Dynamic Storage Management

Mendel Rosenblum

Dynamic Storage Management

Optional readings: Operating Systems: Principles and Practice: None

How to manage a region of memory or storage to satisfy various needs?

- Both in applications and OS
- Both memory (now) and disk storage (later)
- Challenge is unpredictability

Dynamic Storage Operations

- `allocate(size) ⇒ ptr`
  - Allocate a block with a given number of bytes
- `free(ptr)`
  - Free a previously allocated block

Challenge: **unpredictability**

Don't know how long until an allocated block is freed

Super hard problem

General approaches to allocation

- Stack allocation (hierarchical)
  - Restricted support to an particular allocation/free pattern
  - Simple and efficient to implement
- Heap allocation
  - Supports arbitrary allocation/free
  - Hard to implement efficiently

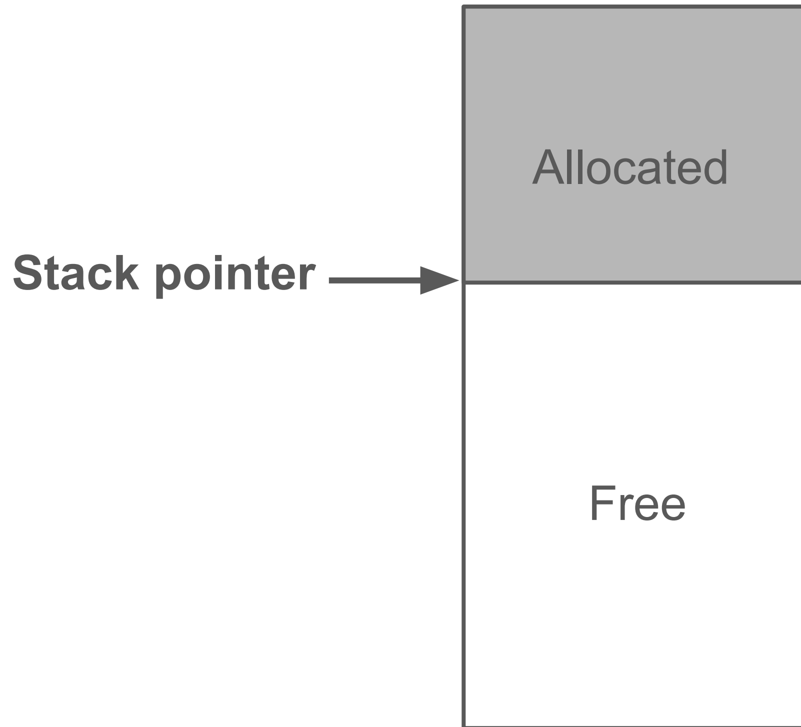
Example of Stack and Heap Allocation & Free

```
int global = 7;           // global data
int* gptr = &global;      // global pointer (contains an address)

void func(int x) {
    int local = x;        // stack variable
    int* lptr = &local;   // stack pointer (contains an address)

    int* heap = new int(42);  // heap object
    lptr = heap;              // pointer now holds heap address

    delete heap;
}
```

Stack Allocation

- Works when last allocated = first freed (LIFO) - **predictable**
  - Single stack pointer implementation:
    - Allocate: adjust pointer
    - Free: adjust pointer back
- Very efficient
  - Example: procedure call. X calls Y calls Y again.
- Other examples of of LIFO allocation patterns:
  - tree traversal
  - expression evaluation,
  - top-down recursive descent parsers

Stack allocators make free space management easy

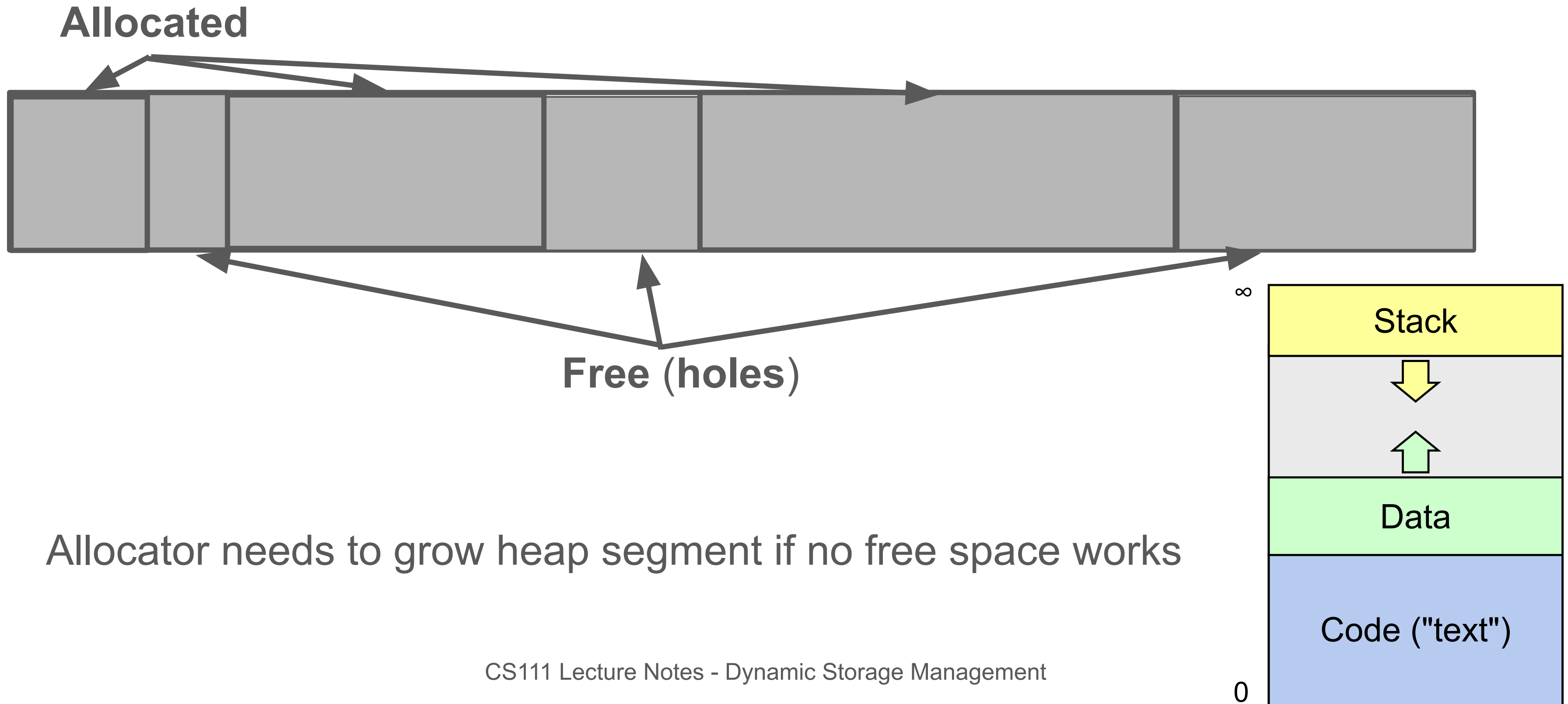


- Allocated space is contiguous in memory
- Free space is contiguous in memory
- Alloc/free ops are add and subtract
- Requires predictability (LIFO) 😞

Heap Allocation (not the heap data structure)

- Needed when freeing is unpredictable
  - Example: tree and graph structures, complex data organizations
- Memory can get broken up into allocated and free chunks

Heap allocation tracking free and in-use

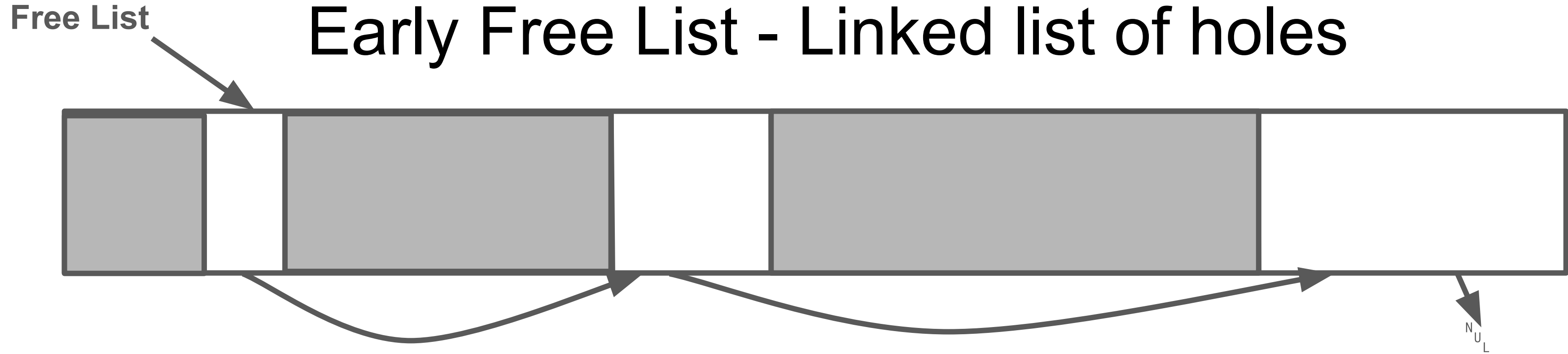


Allocator needs to grow heap segment if no free space works

Fragmentation and Free List

- Goal: Keep the number of holes small, keep their size large
- **Fragmentation**: inefficient use of memory because of lots of small holes
  - Stack allocation has no fragmentation
- Heap allocators need to track free storage (holes): **free list**
  - Many proposed management algorithm for free lists
  - Scan free list looking for a hole that works

Early Free List - Linked list of holes



- Best fit
  - Scan list on allocation, return closest to allocation size, put extra space back on list
- First fit
  - Like best fit but stop scanning as soon as working free region is found

During free operations, merge adjacent free blocks

Issues with first fit and best fit

- Holes tend to fragment
- Resulting holes the size of the small objects
- Failures:
  - Large allocations fail or heap grows
  - Expensive allocations (scanning over small holes)

Slab allocator

- **Slab**: a region of memory that is divided up into chunks of the same size
  - Pool for each popular allocation size
  - Free list for each slab

Slab

Slab allocation and freeing

- Allocation:
  - Go to the slab for the requested size
  - Allocate from its free list
  - If no free objects allocate a new slab, chop it up into free blocks, allocate
- Free:
  - Return block to the free list of the slab containing it
  - If all the blocks in a slab become free, then the slab can be freed

Slab evaluation

- Allocation and freeing super fast in the common case
- Commonly used in OSes
  - A small set of alloc/free sizes (e.g. struct sizes)
- Does it eliminate fragmentation?

No: Unused space inside of a slab

Particularly bad if allocation sizes change

Bit maps for free list implementation

- Bit map - An array of bits
  - Each bit tracks the allocation status of chunk of memory (0 means free, 1 allocated)
- Useful when managing fixed-sized chunks (e.g. within a slab, disk blocks)
- Need to acan arrays of bits to find free blocks
- Will discuss bit maps in more detail on file system lectures

Storage Reclamation

- How to know when dynamically-allocated memory can be freed?
  - When it will no longer be accessed
- Assume only accessible if there exists a pointer to the data
  - No pointer: can't access and safe to free
- Finding pointers
  - Easier when memory is only used in one place (e.g. nodes of a tree)
  - Hard when shared: can't free until all are finished

What can go wrong with memory reclamation?

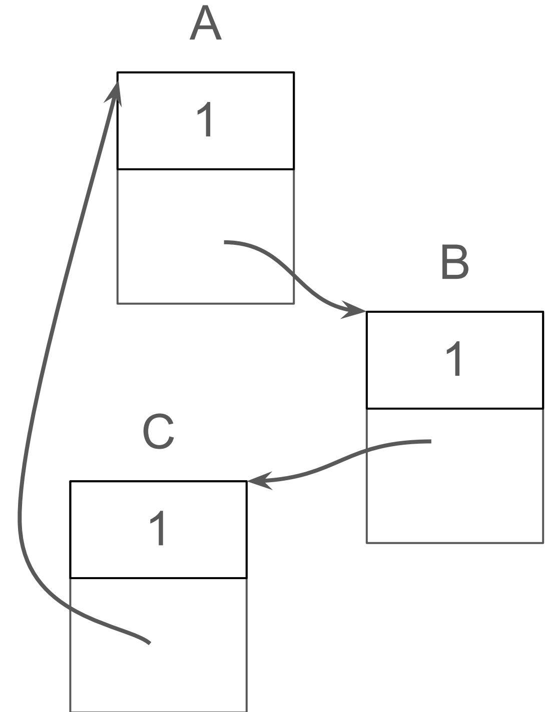
- **Dangling pointers**
  - Free too soon: free memory while it's still being used
- **Memory leaks**
  - Storage gets "lost" because no one freed it even though it can't ever be used again
- Two solutions to problems:
  - Reference counts
  - Garage collection

Which is worst?

Reference Counts

- Reference counting:
  - Object keeps count of pointers to it
  - When an object's count hits zero: free object
- Examples:
  - `std::shared_ptr` in C++
  - Early versions of JavaScript and in Python
  - Inodes in a file system
- What causes problems for reference counting?

  Cycles



Garbage Collection

- Basic operation
  - No free operation: Just delete the pointers
  - Garbage collector:
    - Scan memory finding all live objects (has pointer to it), delete rest
    - Can also **compact** memory to get rid fragmentation
  - Loop: run creating garbage; collect garbage
- Used by some popular modern languages:
  - Java, Javascript, Go, some Python implementations (cycle detector)

One implementation: mark and sweep

- Need to: Find all objects and Find all pointers to objects
- Pass 1 (mark)
  - Go through all statically-allocated and procedure-local variables finding pointers (roots)
  - Mark each object pointer to
    - Recursively mark all objects they point to
  - Need help from language environment to find pointers
- Pass 2 (sweep): copy and compact
  - Go through all objects, copy live objects into contiguous memory
  - Must update pointers of moved object
  - Free any remaining space

Garbage collection is expensive

- 10-20% of all CPU time in systems that use it
- Uses memory inefficiently: 2-5x overallocation
- Long pauses during garbage collection