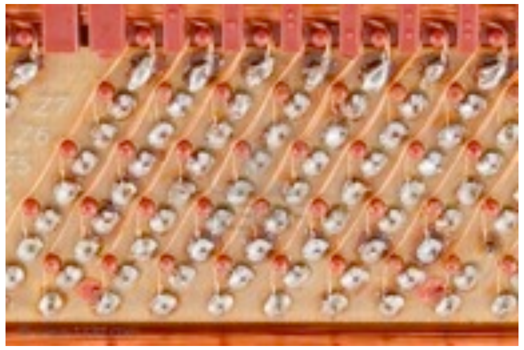
Lecture 8
Dynamic Memory Allocation

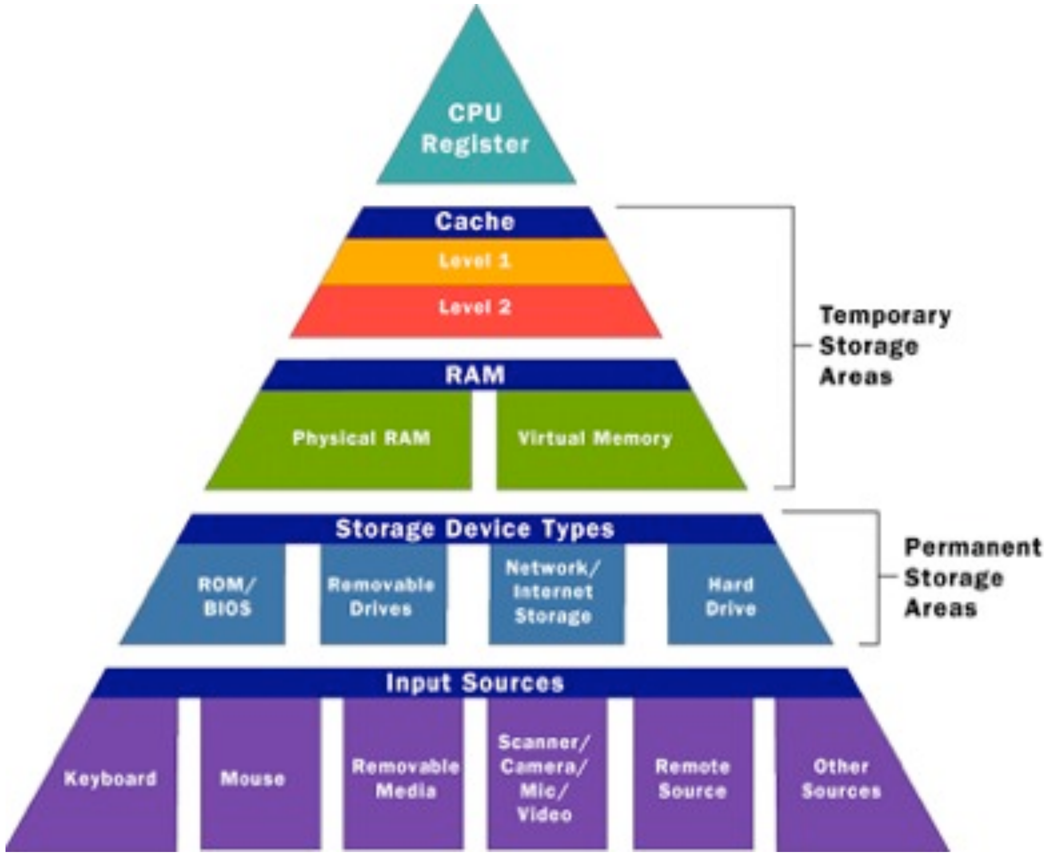
CS240

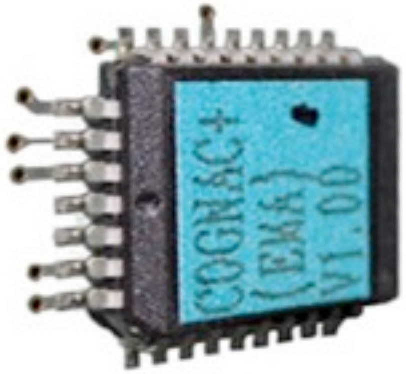
Memory

Computer programs manipulate an abstraction of the computer's memory subsystem

Memory: on the hardware side







@ http://computer.howstuffworks.com/computer-memory.htm/printable

Memory: on the software side

- Each computer programming languages offers *a different abstraction*
- The goal is to make programming easier and improve portability of the source code by hiding irrelevant hardware oddities
- Each language offers a memory API – a set of operations for manipulating memory

  - Sample exam question:
    - *How does the abstraction of memory exposed by the Java programming language differ from that of the C programming language?*

Memory: the Java Story

- Memory is a set of objects with fields, methods and a class + local variables of a method
- Memory is read by accessing a field or local variable
- Memory is modified by writing to a field or local variable
- Location and size of data are not exposed
- Memory allocation is done by call in new

```
public class Main {
  static public
   void main(String[] a){
    Cell c1, c2 = null;
    while (true) {
      c1 = new Cell();

      c2 = c1;
    }
  }
}
```

- Question:
  - Does `main()` terminate?

```
class Cell {Cell next; }
```

Memory: the Java Story

- Memory is a set of objects with fields, methods and a class + local variables of a method
- Memory is read by accessing a field or local variable
- Memory is modified by writing to a field or local variable
- Location and size of data are not exposed
- Memory allocation is done by call in new

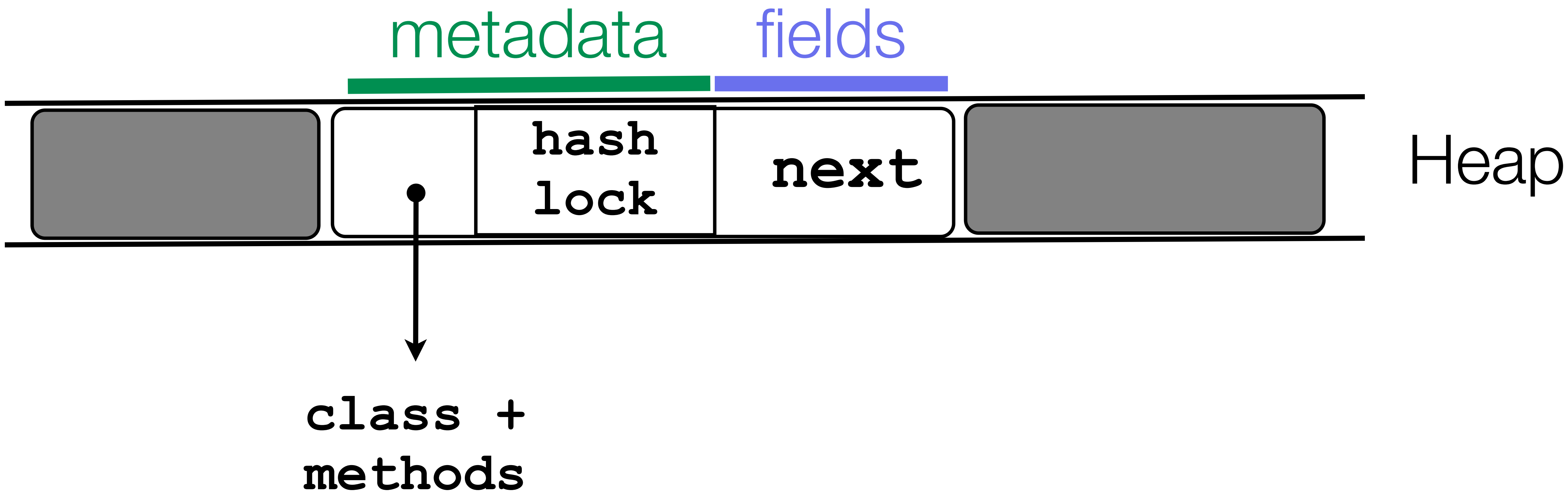
```
public class Main {
   static public
     void main(String[] a){
      Cell c1, c2 = null;
      while (true) {
         c1 = new Cell();
         c1.next = c2;
         c2 = c1;
      }
   }
}

class Cell {Cell next; }
```

- **Question:**
  - Does `main()` terminate?

Memory: the Java Story

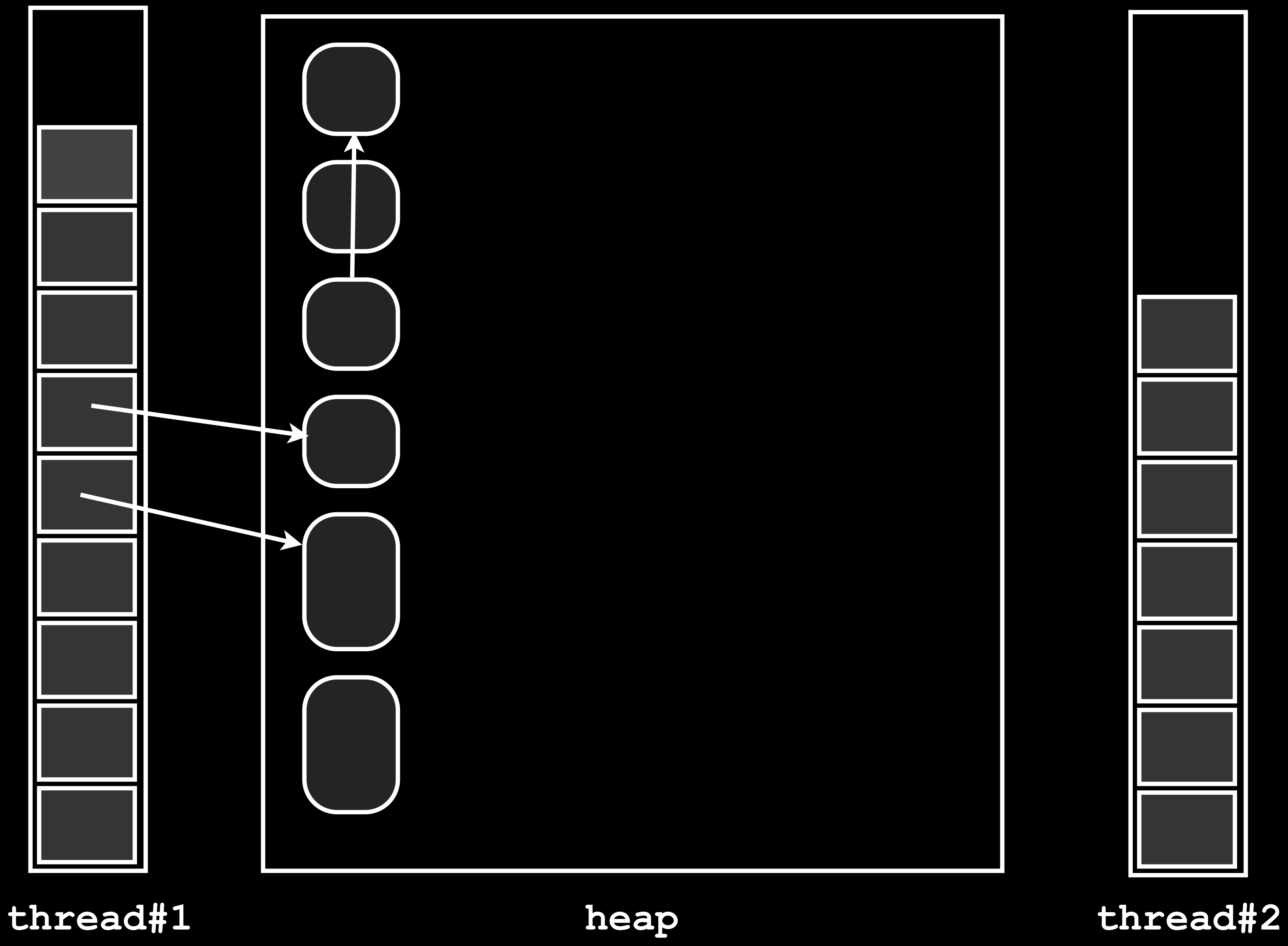
- The semantics of `new` is as follows:
  - Allocate space for the object's fields and metadata fields
  - Initialize the metadata fields
  - Set all fields to null/zero/false
  - Invoke the user defined constructor method



Aparté

- Garbage collection is the technology that gives the illusion of infinite resources
- Garbage collection or GC is implemented by the programming language with the help of the compiler
  - Though for a some well-behaved C programs it is possible to link a special library that provides most of the benefits of GC
  - Question:
    - *How does GC work?*

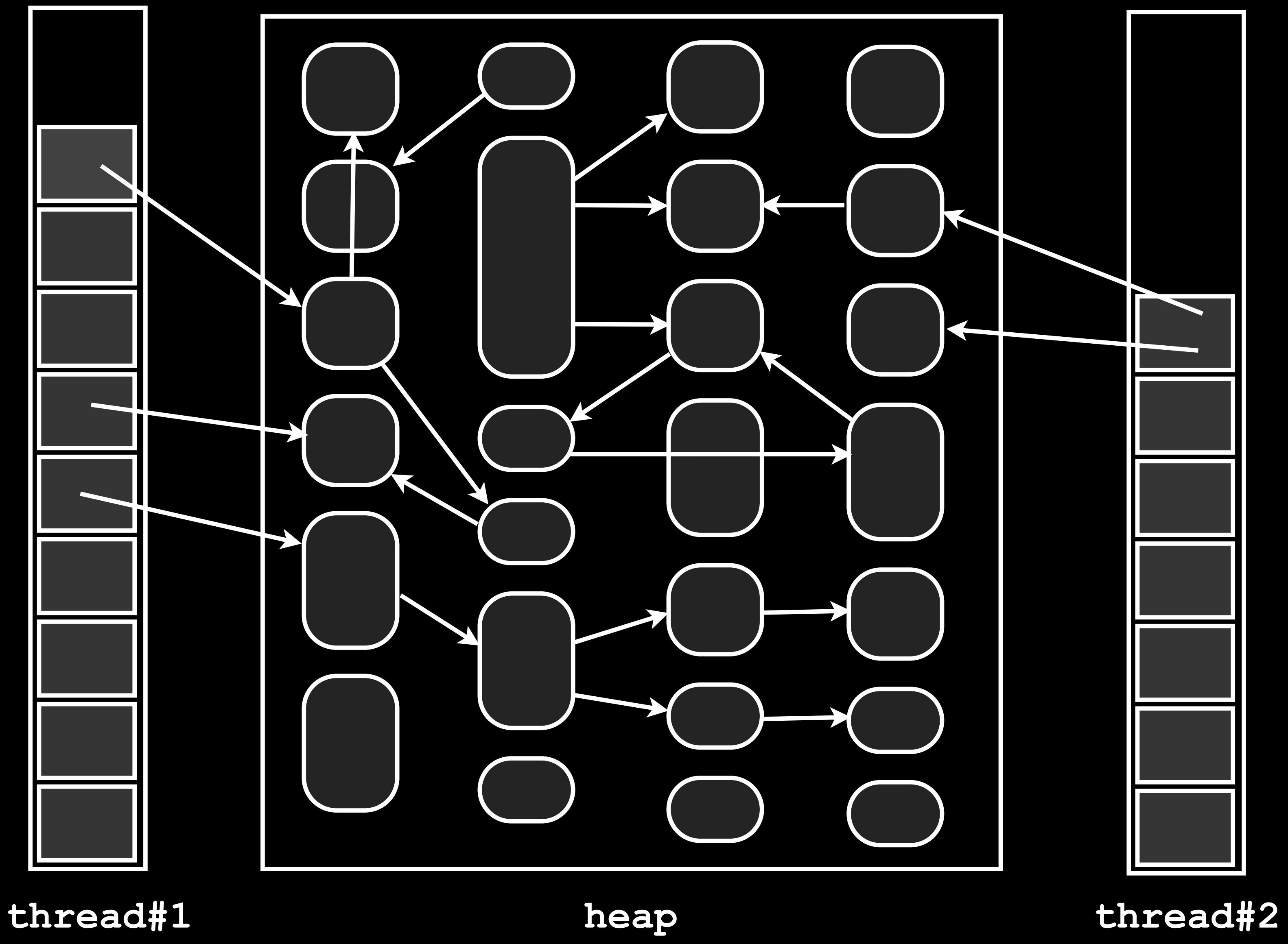
Garbage Collection



Phases

- Mutation
- Stop-the-world
- Root scanning
- Marking
- Sweeping
- Compaction

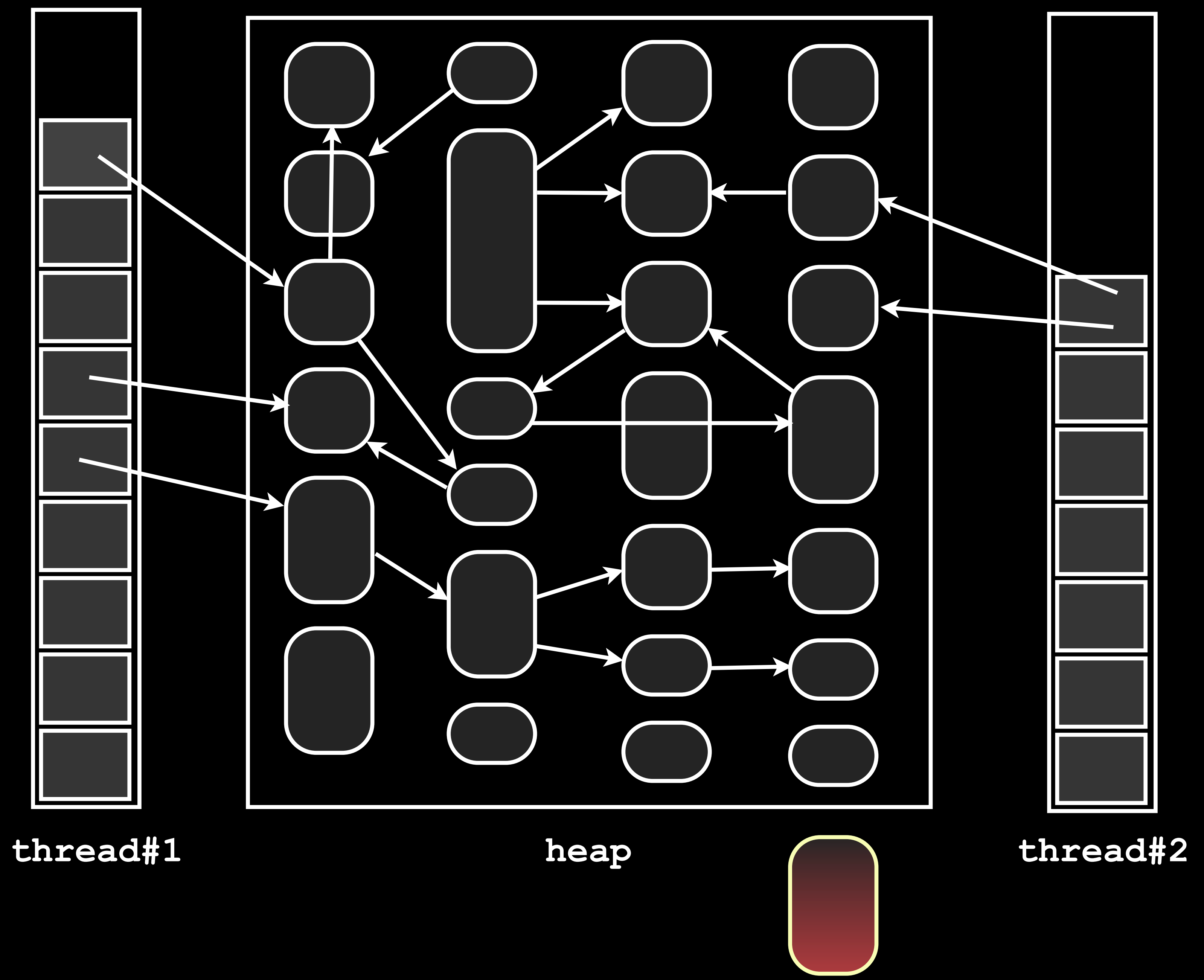
Garbage Collection



Phases

- Mutation
- Stop-the-world
- Root scanning
- Marking
- Sweeping
- Compaction

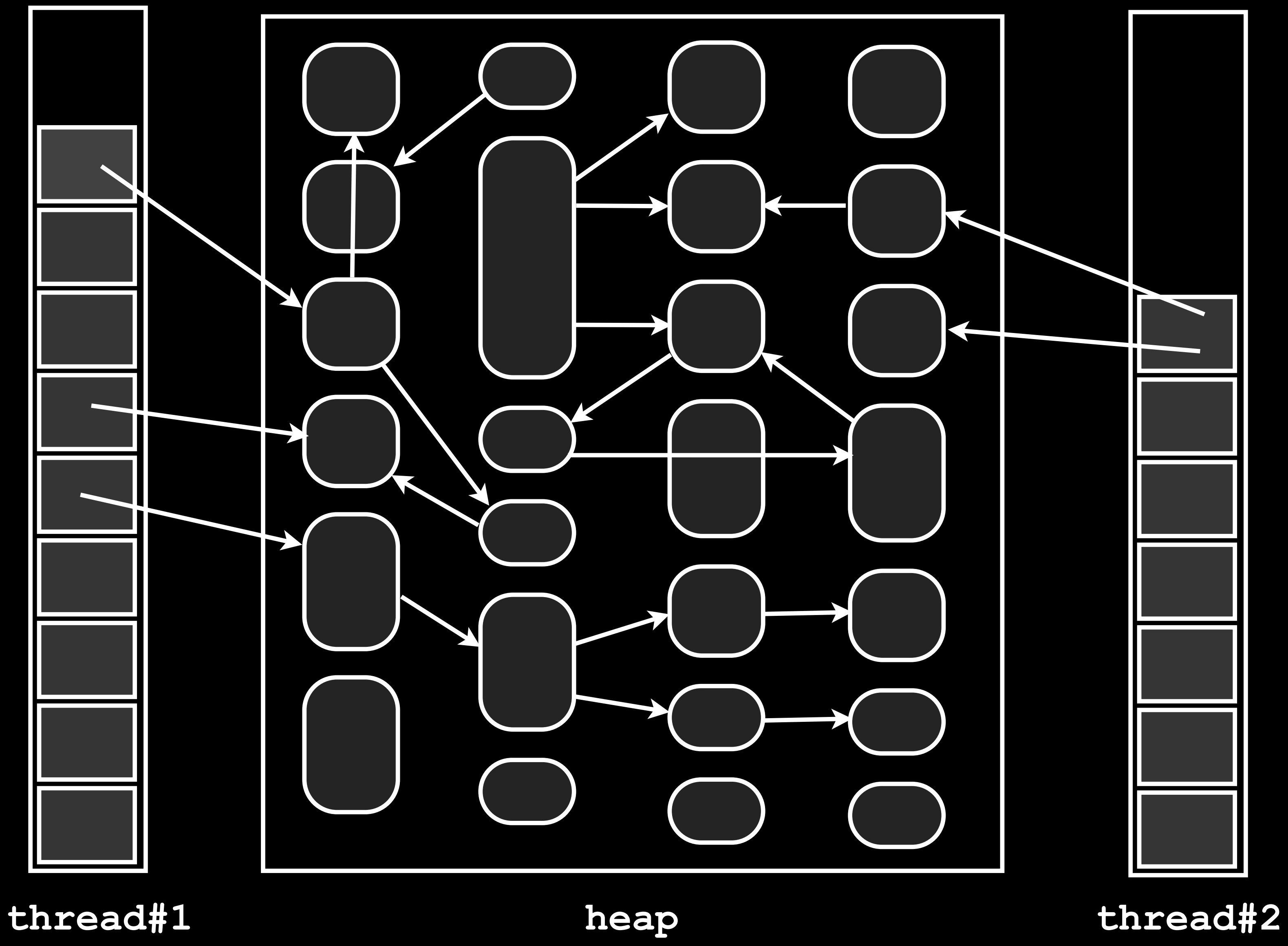
Garbage Collection



Phases

- Mutation
- Stop-the-world
- Root scanning
- Marking
- Sweeping
- Compaction

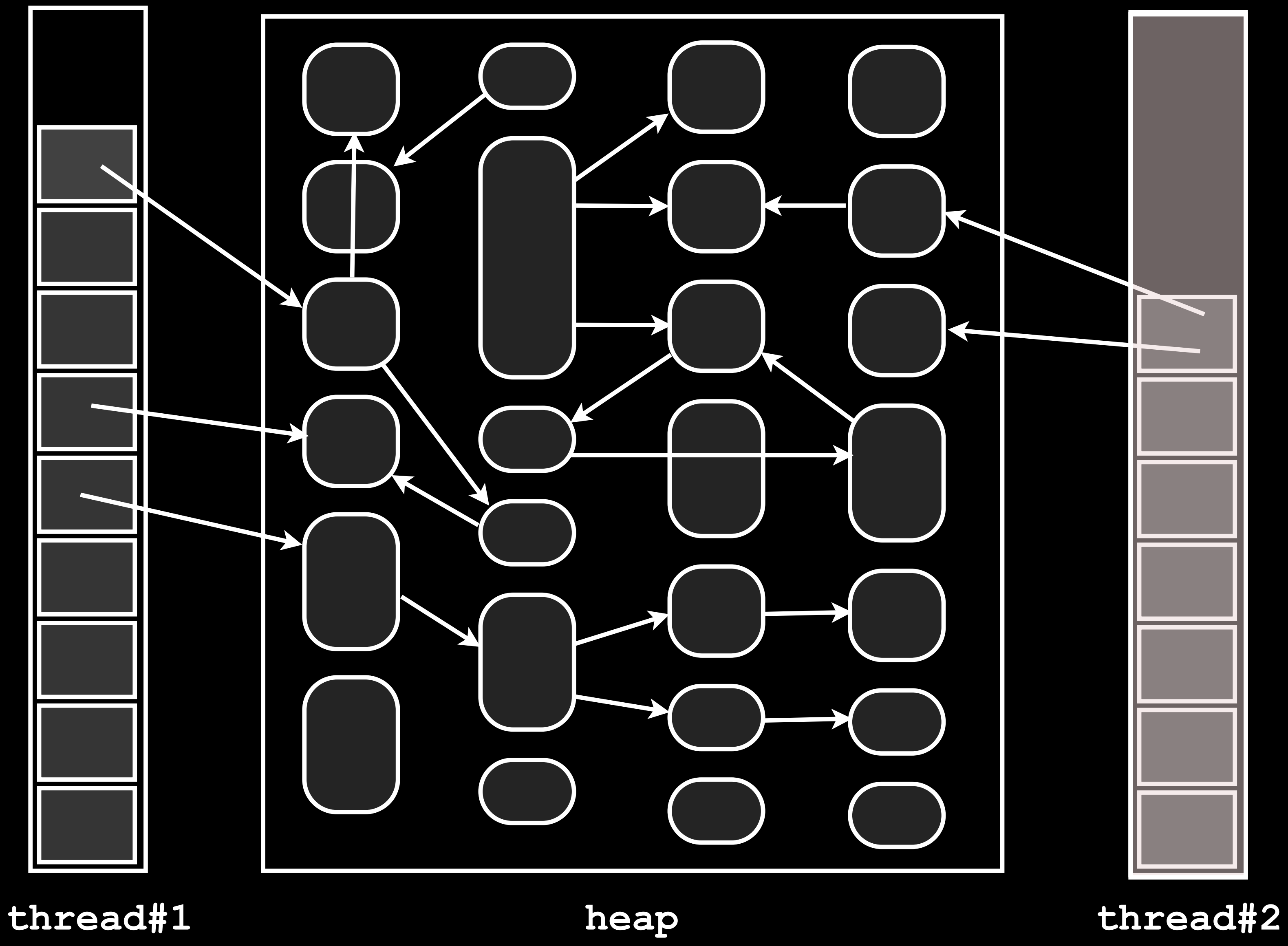
Garbage Collection



Phases

- Mutation
- Stop-the-world
- Root scanning
- Marking
- Sweeping
- Compaction

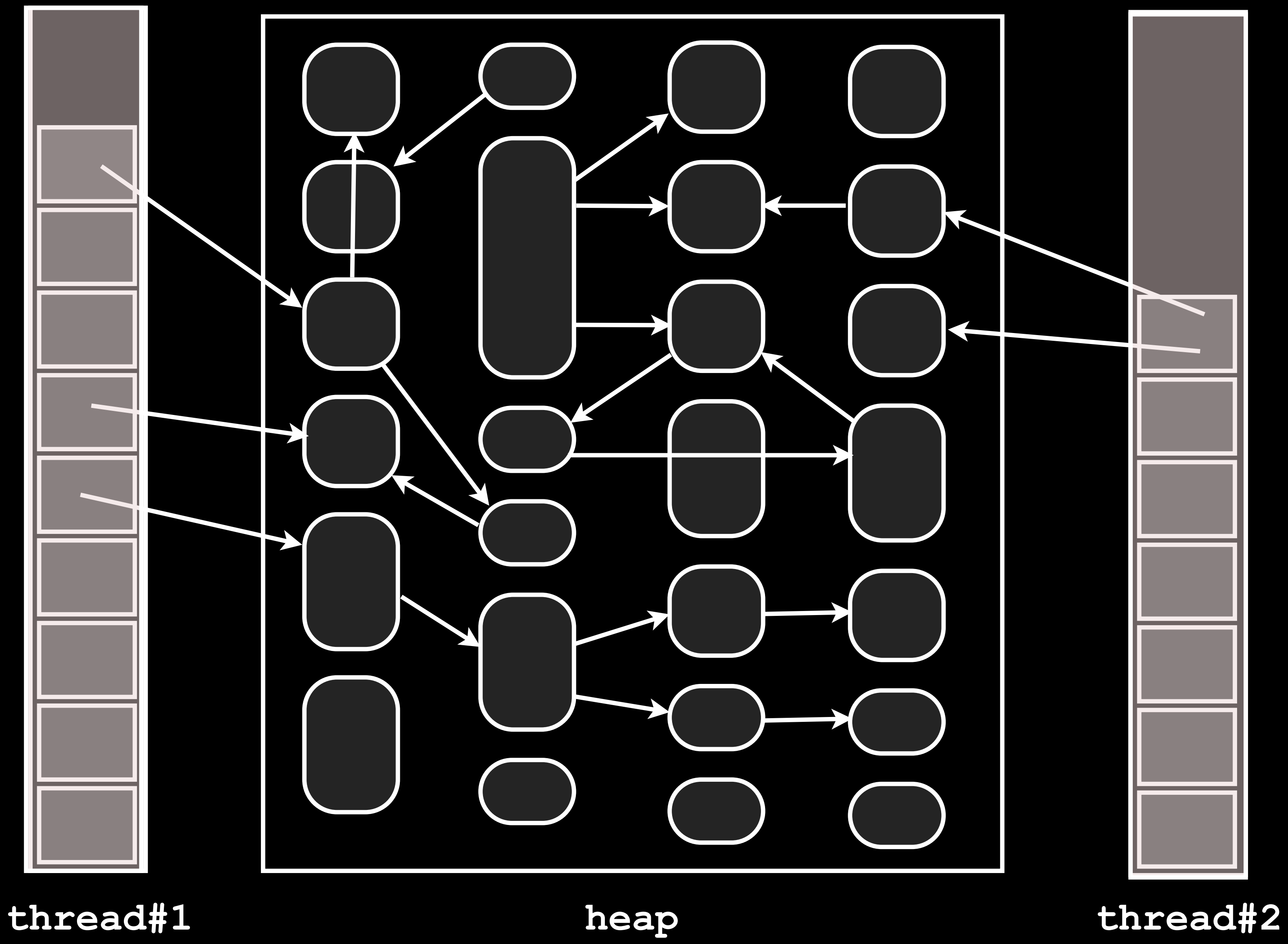
Garbage Collection



Phases

- Mutation
- Stop-the-world
- Root scanning
- Marking
- Sweeping
- Compaction

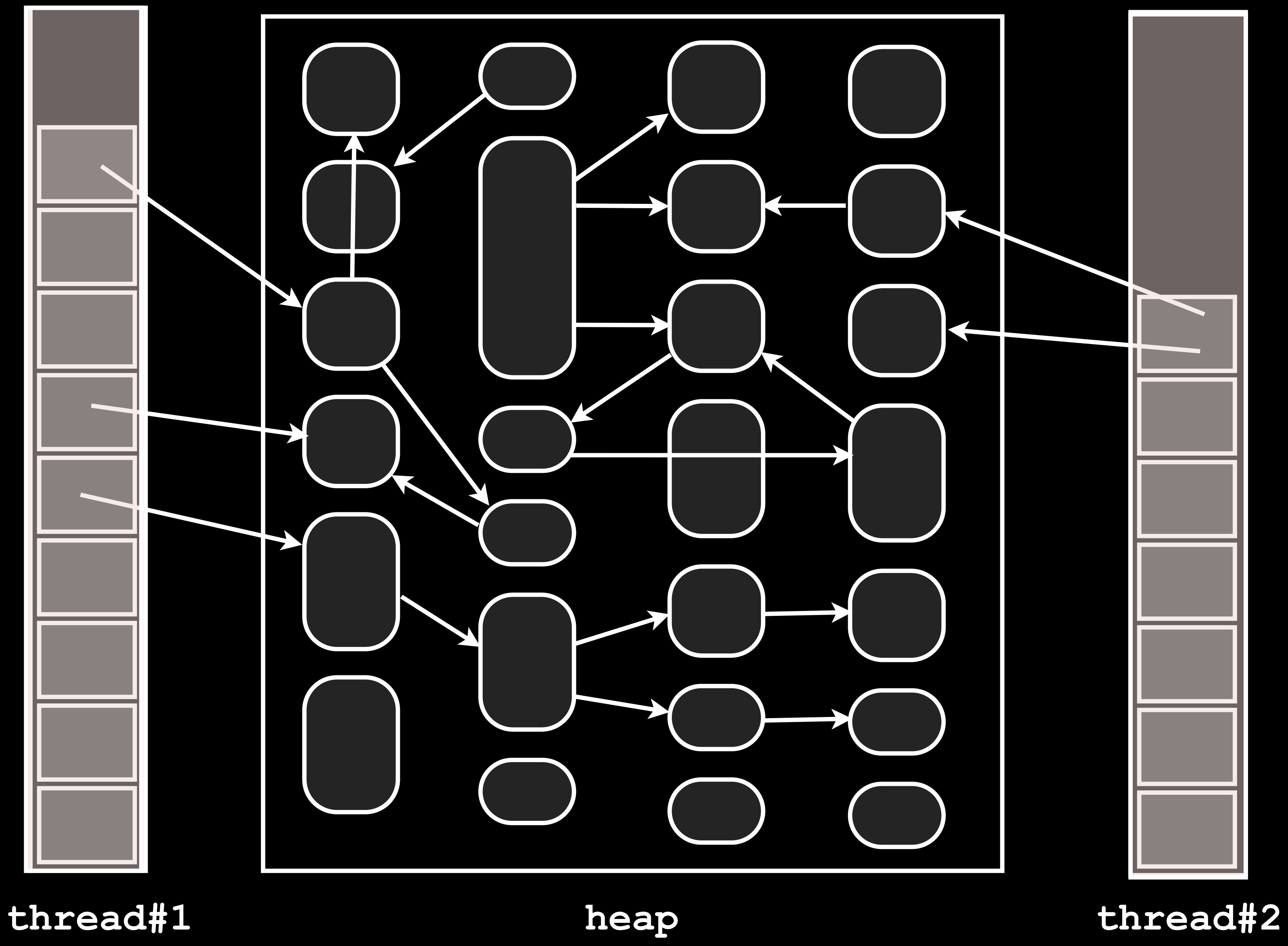
Garbage Collection



Phases

- Mutation
- Stop-the-world
- Root scanning
- Marking
- Sweeping
- Compaction

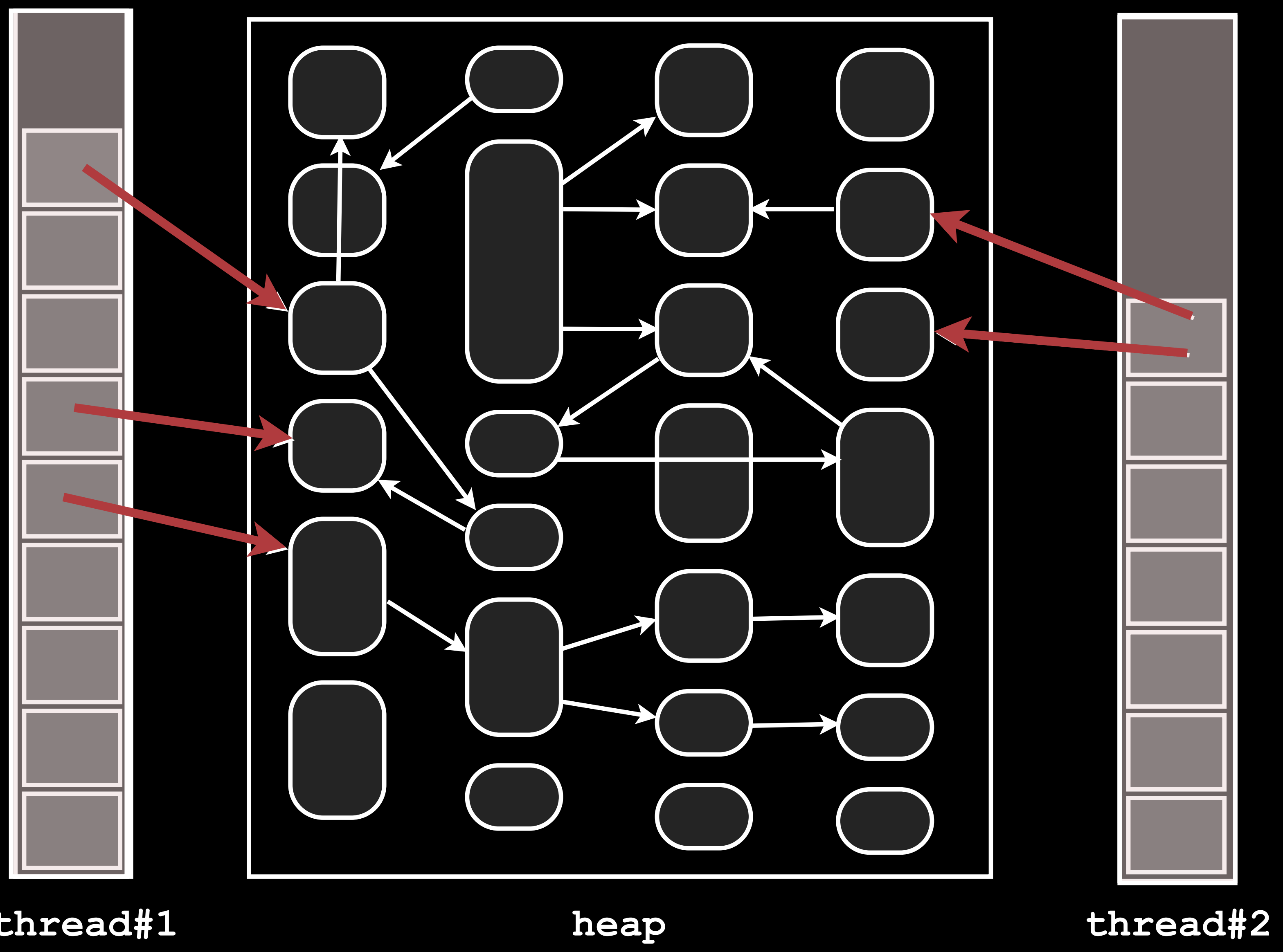
Garbage Collection



Phases

- Mutation
- Stop-the-world
- Root scanning
- Marking
- Sweeping
- Compaction

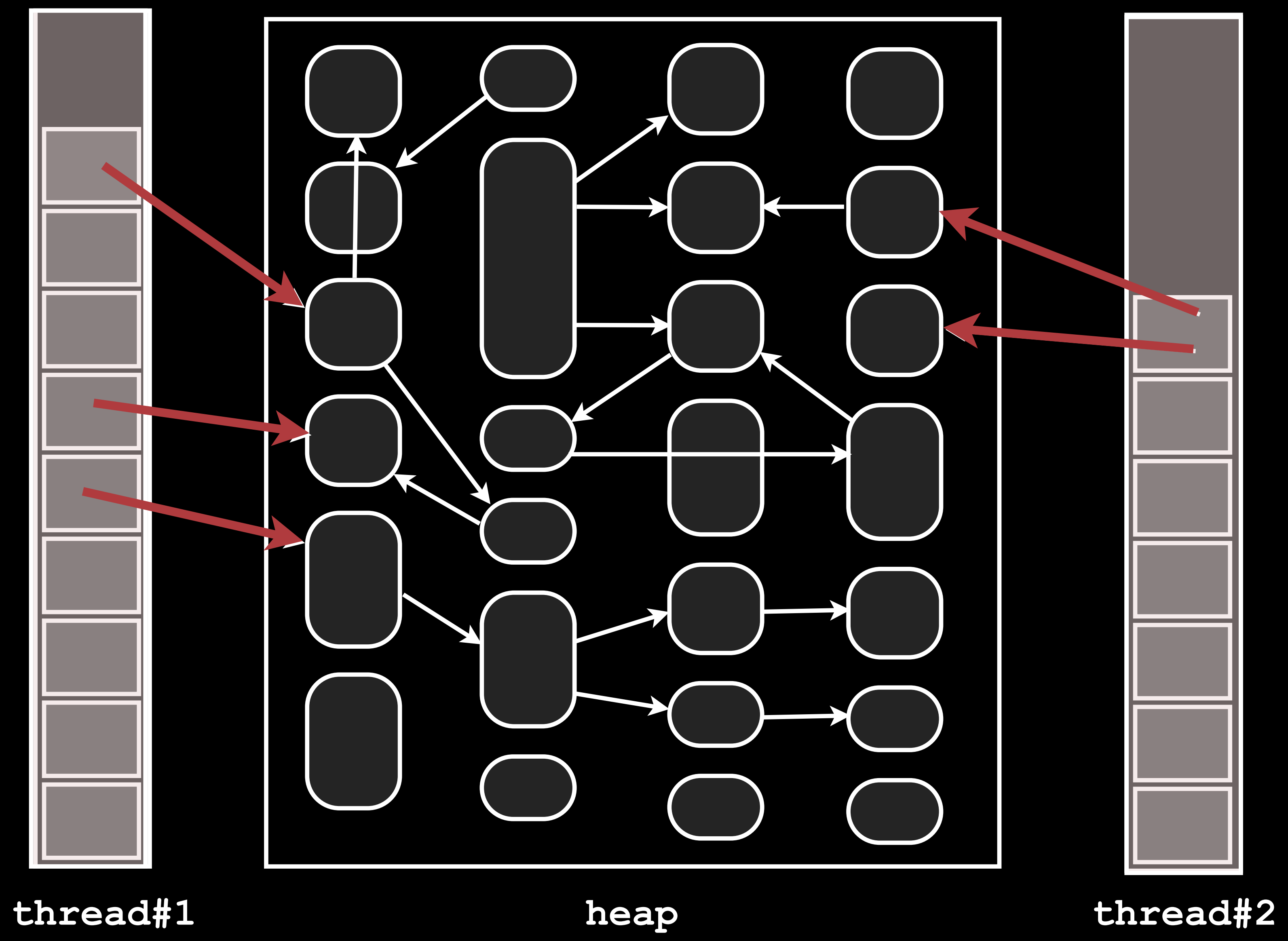
Garbage Collection



Phases

- Mutation
- Stop-the-world
- Root scanning
- Marking
- Sweeping
- Compaction

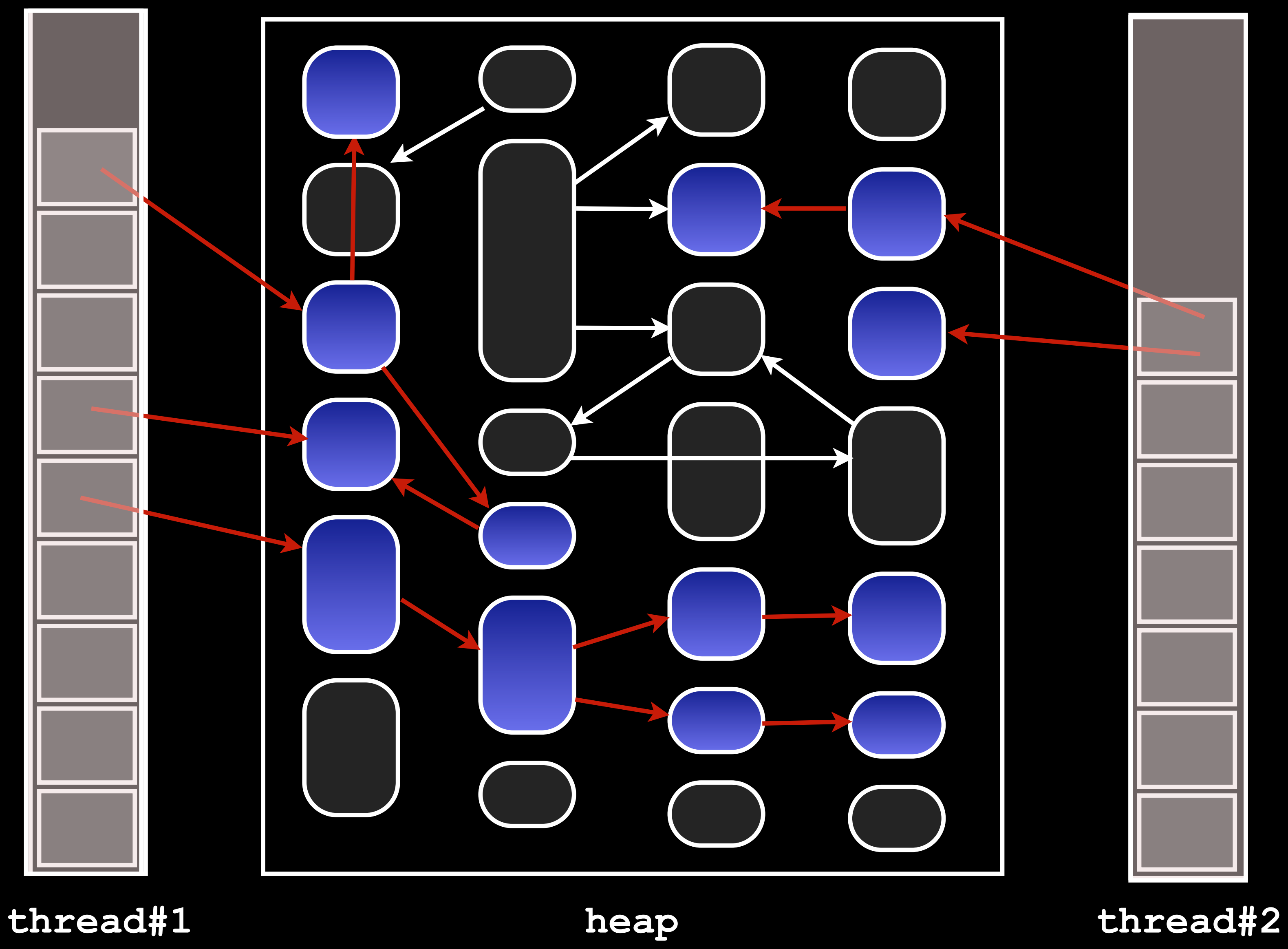
Garbage Collection



Phases

- Mutation
- Stop-the-world
- Root scanning
- Marking
- Sweeping
- Compaction

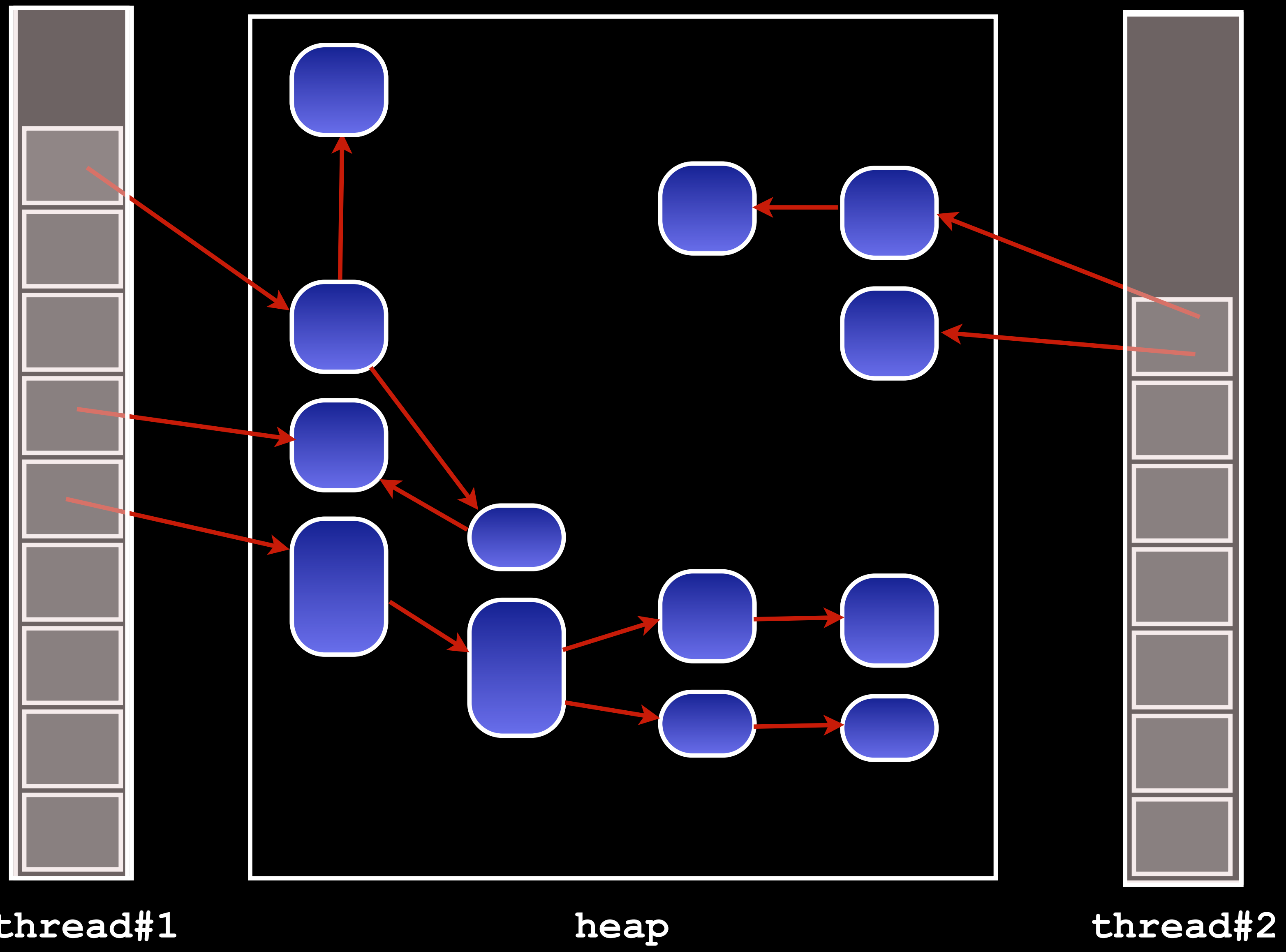
Garbage Collection



Phases

- Mutation
- Stop-the-world
- Root scanning
- Marking
- Sweeping
- Compaction

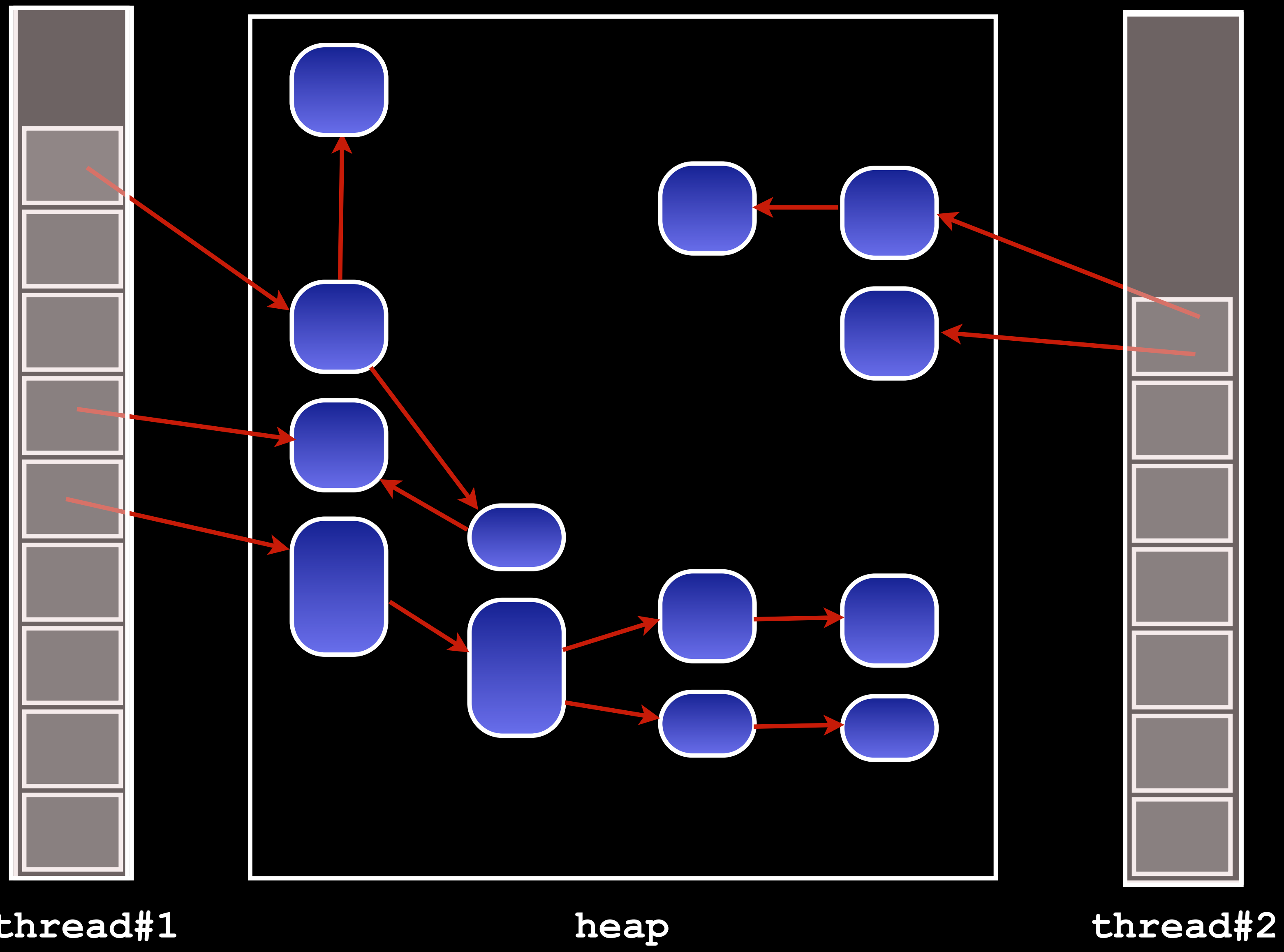
Garbage Collection



Phases

- Mutation
- Stop-the-world
- Root scanning
- Marking
- Sweeping
- Compaction

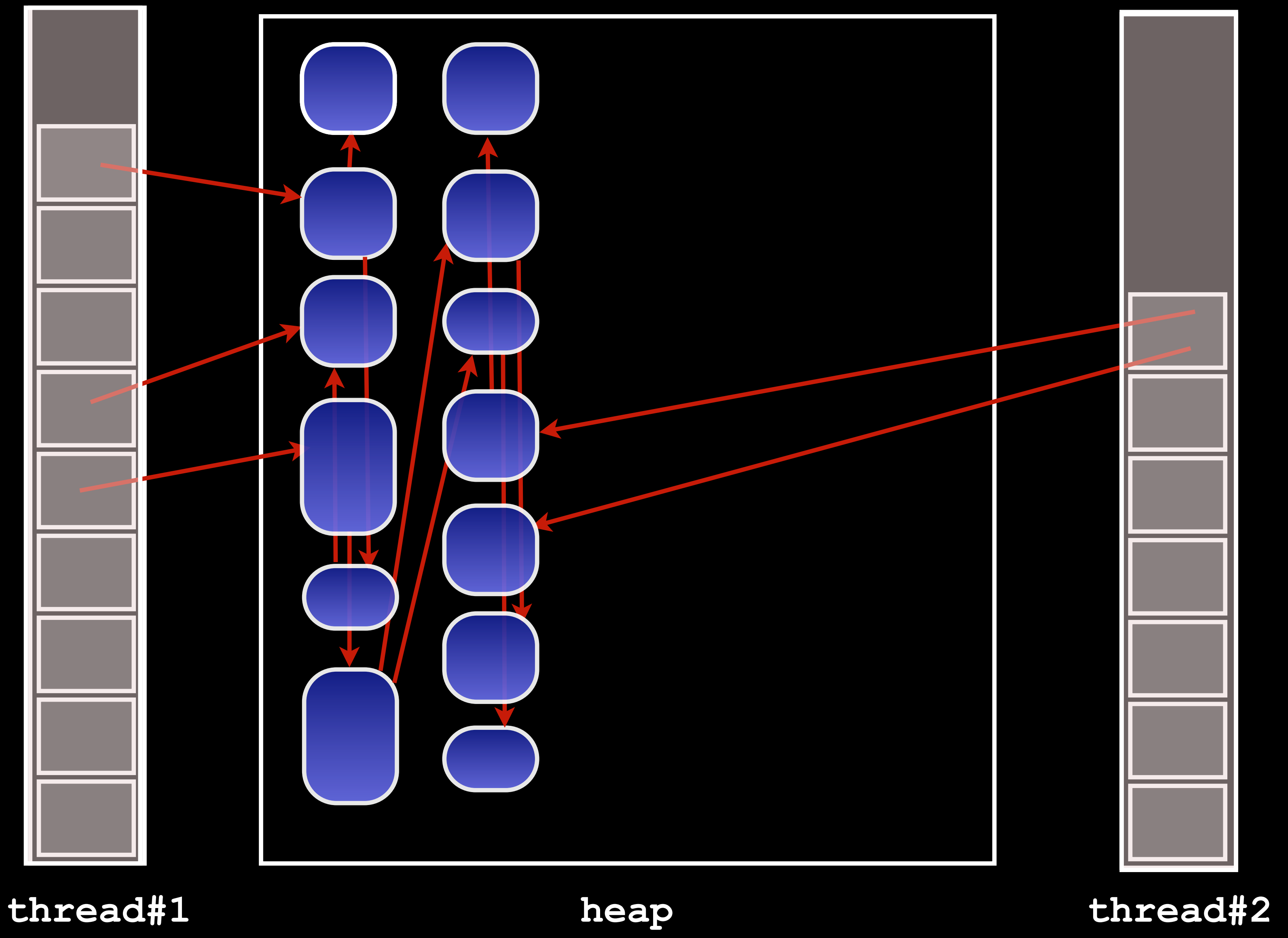
Garbage Collection



Phases

- Mutation
- Stop-the-world
- Root scanning
- Marking
- Sweeping
- Compaction

Garbage Collection



Phases

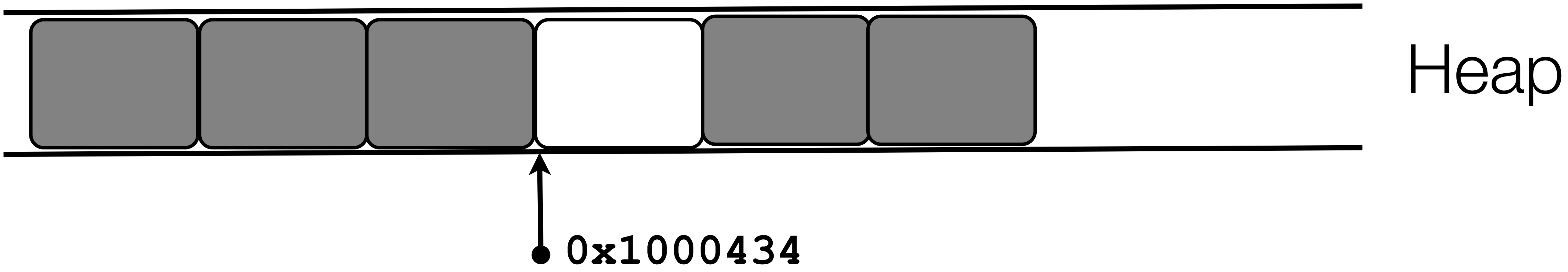
- Mutation
- Stop-the-world
- Root scanning
- Marking
- Sweeping
- Compaction

Isn’t this a course about C?

Yes, Virginia

Memory: the C Story

- C offers a story both simpler and more complex than Java
- Memory is a sequence of bytes, read/written by providing an address
- Addresses are values manipulated using arithmetic & logic operations
- Memory can be allocated:
  - Statically
  - Dynamically on the stack
  - Dynamically on the heap
- Types give the compiler a hint how to interpret a memory addresses



Static and Stack allocation

- Static allocation with the keyword `static`
- Stack allocation automatic by the compiler for local variables
- **`printf`** can display the address of any identifier

```
#include <unistd.h>
#include <stdio.h>

static int sx;
static int sa[100];
static int sy;

int main() {
  int lx;
  static int sz;

  printf("%p\n", &sx);    0x100001084
  printf("%p\n", &sa);    0x1000010a0
  printf("%p\n", &sy);    0x100001230
  printf("%p\n", &lx);    0x7fff5fbff58c
  printf("%p\n", &sz);    0x100001080
  printf("%p\n", &main);  0x100000dfc
```

Static and Stack allocation

- Any value can be turned into a pointer
- Arithmetics on pointers allowed
- Nothing prevents a program from writing all over memory

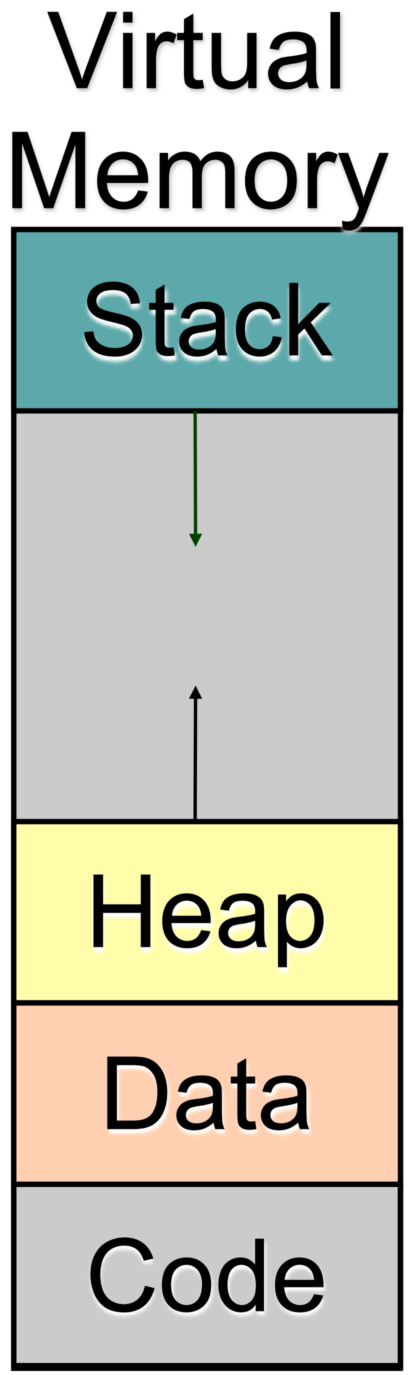
```
static int sx;
static int sa[100];
static int sy;

int main() {
  for(p= (int*)0x100001084;
      p <= (int*)0x100001230;
      p++)
 {
    *p = 42;
  }
  printf("%i\n",sx);
  printf("%i\n",sa[0]);
  printf("%i\n",sa[1]);
```

```
42
42
42
```

Memory layout

- The OS creates a process by assigning memory and other resources
- C exposes the layout as the programmer can take the address of any element (with &)
- Stack:
  - *keeps track of where each active subroutine should return control when it finishes executing; stores local variables*
- Heap:
  - *dynamic memory for variables that are created with malloc, calloc, realloc and disposed of with free*
- Data:
  - *global and static variables*
- Code:
  - *instructions to be executed*



Dynamic Memory: The DIY Way

- A simple dynamic allocation pattern is to ask the OS for a chunk of memory large enough to store *all* data needed
- `sbrk(size)` returns a chunk of memory of `size` words
- The downside is that the programmer must keep track of how memory is used

```
int main() {
 int* x; int* start;
 double* y;
 start = (int*) sbrk(5);
  x = start;
 *x = -42;   x++; y=(double*) x;
 *y = 2.1;   y++; x=(int*) y;
 *x = 42;
 printf("%i\n", *start);   -42
 printf("%i\n", start[0]); -42
 printf("%i\n", start[1]); -858993459
 printf("%i\n", start[2]); 1073794252
 printf("%i\n", start[3]); 42
 printf("%i\n", start[4]); 0
 printf("%i\n", start[5]); 0
 printf("%f\n",            2.100000
    *(double*)(start+1));
```

Dynamic memory management

```
#include <stdlib.h>

void* calloc(size_t n, size_t s)
void* malloc(size_t s)
void  free(void* p)
void* realloc(void* p, size_t s)
```

- Allocate and free dynamic memory

Operations with memory

```
void* memset(void *s, int c, size_t n)
void* memcpy(void *s, const void *s2, size_t n)
```

- Initializing and copying blocks of memory

`malloc(size_t s)`

- Allocates `s` bytes and returns a pointer to the allocated memory.
- Memory is not cleared
- Returned value is a pointer to alloc'd memory or `NULL` if the request fails
- You must cast the pointer

```
p = (char*) malloc(10); /* allocated 10 bytes */
if(p == NULL) { /*panic*/ }
```

CAN FAIL, CHECK THE RETURNED POINTER NOT NULL

`calloc(size_t n, size_t s)`

- Allocates memory for an array of `n` elements of `s` bytes each and returns a pointer to the allocated memory.
- The memory is set to zero
- The value returned is a pointer to the allocated memory or **`NULL`**

```
p = (char*) calloc(10,1); /*alloc 10 bytes */
if(p == NULL) { /* panic */ }
```

CAN FAIL, CHECK THE RETURNED POINTER NOT NULL

What's the difference between `int array[10]` and `calloc(10,4)`

`free(void* p)`

- Frees the memory space pointed to by `p`, which *must* have been allocated with a previous call to `malloc, calloc or realloc`
- If memory was not allocated before, or if `free(p)` has already been called before, undefined behavior occurs.
- If `p` is `NULL`, no operation is performed.
- `free()` returns nothing

```
char *mess = NULL;
mess = (char*) malloc(100);
…
free(mess);
```

FREE DOES NOT SET THE POINTER TO NULL

`realloc(void* p,size_t s)`

- Changes the size of the memory block pointed to by `p` to `s` bytes
- Contents unchanged to the minimum of old and new sizes
- Newly alloc'd memory is uninitialized.
- Unless `p==`NULL, it must have been returned by `malloc, calloc or realloc.`
- If `p==`NULL, equivalent to `malloc(size)`
- If `s==0`, equivalent to `free(ptr)`
- Returns pointer to alloc'd memory, may be different from `p`, or NULL if the request fails or if `s==0`
- If fails, original block left untouched, i.e. it is not freed or moved

`memcpy(void*dest,const void*src,size_t n)`

- Copies `n` bytes from `src` to `dest`
- Returns `dest`
- Does not check for overflow on copy

```
char buf[100];
char src[20] = “Hi there!”;
int type = 9;
memcpy(buf, &type, sizeof(int)); /* copy an int */

memcpy(buf+sizeof(int), src, 10); /*copy 10 chars */
```

`memset(void *s, int c, size_t n)`

- Sets the first `n` bytes in `s` to the value of `c`
  - (`c` is converted to an `unsigned char`)
- Returns `s`
- Does not check for overflow

```
memset(mess, 0, 100);
```

Sizeof matters

- In C, programmers must know the size of data structures
- The compiler provides a way to determine the size of data

```
struct {
 int i; char c; float cv;
} C;

int x[10];
printf("%i\n", (int) sizeof(char));     1
printf("%i\n", (int) sizeof(int));      4
printf("%i\n", (int) sizeof(int*));     8
printf("%i\n", (int) sizeof(double));   8
printf("%i\n", (int) sizeof(double*));  8
printf("%i\n", (int) sizeof(x));       40
printf("%i\n", (int) sizeof(C));       12
```

Sizeof matters

- In C, programmers must know the size of data structures
- The compiler provides a way to determine the size of data

- Do this:

```
int *p = malloc(10 * sizeof(*p));
```

Memory Allocation Problems

- Memory leaks
  - Alloc'd memory not freed appropriately
  - If your program runs a long time, it will run out of memory and slow down the system
  - Always add the free on all control flow paths after a malloc

```
void *ptr = malloc(size);
/*the buffer needs to double*/
size *= 2;
ptr = realloc(ptr, size);
if (ptr == NULL)
  /*realloc failed, original address in ptr
   lost; a leak has occurred*/
  return 1;
```

Memory Allocation Problems

- Use after free
  - Using dealloc'd data
  - Deallocating something twice
  - Deallocating something that was not allocated
    - Can cause unexpected behavior. For example, malloc can fail if "dead" memory is not freed.
    - More insidiously, freeing a region that wasn't malloc'ed or freeing a region that is still being referenced

```
int *ptr = malloc(sizeof (int));
free(ptr);
*ptr = 7; /* Undefined behavior */
```

Memory Allocation Problems

- Memory overrun
  - Write in memory that was not allocated
  - The program will exit with segmentation fault
  - Overwrite memory: unexpected behavior

```
int*y= …
int*x= y+10
for(p= x; p >= y;p++)
{
    *p = 42;
}
```

Memory Allocation Problems

- Fragmentation
  - The system may have enough memory but not in a contiguous region

```
int* vals[10000];

int i;
for (i = 0; i < 10000; i++)
  vals[i] = (int*) malloc(sizeof(int*));

for (i = 0; i < 10000; i = i + 2)
  free(vals[i]);
```

A (simple) malloc

```
#define SIZE 10000
#define UNUSED -1

struct Cell { int sz; void* value; struct Cell* next; };

static struct Cell*free, *used;
static struct Cell cells[SIZE / 10];

void init() {
    void* heap = sbrk(SIZE);
    int i;
    for (i = 0; i<SIZE/10; i++) { cells[i].sz=UNUSED; cells[i].next=NULL; }
    cells[0].sz = 0;
    free = &cells[0];
    free->next = &cells[1];
    free->next->sz = SIZE;
    free->next->value = heap;
    free->next->next = NULL;
    used = &cells[1];
    used->sz = 0;
    used->value = (void*) UNUSED;
}
```

A (simple) malloc

```
#define SIZE 10000
#define UNUSED -1

struct Cell { int sz; void* value; struct Cell* next; };

static struct Cell*free, *used;
static struct Cell cells[SIZE / 10];

void init() {
   void* heap = sbrk(SIZE);
   int i;
   for (i = 0; i<SIZE/10; i++) { cells[i].sz=UNUSED; cells[i].next=NULL; }
   cells[0].sz = 0;
   free = &cells[0];
   free->next = &cells[1];
   free->next->sz = SIZE;
   free->next->value = heap;
   free->next->next = NULL;
   used = &cells[1];
   used->sz = 0;
   used->value = (void*) UNUSED;
}
```

A (simple) malloc

```
void* mymalloc(int size) {
    struct Cell* tmp = free, *prev = NULL;
    if (size == 0) return NULL;
    while (tmp != NULL) {
        if (tmp->sz == size) {
            prev->next = tmp->next;
            tmp->next = used;
            used = tmp;
            return used->value;
        } else if (tmp->sz > size) {
            struct Cell* use = NULL;
            int i;
            for (i = 0; i < SIZE / 10; i++)
                if (cells[i].sz == UNUSED) { use = &cells[i]; use->sz = size; }
            if (use == NULL) return NULL;
            use->next = used;
            use->value = tmp->value;
            tmp->sz -= size;
            tmp->value += size;
            return used->value;
        }
        prev = tmp;
        tmp = tmp->next;
    }
    return NULL;
}
```

A (simple) malloc

```
void myfree(void* p) {
   struct Cell *tmp = used, *prev = NULL;
   while (tmp != NULL) {
      if (tmp->value == p) {
         prev->next = tmp-> next;
         tmp->next = free;
         free = tmp;
         free->sz = UNUSED;
         return;
      }
      prev = tmp;
      tmp = tmp->next;
   }
}
```

Dynamic memory: Checklist

- `NULL` pointer at declaration
- Verify `malloc` succeeded
- Initialize alloc'd memory
- *free* when you *malloc*
- `NULL` pointer after free

Readings and exercises for this lecture

K&R Chapter 5.10 for command line arguments

Write a small program where you free something twice and observe the behavior

Write a small program where you don't free the allocated memory and observe the behavior