Dynamic Memory Access

Textbook Section 11.5

Memory Allocation Review

- **Static Allocation of Memory**
  - memory reservation requirements are determined at compile- (or assembly-) time
- **Dynamic Allocation of Memory**
  - when memory is acquired and released throughout run-time, based on current need

Statically Allocated Singly Linked List

```
data        equu    0       ; node type for a list of words
next        equ     2
NODE_SIZEequ     6
NULL        equ     0       ; why zero for the NULL pointer?

head:       dc.l    node1
node1:      dc.w    1
            dc.l    node2
            ...             ; must the nodes be contiguous?
node2:      dc.w    2       ; must the declaration order be fixed?
            dc.l    NULL
```

In order for data structures to grow (and shrink) at run-time, dynamic allocation of memory must be possible.

Run-time Environment

A high-level language provides a run-time environment. It provides:

- initial program set up (i.e. before calling a “main” function)
- final program tear down (i.e. after “main” returns)
- a set of callable library functions

The run-time environment is responsible for:

- maintaining a heap - a pool of available memory for dynamic allocation,
- plus routines for allocating and deallocating blocks of memory

Run-time Environments

E.g. C provides the standard library functions:

- void *malloc(size_t);   allocates a block from the heap
- void free(void *);   returns a block to the heap

E.g. C++ provides the operators:

- new calls malloc
- delete calls free

ASM programs have no default high-level run-time environment!

They must create and manage their own heaps (or link to an existing run-time environment such as libc)

Heap Management

Heap space must be set aside at the beginning of the program.

Example:

```
heap:    ds.bHEAP_SIZE     ; a pool of available memory
                               ; for use at run-time
```

or, if only one type of node will ever be allocated:

```
heap:    ds.bNUM_NODES*NODE_SIZE

free:    dc.lheap          ; pointer to the next available
                               ; dynamic memory element
```

Heap Strategies - Fixed Size Blocks

1. One-time allocation:

```
allocation: if free == empty then
        set address = NULL
otherwise
        set address = “next available”
        increment “next available” pointer by block size
        return address
```

Heap Strategies - Fixed Size Blocks

1. One-time allocation

Deallocation: n/a

Pros: both operations extremely easy and fast

Cons: possible to run out of available memory despite free memory present
possible to recover but time consuming (i.e. HD defrag)

Heap Strategies - Fixed Size Blocks

2. Reusable nodes

- Maintain a free list of blocks: a list of unallocated dynamic memory
    - in this case each free block points at next free block in heap

Diagram: draw initial free list (all free blocks are initially contiguous)

Heap Strategies - Fixed Size Blocks

2. Reusable nodes

Allocation:

set address = free list head pointer
if address ≠ NULL then remove head block from free list
return address

Deallocation: insert block back into free list at head

Pros: able to reuse nodes

Cons: initial setup requires more time / complex
must maintain list – small amount of time for each operation

Heap Strategies - Variable Size Blocks

1. Use method from *Fixed Size Reusable Nodes*
    - With segregated free lists (“bins”) for each block size
    - Each operation takes the size or the size is additional to the memory size
2. Block Subdivision
    - Begin with one huge block in the free list

Heap Strategies - Variable Size Blocks

2. Block Subdivision

    - During allocation, subdivide when necessary.
    - Can use a first fit strategy: use 1st block which has size ≥ requested.
    - Can use a best fit strategy: search for smallest block which can satisfy the request.
    - If no “big enough” block is found, allocation request fails (or can request another large block of memory from the O/S, and add it to free list).
    - For deallocation, re-insert into free list
    - During deallocation, coalesce with free neighbours when possible

Fragmentation

Problem: Fragmentation is the gradual loss of **useful** space.

There are 2 kinds:

1. **Internal Fragmentation**: space is lost within an allocated block (because of minimum block sizes)
2. **External Fragmentation:** space is lost between allocated blocks

Excessive fragmentation may result in a failed allocation request, even if enough total memory is available!

Fragmentation - Possible Solutions

- Implement compaction: a periodic defragmentation of the heap
- Maintain multiple heaps, each for a different block size
    - Example: glibc under Linux

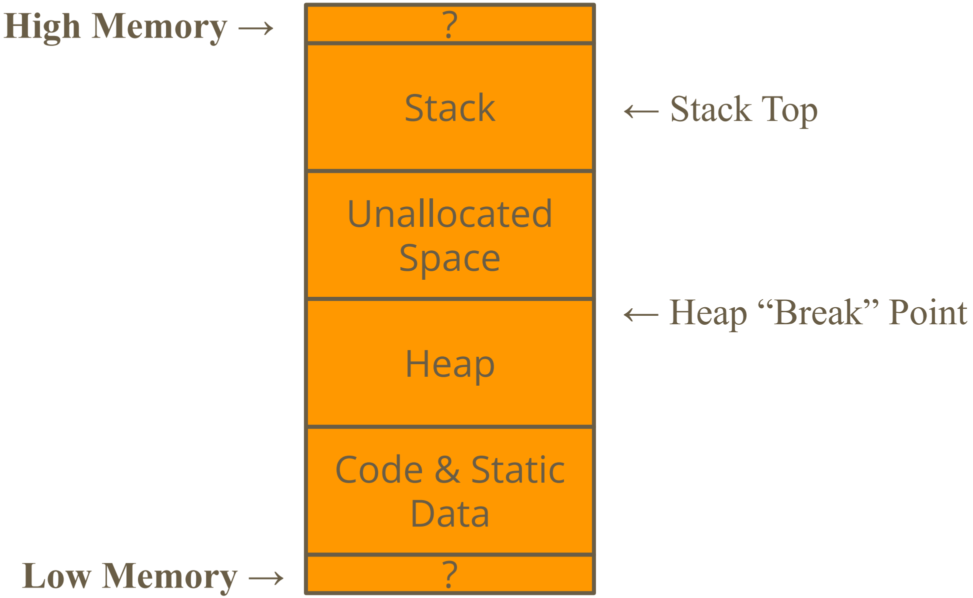
Common Dynamic Allocation Errors

- not checking “out of memory” condition
- memory leaks
- using memory once freed
- freeing memory twice
- overflowing an allocated block

Memory Map - Revisited

Typically, when allocating memory dynamically, the request for more space is ultimately made to the O/S.

Memory map for a Linux process:



Memory Requests From O/S

- Requesting memory from the O/S is “expensive”.
- Heap managers make large, infrequent requests from the O/S and then subdivide the obtained block
- E.g. in Unix/Linux, C’s malloc library function makes infrequent calls to the brk system call, which raises the “break point”

Storage Class

The storage class of a variable refers to the region from where it is allocated. This also relates to when it is allocated and deallocated (i.e. its “lifetime”).

In C/C++ -like languages, the 4 typical storage classes are:

1. static		compile/assembly allocation
2. dynamic	(“heap dynamic”)	run-time allocation from heap
3. automatic	(“stack dynamic”)	run-time allocation on stack
4. register		

This is distinct from variable “scope”, which refers to identifier visibility (e.g. global vs. local variables).