UNIVERSITY of WISCONSIN-MADISON
Computer Sciences Department

CS 537
Intro to Operating Systems

A. Arpaci-Dusseau
Spring 2000

Dynamic Memory Allocation

Questions answered in these notes

- When is a stack appropriate? When is a heap appropriate?
- How can memory be found in the heap?
  - Best fit vs. First fit vs. Worst fit vs. Buddy
- How does one free memory when it is no longer needed?

Readings for this topic:

- Get ready for Chapter 8

Motivation for Dynamic Memory Allocation

How does system manage memory of a single process?

- View: Each process has contiguous logical address space

Next lecture

- Sharing across processes
- Mapping from logical address space to physical address space

Dynamic Storage Management

Static (compile-time) allocation is not possible for data

- Recursive procedures
  - Even regular procedures are hard to predict (data dependencies)
- Complex data structures
  - Storage used inefficiently when reserved statically
  - Must reserve enough to handle worst possible case

```
ptr = Allocate(x bytes)
Free(ptr)
```

Dynamic allocation can be handled in two ways

- Stack allocation:
  - Restricted, but simple and efficient
- Heap allocation:
  - More general, but less efficient
  - More difficult to implement

Stack Organization

Definition: Memory is freed in opposite order from allocation.

alloc(A)
alloc(B)
alloc(C)
free(C)
free(B)
free(A)

When is it useful?

- Memory allocation and freeing are partially predictable
- Allocation is hierarchical
- Example
  - Procedure call frames
  - Tree traversal, expression evaluation, parsing

Stack Implementation

Advance pointer dividing allocated and free space

- Allocate: Increment pointer; Free: Decrement pointer

    alloc(A)
    alloc(B)
    alloc(C)
    free(C)
    alloc(D)
    free(D)
    free(B)
    free(A)

Advantage

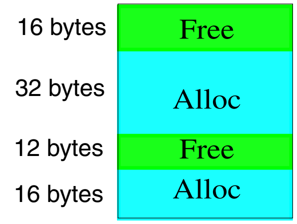
- Keeps all the free space contiguous
- Simple and efficient to implement

Disadvantage: Not appropriate for all data structures

Heap Organization

Definition: Allocate from random locations

- Memory consists of **allocated** areas and **free** areas (or holes).



How to allocate 24 bytes?

When is it useful?

- Allocation and release are unpredictable
- Arbitrary list structures, complex data organizations

    Examples: new in C++, malloc() in C

Advantage: Works on arbitrary allocation and free patterns

Disadvantage: End up with small chunks of free space

Fragmentation

Definition: Free memory that is too small to be usefully allocated

- External: Visible to system
- Internal: Visible to process (e.g., if allocate at some granularity)

Goal

- Number of holes small
- Size of holes large

Stack allocation

- All free space is together in one big chunk

How do we implement heap allocations?

Heap Implementation

Data Structure: Linked list of free blocks

- **free list:** tracks storage not in use

Allocation

- Choose block large enough for request that meets certain criteria
- Update pointers and size variable

Free

- Add block back to list
- Merge adjacent free blocks

```
if (addr of new block == prev addr + size) {
        combine blocks
}
```

Best vs. First vs. Worst

Best fit

- Search the whole list on each allocation
- Choose block that most closely matches size of request
- Can stop searching if see exact match

First fit

- Allocate first block that is large enough
- **Rotating first fit:** Start with next free block each time

Worst fit

- Allocate largest block to request (most leftover space)

Which is best?

Examples

Best Algorithm: Depends on sequence of requests

Example: Memory contains 2 free blocks of size 20 and 15 bytes

- Allocation requests: 10 then 20

- Allocation requests: 8, 12, then 12

Buddy Allocation

Fast, simple allocation for blocks that are 2^n bytes [Knuth 1968]

- Allocation restrictions
  - Block sizes: 2^n
  - Represent allocated units with bitmap

- Allocation strategy for k bytes
  - Raise allocation request to nearest 2^n
  - Search free list for appropriate size
    - Recursively divide larger blocks until reach block of correct size
    - “Buddy” blocks remain free

- Free strategy
  - Recursively coalesce block with buddy if buddy free
  - May coalesce lazily to avoid overhead

Example

1MB of memory

- Allocate: 70KB, 35KB, 80KB
- Free: 70KB, 35KB

Comparison of Allocation Strategies

Best fit

- Tends to leave some very large holes, some very small ones
- Disadvantage: Very small holes can't be used easily

First fit:

- Tends to leave "average" size holes
- Advantage: Faster than best fit

Buddy:

- Organizes memory to minimize external fragmentation
  - Leaves large chunks of free space
  - Faster to find hole of appropriate size
- Disadvantage: Internal fragmentation when not power or 2 request

Memory Allocation in Practice

Malloc() in C:

- Calls **sbrk** to request more contiguous memory from OS
- Add small header to each block of memory
  - Pointer to next free block
  - Size of block
- Where must this header be placed?
- Combination of two data structures

Separate free list for each popular size

- Allocation is fast, no fragmentation
- Inefficient if some are empty while others have lots of free blocks

First fit on list of irregular free blocks

- Combine blocks and shuffle blocks between lists

Reclaiming Free Memory

When can dynamically-allocated memory be freed?

- Easy when a chunk is only used in one place
  - Explicitly call free()
- Hard when information is shared
  --> Can't be recycled until all sharers are finished

Sharing is indicated by the presence of *pointers* to the data

- Without a pointer, can't access data (can't find data)

Two possible problems

- **Dangling pointers:** Recycle storage while it's still being used
- **Memory leaks:** Forget to free storage even when can't be used again
  - Not a problem for short-lived user processes
  - Issue for operating systems and long-running applications

Reference Counts

Idea

- Keep track of the number of references to each chunk of memory
- When reference count reaches zero, free the memory

Example

- Files and hard links in Unix
- Smalltalk
- Objects in distributed systems

Disadvantages

- Circular data structures --> Memory leaks

Garbage Collection

Idea

- Storage isn't freed explicitly (i.e., no `free()` operation)
- Storage freed implicitly when no longer referenced

Approach

- When system needs storage, examine and collect free memory

Advantages

- Works with circular data structures
- Makes life easier on the application programmer

Mark and Sweep Garbage Collection

Requirements

- Must be able to find all objects
- Must be able to find all pointers to objects
  - Compiler must cooperate by marking type of data in memory

Two Passes

- Pass 1: Mark
  - Start with all statically-allocated and procedure-local variables (on stack)
  - Mark each object
  - Recursively mark all objects can reach with a pointer
- Pass 2: Sweep
  - Go through all objects, free those that aren't marked

Garbage Collection in Practice

Disadvantages

- Garbage collection is often expensive: 20% or more of CPU
- Difficult to implement
  - Execute program during garbage collection (incremental)

Languages with Garbage Collection

- LISP (emacs)
- Java

Conservative garbage collection

- Idea: Treat all memory as pointers
- Can be used for C and C++