Lecture 5

Memory Management

- 1) Allocate/assign physical memory to processes
    - explicit requests: malloc(sbrk)
    - implicit: program loading, stack extension
- 2) manage the virtual address space
    - instantiate virtual address space on context switch
    - extend or reduce it on demand
- 3) manage migration to/from secondary storage
    - optimize use of main storage
    - minimize overhead (waste, migrations)

sbrk(2) vs malloc(3)

- sbrk(2): managing size of data segment
    - each address space has a private data segment
    - process can request that it be grown/shrunk
    - sbrk(2) specifies desired ending address
    - this is a coarse and expensive operation so you never want to explicitly call it
- malloc(3): dynamic heap allocation
    - sbrk(2) is called to extend/shrink the heap
    - malloc(3) is called to carve off small pieces
    - mfree(3) is called to return them to the heap

Managing process private data

- 1) loader allocates space for, and copies initialized data from load module
- 2) loader allocates space for, and zeroes uninitialized data from load module
- 3) after it starts, program uses sbrk to extend the process data segment and then puts the newly created chunk on the free list
- 4) Free space “heap” is eventually consumed program uses sbrk to further extend the process data segment and then puts the newly created chunk of the free list

Free lists: keeping track of it all

- fixed sized blocks are easy to track
    - a bit map indicating which blocks are free
- variable chunks require more information
    - a linked list of descriptors, and one per chunk
    - each lists size of chunk, whether it is free
    - each has pointer to next chunk on list
    - descriptors otten at front of each chunk
- allocated memory may have descriptors too

Avoid Creating Small Fragments

- Choose carefully which piece to carve
- careful choices mean longer searches
- optimization means more complex data structures
- cost of reduced fragmentation is more cycles
- A few obvious choices for “smart choices”
    - best fit, worst fit, first fit, next fit
- Choose how to carve carefully to avoid fragmentation

Which chunk: best fit

- Search for the “best fit” chunk
- advantages
    - might find a perfect fit
- disadvantages
    - have to search entire list every time -> inefficient
    - quickly creates very small fragments
    - leads to great amounts of external fragmentation

Which chunk: Worst fit

- search for the “worst fit” chunk
    - the largest size greater/equal to requested size
- advantages
    - tends to create very large fragments
    - slows down fragmentation
- disadvantages
    - still have to search through the entire list every time

Which chunk: first fit

- take first chunk that is big enough
- advantages
    - very short searches
    - creates random sized fragments
- disadvantages
    - the first chunks quickly fragment
    - searches become longer later
    - ultimately it fragments as badly as best fit

Which chunk: next fit

- best of both worlds
    - short searches (maybe shorter than first fit)
    - spreads out fragmentations (like worst fit)
- guess pointers are a general technique
    - think of them as a lazy (non-coherent) cache
    - if they are right, they save a lot of time

    - if they are wrong, the algorithm still works
    - they can be used in a wide range of problems
- After each search, set the guess pointer to the chunk after the one we chose
    - That is the point at which we will begin our next search

Coalescing: De-fragmentation

- All dynamic algorithms have external fragmentation
    - some get it faster, some spread it out more evenly
- We need a way to reassemble fragments
    - check neighbors whenever a chunk is freed
    - recombine w/free neighbors whenever possible
    - free list can be designed to make this easier
        - e.g.: address-sorted doubly linked list of all chunks
        - e.g. “Buddy” allocation w/implicit neighbors
    - counters forces of external fragmentation
- Dynamic equilibrium between carving process and coalescing process
- Coalescing assumes that you can find the addresses adjacent to the chunks in the free list

Coalescing vs. Fragmentation (Carving)

- opposing processes operate in parallel
    - which of the two processes will dominate?
- What fraction of space is typically allocated?
    - coalescing works better with more free space
- how fast is allocated memory turned over?
    - chunks held for long time cannot be coalesced
- how variable are the requested chunk sizes
    - high variability increases fragmentation rate
- how long will the program execute
    - fragmentation, like rust, gets worse with time
- Can coalescing overcome fragmentation?
    - Depends on what you are dealing with

Memory Allocation Requests

- memory requests are not well distributed
    - some sizes are used much more than others
- many key services use fixed-size buffers
    - file systems (disk I/O)
    - network protocols (for packet assembly)
    - standard request descriptors
- these account for much transient use
    - they are continuously allocated and freed

Special Buffer Pools

- There are popular sizes
    - reserve special pools of particular size buffers
    - allocate/free matching requests from those pools
- benefit: improved efficiency
    - much simpler than variable partition algorithm
    - reduces (or eliminates) external fragmentation
- but… we must know how much to reserve
    - too little: buffer pool will become bottleneck
    - too much: we will have lots of idle space

Balancing Space for Buffer Pools

- Many different special purpose pools
    - demand for each changes continuously
    - memory needs to migrate between them
- sounds like a dynamic equilibrium problem!
    - voluntarily managed free space margins
        - a maximum allowable free space per service
        - graceful handling of changing loads
    - claw-back call-back from OS to services
        - OS requests services to free all available memory
        - prompt handling of emergencies
        - Everybody gives back what they can, and eventually you'll have enough space
        - In most cases, we know what the pigs (ones that use lots of memory) are and the OS usually call those ones back

Buffer Pools: Slab Allocation

- requests are not merely for common sizes
    - they are often for the same data structure
    - or even assemblies of data structures
- Initializing and demolition are expensive
    - many fields and much structure are constant
- stop destroying and reinitializing
    - recycle data structures (or assemblies)
    - only reinitialize the fields that must be changed
    - only disassemble to give up the space
    - reduce (or eliminate) carving and coalescing
- Get a huge slab of memory from the OS and then carve the slab into sets of data structures so that they can easily be called

Common Dynamic Memory Errors

- Memory Leak

    - neglect to free memory after you are done with it
    - often happens in (not thought out) error cases
- Buffer overrun
    - writing past beginning or end of allocated chunk
    - usually result of not checking parameter/limits
- Continuing to use it after freeing it
    - may be result of a race condition
    - may be result of returning pointers to locals
    - may simply be “poor housekeeping”

Diagnostic Free List

- standard chunk header
  – free bit, chunk length, next chunk pointer
- allocation audit info
  – tracing down the source of memory leaks
- guard zones
  – detect application buffer over-runs
- zero memory when it is freed
  – detect continued use after chunk is freed

Diagnostic Free Lists can help

- All chunks in list (whether allocated or free)
    - enables us to find state of all memory chunks
- Record of who last allocated each chunk
    - what routine called malloc, when
- Guard zones at beginning and end of chunks
    - limited protection against buffer under/overrun
    - enables us to detect data under/overrun
- Valgrind is a tool that checks for memory mistakes

Memory Leaks

- Programs often forget to heap memory
    - this is why the automatic stack model is so attractive
- losses can be significant in long running processes
    - process grows, consuming ever-more resources
    - degrading system and application performance
- the OS cannot help

    - the heap is managed entirely by user-mode code
    - finding and fixing the leaks is too difficult for most
    - some advocate regular “prophylactic restarts”

Garbage Collection

- Garbage collection is alternative to freeing
    - application allocate objects, never free them
- user-mode memory manage monitors space
    - when it gets low initiate garbage collection
- Garbage Collection
    - search data space finding every object pointer
    - note address/size of all accessible objects
    - compute the compliment (what is inaccessible) -> hence garbage, and no longer being used by the program
    - add all inaccessible memory to the free list

Garbage Collection: TANSTAAFL

- Garbage Collection is expensive
    - scan all possible object references
    - compute an active reference count for each
    - may require stopping application for a long time
- Progressive Background Garbage Collection
    - runs continuously, in parallel with application
    - continuous overhead and competing data access
- The more you need it, the more it costs (the worse it works)
    - more frequent garbage collection scans
    - yielding less free memory per scan

Finding all **accessible** data

- Object oriented languages often enable this
    - all object-references are tagged
    - all object descriptors include size information
- It is often possible for system resources
    - where all resources and references are known (e.g. we know who has which files open)
- Resources can be designed with GC in mind
    - but, in the general case, it may be impossible

General Case GC: What’s so hard?

- Compiler can know static/automatic pointer
- How do we identify pointers in the heap?
    - search data segment for address like values?
    - a number or string could like an address

- How do we know if pointers are still live?
    - a value doesn't mean the code is still using it
- What kind of structure does it point to?
    - we need to know how much memory to free?
- Only possible if all data/pointers are tagged
    - which is, in general, not the case
- Garbage Collection is an option if you've designed your data structures with GC in mind

GC vs Reference Counting

- Every time we do something like a dup or open, we increment the reference counter, and close decrements the reference counter
- Once the reference count goes to zero, we recycle our resources automatically
- What if there are multiple pointers to object?
    - when can we safely delete it
- Associate a reference count w/each object
    - increment count on each reference creation
    - decrement count on each reference deletion
    - delete object when reference count hits zero
- This is not the same as Garbage Collection
    - it requires explicit close/release operations
    - doesn't involve searching for unreferenced objs
    - correct count maintenance may not be free

Garbage Collection and Defragmentation

- Garbage Collection: seeking out no-longer-in use resources and recycling them for reuse
- Defragmentation: reassigning and relocating previously allocated resources to eliminate external fragmentation to create densely allocated resources with large contiguous pools of free space
- Both are techniques for remedying space that has been rendered unusable by unfortunate combinations of allocation events
- these two techniques are often applied in tandem, with garbage collection first then defragmentation

Garbage Collection

- How does an allocated resource come to be returned to the free pool?
- With many resources, allocated resources are freed by an explicit or implicit action on the client's part
    - calling close(2) on a file
    - calling free(3) on memory obtained from malloc(3)

    - invoking the delete operator on a C++ object
    - return from a C/C++ subroutine
    - calling exit(2) to terminate a process
- If a resource is sharable by multiple concurrent clients (an open file or a shared memory segment), we cannot free it simply because one client called close. The resource manager may maintain an active reference count for each resource
    - increment the reference count whenever a new reference to the object is obtained
    - decrement whenever a reference is released such as a call to close
    - automatically free the object when the reference count reaches zero
- In languages where it is possible to copy resource references, like pointers, references can be copied or destroyed without invoking any OS services ->OS will not know about these resource references
- Many modern languages do not require programs to explicitly free some resources such as memory
- some resources (DHCP) addresses may never be explicitly freed, and so must be automatically recycled after they have gone unused for a specific period of time
- for some resources, allocation/release operations may be so frequent that the cost of keeping track of them becomes a significant performance overhead
- Resources are allocated but never specifically freed (explicitly)
- when the pool of available resources becomes dangerously small, the system initiates garbage collection
    - begin with a list of all of the resources that originally existed
    - scan the process to find all resources that are still reachable
    - each time a reachable resource is found, remove it from the original resource list
    - anything that is still in the list at the end of the scan can be freed if it is no longer referenced by the process
- Garbage collection only works if it is possible to identify all active resource references
- Disadvantages: The system may run very nicely for a long time, and then suddenly stop for garbage collection. The workaround to this is to run a garbage collector that continuously frees resources that are not used
    - Non-deterministic cleanup of resources, since garbage collection is delayed until the pool of allocated resources becomes dangerously small, if you want the memory to be freed instantaneously, garbage collections won't do a good job
    - There is an overhead for keeping track of the resources that were allocated, as when the GC collects it is common to see small hangs

Defragmentation

- All variable partition memory allocation algorithms eventually result in a loss of useful storage due to external fragmentation
    - free memory is broken up into disconnected shards, none of which are large enough to be useful

    - Coalescing can counter external fragmentation but is only useful if adjacent memory chunks happen to be free at the same time
- How important is contiguous allocation?
    - if we are allocating blocks of disk, the only cost of non-contiguous allocation may be longer seeks -> slower file I/O
- Garbage Collection analyzes the allocated resources to determine which ones are still in use. Defragmentation goes garther and actually changes which resources are allocated
- Flash Management
    - NAND Flash is a pseudo-Write-Once-Read-Many medium. After a block has been written, some serious processing and voltage will be required to erase it before it can be rewritten with new data
    - When the OS does a write to an SSD, firmware in the Flash Controller assigns a new block to receive the new-data, leaving the old data no longer referenced
- Disk Space Allocation
    - File reads and writes can progress orders of magnitudes more quickly if logically consecutive blocks are stored contiguously on disk.
    - But eventually, after many generations of files being created and deleted, the free space and files become discontiguous and therefore a lot slower.
    - 1) choose an area of the disk where we want to create contiguous free space and files
    - 2) for each file in that area, copy it to some other place, to free up space in the area we want to defragment
    - 3) after all files have been copied out of that area, we can coalesce all the free space into one huge contiguous chunk
    - 4) choose a set of files to be moved into the newly contiguous free space, and copy them into it
    - 5) repeat the process until all of the files and free space are contiguous
- Defragmentation physically attempts to organize the contents of the mass storage device used to store files into the smallest number of contiguous regions.
- Defragmentation also attempts to create larger regions of free space using compaction(the reduction of the number of data elements, bandwidth, cost, and time for the generation, etc.) to impede the return of fragmentation.

Chapter 12

- Every address generated by a user program is a virtual address
- The OS is just providing an illusion to each process, specifically that it has its own large and private memory

Chapter 13: The Abstraction: Address Space

13.1: Early Systems

- Early System only had the OS code and memory to store the data segments

13.2: Multiprogramming and Time Sharing

- One way to implement time sharing would be to run one process for a short while, giving it full access to all memory, then stop it, save all of its state to some kind of disk, load some other process' state, run it for a while and thus implement some crude sharing mechanisms
    - As memory grows, this is way too slow
    - While saving and restoring register-level states like PC and general-purpose registers is relatively fast, when you have to load entire contents of memory to disk, it is extremely slow
    - Therefore, we'd rather leave the processes in memory while switching between them

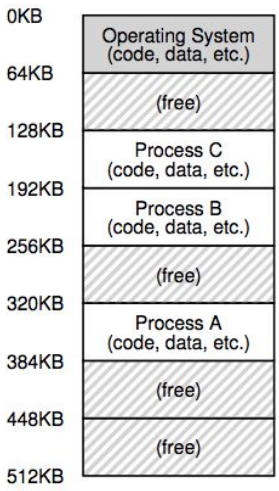


Figure 13.2: **Three Processes: Sharing Memory**

    -
    - While one process is running, the others sit on the ready queue
    - Issues: Allowing multiple programs to reside concurrently in memory makes protection an important issue;
        - you don't want a process to be able to read or write some other process' memory

13.3 Address Space

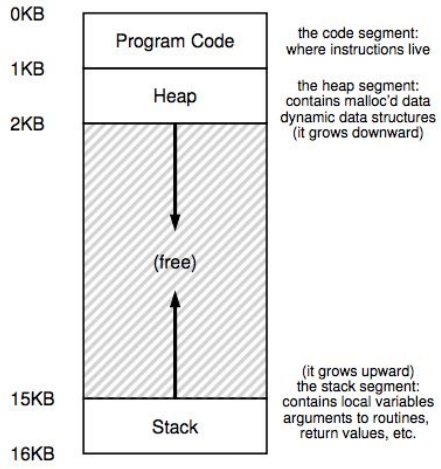


Figure 13.3: **An Example Address Space**

- The address space is the running program's view of memory in the system
- **Code**: The instructions that are put into the address space
- **Stack:** The program uses a stack to store all of its parameters, local variables, return values to and from routines, and where the function was passed in.
- **Heap:** The heap is used for dynamically allocated variables and resources such as when you call malloc() from C or when you call new from object oriented languages such as Java or C++
- The code segment where the instructions live is places at the top of the address because it is static and thus easy to place in memory. Therefore, we know that it won't need anymore space then what the program allocates
- The Heap and Stack may grow/shrink when the program runs. The heap is placed above the stack because it grows downwards while the stack grows upwards.
- When we describe the address space we are describing the abstraction that the OS is providing for the running program.
    - The program isn't in memory at physical addresses 0 through 16KB in the diagram
    - Rather, it is loaded into some arbitrary address that is available
- When the process is loaded into memory at a different address, this is called virtualizing memory because the running program thinks it is loaded into memory at a particular address and has a potentially very large address space
- Example: When process A tries to perform a load at address 0, the OS has to make sure that it doesn't get loaded into address 0 but address 320KB where A is loaded into memory.
- **Tip: The Principles of Isolation**
    - Isolation is a key principle in building reliable systems
    - If two entities are properly isolated from one, then one can fail without changing the state of the other

13.4: Goals

- One major goal of a virtual memory (VM) system is **transparency**: the illusion provided by the OS should not be visible to applications
- Another goal of VM is **efficiency**: The OS should strive to make the virtualization as efficient as possible in terms of time and space
    - The OS will have to rely on hardware support for efficiency such as the TLB
- The final goal of the OS is **protection**: The OS should make sure to protect processes from one another as well as the OS itself from other processes.
    - When one process performs a load, a store, or an instruction fetch, it should not be able to access or affect in any way the memory contents of other processes or the OS itself
    - Each process should be running in its own cocoon, safe from the ravages of other faulty or malicious processes

13.5: Summary

- The VM system is responsible for providing the illusion of a large sparse, private address space to programs, which hold all of their instructions and data therein
- The OS will make sure to protect programs from one another and properly achieve isolation through the use of hardware and great amounts of low-level machinery and mechanisms
- When we print out an address like a value of a pointer, we are printing the virtual address. Only the OS knows the actual physical address of the running process

Chapter 14: Interlude: Memory API

14.1: Types of Memory

- Two types of memory allocated
    - **Stack memory**: allocations and deallocations of stack memory are initialized within the compiler implicitly.
        - This is often known as the automatic memory because it is done for you
        - Declaring memory in stack memory is easy: Just create a local variable inside a function and the compiler will allocate and deallocate the memory for you
    - **Heap memory**: allocations and deallocations are explicitly handled by the programmer
        - You can allocate and deallocate by using commands such as malloc()

14.2: The malloc() call

- Pass it a size asking for room on the heap, and it either succeeds and gives you back a pointer to the newly allocated space, or fails and returns NULL
- All you need to include is stdlib.h
- Use the sizeof function to malloc a certain amount of memory

    - ex: double *d = (double*) malloc(sizeof(double))
    - Sizeof is thought of as a compile time operator and not a function call
- sizeof(int) is 4 and sizeof(double) is 8 and sizeof(pointer) is 4 for 32-bit machines and 8 for 64-bit machines
- Be careful when allocating space for strings
    - use convention: malloc(strlen(s) + 1)
    - makes sure to get the length of the string plus the EOF character so that there is enough space.
    - Using sizeof on strings can be potentially difficult
- malloc() returns a pointer to type void, so you can cast it to a type of your choice such as a double or int

14.3: The free() call
- To free heap memory that is no longer in use, use the free() call
- The routine takes one argument, a pointer that was returned by malloc()

14.4: Common Errors
- There are a number of common errors that arise in the use of malloc() and free()
- Correct memory management has been such a problem that many languages have support for automatic memory management
    - garbage collector runs and figures out what memory you no longer need and frees it for you
- **Forgetting to Allocate Memory**
    - Many routines expect memory to be allocated before you call them, such as strcpy(dst, stc) which copies a string from a source pointer to a destination pointer
    - If you do not allocate memory beforehand, a segmentation will be caused and your program will break
- **Not Allocating Enough Memory**
    - When you don't allocate enough memory, a buffer overflow may occur
    - Buffer overflows can lead to numerous security vulnerabilities in systems
    - Sometimes even though you don't allocate enough memory, your program still runs correctly
        - Just because it runs correctly doesn't mean that your program is correct!
- **Forgetting to Initialize Allocated Memory**
    - With this error, you call malloc() properly but forget to fill in some values into your newly-allocated data type. This can cause an uninitialized read, where it reads random data with unknown value from the heap
- **Forgetting to Free Memory**
    - Another common error is known as a memory leak, and it occurs when you forget to free the memory

    - Note that using a garbage-collected language doesn't help: if you still have a reference to some chunk of memory, no garbage collector will ever free it, and thus memory leaks remain a problem
- **Freeing Memory Before You Are Done With It**
    - Sometimes a program will free memory before it is finished using it, resulting in a dangling pointer
    - This can crash the program or overwrite valuable memory that you previously had
- **Freeing Memory Repeatedly**
    - When a program frees memory more than once, it is called a **double free**, making the memory-allocation library confused and often causing undefined behavior
- **Calling free() Incorrectly**
    - If you pass in a value to free() that you did not malloc() earlier, it can lead to an invalid free, confusing the memory-allocation library
- Aside: Why No Memory is Leaked When a Process Exits
    - Two levels of memory management within the OS: One for the process when they run, managed by the OS. The other is within the process itself, for example, within the heap when you call malloc()
    - The OS will reclaim all the memory of the process when the program is finished (including the stack, code, and heap pages)
        - Ensures that no memory is lost even though you forget to free() it
        - Therefore, for short-lived programs, not freeing the memory allocated may not be a problem
    - Remember that the OS has primary and secondary storage as well:
        - It has all the running/ready programs in the primary storage. Primary storage is very expensive and has limited amounts of space
        - Secondary storage has "blocked" processes and also makes room for "ready" processes in the primary storage
- Summary
    - Purify and Valgrind are both excellent at helping the user locate the source of memory-related problems such as invalid freeing and memory leaks

14.5: Underlying the OS Support

- malloc() and free() are not system calls, but rather they are **library calls**
- Thus the malloc library manages space within your virtual address space, but itself is built on top of some system calls which call into the OS to ask for some memory release back to the system
    - in short, it is a library call that is built on top of a series of system calls to the OS for some memory release
- Example of such a system call: brk
    - Used to change the location of the program's break, the location of the end of the heap

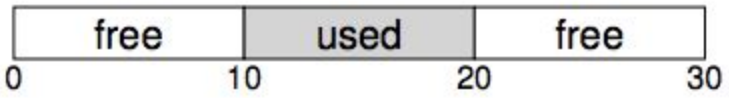
    - It takes one argument (the address of the new break) and increases/decreases the size of the heap based on whether the new break is larger or smaller than the break
    - *Never tried to directly call brk or sbrk (passes and increment)
- You can also obtain memory from the OS through the mmap() system call.
    - By passing in the correct arguments, you can create anonymous memory regions within your program -- a region which is not associated with any particular file but rather with swap space. This memory can be treated as a heap as well

14.6: Other Calls

- There are a few other calls such as calloc() which allocates memory and also zeros it before returning
- realloc() adds a new region of memory and copies the old region into it, returning a pointer to the new region

Chapter 17: Free-Space Management

- Managing free space can certainly be easy, with the concept of paging. It is easy when the space you are managing is divided into fixed-size units
    - keep a list of fixed-size units: when a client requests one of them, return the first entry
- Free-space management becomes more difficult when the the free space consists of variable sized units
- OS managing physical memory when using **segmentation** to implement virtual memory
    - The problem that occurs with this is **external fragmentation**
        - The free space gets chopped into little pieces of different sizes and is thus fragmented
        - Requests may fail because there are no segments that can satisfy the request. There may be no contiguous free spaces that are available, even if the total amount of free space exceeds the request
    - Example of external fragmentation



    -
    - The total free space is 20 bytes. However, a request for 15 bytes will still fail because the memory is fragmented into 10 byte segments
    -

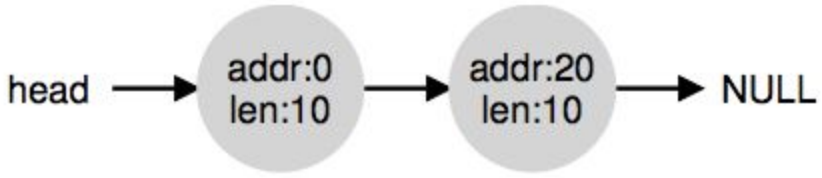
Chapter 17.1: Assumptions

- Assume a basic interface such as that provided by malloc() and free()
- Specifically void *malloc(size_t size) takes a single parameter which is the number of bytes requested by the application

    - it returns a pointer to an available region of that size or greater
- The space that this library deals with is traditionally called the heap, and the data structure used to manage free space in the heap some kind of **free list**
    - This structure contains references to all of the free chunks of space in the managed region of memory
- Allocators could of course also encounter the problem of **internal fragmentation**
    - if an allocator hands out chunks of memory bigger than that requested, any unasked for and therefore unused chunk of free space is known as internal fragmentation (because the waste occurs inside the allocated units)
- Assume that once memory is handed out to a client, it cannot be relocated to another location in memory
    - ex: if a program calls malloc() and is given a pointer to some space within the heap, that memory region is essentially “owned” by the program
- Thus, no compaction of free space is possible, which could be potentially useful when dealing with external fragmentation
    - However, it can be used to deal with fragmentation when implementing segmentation in the OS
- Assume that the allocator manages a contiguous region of bytes. In some cases, an allocator could ask for that region to grow
    - ex: a user-level memory-allocation library might call into the kernel to grow the heap (via the sbrk system call) when it runs out of space
    - For simplicity, we will assume that the region is a fixed size owned by that file

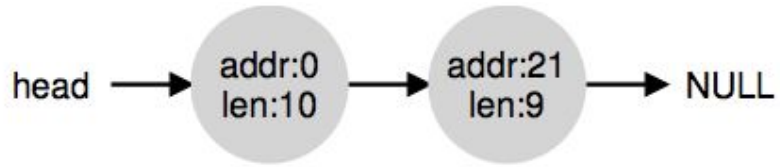
17.2: Low-Level Mechanisms

- **Splitting and Coalescing**
    - A free list contains a set of elements that describe the free space still remaining in the heap
    - Thus assuming that we have a 30-byte heap where the first 10 bytes are free, second 10 bytes are used, and the third 10 bytes are free, we can assume the below diagram



    -
    - A request for anything greater than 10 bytes will result in returning a NULL
    - However, what happens when the client requests for something less than 10 bytes?
        - The allocator will perform an action known as **splitting,** where it will find a free chunk of memory that can satisfy the request and split it into two.
        - The first chunk it will return to the caller and the second chunk will remain on the list

        - Thus, if the client requests for one byte, the diagram above will now look like the following



        -
        - The region now starts at 21 instead of 20, and the length of the free region is now 9
    - A corollary mechanism found in many allocators is known as coalescing of free space
    - Ex: if free(10) is called, then if you don't think about it you may end up with three sections of 10 bytes of free space
    - The allocator coalesces the free space into one big chunk (in this case of 30 bytes) if it can fit between two chunks of free space
    - The allocator can ensure that large chunks of free space are available on the heap for the application to use
- **Tracking the Size of Allocated Region**
    - The free() library call does not take a size parameter
        - it is assumed that given a pointer, the malloc library can quickly determine the size of the region of memory being freed and thus incorporate the space back into the free list
        - To accomplish this, most allocators store a little bit of extra information in a header block which is kept in memory, usually just before the handed-out chunk of memory

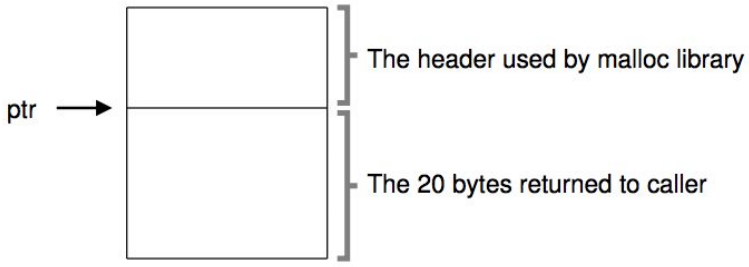


Figure 17.1: **An Allocated Region Plus Header**

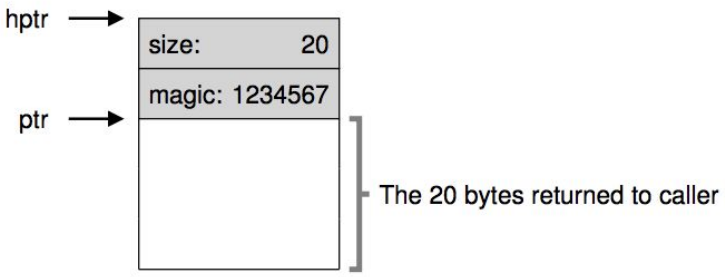


Figure 17.2: **Specific Contents Of The Header**

        -

        - The header minimally contains the size of the allocated region
            - it may also contain additional pointers to speed up deallocation, a magic number to provide additional integrity checking, and other information
        - When the user calls free(ptr), the library then uses simple pointer arithmetic to figure out where the header begins
        - Library determines whether the magic number matches the expected value and calculate the size of the newly freed region
        - The size of the free region is the size of the header plus the size of the space allocated to the user.
        - When a user requests N bytes of memory, the library does not search for a free chunk of size N, but rather for N+header size
- **Embedding a Free List**
    - You can't call malloc() from within the library
        - therefore, you need to build the list inside the free space itself
    - Assume we have a 4096-byte chunk of memory to manage (i.e. the heap is 4KB)

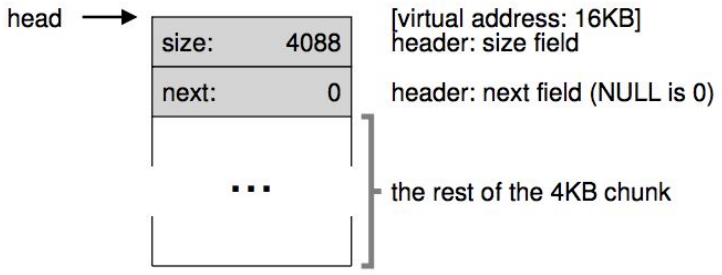


Figure 17.3: **A Heap With One Free Chunk**

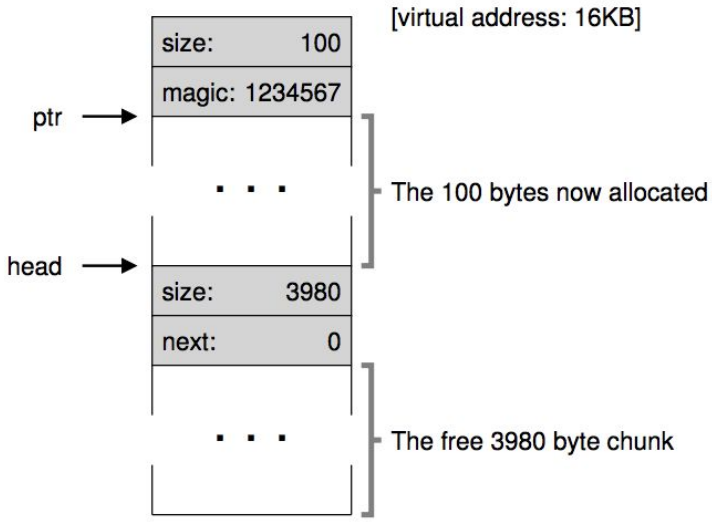
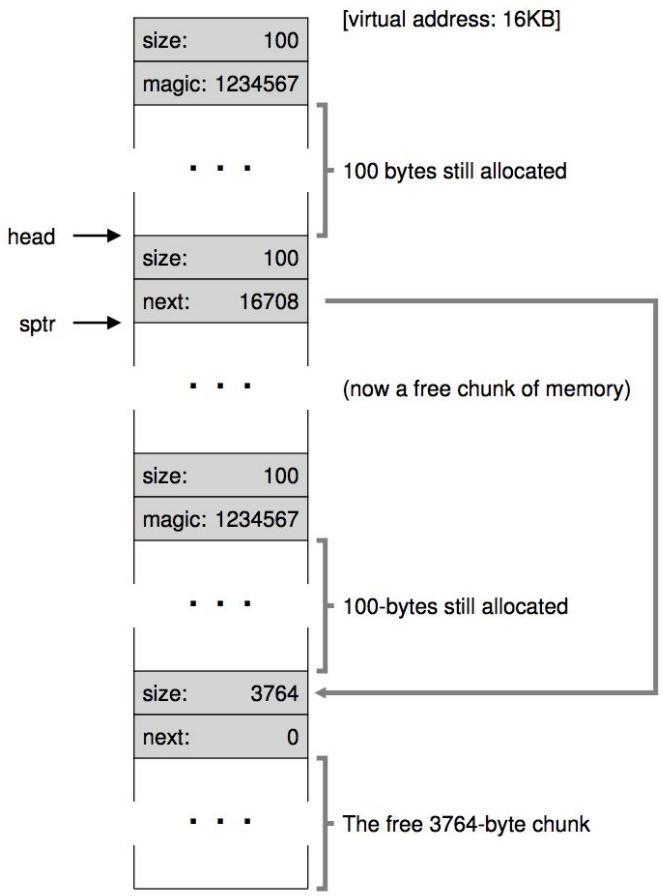


Figure 17.4: **A Heap: After One Allocation**

○



- Figure 17.6: **Free Space With Two Chunks Allocated**

- Growing the Heap
  - What should we do when the heap runs out of space?
    - In most cases, we will just fail or return NULL
    - Most allocators start off with small-size heaps, then make requests to the OS using the sbrk system call
    - The OS finds free physical pages, maps them into the address space of the requesting process, and then returns the value of the end of the new heap
    - Make sure to include the values for the headers as well

17.3: Basic Strategies

- The ideal allocator is both fast and minimizes fragmentation
- **Best fit**
  - The best fit strategy is searches through the free list and finds chunks of free memory that are or bigger than the requested size
  - Return the one that is the smallest size out of the candidates -> thus called best fit
  - Advantages: Best fit tries to reduce wasted space

    - Disadvantages: Naive implementations pay a heavy performance penalty when performing an exhaustive search for the correct free block
- **Worst fit**
    - Find the largest chunk and return the requested amount and keep the remaining chunk on the free list
    - Advantages: tries to leave big chunks free instead of the small ones that arise from the best fit choice
    - Disadvantages: once again, exhaustive searches on the free list is very inefficient
    - The worst fit algorithm has been shown to lead to fragmentation while having the highest overhead
- **First Fit**
    - The first fit method chooses the first chunk of memory that the fits the request
        - The remaining space is kept free for subsequent requests
    - Advantages: Very efficient because it does not perform a full search of the free list in the heap
    - Disadvantages: Sometimes pollutes the beginning of the free list with smaller chunks, so subsequent calls would take longer
        - Fixes: one approach is to use address-based ordering by keeping the list ordered by the address of the free space, so coalescing becomes easier and fragmentation is reduced
- Examples

Here are a few examples of the above strategies. Envision a free list with three elements on it, of sizes 10, 30, and 20 (we'll ignore headers and other details here, instead just focusing on how strategies operate):

Assume an allocation request of size 15. A best-fit approach would search the entire list and find that 20 was the best fit, as it is the smallest free space that can accommodate the request. The resulting free list:

As happens in this example, and often happens with a best-fit approach, a small free chunk is now left over. A worst-fit approach is similar but instead finds the largest chunk, in this example 30. The resulting list:

    -

17.4: Other Approaches

- **Segregated Lists**
    - If a particular application has one or a few popular-sized request that it makes, keep a separate list just to manage objects of that size, while all other requests are managed by a separate more general memory allocator

    - Advantages: By having a chunk of memory dedicated for one of particular size requests, the efficiency is heavily increased and fragmentation is much less of a concern
    - Disadvantages: How much memory should one dedicate to the pool of memory that holds specialized requests?
        - **Slab allocator**: When the kernel boots up, it allocates a number of object caches for kernel objects that are likely to be requested frequently (such as locks, file-system inodes, etc.).
            - The object caches are each segregated free lists of a given size and serve memory allocation and free requests quickly
            - When a given cache is running low on memory, it makes request for some **slabs of memory** from a more general memory allocator (total amount requested being a multiple of the page size and object in question)
            - When the reference counts of the objects within a given slab all go to zero, the general allocator can reclaim them from the specialized allocator
            - Keeps free objects on the lists in a pre-initialized state
            - Avoids frequent initialization and destruction because those actions can be very expensive, and lowers overhead significantly
- **Buddy Allocation**
    - Because coalescing is critical for an allocator, some approaches have been designed around making coalescing simple in order to decrease fragmentation
    - **Binary buddy allocation**
        - Free memory conceptually thought of as one big space of 2^N. When a request for memory is made, the search for free space recursively divides free space by two until a block that is big enough to accommodate the request is found. Then the requested block is returned
        - Disadvantages: Can suffer from internal fragmentations because it can only return to the client powers of two size blocks
        - Example: When returning the 8KB block, the allocator checks whether the “buddy” 8KB block is free and if so coalesces the two blocks into a 16KB block
            - Then the 16KB is checked with the “buddy” 16KB block and continues coalescing recursively
        - Does a very good job of handling coalescing so therefore is able to efficiently avoid coalescing
        - Recursion can, however, be slow an efficient at times
- Other Ideas
    - One major problem with any of the previous approaches that we mentioned is their lack of scaling

- Searching lists can be quite slow and thus advanced allocators use more complex data structures to address these costs, trading simplicity for performance