15-213

Introduction to Computer Systems

Memory Management I: Dynamic Storage Allocation Oct 8, 1998

Topics

- User-level view
- Policies
- Mechanisms

Harsh Reality #3

Memory Matters

Memory is not unbounded

- It must be allocated and managed
- Many applications are memory dominated
  - Especially those based on complex, graph algorithms

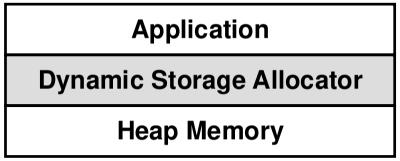
Memory referencing bugs especially pernicious

- Effects are distant in both time and space

Memory performance is not uniform

- Cache and virtual memory effects can greatly affect program performance
- Adapting program to characteristics of memory system can lead to major speed improvements

Dynamic Storage Allocation



Application

- Requests and frees contiguous blocks of memory

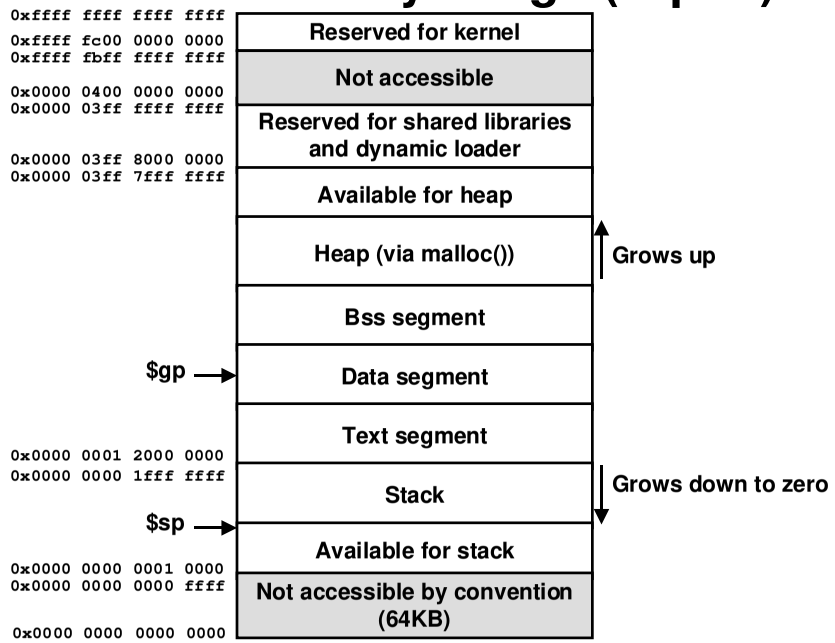
Allocator (e.g., Unix malloc package)

- Provides an abstraction of memory as a set of blocks
- Doles out free memory blocks to application
- Keeps track of free and allocated blocks

Heap memory

- region starting after bss segment

Process memory image (Alpha)



Malloc package

void *malloc(int size)

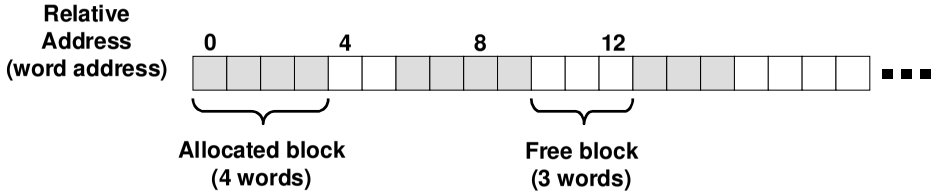
- **if successful:**
  - returns 8-byte aligned pointer to memory block of at least size bytes
  - is size=0, returns NULL
- **if unsuccessful:**
  - returns NULL

void free(void *p)

- **returns block pointed at by p to pool of available memory**
- **p must come from a previous call to malloc().**

Definitions and assumptions

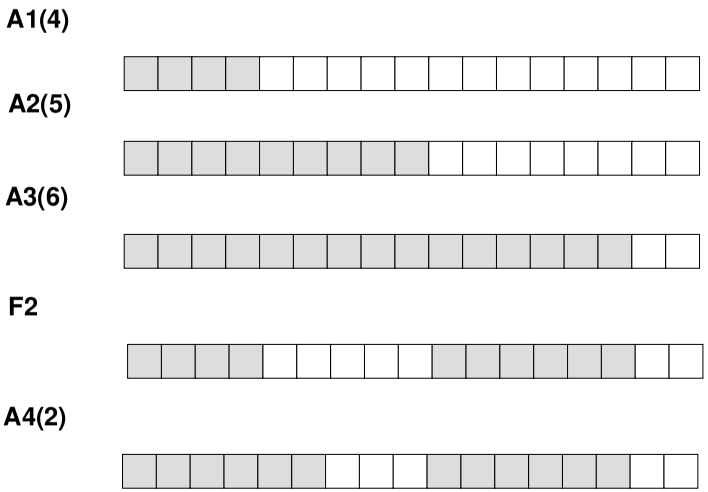
Heap Memory (fixed size)



Program Primitives:

- **Allocation request: Ai(n)**
  - Allocate n words and call it block i
  - Example: A7(128)
- **Free request: Fj**
  - Free block j
  - Example: F7

Allocation example



Constraints

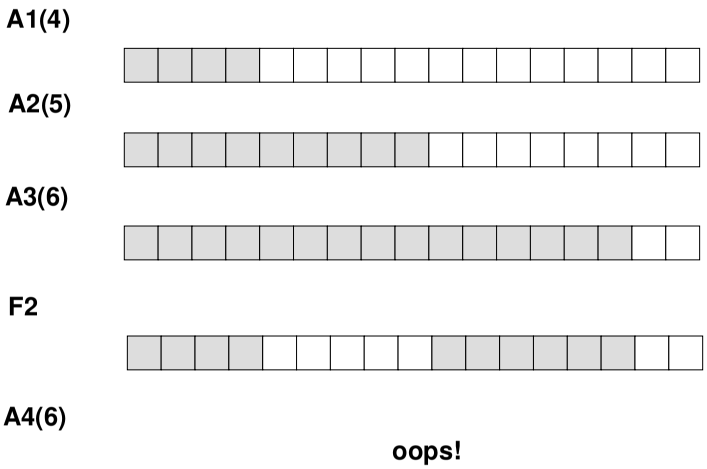
Applications:

- **Can issue arbitrary sequence of allocation and free requests**
- **Free requests must correspond to an allocated block**

Allocators

- **Can't control number or size of allocated blocks**
- **Must respond immediately to all allocation requests**
  - i.e., can't reorder or buffer requests
- **Must allocate blocks from free memory**
  - i.e., can only place allocated blocks in free memory
- **Must align blocks so they satisfy all alignment requirements**
  - usually 8 byte alignment
- **Can only manipulate and modify free memory**
- **Can't move the allocated blocks once they are allocated**
  - i.e., compaction is not allowed

Fragmentation



Fragmentation (cont)

Def: *(external) fragmentation* is the inability to reuse free memory.

- **possible because applications can free blocks in any order, potentially creating holes.**

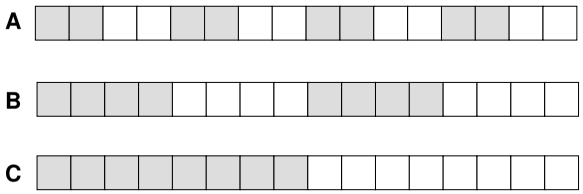
Minimizing fragmentation is the fundamental problem of dynamic resource allocation...

Unfortunately, there is no good operational definition.

Function of

- **Number and sizes of holes,**
  - Placement of allocated blocks,
  - Past program behavior (pattern of allocates and frees)
- **Future program behavior.**

Fragmentation (cont)



Which heaps have a fragmentation problem?

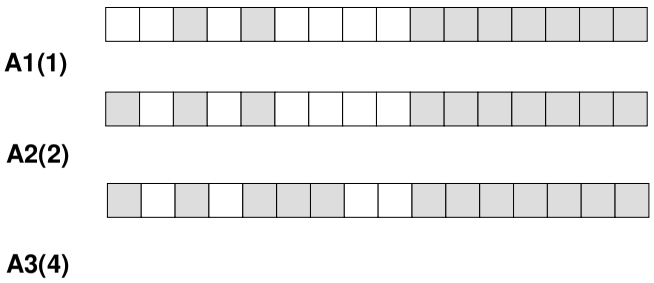
It depends...

- **Qualitatively, C has fewer and bigger holes.**
- **But fragmentation occurs only if program needs a large block.**
- **Still, C is probably less likely to encounter problems.**

Definitive answer requires a model of program execution.

Fragmentation (cont)

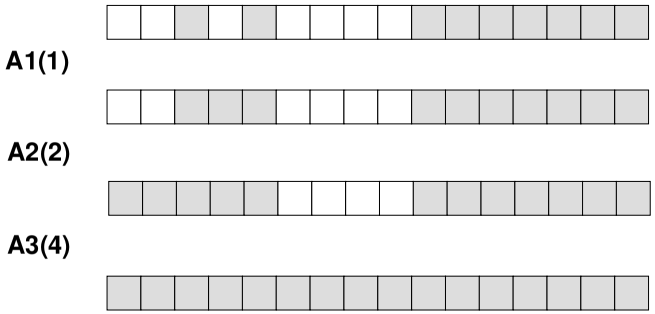
"First Fit"



The policy for placing allocated blocks has a big impact on fragmentation

Fragmentation (cont)

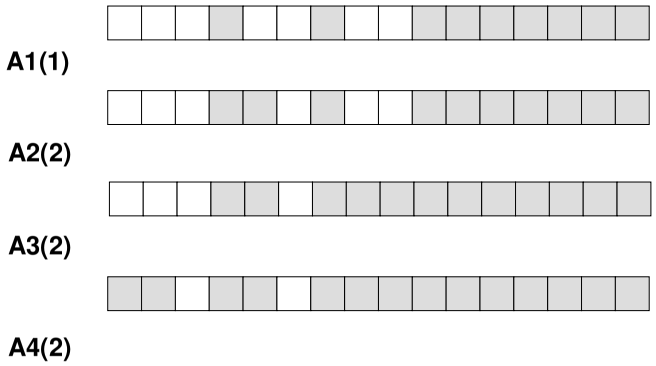
"Best Fit"



But "best fit" doesn't always work best either

Fragmentation (cont)

"Best Fit"

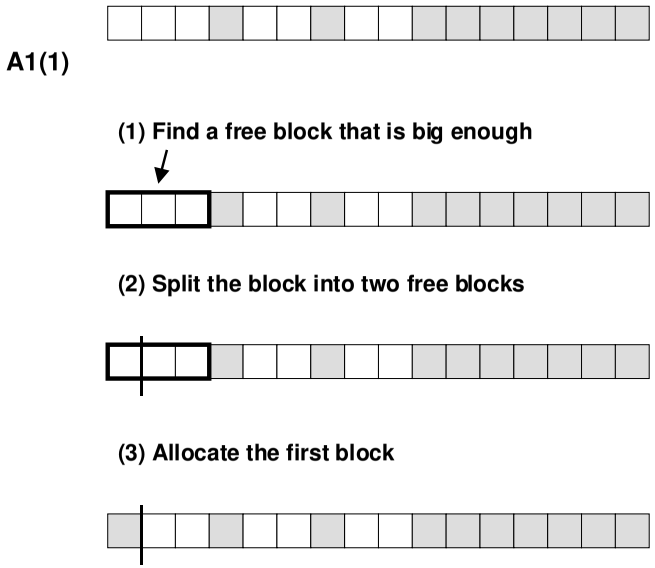


oops!

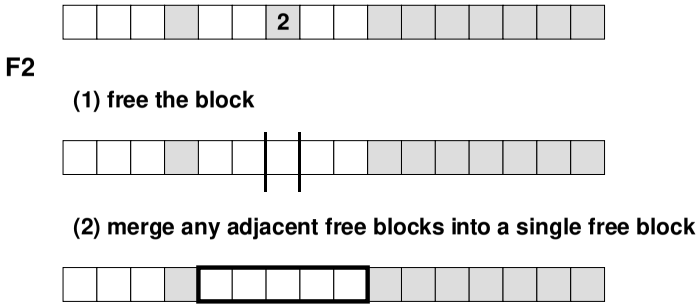
Fragmentation (cont)

"First Fit"

Splitting



Coalescing



Crucial operation for any dynamic storage allocator

Can be:

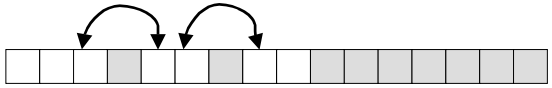
- **immediate (performed at every free request)**
- **deferred (performed every k free requests or when necessary)**

Organizing the set of free blocks

Some data structure needed to organize the search of the free blocks.

Efficient implementations use the free blocks themselves to hold the necessary link fields.

- **disadvantage: every allocated block must be large enough to hold the link fields (since the block could later be freed)**
- **imposes a minimum block size**
- **could result in wasted space (internal fragmentation)**



Common approach: list of free blocks embedded in an array of allocated blocks.

Organizing the set of free blocks

address ordering

– no memory overhead

doubly linked list of free blocks

– simple, popular, reasonable memory overhead

– might not scale to large sets of free blocks

tree structures

– more scalable

– less memory efficient than lists

segregated free lists

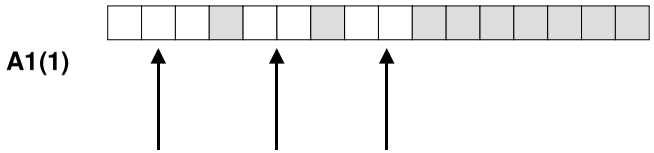
– different free lists for different size classes of free blocks.

– internal fragmentation

Placement policies

When a block is allocated, we must search the free list for a free block that is large enough to satisfy the request (*feasible free block*).

***Placement policy* determines which feasible free block to choose.**

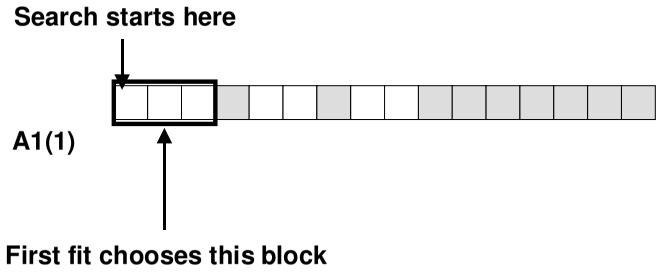


Each of these free blocks is feasible.
Where do we place the allocated block?

Placement policies (cont)

first fit

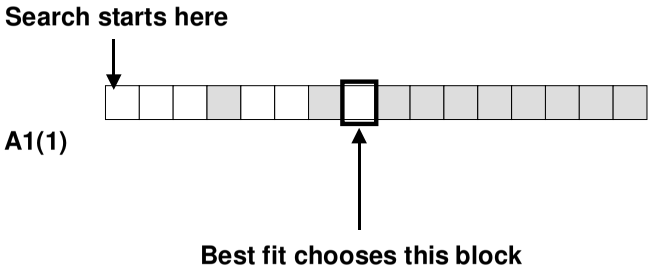
- search list from beginning, choose first free block that fits.
- simple and popular
- can increase search time, because “splinters” can accumulate near the front of the list.
- simplicity lends itself to tight inner loop
- might not scale well for large free lists.



Placement policies (cont)

best fit

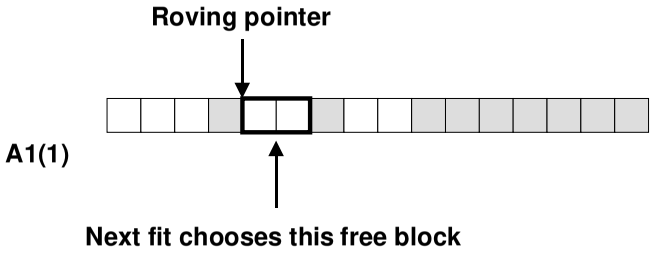
- choose free block that fits the best
- motivation is to try to keep fragments (what’s left over after splitting) as small as possible
- can backfire if blocks almost fit, but not quite.



Placement policies (cont)

next fit [Knuth]

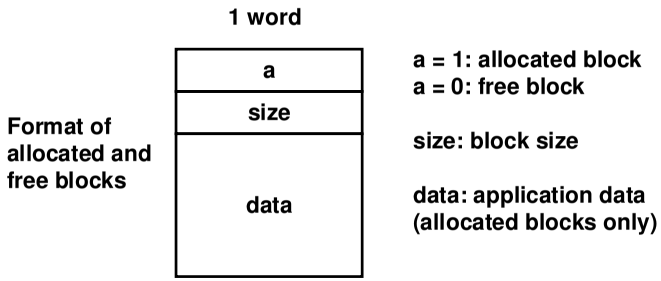
- like first fit, but instead of starting each search at the beginning...
- use a *roving pointer* to remember where last search was satisfied.
- begin next search at this point.
- motivation is to decrease average search time.
- potential disadvantage: can scatter blocks from one program throughout memory, adversely affecting locality.



Implementation Issues

The simplest allocator

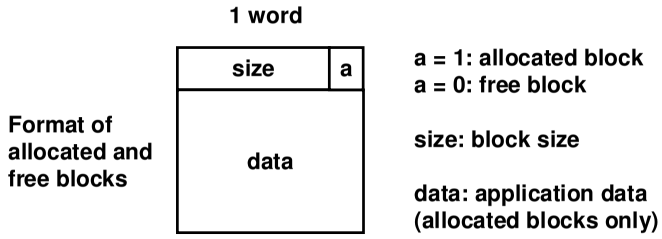
- **allocate time: linear in total number of blocks**
- **free time: linear in total number of blocks**
- **min block size: two words**



Implementation issues

A simple space optimization:

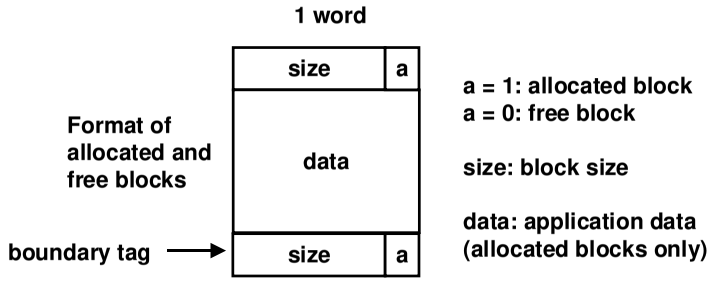
- exploit unused lower order size bits
- block size always a multiple of the wordsize
- reduces minimum block size from 2 words to 1 word



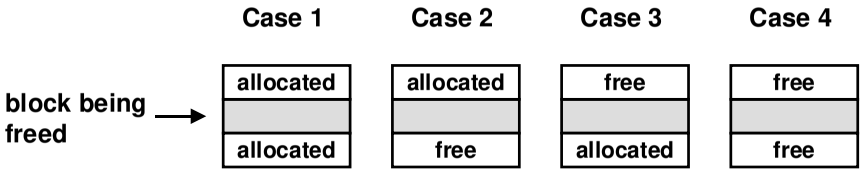
Implementation issues

Boundary tags [Knuth73]

- replicate size/allocated word at bottom of free blocks
- allocate time: linear in total number of blocks
- free time: constant time
- minimum block size: 2 words



Constant time coalescing



Food for thought

How can we use a list of free block to reduce the search time to linear in the number of free blocks?

Can we avoid having two conditionals in the inner loop of the free block list traversal

- one to check size
- one to check that entire list has been searched

Can we implement a free list algorithm with constant time coalescing and a minimum block size of three words instead of four words?

Internal fragmentation

Internal fragmentation is wasted space inside allocated blocks:

- **minimum block size larger than requested amount**
  - e.g., due to minimum free block size, free list overhead
- **policy decision not to split blocks**
  - e.g., allocating from segregated free lists (see [Wilson85])

Much easier to define and measure than external fragmentation.

Source of interesting computer science forensic techniques in the context of disk blocks

- **contents of "slack" at the end of the last sector of a file contain directory entries.**
- **provide a snapshop of the system that copied the file.**

For more information

D. Knuth, "The Art of Computer Programming, Second Edition", Addison Wesley, 1973

- **the classic reference on dynamic storage allocation**

Wilson et al, "Dynamic Storage Allocation: A Survey and Critical Review", Proc. 1995 Int'l Workshop on Memory Management, Kinross, Scotland, Sept, 1995.

- **comprehensive survey**
- **/afs/cs/academic/class/15-213/doc/dsa.ps**