

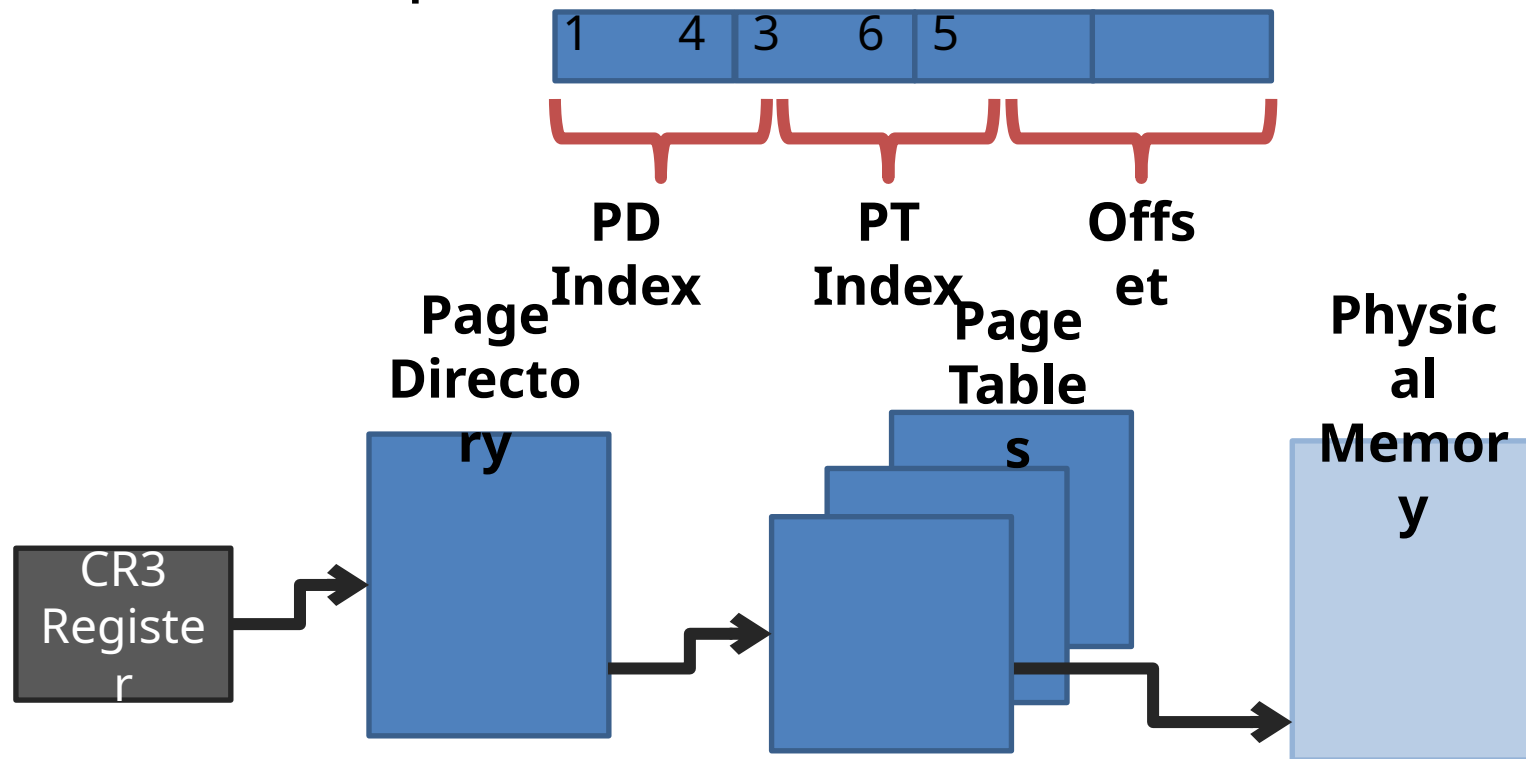
CS 5600

Computer Systems

Lecture 8: Free Memory Management

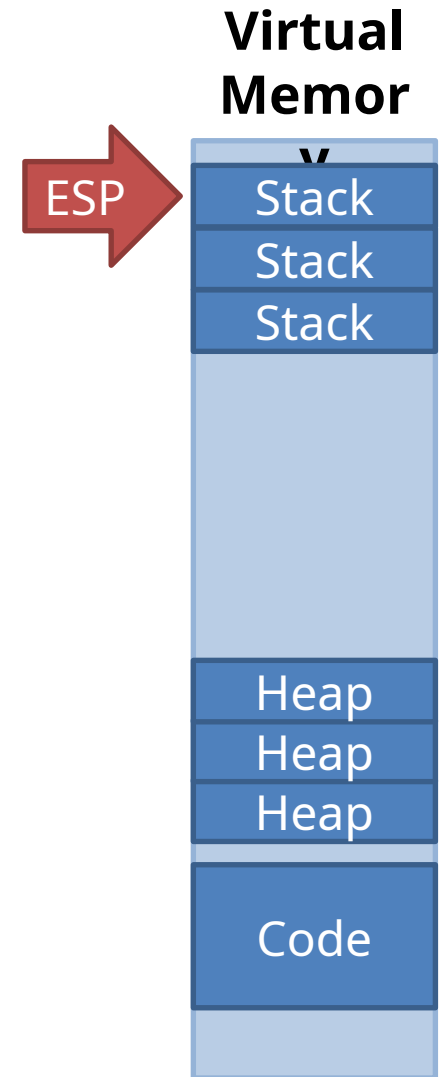
Recap of Last Week

- Last week focused on virtual memory
 - Gives each process the illusion of vast, empty memory
 - Offers protection and isolation



Dynamic Allocation of Pages

- Page tables allow the OS to dynamically assign physical frames to processes on-demand
 - E.g. if the stack grows, the OS can map in an additional page
- On Linux, processes use `sbrk()/brk()/mmap()` to request additional heap pages
 - But, these syscalls only allocates memory in multiples of 4KB pages



What About malloc() and free()?

- The OS only allocates and frees memory in units of 4KB pages
 - What if you want to allocate <4KB of memory?
 - E.g. `char * string = (char *) malloc(100);`
- Each process manages its own heap memory
 - On Linux, glibc implements *malloc()* and *free()*, manages objects on the heap
 - The JVM uses a `garbage collector` to manage the heap
- There are many different strategies for managing free memory

Free Space Management

- Today's topic: how do processes manage free memory?
 1. Explicit memory management
 - Languages like C, C++; programmers control memory allocation and deallocation
 2. Implicit memory management
 - Languages like Java, Javascript, Python; runtime takes care of freeing useless objects from memory
- In both cases, software must keep track of the memory that is in use or available

Why Should You Care?

- Regardless of language, all of our code uses dynamic memory
- However, there is a performance cost associated with using dynamic memory
- Understanding how the heap is managed leads to:
 - More performant applications
 - The ability to diagnose difficult memory related errors and performance bottlenecks

Key Challenges

- Maximizing CPU performance
 - Keeping track of memory usage requires effort
- Maximize parallelism
 - Heap memory is shared across threads
 - Thus, synchronization may be necessary
- Minimizing memory overhead
 - Metadata is needed to track memory usage
 - This metadata adds to the size of each object
- Minimize fragmentation
 - Over time, deallocations create useless gaps in memory

- Free Lists
 - Basics
 - Speeding Up *malloc()* and *free()*
 - Slab Allocation
 - Common Bugs
- Garbage Collectors
 - Reference Counting
 - Mark and Sweep
 - Generational/Ephemeral GC
 - Parallel Garbage Collection

Setting the Stage

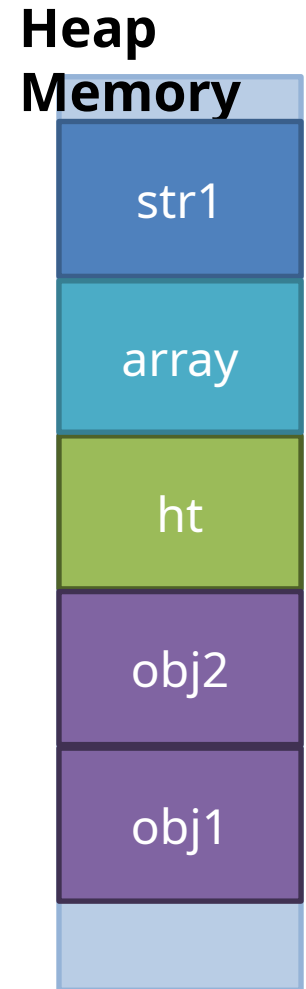
- Many languages allow programmers to **explicitly** allocate and deallocate memory
 - C, C++
 - *malloc()* and *free()*
- Programmers can *malloc()* any size of memory
 - Not limited to 4KB pages
- *free()* takes a pointer, but not a size
 - How does *free()* know how many bytes to deallocate?
- Pointers to allocated memory are returned to the programmer
 - As opposed to Java or C# where pointers are “managed”
 - Code may modify these pointers

Requirements and Goals

- Keep track of memory usage
 - What bytes of the heap are currently allocated/unallocated?
- Store the size of each allocation
 - So that *free()* will work with just a pointer
- Minimize fragmentation...
 - ... without doing compaction or relocation
 - More on this later
- Maintain higher performance
 - $O(1)$ operations are obviously faster than $O(n)$, etc.

Heap Fragmentation

```
obj * obj1, * obj2;  
hash_tbl * ht;  
int array[];  
char * str1, * str2;  
...  
free(obj2);  
free(array);  
...  
str2 = (char *) malloc(300);
```



- This is an example of **external** fragmentation
 - There is enough empty space for str2, but the space isn't usable
- As we will see, internal fragmentation may also be an issue

- Design challenge: linked lists are dynamic data structures
- Dynamic data structures go on the heap
- But in this case, we are implementing the heap?!

• A free list is a data structure for maintaining a list of free memory blocks

• Three key components

1. A linked-list that records free regions of memory
 - Free regions get split when memory is allocated
 - Free list is kept in sorted order by memory address
2. Each allocated block of memory has a header that records the size of the block
3. An algorithm that selects which free region of memory to use for each allocation request

Free List Data Structures

- The free list is a linked list
- Stored in heap memory, alongside other data
- For *malloc(n)*:

num_bytes = *n* +

`sizeof(header)`
`typedef struct node_t {`

`int size;`

`struct node_t *`

`next;`

`} node;`

`typedef struct header_t`
`{`

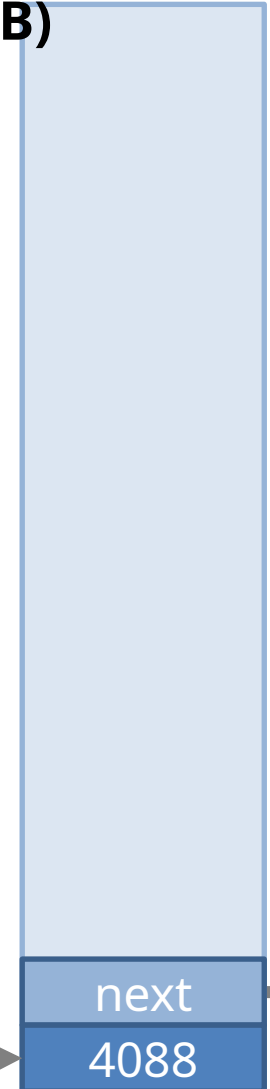
`int size;`

- Linked list of regions of free space
- size = bytes of free space

- Header for each block of allocated space
- size = bytes of allocated space

`node *`
`head`

Heap Memory
(4KB)



Code to Initialize a Heap

// mmap() returns a pointer to a chunk of free space

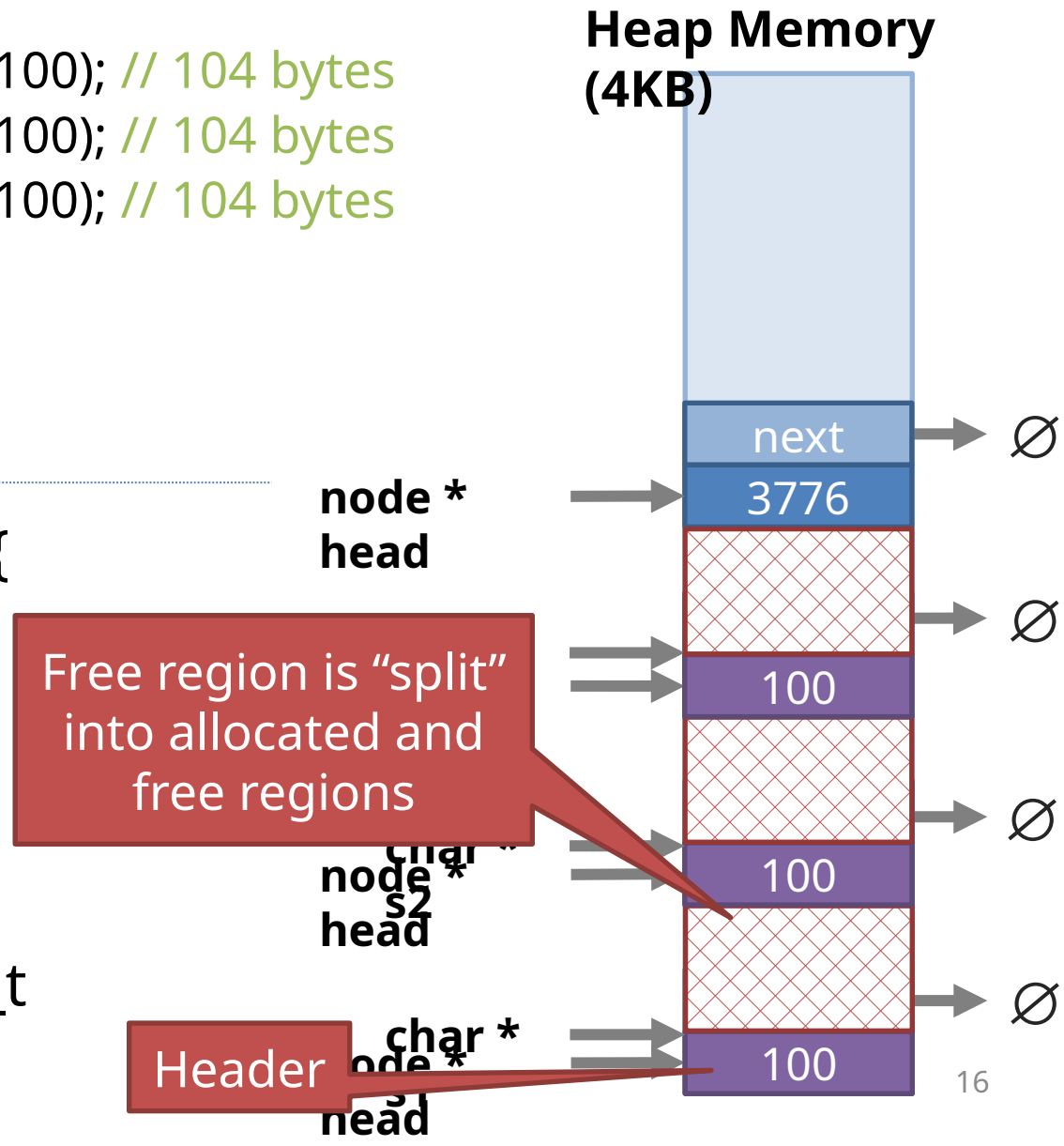
```
node * head = mmap(NULL, 4096,      PROT_READ |  
PROT_WRITE,  
                    MAP_ANON | MAP_PRIVATE, -1, 0);  
head->size = 4096 - sizeof(node);  
head->next = NULL;
```

Allocating Memory (Splitting)

```
char * s1 = (char *) malloc(100); // 104 bytes
char * s2 = (char *) malloc(100); // 104 bytes
char * s3 = (char *) malloc(100); // 104 bytes
```

```
typedef struct node_t {
    int size;
    struct node_t *
next;
} node;
```

```
typedef struct header_t
{
    int size;
```



Freeing Memory

- The free list is kept in sorted order
 - free() is an O(n) operation

```
free(s2); // returns 100 +
free(s1); // returns 100 +
free(s3); // returns 100 +
```

All memory is free, but the free list divided into four regions

```
typedef struct node_t {
    int size;
    struct node_t *
```

```
next;
} node;
```

```
typedef struct node_t {
    int size;
```

```
};
```

These pointers are "dangling": they still point to heap memory, but the pointers are invalid

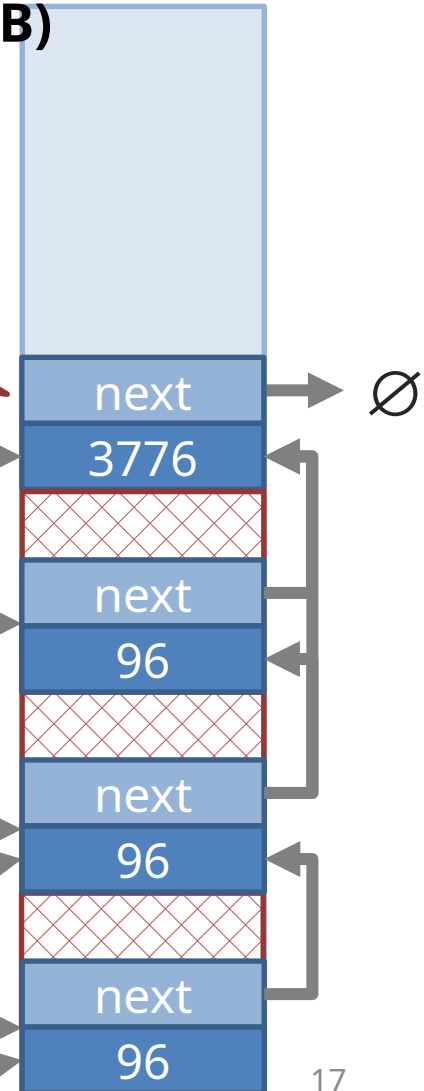
node *
head

char *
s3

char *
node *
head

char *
node *
head

Heap Memory
(4KB)



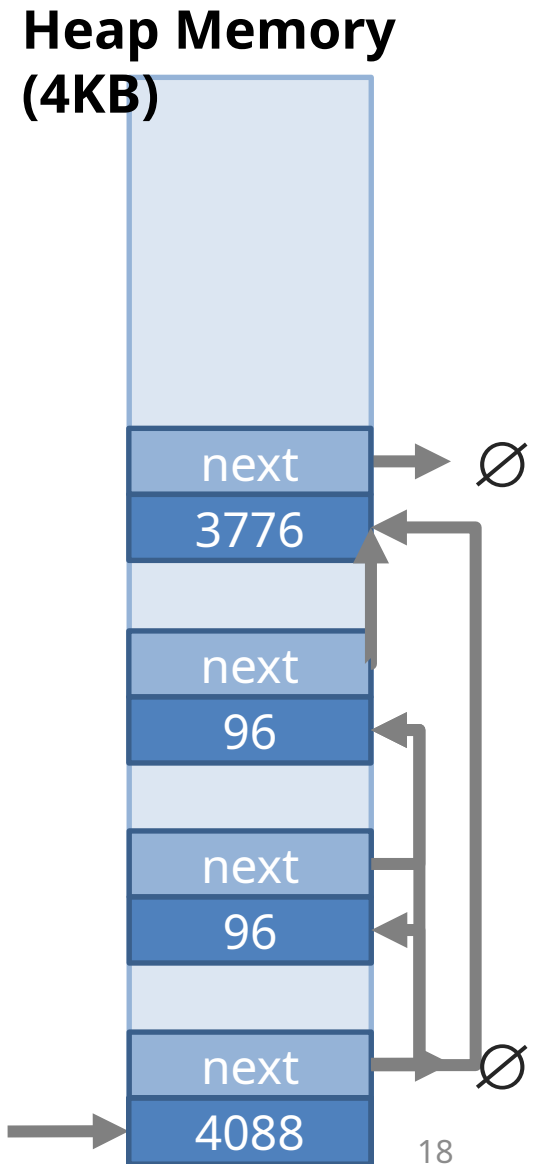
Coalescing

- Free regions should be merged with their neighbors
 - Helps to minimize fragmentation
 - This would be $O(n^2)$ if the list was not sorted

```
typedef struct node_t {
    int size;
    struct node_t *
next;
} node;
```

```
typedef struct header_t
{
    int size;
```

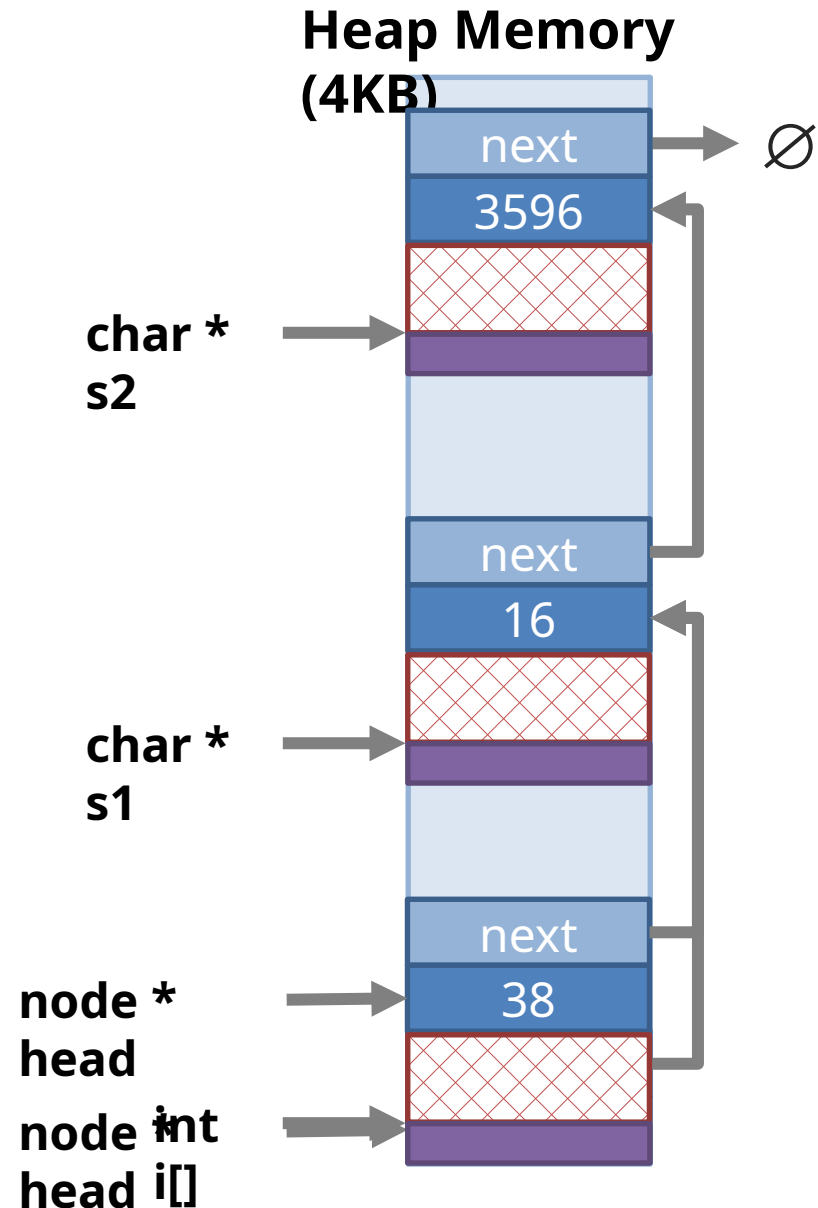
```
node *
head
```



Choosing Free Regions (1)

```
int i[] = (int*) malloc(8);  
// 8 + 4 = 12 total bytes
```

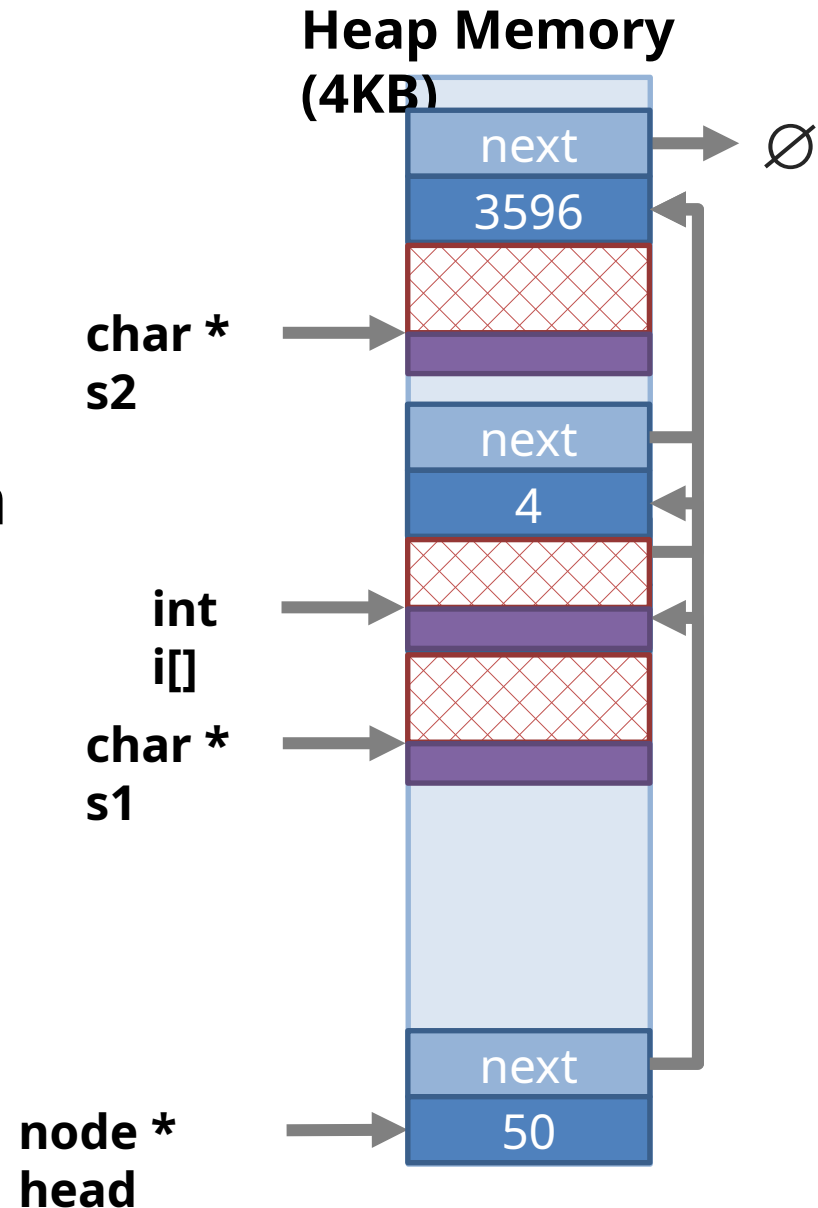
- Which free region should be chosen?
- Fastest option is **First-Fit**
 - Split the first free region with ≥ 8 bytes available
- Problem with First-Fit?
 - Leads to external fragmentation



Choosing Free Regions (2)

```
int i[] = (int*) malloc(8);  
// 8 + 4 = 12 total bytes
```

- Second option: **Best-Fit**
 - Locate the free region with size closest to (and \geq) 8 bytes
- Less external fragmentation than First-fit
- Problem with Best-Fit?
 - Requires $O(n)$ time



Basic Free List Review

- Singly-linked free list
- List is kept in sorted order
 - *free()* is an $O(n)$ operation
 - Adjacent free regions are coalesced
- Various strategies for selecting which free region to use for a given *malloc(n)*
 - First-fit: use the first free region with $\geq n$ bytes available
 - Worst-case is $O(n)$, but typically much faster
 - Tends to lead to external fragmentation at the head of the list
 - Best-fit: use the region with size closest (and \geq) to n
 - Less external fragments than first-fit, but $O(n)$ time

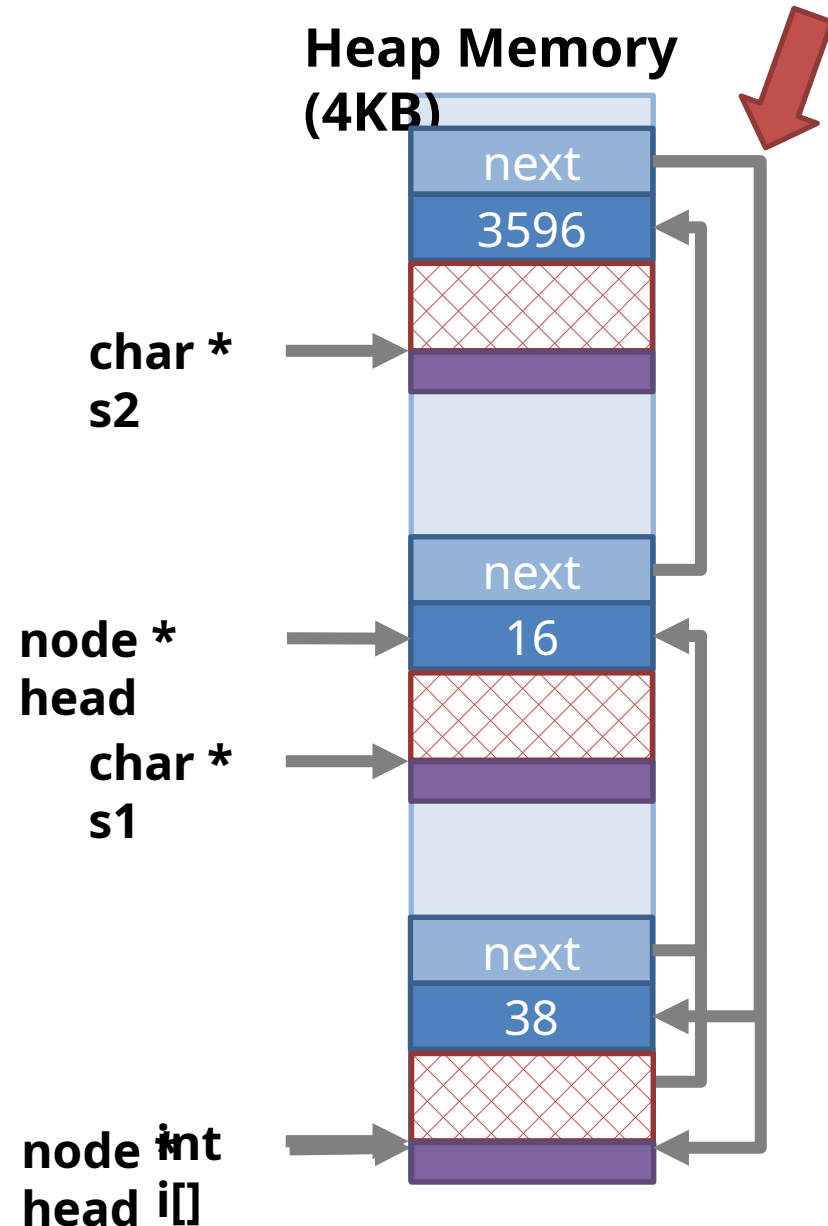
Improving Performance

1. Use a circular linked list and **Next-Fit**
 - Faster than Best-Fit, less fragmentation than First-fit
2. Use a doubly-linked free list with footers
 - Good: makes free() and coalesce $O(1)$ time
 - Bad: small amount of memory wasted due to headers and footers
3. Use bins to quickly locate appropriately sized free regions
 - Good: much less external fragmentation, $O(1)$ time
 - Bad: much more complicated implementation
 - Bad: some memory wasted due to internal fragmentation

Circular List and Next-Fit

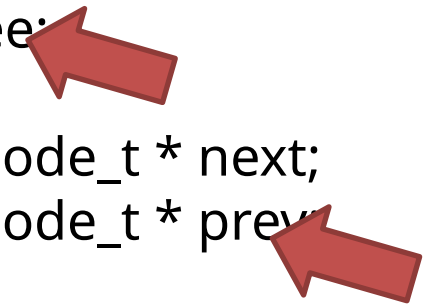
```
int i[] = (int*) malloc(8);
```

1. Change to a singly-linked, circular linked list
2. Use First-Fit, but move head after each split
 - Known as **Next-Fit**
 - Helps spread allocations, reduce fragmentation
 - Faster allocations than Best-Fit



Towards $O(1)$ *free()*

```
typedef struct node_t {  
    bool free;  
    int size;  
    struct node_t * next;  
    struct node_t * prev;  
} node;
```



```
typedef struct header_t {  
    bool free;  
    int size;  
} header;
```

```
typedef struct footer_t {  
    int size;  
} footer;
```

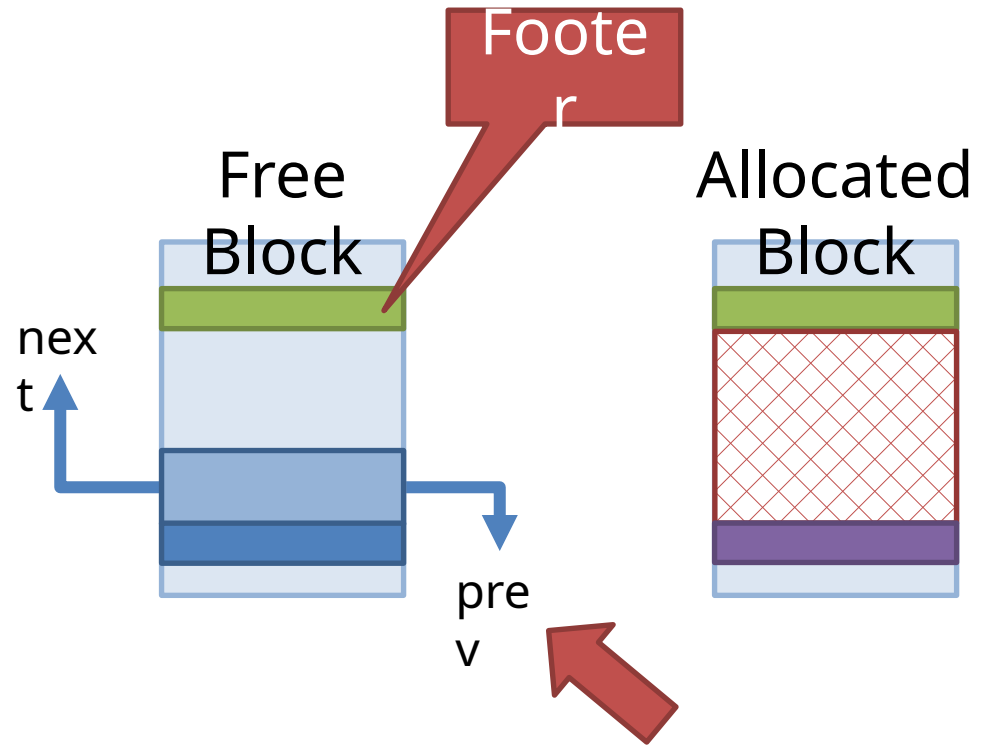
- *free()* is $O(n)$ because the free list must be kept in sorted order
- Key ideas:
 - Move to a doubly linked list
 - Add footers to each block
- Enables coalescing without sorting the free list
 - Thus, *free()* becomes $O(1)$

Example Blocks

```
typedef struct node_t {  
    bool free;  
    int size;  
    struct node_t * next;  
    struct node_t * prev;  
} node;
```

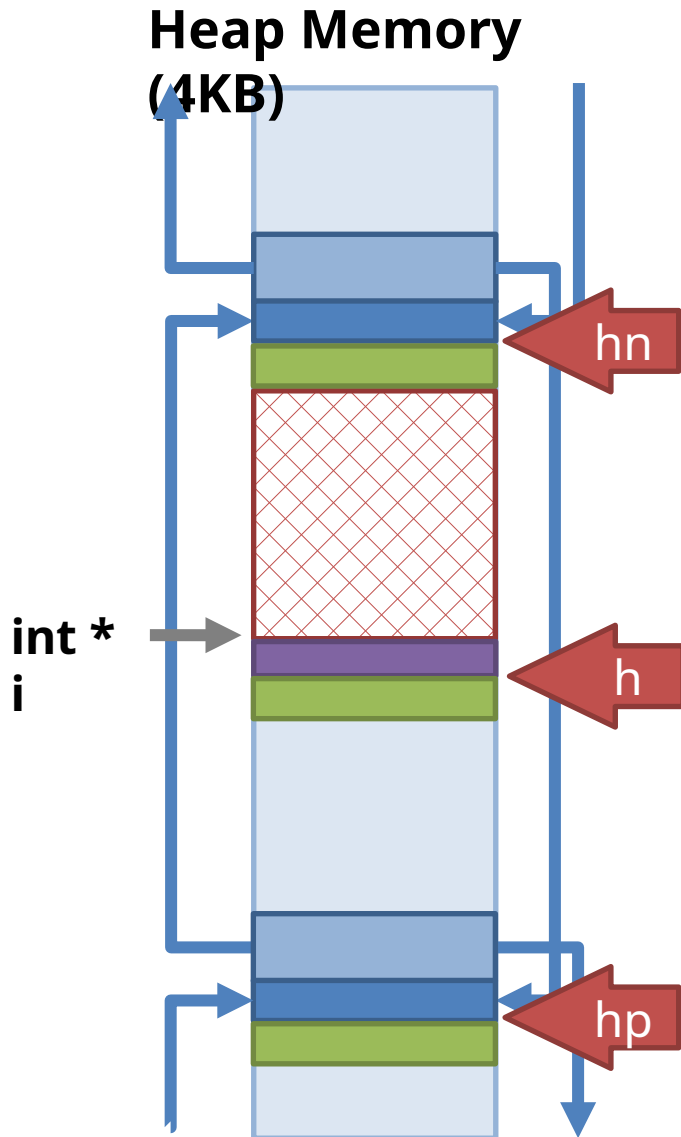
```
typedef struct header_t {  
    bool free;  
    int size;  
} header;
```

```
typedef struct footer_t {  
    int size;  
} header;
```



Locating Adjacent Free Blocks

- Suppose we have *free(i)*
- Locate the next and previous free blocks



```
char * p = (char *) i; // for convenience
// header of the current block
header * h = (header *) (p - sizeof(header));
// header of the next block
header * hn = (header *) (p + h->size +
sizeof(footer));
// previous footer
footer * f = (footer *) (p - sizeof(header) -
sizeof(footer));
// previous header
header * hp = (header *)
((char *) f - f->size - sizeof(header));
```

Coalescing is $O(1)$

```
node * n = (node *) h, nn, np;  
n->free = true;
```

```
if (hn->free) { // combine with the next free block
```

```
    nn = (node *) hn;
```

```
    n->next = nn->next;    n->prev = nn->prev;
```

```
    nn->next->prev = n;    nn->prev->next = n;
```

```
    n->size += nn->size + sizeof(header) + sizeof/footer);
```

```
    ((footer *) ((char *) n + n->size))->size = n->size;
```

```
}
```

```
if (hp->free) { // combine with the previous free block
```

```
    np = (node *) hp;
```

```
    np->size += n->size + sizeof(header) + sizeof/footer);
```

```
    ((footer *) ((char *) np + np->size))->size = np->size;
```

```
}
```

```
if (!hp->free && !hn->free) {
```

```
    // add the new free block to the head of the free list
```

```
}
```

- Be careful of corner cases:
 - The first free block
 - The last free block

Speeding Up *malloc()*

- At this point, *free()* is $O(1)$
- But *malloc()* still has problems
 - Next-Fit: $O(1)$ but more fragmentation
 - Best-Fit: $O(n)$ but less fragmentation
- Two steps to speed up *malloc()*
 1. Round allocation requests to powers of 2
 - Less external fragmentation, some internal fragmentation
 2. Divide the free list into **bins** of similar size blocks
 - Locating a free block of size *round(x)* will be $O(1)$

Rounding Allocations

- *malloc(size)*

size += sizeof(header) + sizeof/footer; // will always be >16 bytes

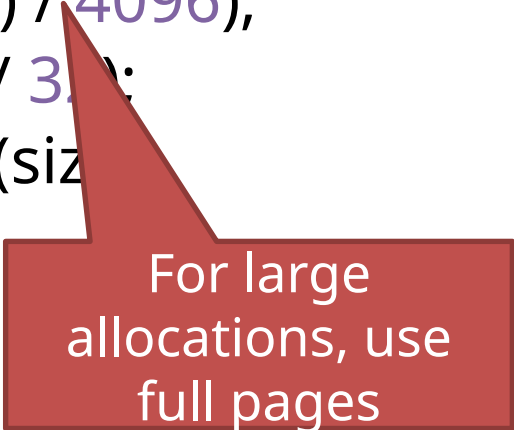
if (size > 2048) size = 4096 * ((size + 4095) / 4096);

else if (size < 128) size = 32 * ((size + 31) / 32);

else size = round_to_next_power_of_two(size);

- Examples:

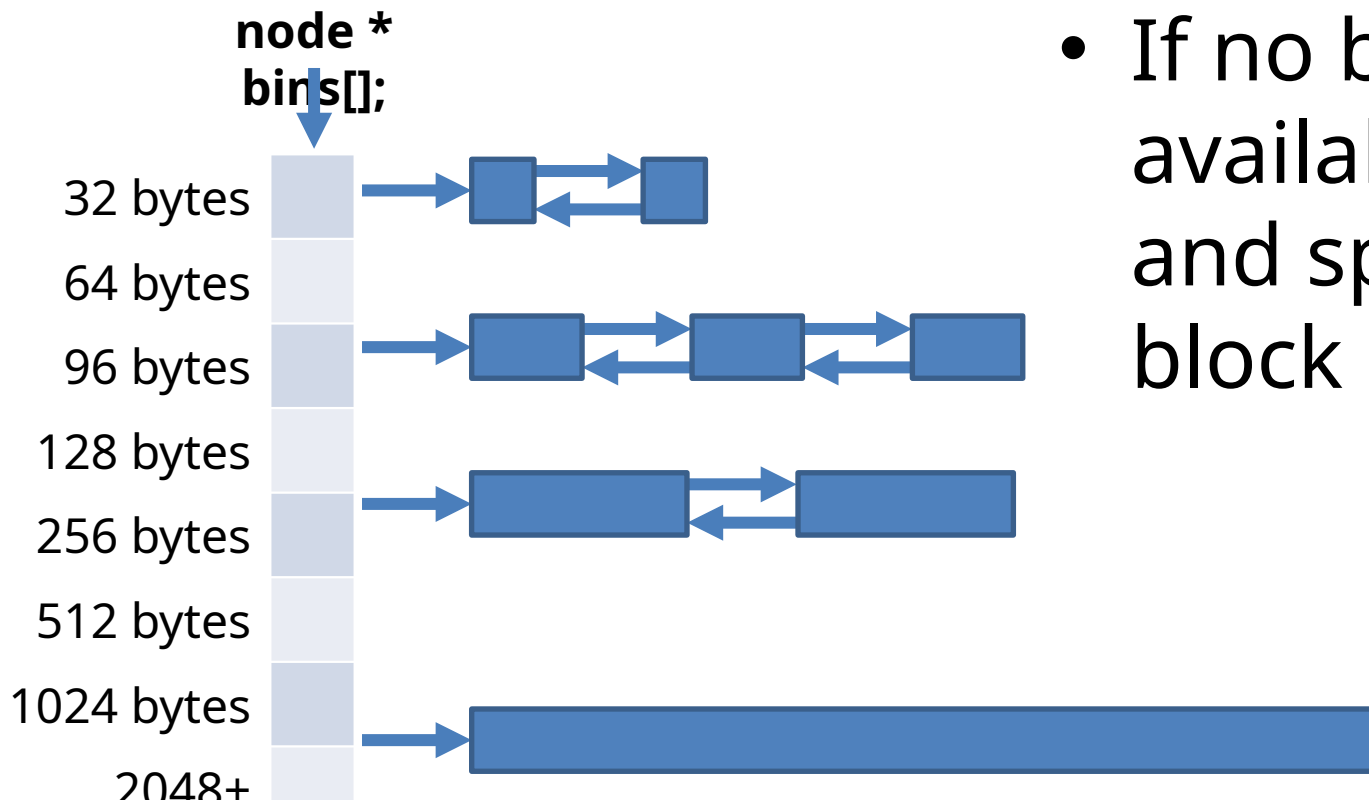
- *malloc(4)* → 32 bytes
- *malloc(45)* → 64 bytes
- *malloc(145)* → 256 bytes



For large allocations, use full pages

Binning

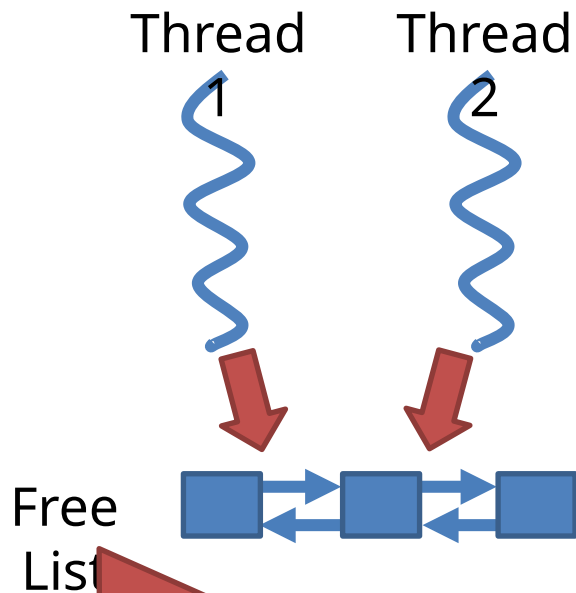
- Divided the free list into bins of exact size blocks
- Most allocations handled in $O(1)$ time by pulling a free block from the appropriate list



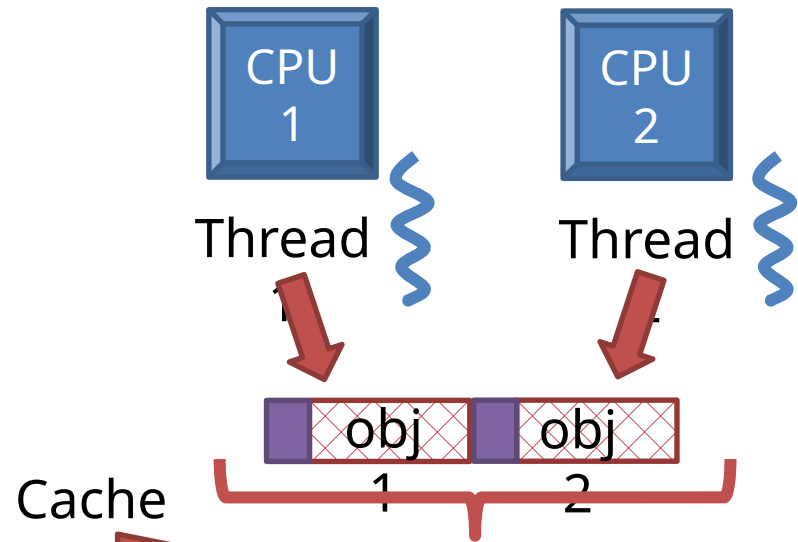
- If no block is available, locate and split a larger block

Next Problem: Parallelism

- Today's programs are often parallel
- However, our current memory manager has poor performance with >1 threads



The free list is shared, thus it must be protected by a mutex



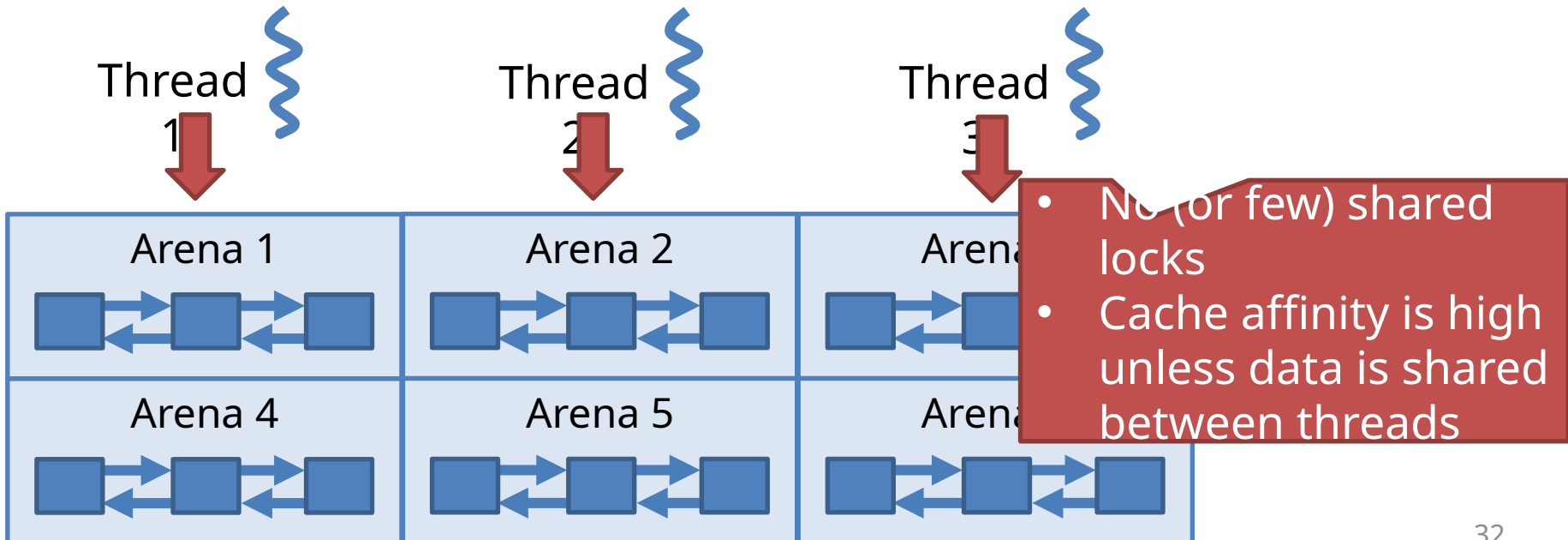
Cache

Line contention in memory

- Objects for different threads may share the same cache line
- This causes contention between CPU

Per-Thread Arenas

- To reduce lock and CPU cache contention, divide the heap into arenas
 - Each arena has its own free list
 - Each thread is assigned to several arenas



Two More Things

- How can you make your code manage memory more quickly?
 - Slab allocation
- Common memory bugs
 - Memory leaks
 - Dangling pointers
 - Double free

Speeding Up Your Code

- Typically, the memory allocation algorithm is not under your control
 - You don't choose what library to use (e.g. glibc)
 - You don't know the internal implementation
- How can you make your code faster?
 - Avoid the memory allocator altogether!
 - Use an **object cache** plus **slab allocation**

```
template<class T> class obj_cache {  
    private:
```

```
        stack<T *> free_objs;
```

```
        void allocate_slab() {
```

```
            malloc(sizeof(T) * 10);
```

```
            free_objs.push(&objs[x]);
```

```
        }
```

```
    public:
```

objects

```
    obj_cache() { allocate_slab(); } // start by pre-allocating some
```

```
    T * alloc() {
```

```
        allocate_slab(); // allocate more if we run out
```

available object

```
    }
```

```
    void free(T * obj) { free_objs.push(obj); } // return obj to the pool
```

```
};
```

- Objects are allocated in bulk
- Less space wasted on headers and footers

```
    T * objs = (T *)
```

```
    for (int x = 0; x < 10; ++x)
```

```
        if (free_objs.empty())
```

```
            T * obj = free_objs.top();
```

```
            free_objs.pop();
```

```
            return obj; // return an
```

Two More Things

- How can you make your code manage memory more quickly?
 - Slab allocation
- Common memory bugs
 - Memory leaks
 - Dangling pointers
 - Double free

Memory Management Bugs (1)

```
int search_file(char * filename, char * search) {
    unsigned int size;
    char * data;
    FILE * fp = fopen(filename, "r"); // Open the file
    fseek(fp, 0, SEEK_END);           // Seek to the end of the file
    size = ftell(fp);                 // Tell me the total length of the file
    data = (char *) malloc(size * sizeof(char)); // Allocate buffer
    fseek(fp, 0, SEEK_SET);           // Seek back to the beginning of the file
    fread(data, 1, size, fp);         // Read the file
    return strstr(data, search) > 0;
}

void main(int argc, char ** argv) {
    if (search_file(argv[1], argv[2])) printf("String '%s' found in file '%s'\n", argv[2], argv[1]);
    else printf("String '%s' NOT found in file '%s'\n", argv[2], argv[1]);
}
```

- We forgot to free(data)!
- If this program ran for a long time, eventually it would exhaust all available virtual memory

Memory Management Bugs (2)

- Dangling pointer

```
char * s = (char *) malloc(100);
```

```
...
```

```
free(s);
```

```
...
```

```
puts(s);
```

- Behavior is nondeterministic
- If the memory has not been reused, may print `s`
- If the memory has been recycled, may print garbage

- Double free

```
char * s = (char *) malloc(100);
```

```
...
```

```
free(s);
```

```
...
```

```
free(s);
```

- Typically, this corrupts the free list
- However, your program may not crash (nondeterminism)
- In some cases, double free bugs are **exploitable**

- Free Lists
 - Basics
 - Speeding Up *malloc()* and *free()*
 - Slab Allocation
 - Common Bugs
- Garbage Collectors
 - Reference Counting
 - Mark and Sweep
 - Generational/Ephemeral GC
 - Parallel Garbage Collection

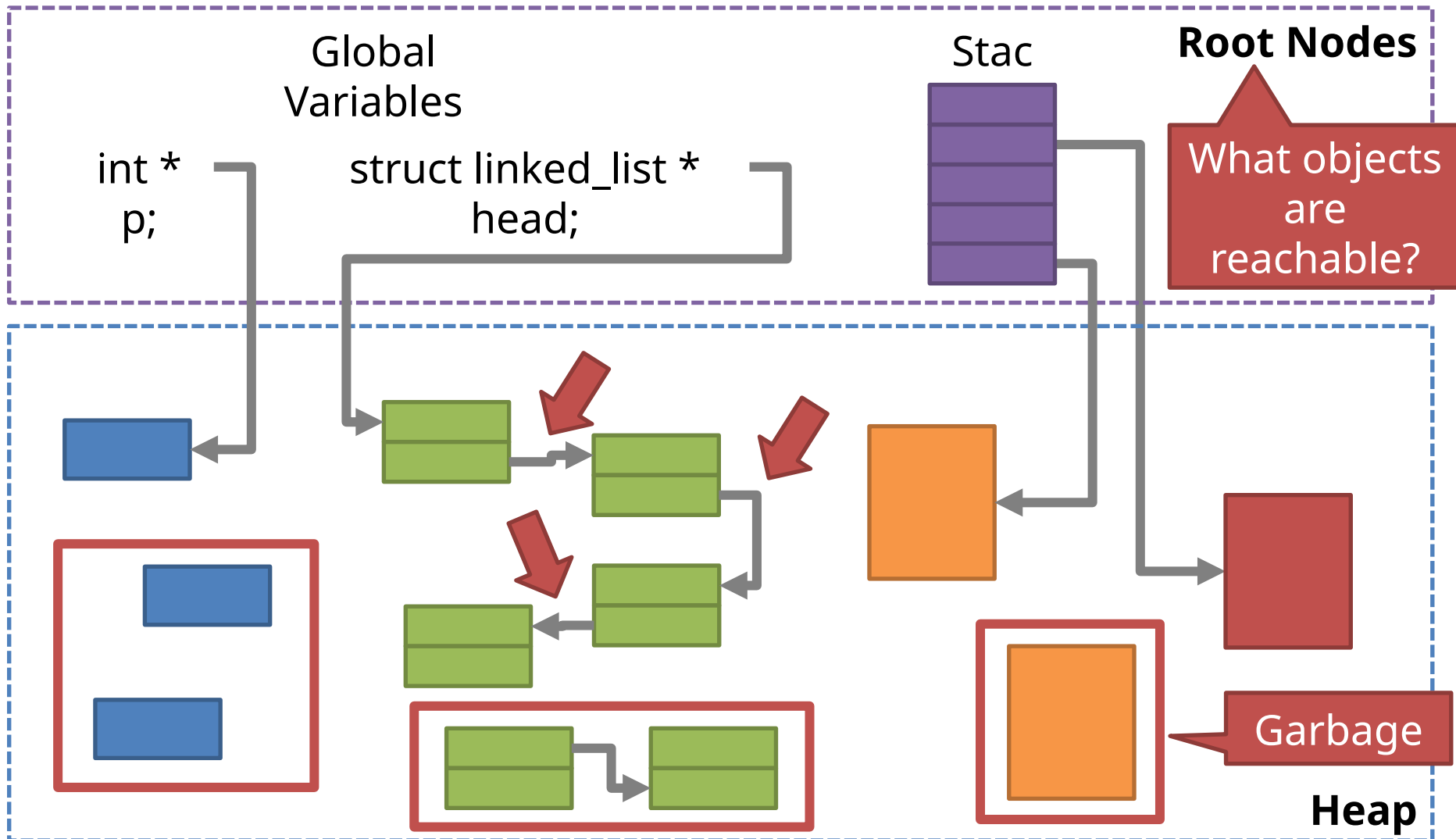
Brief Recap

- At this point, we have thoroughly covered how *malloc()* and *free()* can be implemented
 - Free lists of varying complexity
 - Modern implementations are optimized for low fragmentation, high parallelism
- What about languages that automatically manage memory?
 - Java, Javascript, C#, Perl, Python, PHP, Ruby, etc...

Garbage Collection

- Invented in 1959
- Automatic memory management
 - The GC reclaims memory occupied by objects that are no longer in use
 - Such objects are called **garbage**
- Conceptually simple
 1. Scan objects in memory, identify objects that cannot be accessed (now, or in the future)
 2. Reclaim these garbage objects
- In practice, very tricky to implement

Garbage Collection Concepts



Identifying Pointers

- At the assembly level, anything can be a pointer

```
int x = 0x80FCE42;
```

```
char * c = (char *) x;    // this is legal
```

- Challenge: how can the GC identify pointers?
 1. Conservative approach: assume any number that might be a pointer, is a pointer

- Problem: may erroneously determine (due to false pointers) that some blocks of memory are in use

2. Deterministic approach: use a type-safe language that does not allow the

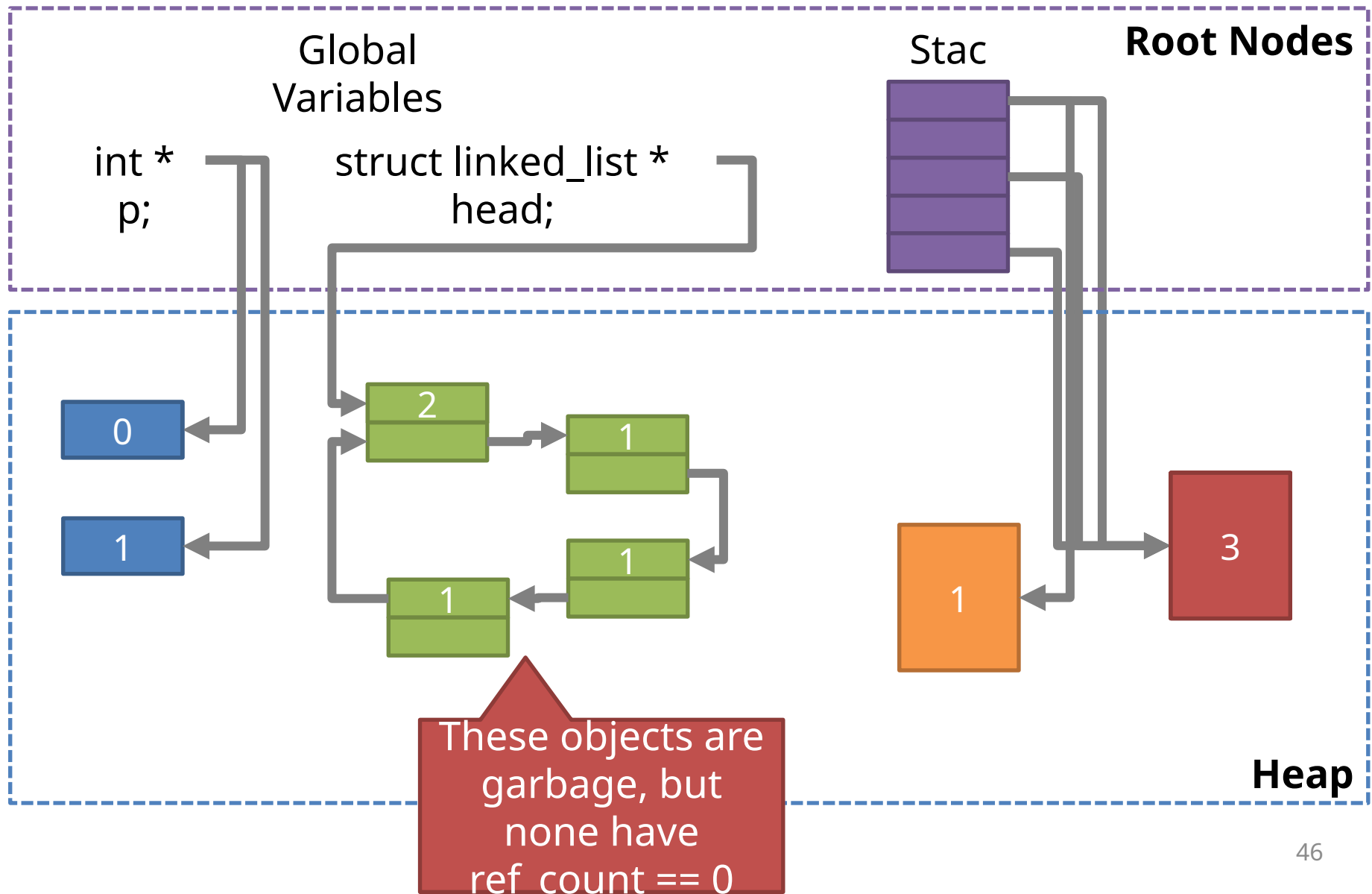
Approaches to GC

- Reference Counting
 - Each object keeps a count of references
 - If an objects count == 0, it is garbage
- Mark and Sweep
 - Starting at the roots, traverse objects and “mark” them
 - Free all unmarked objects on the heap
- Copy Collection
 - Extends mark & sweep with compaction
 - Addresses CPU and external fragmentation issues
- Generational Collection
 - Uses heuristics to improve the runtime of mark & sweep

Reference Counting

- Key idea: each object includes a `ref_count`
 - Assume `obj * p = NULL;`
 - `p = obj1; // obj1->ref_count++`
 - `p = obj2; // obj1->ref_count--, obj2->ref_count++`
- If an object's `ref_count == 0`, it is garbage
 - No pointers target that object
 - Thus, it can be safely freed

Reference Counting Example



Pros and Cons of Reference Counting

The Good

- Relatively easy to implement
- Easy to conceptualize

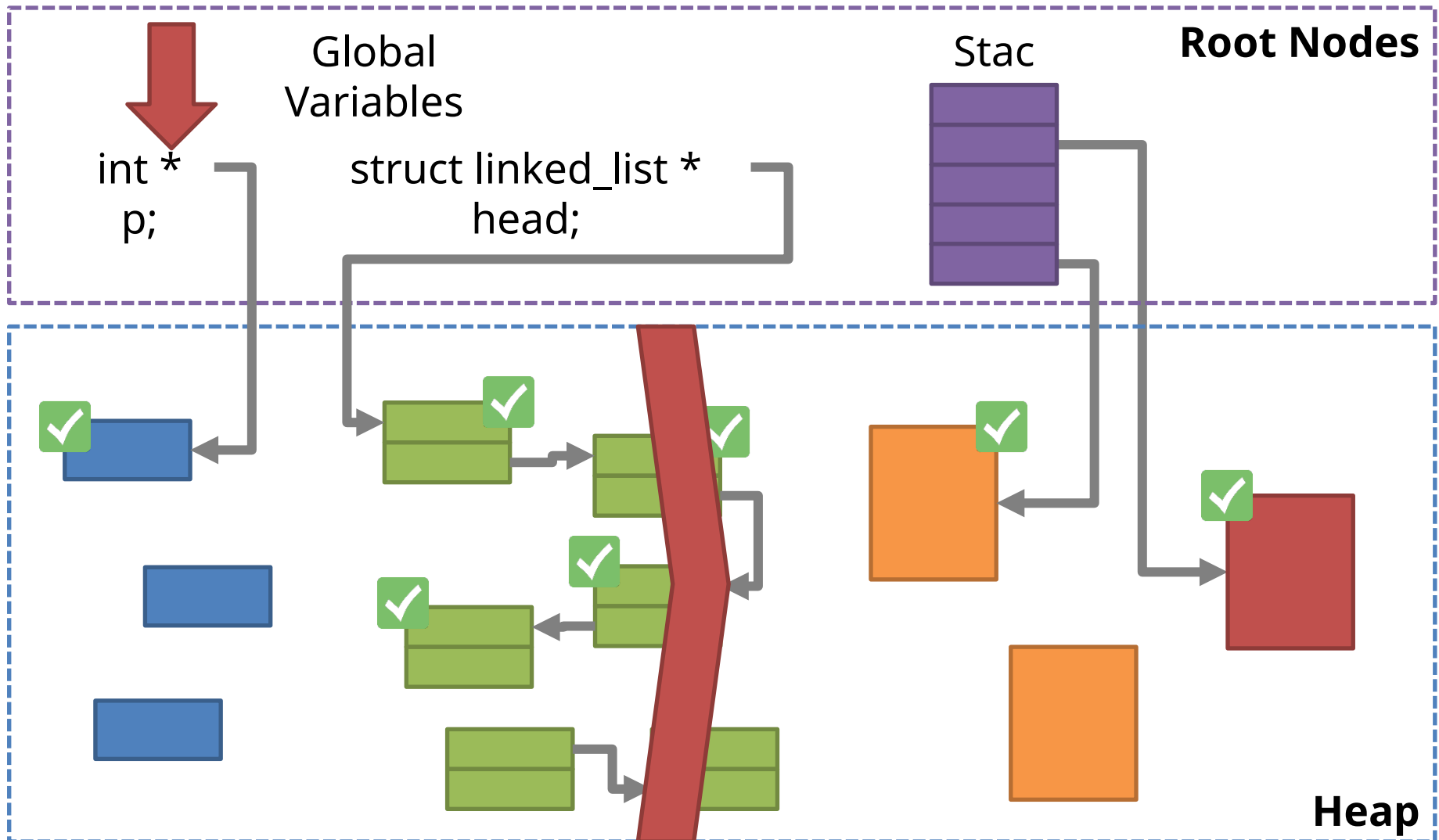
The Bad

- Not guaranteed to free all garbage objects
- Additional overhead (`int ref_count`) on all objects
- Access to `obj->ref_count` must be synchronized

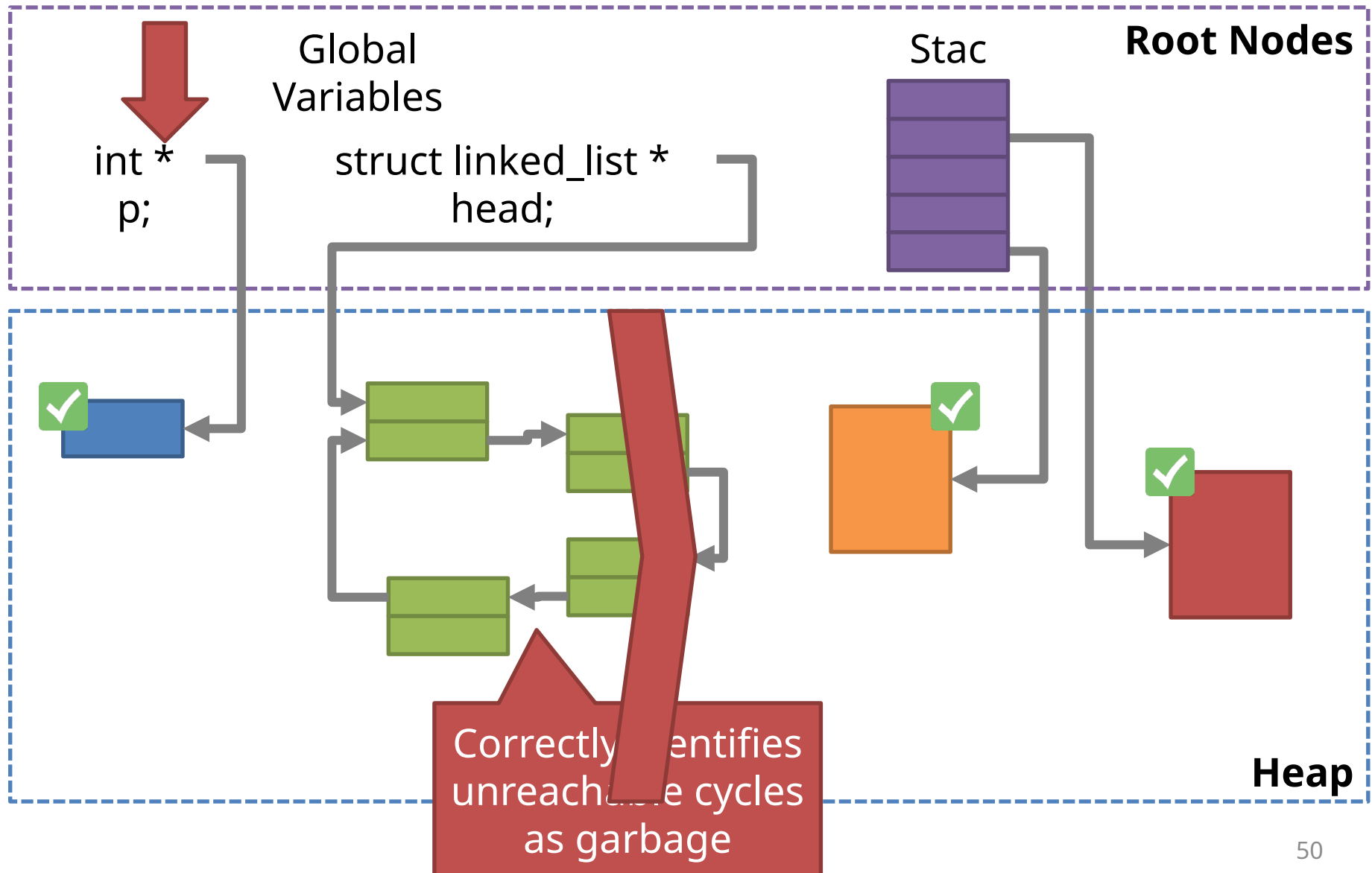
Mark and Sweep

- Key idea: periodically scan all objects for reachability
 - Start at the roots
 - Traverse all reachable objects, mark them
 - All unmarked objects are garbage

Mark and Sweep Example



Mark and Sweep Example



Pros and Cons of Mark and Sweep

The Good

- Overcomes the weakness of reference counting
- Fairly easy to implement and conceptualize
- Guaranteed to free all garbage objects

Be careful. If you forget to set a reference to NULL, it will never be collected (i.e. Java can leak memory)

The Bad

- Mark and sweep is CPU intensive
 - Traverses all objects reachable from the root
 - Scans all objects in memory freeing unmarked objects
- Naïve implementations “stop the world” before collecting
 - Threads cannot run in parallel with the GC
 - All threads get stopped while the GC runs

Copy Collection

- Problem with mark and sweep:
 - After marking, all objects on the heap must be scanned to identify and free unmarked objects
- Key idea: use compaction (aka relocation)
 - Divide the heap into *start space* and *end space*
 - Objects are allocated in *start space*
 - During GC, instead of marking, copy live object from *start space* into *end space*
 - Switch the *space* labels and continue

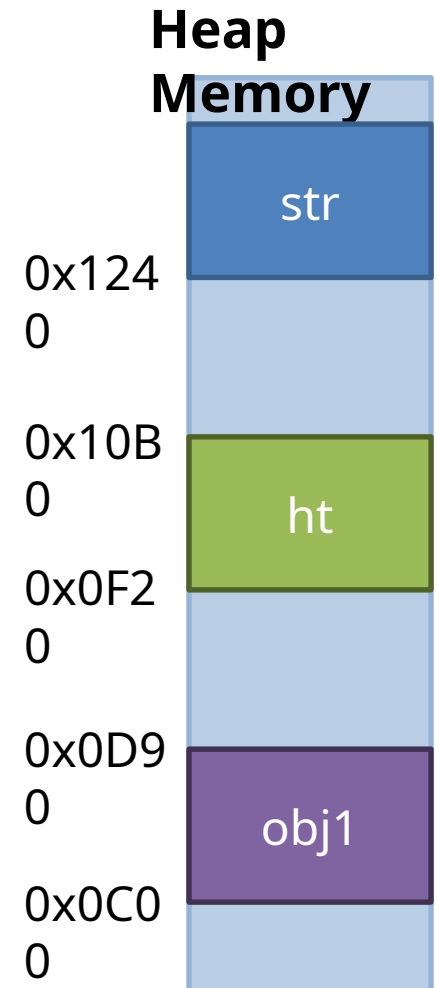
Compaction/Relocation

String str2 = new String();

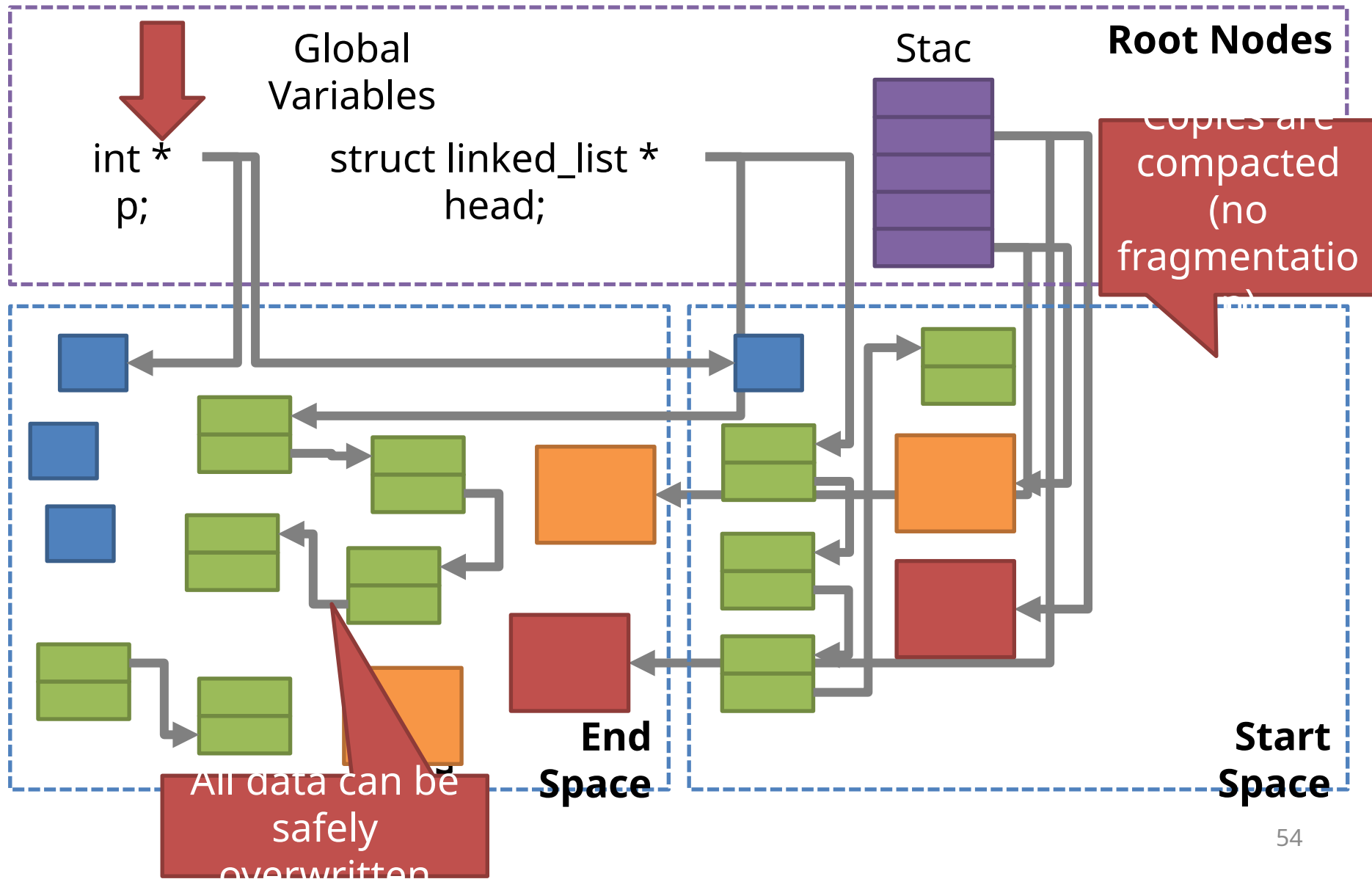
Pointer	Value	Location
obj1	0x0D90	0x0D90
ht	0x0F20	0x0F20
str	0x1240	0x1240
str2	???	???



- One fragmentation action
 - Copy allocated blocks of memory into a contiguous region of memory
 - Repeat this process periodically
- This only works if pointers are boxed, i.e. managed by the runtime



Copy Collection Example



Pros and Cons of Copy Collection

The Good

- Improves on mark and sweep
- No need to scan memory for garbage to free
- After compaction, there is no fragmentation

The Bad

- Copy collection is slow
 - Data must be copied
 - Pointers must be updated
- Naïve implementations are not parallelizable
 - “Stop the world” collector

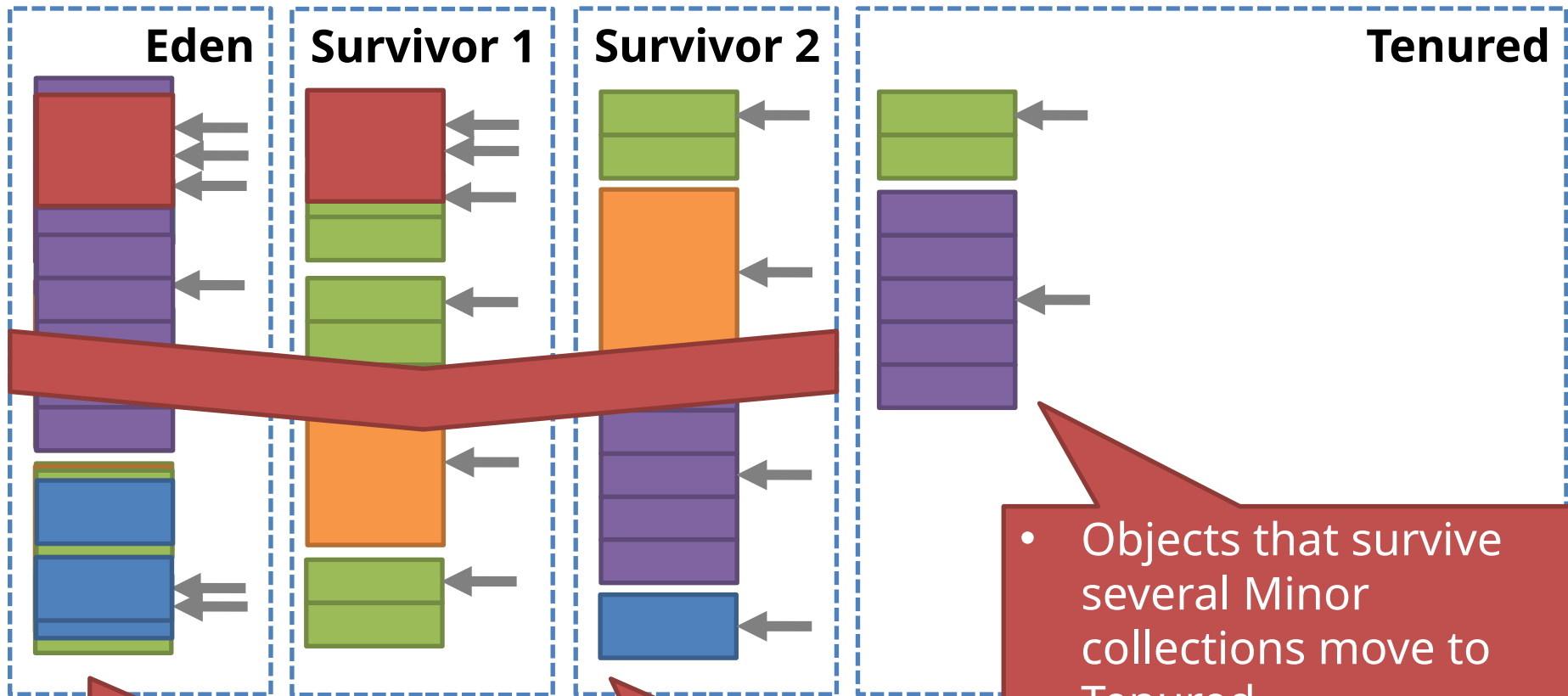
Generational Collection

- Problem: mark and sweep is slow
 - Expensive full traversals of live objects
 - Expensive scan of heap memory
- Problem: copy collection is also slow
 - Expensive full traversals of live objects
 - Periodically, all live objects get copied
- Solution: leverage knowledge about object creation patterns
 - Object lifetime tends to be inversely correlated with likelihood of becoming garbage (generational hypothesis)
 - Young objects die quickly – old objects continue to live

Garbage Collection in Java

- By default, most JVMs use a generational collector
- GC periodically runs two different collections:
 1. Minor collection – occurs frequently
 2. Major collection – occurs infrequently
- Divides heap into 4 regions
 - Eden: newly allocated objects
 - Survivor 1 and 2: objects from Eden that survive minor collection
 - Tenured: objects from Survivor that survive several minor collections

Generational Collection Example



- Minor collection occurs whenever Eden gets full
- Live objects are copied to Survivor

- Survivor 1 and 2 rotate as destinations for a copy collector

- Objects that survive several Minor collections move to Tenured
- Tenured objects are only scanned during Major collection
- Major collections occur infrequently

More on Generational GC

- Separating young and old objects improves performance
 - Perform frequent, minor collections on young objects
 - No need to scan old objects frequently
- Copy collection reduces fragmentation
 - Eden and Survivor areas are relatively small, but they are frequently erased

Parallel and Concurrent GC

- Modern JVMs ship with multiple generational GC implementations, including:
 - The Parallel Collector
 - Runs several GC threads in parallel with user threads
 - Multiple GC threads take part in each minor/major collection
 - Best choice if your app is intolerant of pauses
 - The Concurrent Mark and Sweep Collector
 - Also implements multi-threaded GC
 - Pauses the app, uses all CPU cores for GC
 - Overall fastest GC, if your app can tolerate pauses

malloc()/free() vs. GC

Explicit Alloc/Dealloc

- Advantages:
 - Typically faster than GC
 - No GC “pauses” in execution
 - More efficient use of memory
- Disadvantages:
 - More complex for programmers
 - Tricky memory bugs
 - Dangling pointers
 - Double-free
 - Memory leaks
 - Bugs may lead to security vulnerabilities

Garbage Collection

- Advantages:
 - Much easier for programmers
- Disadvantages
 - Typically slower than explicit alloc/dealloc
 - Good performance requires careful tuning of the GC
 - Less efficient use of memory
 - Complex runtimes may have security vulnerabilities
 - JVM gets exploited all the

Other Considerations

- Garbage collectors are available for C/C++
 - Boehm Garbage Collector
 - Beware: this GC is conservative
 - It tries to identify pointers using heuristics
 - Since it can't identify pointers with 100% accuracy, it must conservatively free memory
- You can replace the default *malloc()* implementation if you want to
 - Example: Google's high-performance tcmalloc library
 - <http://goog-perftools.sourceforge.net/doc/tcmalloc.html>

Sources

- Slides by Jennifer Rexford
 - <http://www.cs.princeton.edu/~jrex>
- Operating Systems: Three Easy Pieces, Chapter 17 by Remzi and Andrea Arpaci-Dusseau
 - <http://pages.cs.wisc.edu/~remzi/OSTEP/vm-freespace.pdf>
- Java Platform, Standard Edition HotSpot Virtual Machine Garbage Collection Tuning Guide by Oracle (Java SE v. 8)
 - <http://docs.oracle.com/javase/8/docs/technotes/guides/vm/gctuning/toc.html>