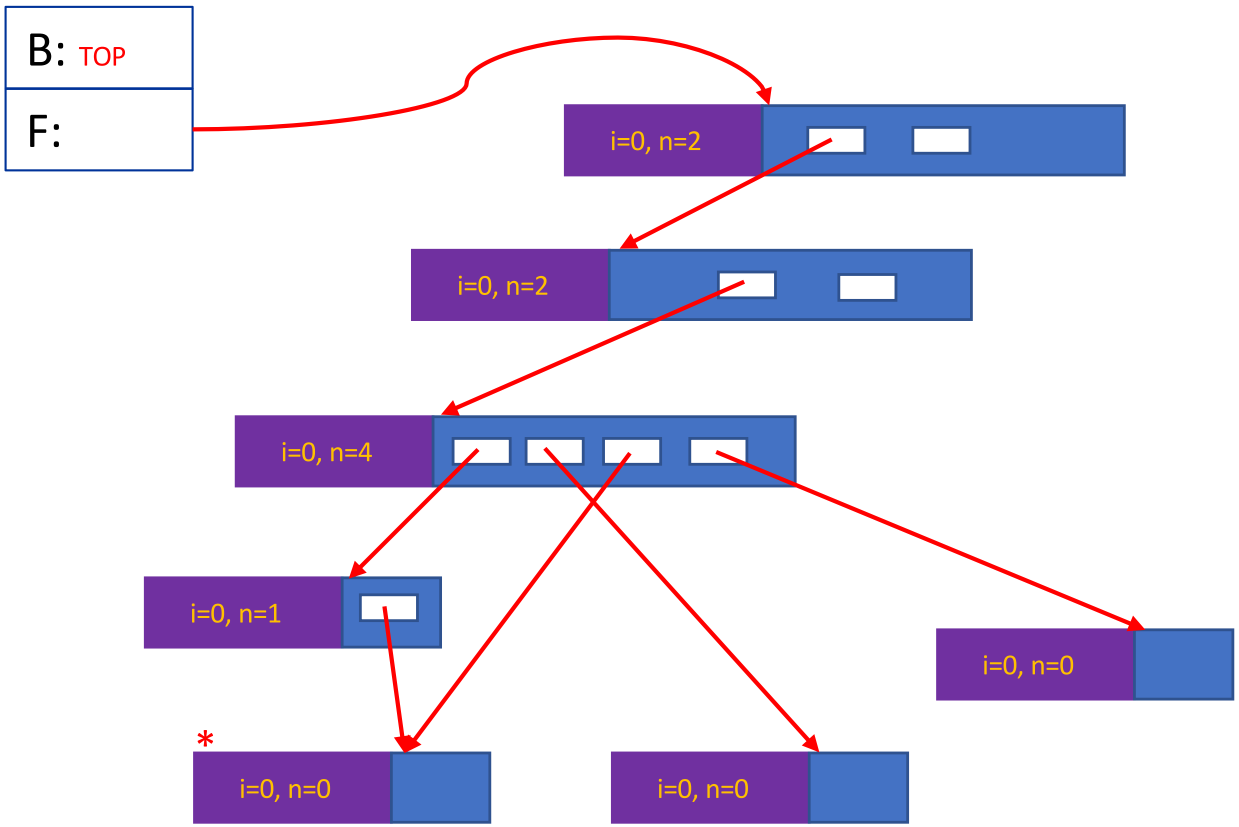
The Deutsch-Schorr-Waite pointer-reversal technique was introduced in Lecture 21. This technique can also be used during the mark phase of Mark-Scan garbage collection

Lecture 21 discussed the use of pointer-reversal for a binary tree, where each node (block) had at most two pointers to subtrees. For a system that uses variable-sized blocks, a block could contain more than two pointers and so the algorithm must be varied as follows:

- extend the block header with two pieces of information: the "*n-field*" contains the number of pointers contained in the data area of the block, and the "*i-field*" indicates the number of those pointers that have been partially or fully processed
- both *n-field* and *i-field* must have enough bits to represent the largest number of pointers that can be held in the largest possible block
- pointers are processed in a fixed and reproducible order, and *i-field* tracks the progress of the mark-phase
- *i-field* is used to determine which is the next pointer to process after a depth-first search returns back to the current block, and also which location to use for holding the back pointer
- Notice that the *i-field* can also double as a mark bit

FUNCTIONAL PROGRAMMING
MARK-SCAN GARBAGE COLLECTION

POINTER REVERSAL – VARIABLE SIZED BLOCKS



The diagrams on this slide and the following slides illustrate the extension of a block header to include *n-field* and *i-field* so that pointer reversal can be used for variable-sized blocks

Headers are shown in purple, and data areas are in blue. The values of *i-field* and *n-field* are shown as "i" and "n" in orange

In this example the block marked with a red star has two parents

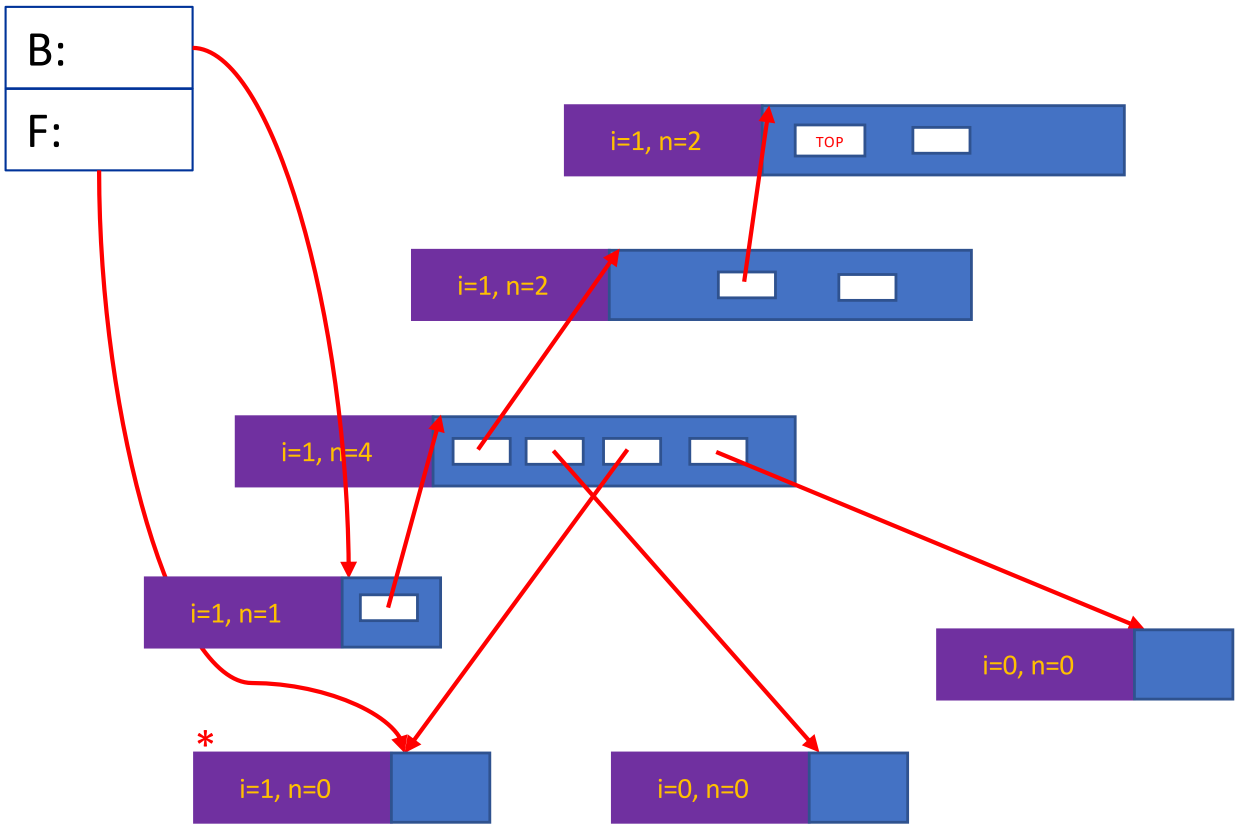
All blocks start with *i = 0*, which indicates that they have not yet been visited by *mark()*

FUNCTIONAL PROGRAMMING
MARK-SCAN GARBAGE COLLECTION

In this diagram the *mark()* function has used pointer-reversal to follow the first pointers of each of the blocks, starting from the top until reaching the red-starred block at the bottom

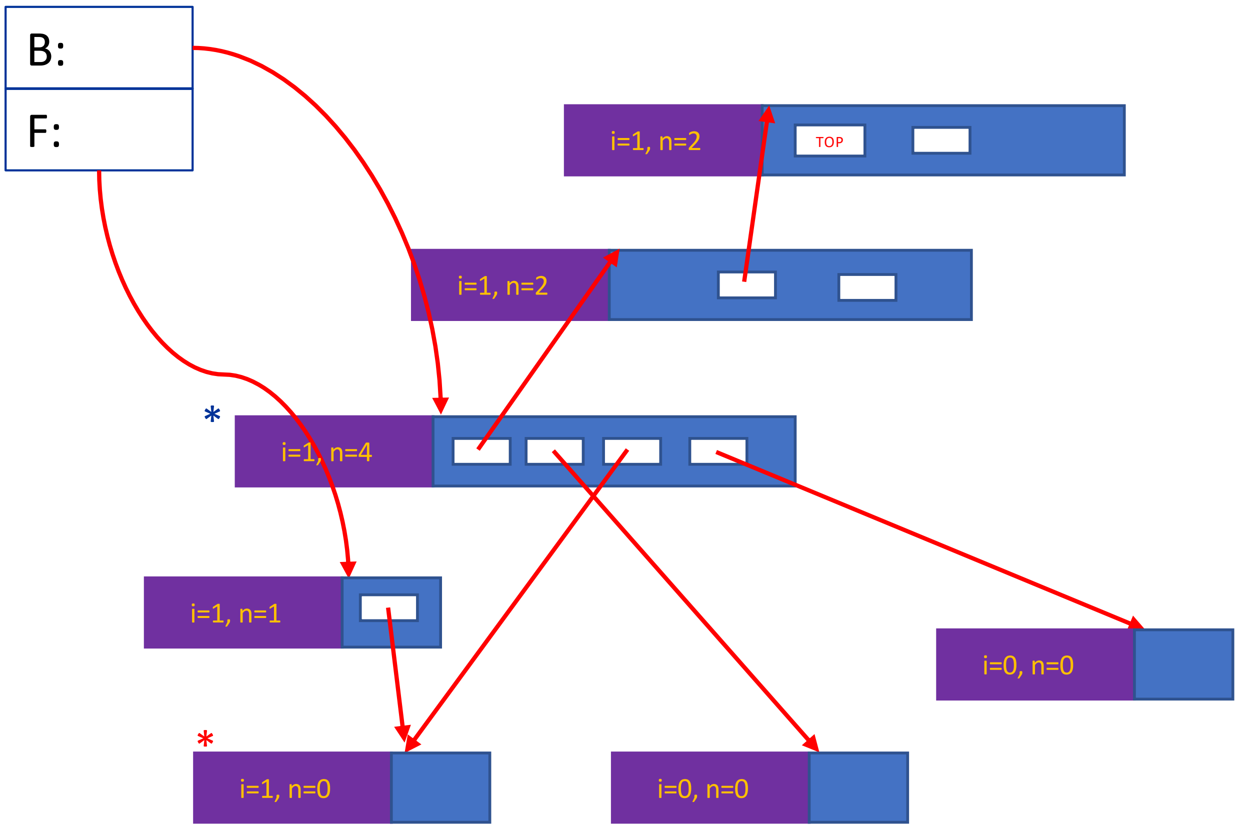
Each block that *mark()* has started to process has *i>0*, whereas blocks that have not yet been visited by *mark()* still have *i=0*

POINTER REVERSAL – VARIABLE SIZED BLOCKS



FUNCTIONAL PROGRAMMING
MARK-SCAN GARBAGE COLLECTION

POINTER REVERSAL – VARIABLE SIZED BLOCKS



After the red-starred block has been processed (it had no pointers to follow), *mark()* rewinds one step, as shown in this diagram

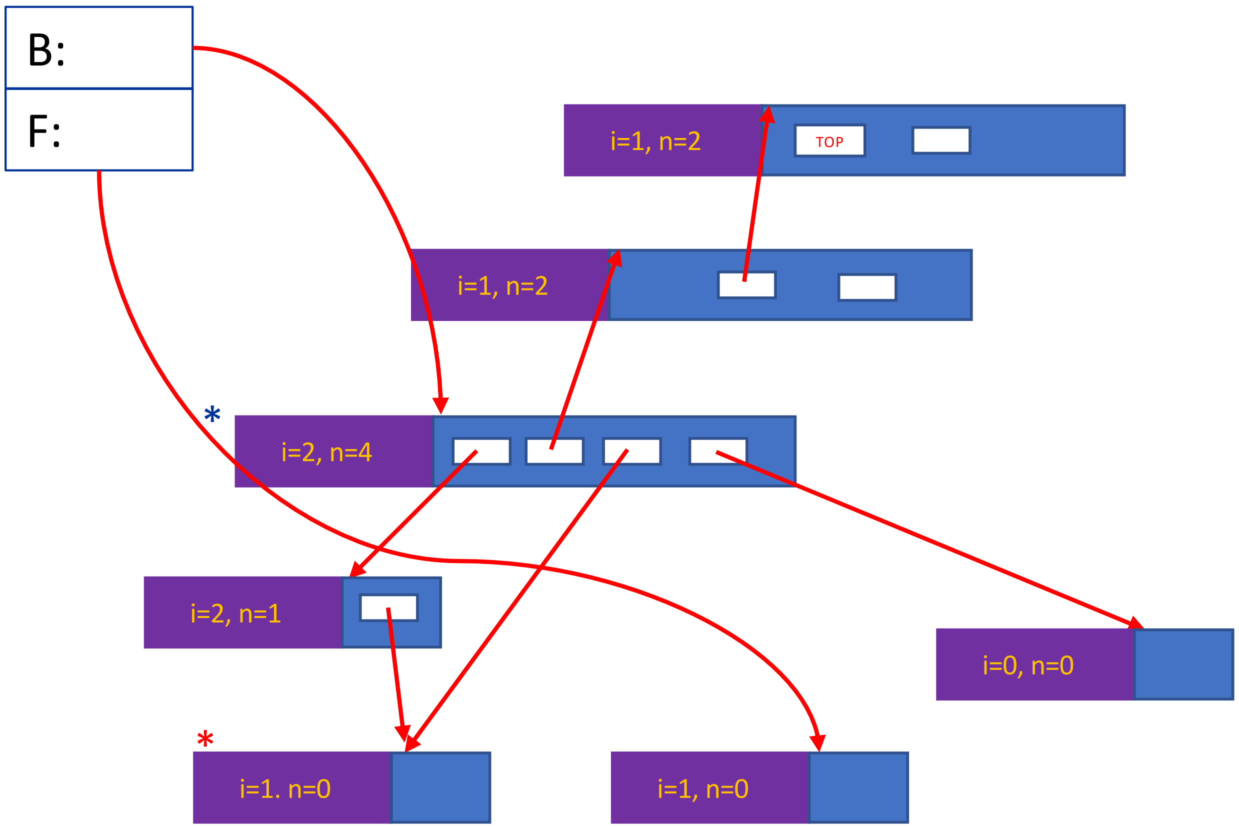
Now the red-starred block has *i>0*, indicating that *mark()* has started to process the block, and *i>n*, indicating that *mark()* has finished processing all pointers in the block

The blue-starred block has not quite finished processing its first pointer, so we still have *i=1*, which also serves to indicate that the <u>reversed</u> pointer is held in the location that previously held the first pointer (and that the address previously held in that location of the blue-starred block is currently held in "F")

The next step is for *mark()* to start to process the second pointer in the blue-starred block: this is shown on the next slide

FUNCTIONAL PROGRAMMING
MARK-SCAN GARBAGE COLLECTION

POINTER REVERSAL – VARIABLE SIZED BLOCKS



Here, *mark()* has started to process the second pointer in the data area of the blue-starred block

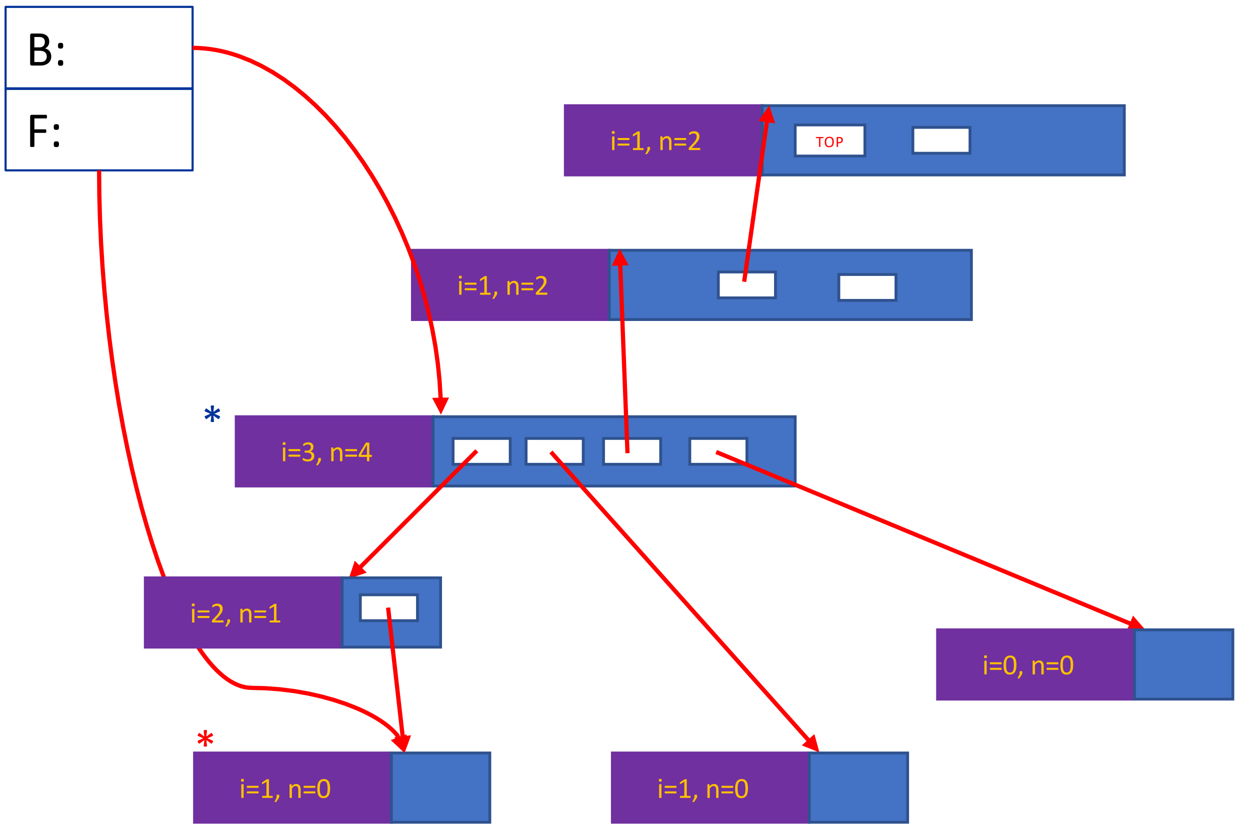
For the blue-starred block notice that it now has a value of *i=2*, which indicates that *mark()* is currently processing the second pointer in the data area, and also indicates that the reversed pointer is now held in the location that previously held the second pointer

Also notice that the first child block (that has now been fully processed) has a value of *i>n*, indicating that it has been fully processed

Mark() processes the second child block very quickly (it has no pointers in its data area) and needs to process the blue-starred block's next pointer. Using the address held in "B", *mark()* can see that the reversed pointer is currently held in the location of the second pointer, so it knows how to manipulate the pointers to start processing the next child block (see next slide)

FUNCTIONAL PROGRAMMING
MARK-SCAN GARBAGE COLLECTION

POINTER REVERSAL – VARIABLE SIZED BLOCKS



Here, *mark()* has started to process the third pointer in the data area of the blue-starred block

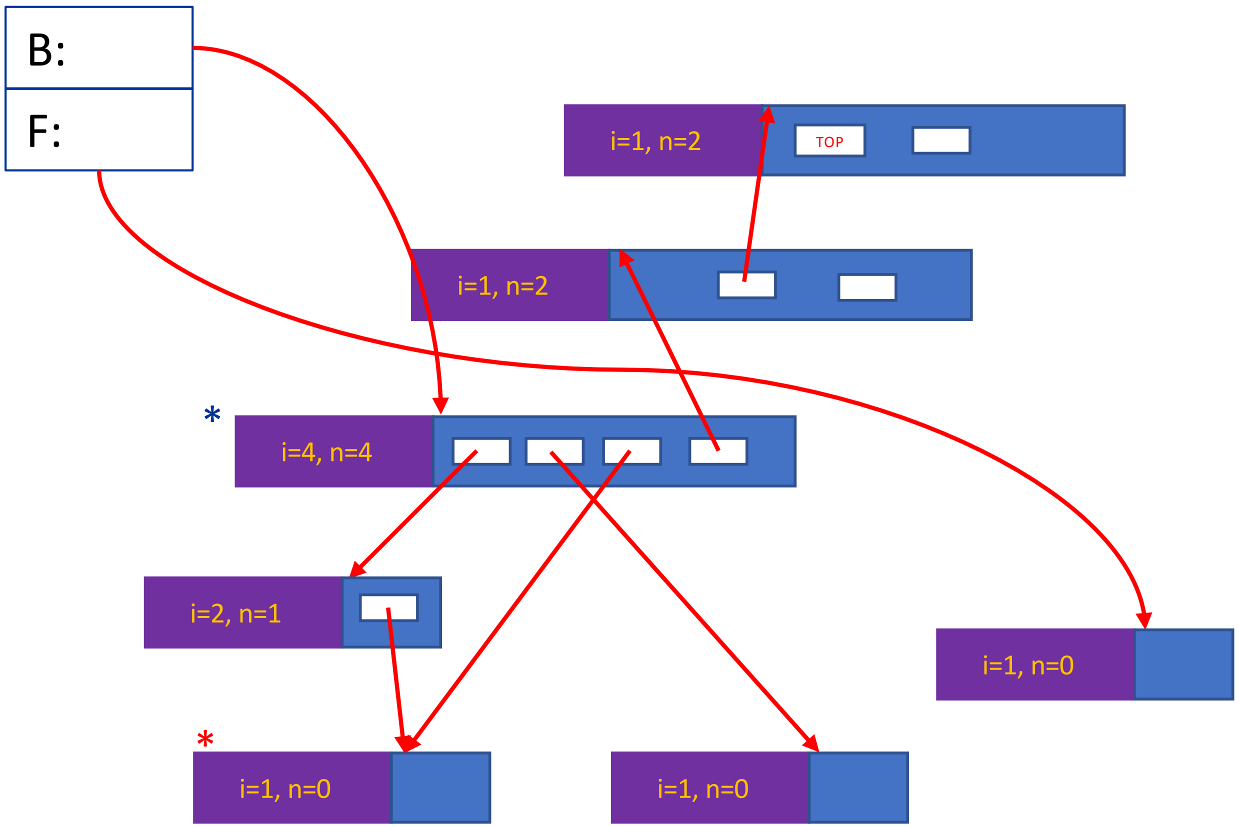
For the blue-starred block notice that it now has a value of i=3, which indicates that *mark()* is currently processing the third pointer in the data area, and also that the reversed pointer is now held in the location that previously held the third pointer

Mark() immediately sees that the block now pointed to by F has a value *i>n*, which means that it has previously been fully visited by *mark()* and does not need to be visited again

Mark() needs to move straight on to process the next pointer for the blue-starred block. Using the address held in "B", *mark()* can see that the reversed pointer is currently held in the location of the third pointer, so it knows how to manipulate the pointers to start processing the next child block (see next slide)

FUNCTIONAL PROGRAMMING
MARK-SCAN GARBAGE COLLECTION

POINTER REVERSAL – VARIABLE SIZED BLOCKS



On this slide, *mark()* has started to process the fourth and final pointer in the data area of the blue-starred block

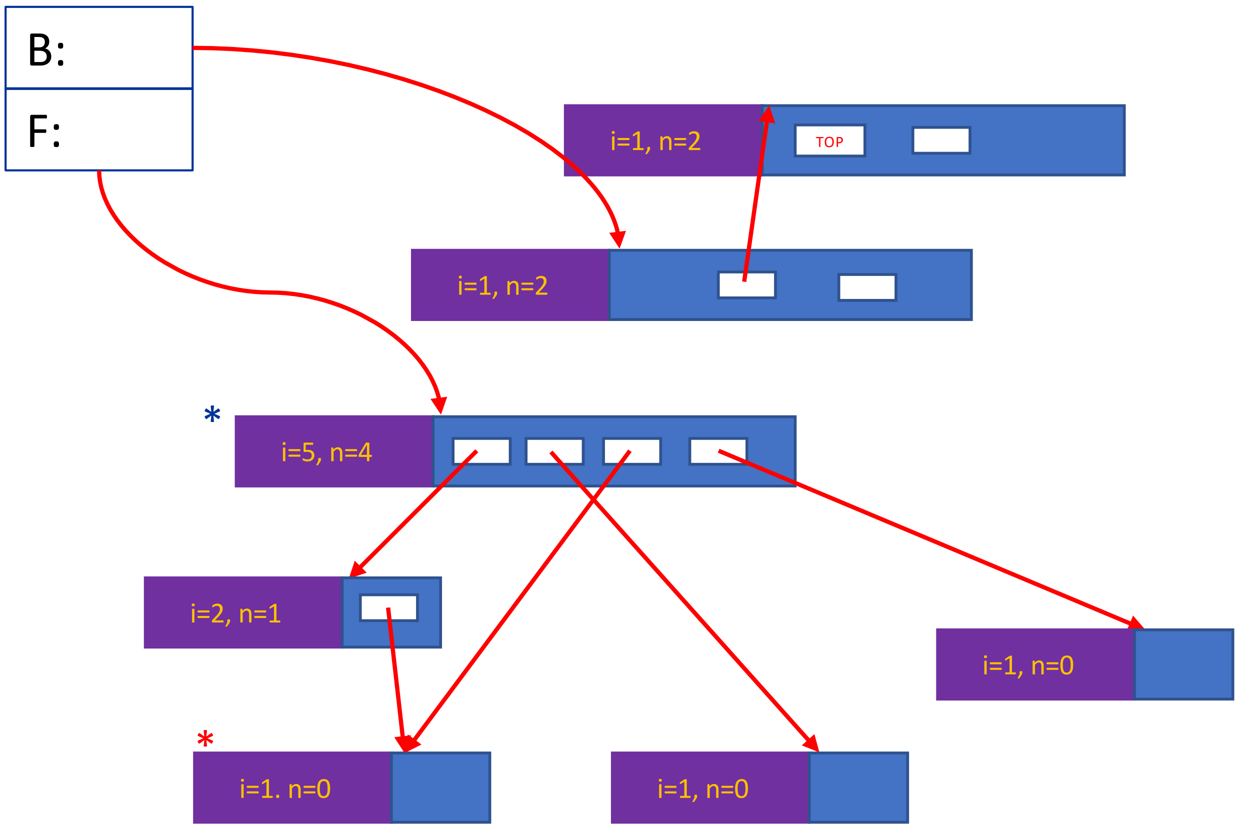
For the blue-starred block notice that it now has a value of i=4, which indicates that *mark()* is currently processing the fourth pointer in the data area, and also that the reversed pointer is now held in the location that previously held the fourth pointer

Mark() processes the fourth child block very quickly (it has no pointers in its data area) and needs to process the blue-starred block's next pointer (if any more exist). Using the address held in "B", *mark()* can see that the reversed pointer is currently held in the location of the fourth pointer, and can see that i=4 and n=4 so that it must have now processed the final child block for the blue-starred block. It knows how to manipulate the pointers to finish processing the blue-starred block and rewind upwards to consider its parent block (see next slide)

FUNCTIONAL PROGRAMMING
MARK-SCAN GARBAGE COLLECTION

Here, *mark()* has completely finished processing the blue-starred block. It has *i>0* which means it has been visited by *mark()* and it has *i>n* which means that all pointers in its data area have been fully processed

POINTER REVERSAL – VARIABLE SIZED BLOCKS



FUNCTIONAL PROGRAMMING
MARK-SCAN GARBAGE COLLECTION

POINTER REVERSAL: FOR AND AGAINST

For:

- PR only require small constant space
- Helpful when memory is low

Against:

- additional per-block space overhead (1 to $\lceil \log_2 N \rceil$ bits)
- much slower than stack traversal:
  - block visits: stack traversal = 2+
PR = (n+1) visits
  - more work: stack traversal = pop, mark, n*push
PR = update B, F, n*pointers, flags,...

For and against pointer reversal (PR)

For

- PR algorithms only require small constant space in which to operate
- this is helpful for garbage collection, since garbage collection is typically run when memory is running out

Against

- each block requires additional space overhead – at least a single bit to indicate when PR has unwound into the second pointer, and at worst $\lceil \log_2 N \rceil$ bits where *N* is the maximum number of pointers that can exist in the largest block
- the time performance of PR is much worse than standard stack traversal (perhaps 50% slower for shallow structures)

  Block visits:
  - stack traversal visits each block at least twice
  - yet PR visits each block at least (*n+1*) times (*n* is the number of pointers in the block), requiring extra memory fetches

  More work:
  - stack traversal pops a block pointer and then marks and pushes each child block
  - PR must update multiple values each time it visits a block (B, F, pointer fields in multiple blocks, flag bits in block headers, etc)

FUNCTIONAL PROGRAMMING
MARK-SCAN GARBAGE COLLECTION

LAZY SCANNING

- for a program that produces medium-large amounts of garbage, scan() much slower than mark()
  - because %age live blocks in heap is small
  - *mark()* only visits love blocks
  - *scan()* visit all blocks in heap
- speed is important, because "embarrassing pause" of evaluation during GC
- can reduce embarrassing pause by running *scan()* concurrently with evaluation (resume evaluation after *mark()*)
  - NB program never sees GC mark bits and can't access garbage blocks (so unaware of scan activities)

Lazy scanning

For a program that produces a medium to large amount of garbage, the scan phase is much slower than the mark phase. This is because the number of live blocks will be significantly smaller than the total number of blocks in the heap:

- the mark phase only visits live blocks (though the accesses are essentially random, so VM/cache performance may be problematic), whereas
- the scan phase must visit **all** blocks in the heap (though access is sequential, which should be better for VM/cache performance)

Recall that evaluation of the program normally pauses while garbage collection is performed. And from the above we know that often the scan phase contributes most to that pause

This "embarrassing pause" can be reduced if the scan phase can be performed lazily, interspersed with normal program evaluation. The cost is a slight increase in complexity and a slight reduction in performance of the running program while the scan phase is being completed incrementally

Notice that the program never sees the garbage collection mark bits, and cannot access garbage blocks (so is unaware of scan activities like linking garbage blocks onto the free list)

FUNCTIONAL PROGRAMMING
MARK-SCAN GARBAGE COLLECTION

LAZY SCANNING: IMPERATIVE PSUEDO-CODE

```
malloc() =
   while S < HeapEndAddress {
      markbit = read(S)
      write(S,0)
      result = S+Headersize
      S = S + Headersize + Datasize
      if (markbit == 0) return(result)
   }
   mark()
   while S < HeapEndAddress {
      markbit = read(S)
      write(S,0)
      result = S+Headersize
      S = S + Headersize + Datasize
      if (markbit == 0) return(result)
   }
   abort "Out of memory"
```

Lazy scanning continued

The simplest way to evaluate the program and perform the scan phase simultaneously is to do a fixed amount of scanning each time *malloc()* is called

Pseudo-code for Hughes's lazy scanning algorithm for **fixed-size** blocks **with no free list** is illustrated on this slide (based loosely on Jones & Lins 1996, Page 89)

The variable S is defined globally (its value persists between calls to *malloc()*) and is set to HeapStartAddress at the start of program evaluation

In this simple version of the code:

- If the block is free, the statement write(S,0) is unnecessary but does no harm (except to reduce performance)
- If the block is live, the statement result=S+Headersize is unnecessary but does no harm

If a free block is not found in the first scan of the heap, *mark()* is called to mark all live cells and then the scan is repeated. If the second scan fails, the heap is clearly out of memory

Notice how this lazy scanning strategy does not need a free list!

Out of scope for this module (2022), but interesting: can you modify this code so that it works for variable sized blocks? You will need to use a free list – how will you manage that free list?

FUNCTIONAL PROGRAMMING
MARK-SCAN GARBAGE COLLECTION

FOR AND AGAINST MARK-SCAN GC

Against	For
mark() visits every live block, and scan() visits every block in the heap	low admin space overheads (per-block and overall), and no overhead on copying/deleting pointers to blocks
GC will become more frequent as the "residency" increases (% heap taken up by live blocks)	the code for mark() and scan() is simple and small, taking up little memory
fragmentation may occur, and there is no (natural) compaction	It is an in-place technique: live blocks do not move, and a compactor can be run separately
	The performance impact of interaction with VM and caching systems (during the running of *mark()* and *scan()*) is surprisingly good
	It is able to recover cyclic structures in the heap (e.g. cyclic data structures)
There is an embarrassing pause while the GC runs, which is not good for real-time, highly-interactive or distributed systems	though Mark-Scan GC can be modified to run concurrently with program evaluator (if they communicate with each other)
	It can be modified to use a bitmap of mark bits (instead of keeping them in block headers), which can further improve speed

For and Against

mark() visits every live block, and *scan()* visits every block in the heap, though space overheads are low (per-block and overall), and no overhead is placed on copying or deleting pointers to blocks

GC will become more frequent as the "residency" increases (the percentage of the heap taken up by live blocks)

Code is simple and small, taking up little memory

It is an in-place technique: live blocks do not move, fragmentation may occur, and there is no compaction (though a compactor can be run separately)

The performance impact of the algorithm's interaction with VM and caching systems (during the running of *mark()* and *scan()*) is surprisingly good

It is able to recover cyclic structures in the heap (e.g. cyclic data structures) – whereas other GC techniques may not

There is an embarrassing pause while the GC runs, which is not good for real-time, highly-interactive or distributed systems, though Mark-Scan GC can be modified to run concurrently with program evaluator (if they communicate with each other)

Mark-Scan GC can be modified to use a bitmap of mark bits (instead of keeping them in block headers), which can further improve speed

FUNCTIONAL PROGRAMMING
MARK-SCAN GARBAGE COLLECTION

Mark-Scan and Miranda

Miranda uses Mark-Scan garbage collection with lazy scanning, and Miranda's implementation is completely stackless – it uses pointer-reversal both for marking and for its execution stack

MARK-SCAN AND MIRANDA

- Miranda uses Mark-Scan garbage collection with lazy scanning
- Miranda's implementation is completely stackless – it uses pointer-reversal both for marking and for its execution stack

FUNCTIONAL PROGRAMMING
MARK-SCAN GARBAGE COLLECTION

SUMMARY

- Overview
- Assumptions
- Simple imperative pseudo-code
- Using a marking stack
- Pointer reversal
- Lazy scanning
- For and against Mark-Scan GC
- Mark-Scan and Miranda

In summary, this lecture has explored the first of three canonical garbage collection algorithms – Mark Scan Garbage Collection

The lecture provided simple imperative pseudo-code and discussed implementation issues such as the use of a marking stack versus the use of pointer reversal, and the use of a variant of the scan phase known as "lazy scanning"

The lecture then ended with a summary of good and bad characteristics of Mark-Scan Garbage Collection, and a few words about its use in Miranda