

Functional Programming

Christopher D. Clack

FUNCTIONAL PROGRAMMING

Lecture 25

COPYING GARBAGE COLLECTION

FUNCTIONAL PROGRAMMING

COPYING GARBAGE COLLECTION

CONTENTS

- Overview
- Assumptions
- Simple imperative pseudo-code
- Cheney's iterative copying garbage collector
- For and against Copying GC

In this lecture we will explore the second of three canonical garbage collection algorithms – Copying Garbage Collection

The lecture provides simple imperative pseudo-code for a recursive version of copying GC and then presents Cheney's iterative copying garbage collection algorithm, together with a worked example

The lecture ends with a summary of good and bad characteristics of Copying Garbage Collection

FUNCTIONAL PROGRAMMING

COPYING GARBAGE COLLECTION

OVERVIEW

- Copying GC triggered by *malloc()* when free memory becomes low
- Program evaluation typically pauses during GC
- Garbage is detected by tracing all live pointers & copying all live blocks to a different part of memory
- Garbage is collected / made available for re-use by making its memory available after the next GC
- After GC, in the part of memory from which allocation will be satisfied, all live blocks are at one end of memory and all free blocks at the other end – pointer increment allocation can be used
- Live blocks are copied: after GC there is no fragmentation

Triggering the garbage collection

Copying 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 Copying garbage collection algorithm automatically identifies garbage by following all live pointers and copying to another part of memory every block of memory that can be reached by following all live pointers

Collecting garbage

The Copying algorithm makes garbage available for re-use after the next GC cycle. It does this by alternating the part of memory from which allocation is satisfied

- initially allocate from Part 1
- after the 1st GC, allocate from Part 2
- after the 2nd GC, allocate from Part 1

Interaction with memory allocation

After GC, in the part of memory from which allocation will be satisfied, all live blocks are at one end of memory and all free blocks at the other end – pointer increment allocation can be used

Fragmentation

Live blocks are copied: immediately after GC there is no fragmentation

FUNCTIONAL PROGRAMMING

COPYING GARBAGE COLLECTION

ASSUMPTIONS

- *malloc()* allocates free memory using pointer increment
- Memory is divided into two semi-spaces: one contains current data, the other contains obsolete data
- all live blocks are reachable by tracing all live pointers
- can distinguish pointers from other data
- all live pointers can be found by tracing from a "Root Set" of pointers
- each block header contains a "forwarding" bit
- evaluation pauses during GC

Assumptions

Copying garbage collection assumes:

- normally, the memory allocator uses pointer-increment allocation
- Memory is divided into two semi-spaces: one contains current data, the other contains obsolete data
- 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 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 called the "Root Set"
- all blocks have an additional bit in the block header that indicates whether a forwarding address exists for this block (see later)
- that (in the simple version of Copying GC) program evaluation pauses during GC and resumes after the GC has finished

FUNCTIONAL PROGRAMMING

COPYING GARBAGE COLLECTION

SIMPLE IMPERATIVE PSEUDO-CODE


init() =

```
tospace = HeapStartAddress
fromspace = tospace + (Heapsize/2) + 1
free = tospace
top_of_space = tospace + (Heapsize/2)
```

malloc(n) =

```
if (free + n > top_of_space) flip()
if (free + n > top_of_space) abort "Out of memory"
result = free + Headersize
write(free, (0, n))
free = free + n + Headersize
return (result)
```

Initialise the header with 0 ("not forwarded") and the size of the data area



flip() =

```
tmp = fromspace
fromspace = tospace
tospace = tmp
top_of_space = tospace + (Heapsize/2)
free = tospace
for each R in RootSet { R = copy(R) }
```

Copying GC requires an initialisation function just before the program starts to be evaluated, a pointer-increment memory allocator, and a GC function during which live blocks are copied from one semi-space to another. The pseudo-code (left) is based on Jones & Lins 1996, Pages 29 & 30)

Init(): One semi-space (called "tospace") initially occupies the lower half of heap memory, and the other ("fromspace") initially occupies the top half of heap memory. Global variables *tospace*, *fromspace*, *free* and *top_of_space* are initialised

malloc(): this code first checks whether allocation of the required block of size *n* would exceed the end of *tospace* (which is the semi-space from which allocation is always satisfied) – if so, it calls the GC function *flip()*. It then makes the same check again – if the test fails again (despite GC), it causes the program to abort with an "Out of memory" error. Otherwise, it uses very fast pointer-increment allocation where *free* acts as the TOP pointer mention in Lecture 23

flip(): the flip() code starts by flipping the global variables *fromspace* and *tospace*, so that they now refer to the opposite semi-space. The global variables *free* and *top_of_space* are reset appropriately and then all live blocks in *fromspace* are copied to *tospace* – this is done by transitively following all pointers in RootSet and copying every block found

FUNCTIONAL PROGRAMMING

COPYING GARBAGE COLLECTION

SIMPLE IMPERATIVE PSEUDO-CODE

```
copy(P) =  
  if (atomic(P) or (P==nil)) return(P)  
  (forwarded, size) = read(P - Headersize)  
  forwardingaddress = read(P)  
  if (forwarded == 0) {  
    newaddr = free  
    free = free + size + Headersize  
    forwardingaddress = newaddr + Headersize  
    temp = read(P)  
    write(P, forwardingaddress)  
    write(P-Headersize, 1)  
    write(forwardingaddress, copy(temp))  
    for i = 1 to n-1 {  
      temp = copy(read(P+i))  
      write(forwardingaddress+i, temp)  
    }  
  }  
  return(forwardingaddress)
```

reserve space before any recursive call to copy()

setup forwarding address before any recursive call to copy()

The copy() function is complex – it must ensure that the topology of shared structures (including cyclic pointers) are copied faithfully. The recursive pseudo-code shown on this slide is loosely based on Jones&Lins 1996 – Page 30

Preservation of sharing is achieved by leaving a "forwarding address" in each *fromspace* block when it is copied – this is the address in *tospace* of the copy of that block. Whenever a block in *fromspace* is visited, *copy()* checks to see whether it has already been copied – if so, the forwarding address is returned rather than allocating space in *tospace* (notice how allocating memory in *tospace* is done directly, not using *malloc()*). To ensure termination and correct sharing, the forwarding address is set to point to newly allocated memory before data in the *fromspace* block are copied

The forwarding address can be held in the first word of the data area of the block being copied

Notice that the value *P* might be:

- a non-pointer ("atomic") or the value *nil* (in which case *copy(P)* returns *P*),
- a pointer to a block that has already been copied (in which case *copy(P)* returns the forwarding address), or
- a pointer to a block (in which case *copy(P)* copies it to *tospace*, sets the forwarding address, and returns the forwarding address)

FUNCTIONAL PROGRAMMING

COPYING GARBAGE COLLECTION

CHENEY'S ITERATIVE COPYING GC

flip() =

```
(fromspace, tospace) = (tospace, fromspace)
top_of_space = tospace + (Heapsize/2)
free = scan = tospace
for each R in RootSet { R = copy(R) }
while (scan < free) {
    (forwarded, size) = read(scan)
    for M in childrenlocs(scan) { write(M, copy(read(M))) }
    scan = scan + size + H }
```

copy(P) =

```
(forwarded, n) = read(P - H)
addr = read(P)
if (forwarded == 1) { return(addr) }
addr = free + H
free = addr + n
for i = 0 to n-1+H { write(addr-H+i, read(P+i-H)) }
write(P, addr)
write(P - H, 1)
return(addr)
```

swap
fromspace/tospace



The previous slide gave simple recursive imperative pseudo-code, but recall from the last lecture that recursive code can be slow and risks the stack overflowing

Cheney's algorithm provides iterative Copying GC. Instead of storing branch points of the active graph of live blocks in a stack, it uses a queue. Pointers *scan* and *free* point to each end of this queue, and the queue is the set of new nodes in *tospace*

The function *copy()* is now non-recursive. It copies a block into *tospace* but does not copy any pointers held in its data area. Any block existing in *tospace* between *scan* and *free* is considered to be only partly copied. The value *H* is Headersize

The processing of pointers held in the data areas of blocks copied into *tospace* is now effected by the function *flip()*, which iteratively looks at every block between *scan* and *free*, calling *copy()* to make copies into *tospace* of any pointers that are found

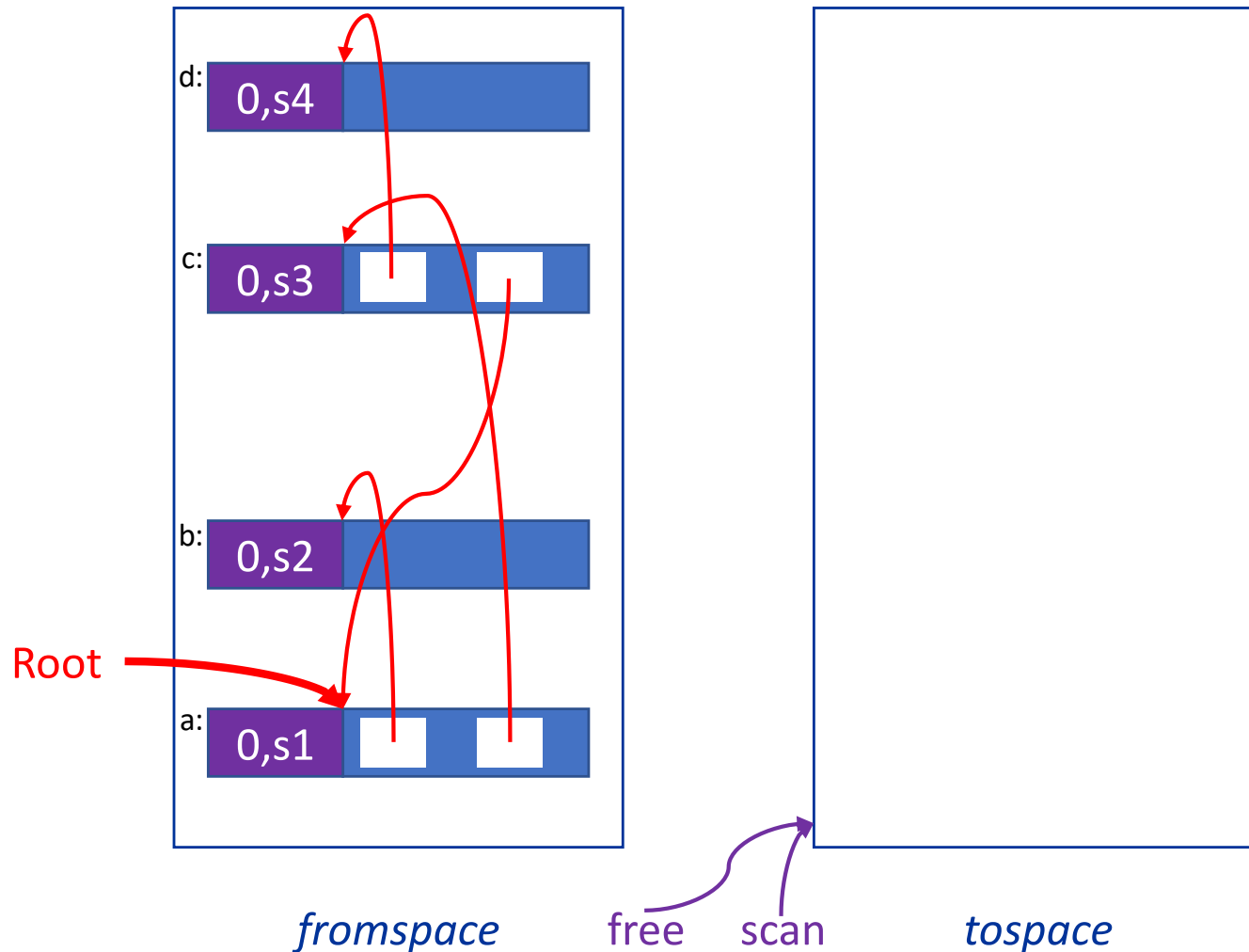
Notice that every time *flip()* calls *copy()*, the value of *free* may be increased as the result of a new block being copied into *tospace*. Thus, *flip()* is continually playing catch-up, moving *scan* forwards towards a moving *free*. Eventually *free* stops moving and *scan* catches up, at which point the GC has finished

The following slides provide a worked example of Cheney's iterative algorithm for Copying GC

FUNCTIONAL PROGRAMMING

COPYING GARBAGE COLLECTION

CHENEY'S ITERATIVE COPYING GC



This slide illustrates the memory layout after *flip()* has been called, the definitions of *fromspace* and *tospace* have been swapped, and the variables *scan* and *free* have been initialised to the first free location in *tospace*

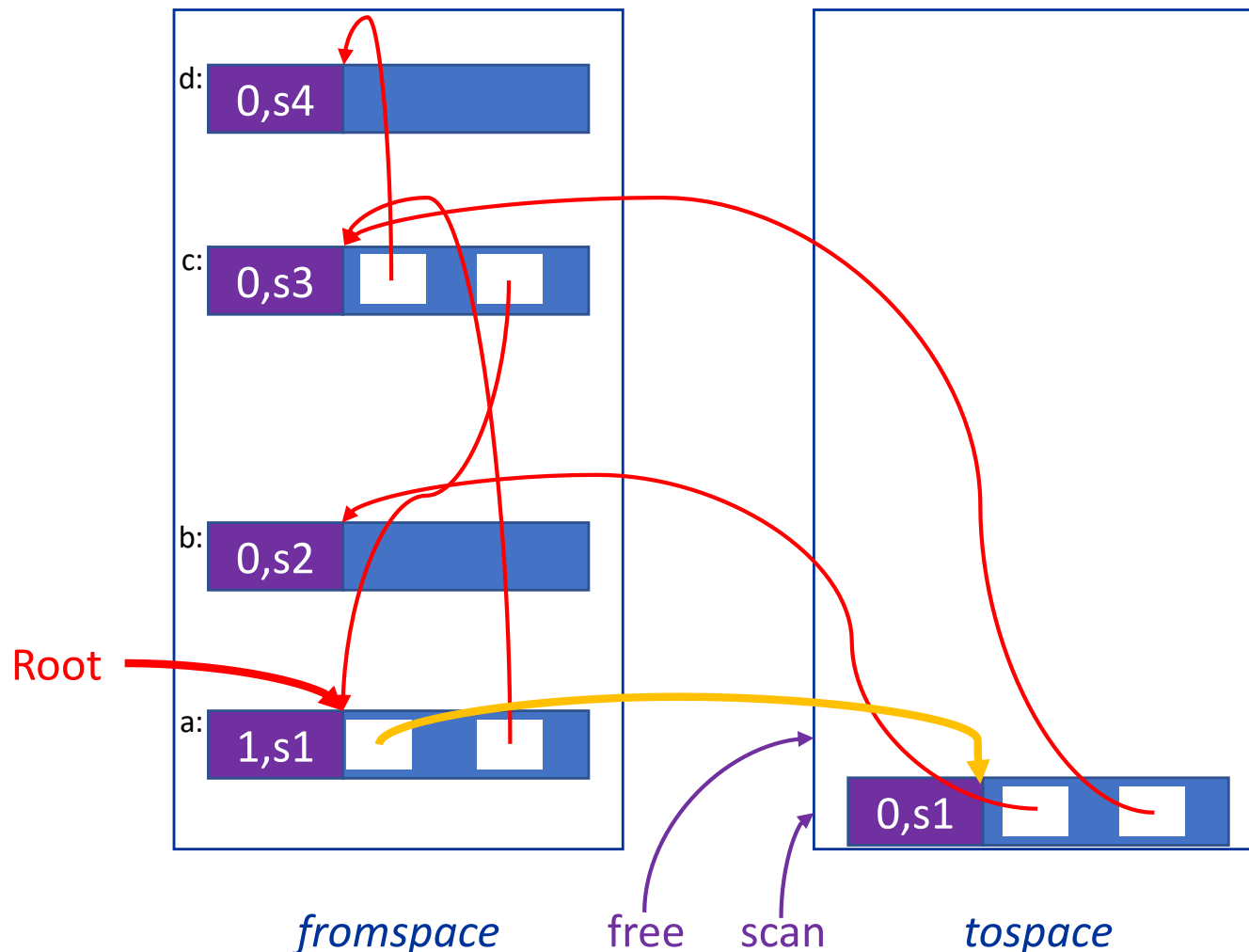
There are four live blocks in *fromspace* that need to be copied into *tospace*, and the Root Set only contains one pointer. We have annotated the blocks a, b, c and d

flip() is about to call *copy()*, passing it the root pointer which is the address of the data area of block "a"

FUNCTIONAL PROGRAMMING

COPYING GARBAGE COLLECTION

CHENEY'S ITERATIVE COPYING GC



The function *copy()* reads the *forwarded* bit 0 and the *size* *s1* from the header of block a

forwardingaddress is set to the contents of the first address of the data area of block a

forwarded=0, so *copy()* creates a new block of size (*s1* + H) in *tospace*, starting at address *free* and copying every word from block a (including its header) to *tospace*

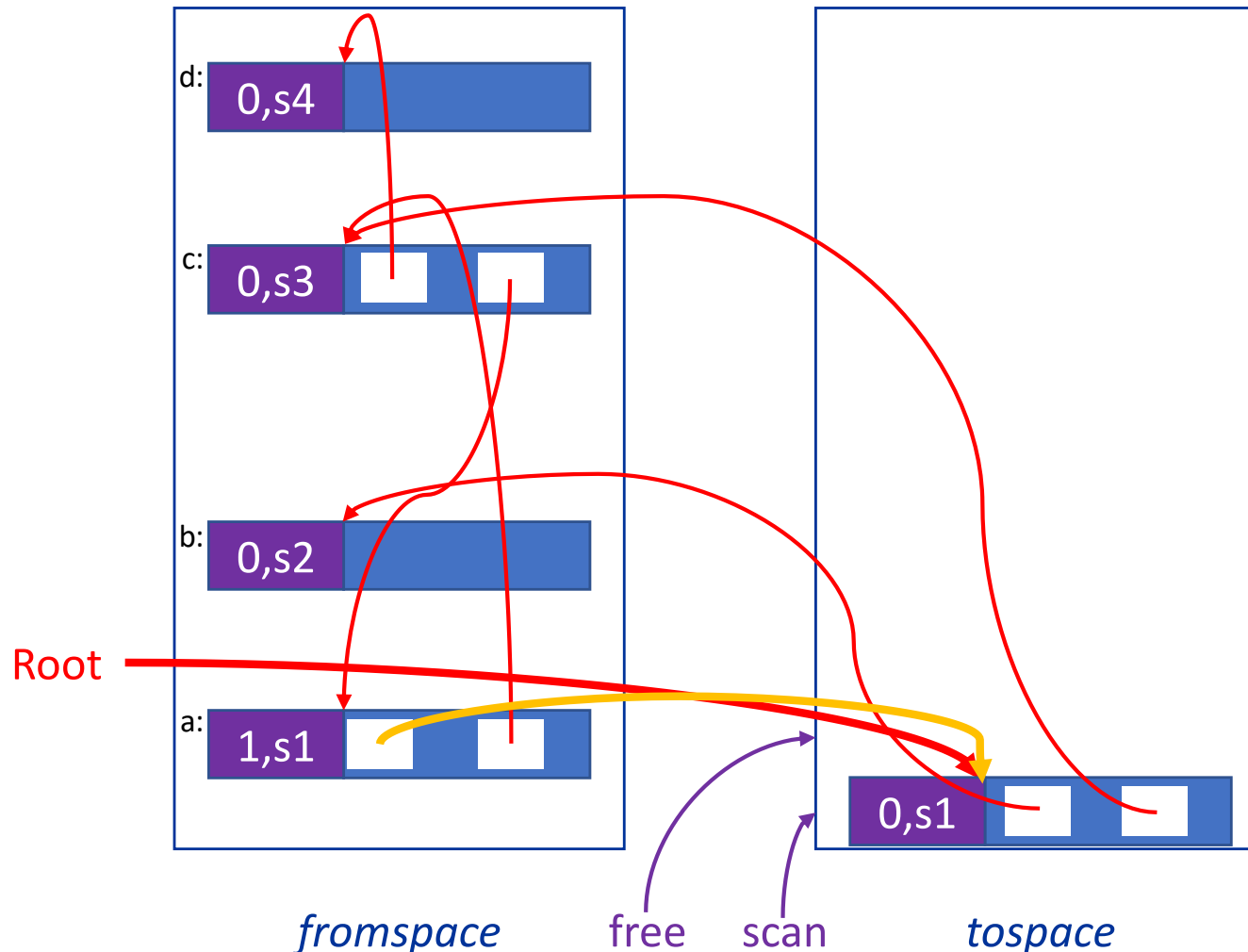
The block "a" in *fromspace* has had its header updated to indicate the existence of a forwarding address, the forwarding address (the orange pointer) has overwritten the first word of block a's data area

free has been updated to point to the first free block above the new copy of block "a" in *tospace*

FUNCTIONAL PROGRAMMING

COPYING GARBAGE COLLECTION

CHENEY'S ITERATIVE COPYING GC



After the function `copy()` returns the forwarding address of the new copy in *tospace* of block "a", the function `flip()` first updates the root pointer and then enters the while loop that does the scanning

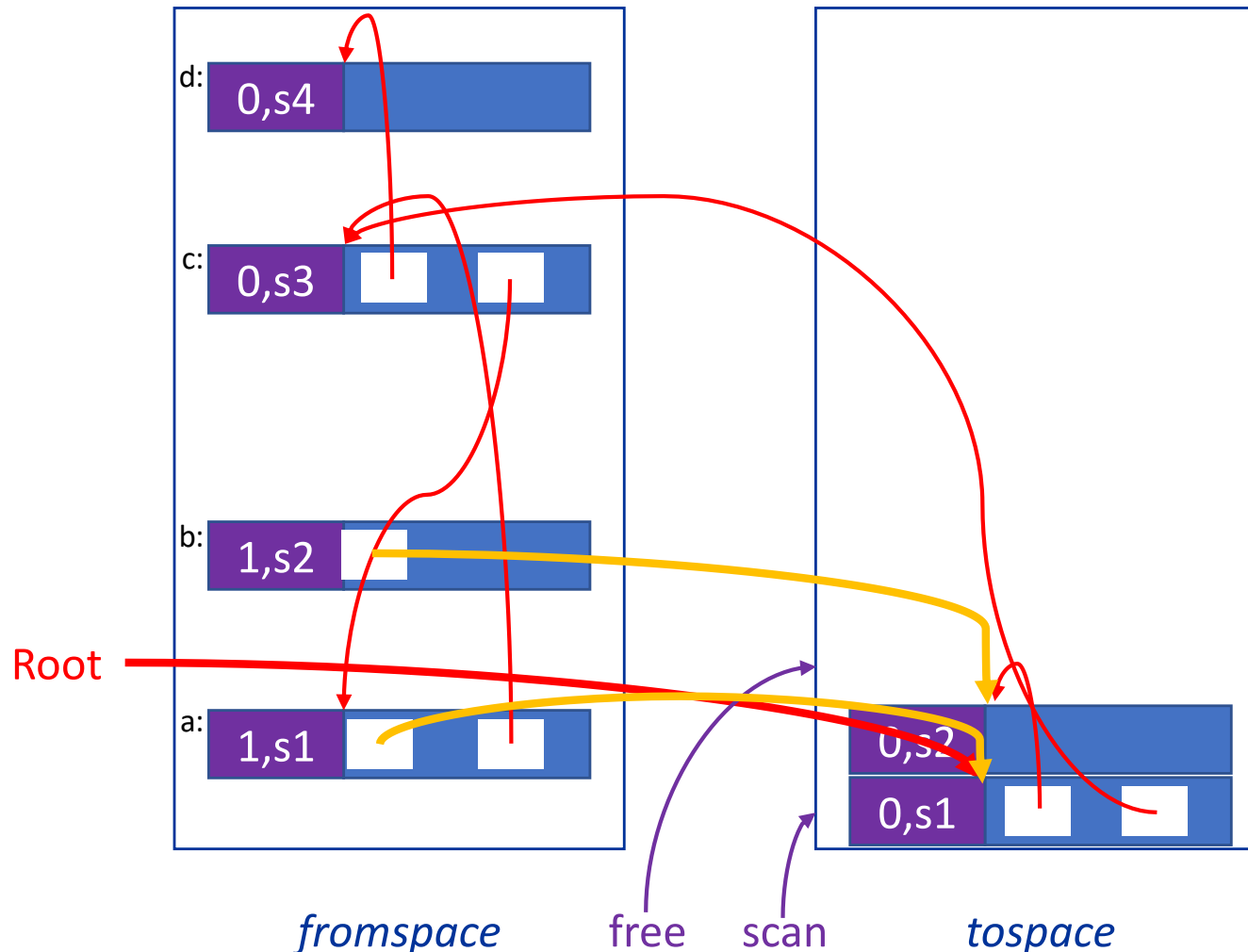
Notice that *scan* is the address of the header of the new copy in *tospace* of block "a". `flip()` calls `childrenlocs(scan)` to look in the data area of this new copy of block "a" to find the set of locations in that data area that contain pointers (pointing back into *fromspace*)

This figure shows the memory layout when the first of those pointers (to block b in *fromspace*) has been found and `flip()` is about to call `copy()`, passing it the address in *fromspace* of block "b"

FUNCTIONAL PROGRAMMING

COPYING GARBAGE COLLECTION

CHENEY'S ITERATIVE COPYING GC



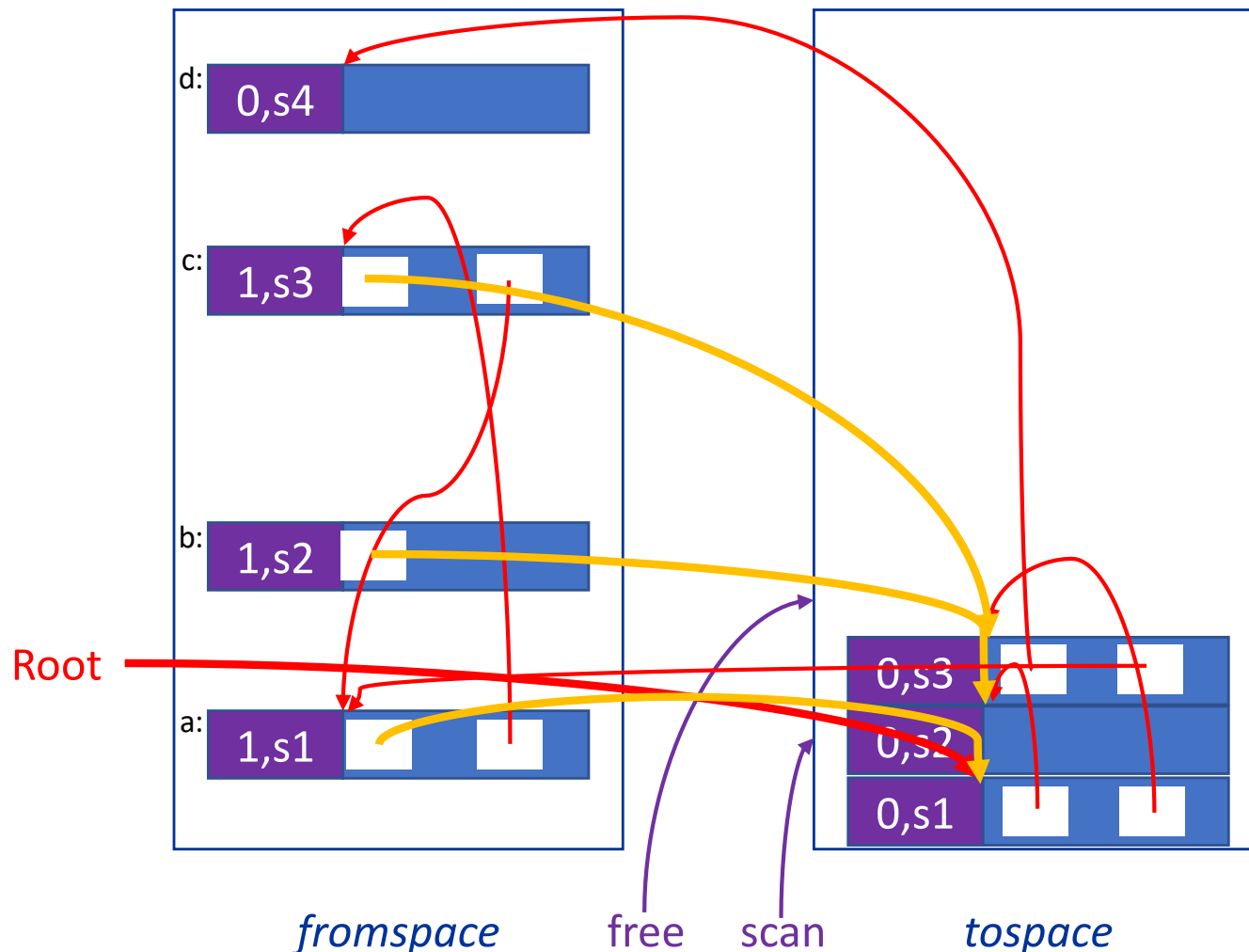
This figure shows the memory layout after *copy()* has copied block "b" from *fromspace* to *tospace*, *free* has been update to point to the next free location in *tospace*, the header for block "b" in *fromspace* has been updated, a forwarding pointer has been saved at the start of its data area, *copy()* has returned the forwarding pointer and this has been used to update the appropriate location in the copy of block "a" held in *tospace*

The function *flip()* will now proceed to the next iteration in its for loop, to process the second pointer held in the *tospace* copy of block "a" (see next slide)

FUNCTIONAL PROGRAMMING

COPYING GARBAGE COLLECTION

CHENEY'S ITERATIVE COPYING GC



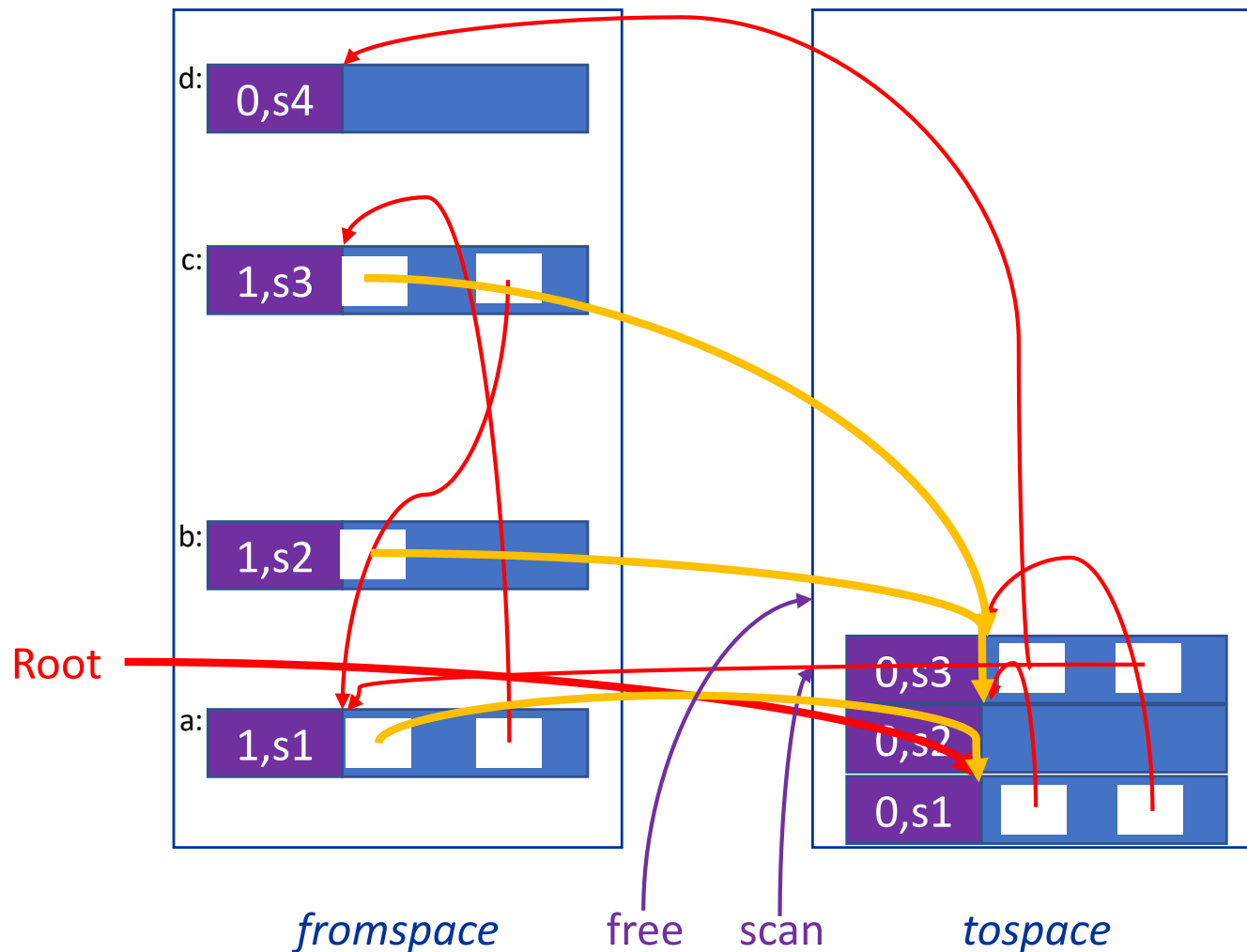
This figure shows the memory layout after the second pointer in the data area of the *tospace* copy of block "a" has been processed, *copy()* has returned the forwarding address and the location in the data area holding the pointer has been updated

In this diagram *flip()* has also just updated the variable *scan* so that it points to the start of the next block in *tospace*. Because *scan* has not yet caught up with *free*, *flip()* will execute another iteration of its while loop to process the next block. This of course has no child pointers to process, and so *scan* will be updated (see next slide)

FUNCTIONAL PROGRAMMING COPYING GARBAGE COLLECTION

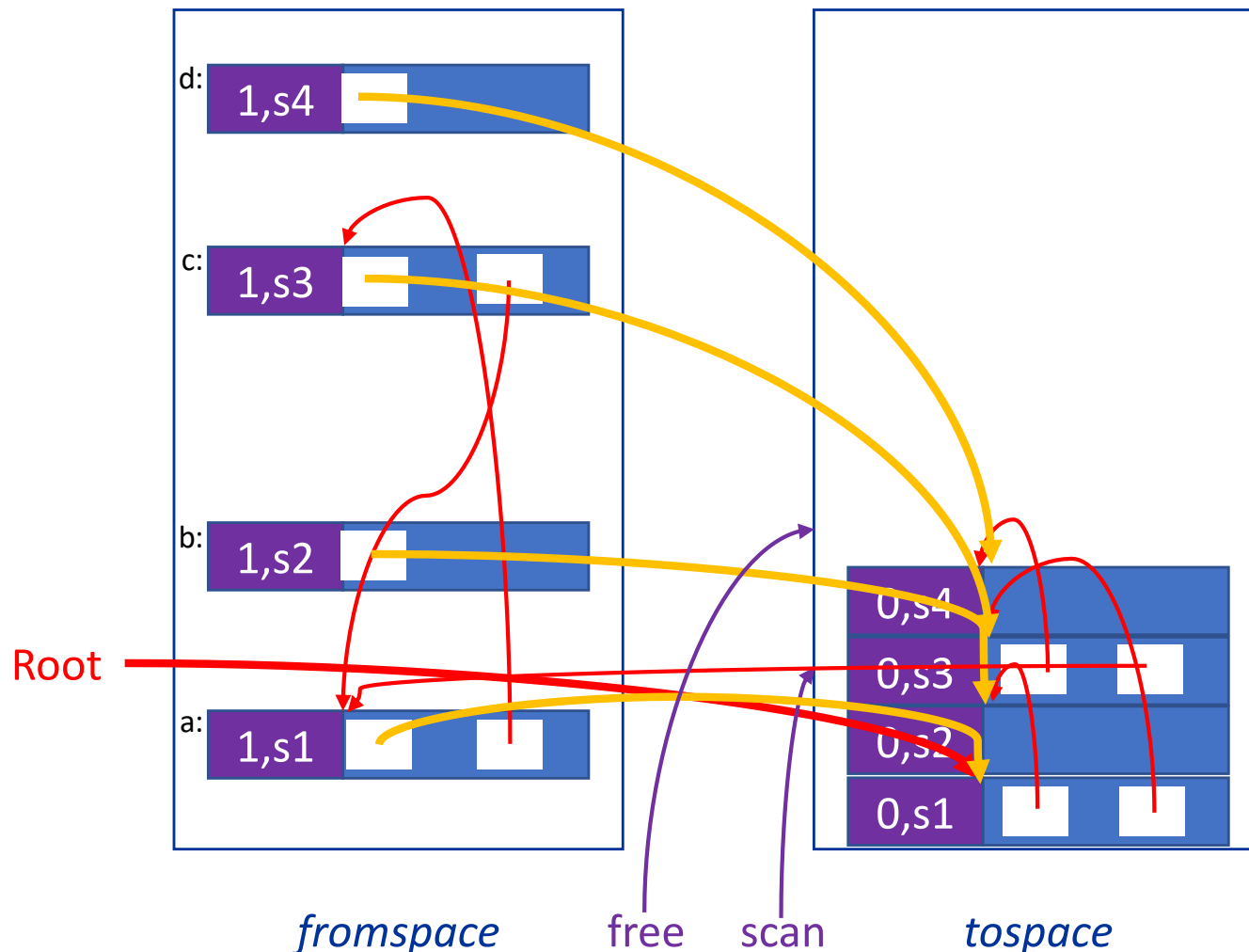
Here *scan* has been updated and has not yet caught up with *free*: there is another iteration of the while loop that must be processed, to check the child pointers inside the *tospace* copy of block "c" (see next slide)

CHENEY'S ITERATIVE COPYING GC



FUNCTIONAL PROGRAMMING COPYING GARBAGE COLLECTION

CHENEY'S ITERATIVE COPYING GC



The call to `copy()` to process the first pointer held in the data area of the *tospace* copy of block "c" has caused a copy of block "d" to be made in *tospace*, and its return value of the forwarding pointer has been used to update the pointer held in the data area of the *tospace* copy of block "c"

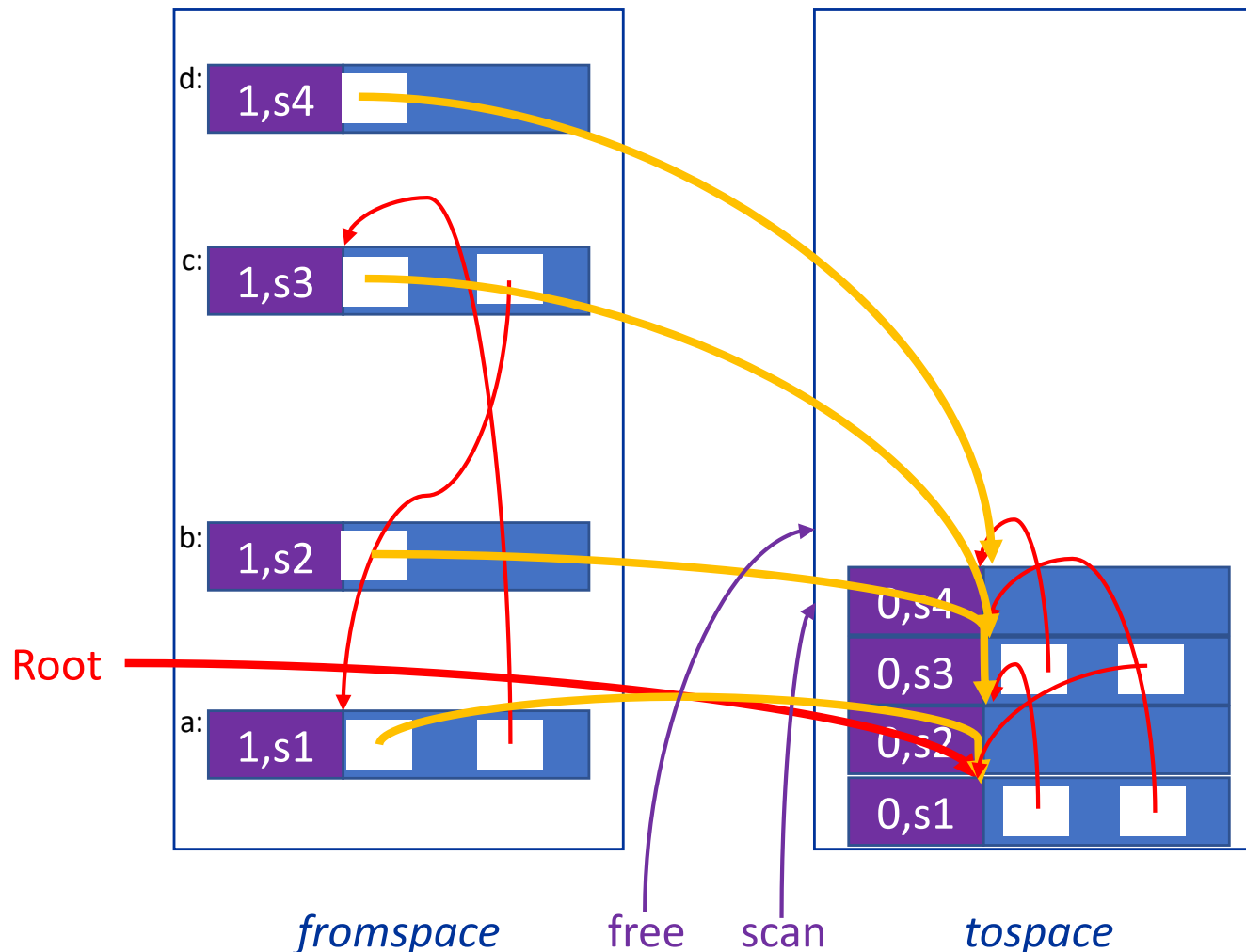
The header of block "d" has been updated, and a forwarding pointer has been written into the first address of its data area

`flip()` has not yet updated *scan* because it must finish the for loop, which must process the second pointer in the *tospace* copy of block "c" (see next slide)

FUNCTIONAL PROGRAMMING

COPYING GARBAGE COLLECTION

CHENEY'S ITERATIVE COPYING GC

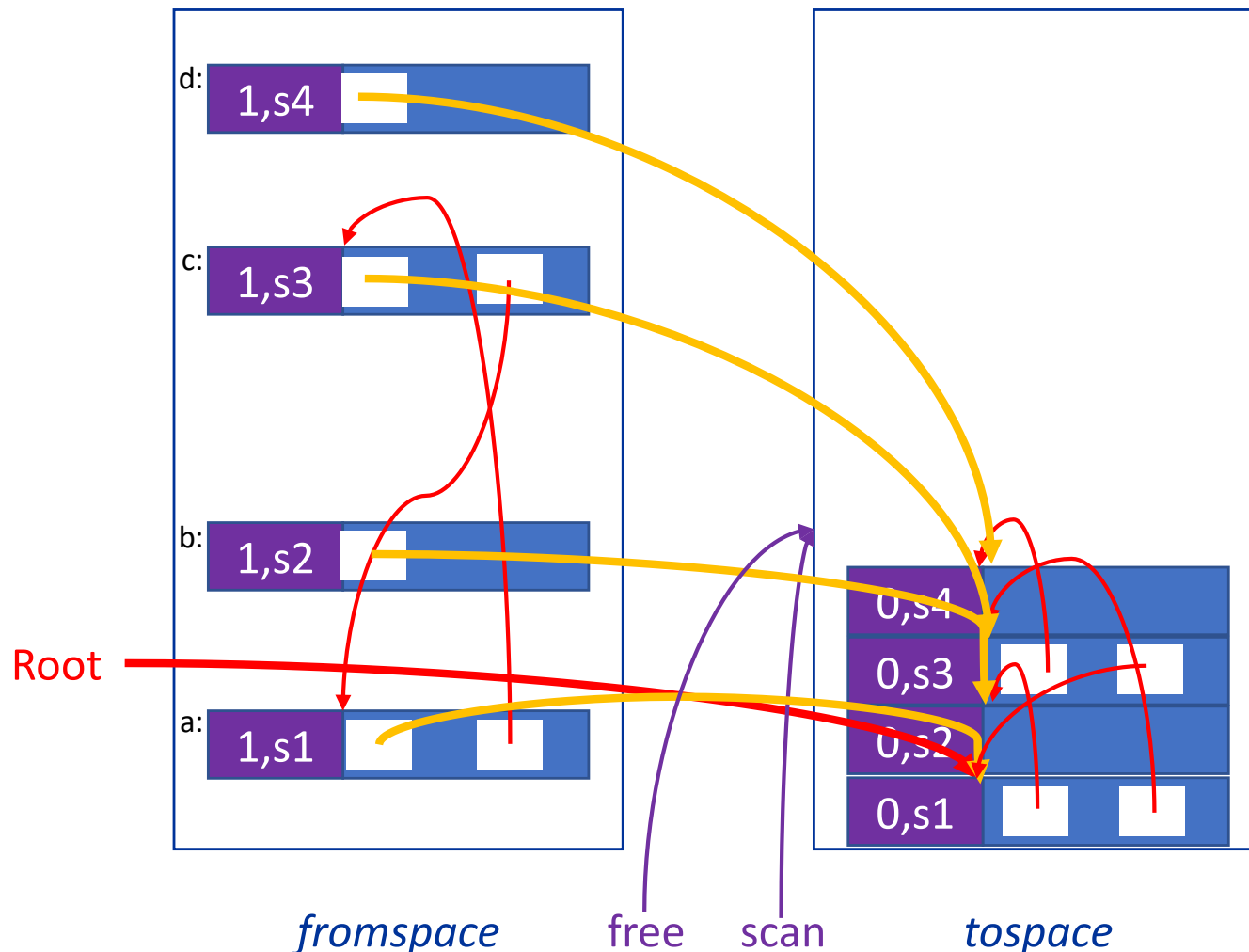


The call to *copy()* to process the second pointer held in the data area of the *tospace* copy of block "c" has read the header of the *fromspace* copy of block "a" and discovered that it has a forwarding address – *copy()* therefore returned the forwarding address, and *flip()* caused the second pointer in the data area of the *tospace* copy of block "c" to be updated to be that forwarding address

flip() has updated scan to point to the copy in *tospace* of block "d"

FUNCTIONAL PROGRAMMING COPYING GARBAGE COLLECTION

CHENEY'S ITERATIVE COPYING GC



Processing the children of the *tospace* copy of block "d" was easy – there were no children!

Because no new blocks were copied into *tospace*, *free* has not changed

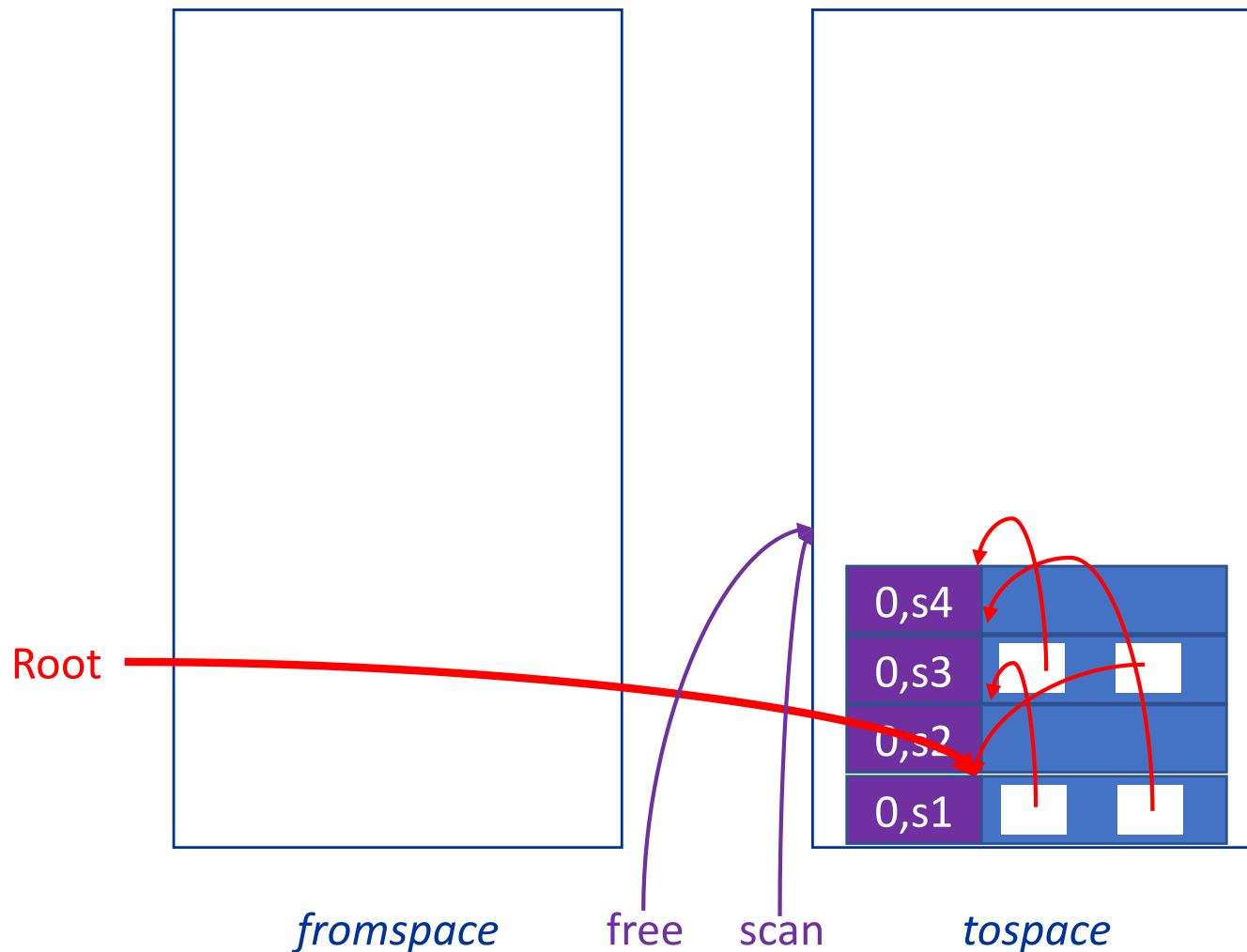
flip() has updated scan and it is now the same as *free*

Copying GC has now finished. All live blocks in *fromspace* have been copied to *tospace*, and all pointers within the data areas of blocks in *tospace* now point to addresses in *tospace*. Specifically, there are no pointers from *tospace* into *fromspace*

Program evaluation can proceed, and anything remaining in *fromspace* is inaccessible to the program and no longer needed by the Copying garbage collector (see next slide)

FUNCTIONAL PROGRAMMING COPYING GARBAGE COLLECTION

CHENEY'S ITERATIVE COPYING GC



This is the final position of the memory layout after the Copying garbage collection (using Cheney's iterative algorithm) has finished

Any memory allocation requests are now satisfied from *tospace* – if *tospace* starts to get full then garbage collection will be triggered again by *malloc()*

If *malloc()* calls another garbage collection, *tospace* will then be the memory area on the left. As the program proceeds, it alternates between using the memory on the left and the memory on the right

FUNCTIONAL PROGRAMMING COPYING GARBAGE COLLECTION

FOR AND AGAINST COPYING GC

Against	For
The required size of the heap is double that of non-copying GCs	Pointer-increment allocation costs are very low, the out-of-memory check is very fast, and there's no overhead on copying/deleting pointers to blocks
Unless both semi-spaces exist simultaneously in RAM, a copying collector will suffer more VM page faults overall than mark-scan GC because it uses twice as many pages	Natural compaction means (i) the working set of VM pages is smaller than Mark-Scan, so reducing page faults while the program (not the GC) runs, and (ii) cache performance of the running program is better than Mark-Scan
GC will become more frequent as the "residency" increases (% heap taken up by live blocks)	Fragmentation is eliminated immediately after a garbage collection (the technique is naturally compacting)
There is an embarrassing pause while the GC runs, which is not good for real-time, highly-interactive or distributed systems	though Copying GC can be modified to run concurrently with program evaluator (if they communicate with each other)
	It is able to recover cyclic structures in the heap (e.g. cyclic data structures)
	It only visits the live blocks, not the entire heap

For and Against

Pointer-increment allocation costs are very low, the out-of-memory check is very fast, and there's no overhead on copying/deleting pointers to blocks

Fragmentation is eliminated immediately after a garbage collection (the technique is naturally compacting)

However, the required size of the heap is double that of non-copying GCs

Unless both semi-spaces can exist simultaneously in physical memory, a copying collector will suffer more VM page faults overall than mark-scan GC because it uses twice as many pages

Natural compaction means (i) the working set of VM pages is smaller than Mark-Scan, so reducing page faults while the program (not the GC) runs, and (ii) cache performance of the running program is better than Mark-Scan

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

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 Copying GC can be modified to run concurrently with program evaluator (if they communicate with each other)

It only visits live blocks, not the entire heap

FUNCTIONAL PROGRAMMING

COPYING GARBAGE COLLECTION

SUMMARY

- Overview
- Assumptions
- Simple imperative pseudo-code
- Cheney's iterative copying garbage collector
- For and against Copying GC

In summary, this lecture has explored the second of three canonical garbage collection algorithms – Copying Garbage Collection

The lecture has provided simple imperative pseudo-code for a recursive version of copying GC and then presented Cheney's iterative copying garbage collection algorithm, together with a detailed worked example

The lecture ended with a summary of good and bad characteristics of Copying Garbage Collection