

Dynamic Storage Allocation

CS 414: Operating Systems
Spring 2008

Memory Allocation

- ♦ Static Allocation (fixed in size)
 - Sometimes we create data structures that are “fixed” and don’t need to grow or shrink.
- ♦ Dynamic Allocation (change in size)
 - At other times, we want to **increase** and decrease the size of our data structures to accommodate changing needs.
- ♦ Often, real world problems mean that we don’t know how much space to declare, as the number needed will change over time.

Static Allocation

- ♦ Done at compile time.
- ♦ Global variables: variables declared “ahead of time,” such as fixed arrays.
- ♦ Lifetime = entire runtime of program
- ♦ Advantage: efficient execution time.
- ♦ Disadvantage?
 - If we declare more static data space than we need, we **waste space**.
 - If we declare less static space than we need, we are **out of luck**.

Dynamic Allocation

- ♦ Done at run time.
- ♦ Data structures can grow and shrink to fit changing data requirements.
 - We can allocate (create) additional storage whenever we need them.
 - We can de-allocate (free/delete) dynamic space whenever we are done with them.
- ♦ Advantage: we can always have exactly the amount of space required - **no more, no less**.
- ♦ For example, with pointers to connect them, we can use dynamic data structures to create a chain of data structures called a linked list.

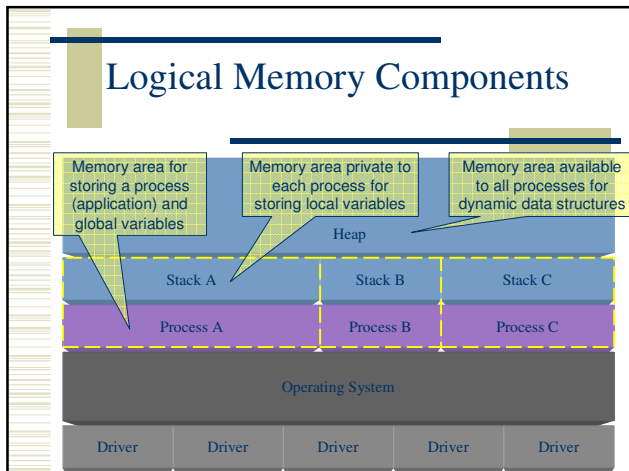
Dynamic Allocation

- ◆ Local Variables
 - Lifetime = duration of procedure activation (as long as that method is active)
 - Advantage: efficient storage use.
 - **Stack Allocation** (all variables allocated within the procedure scope unless declared static).
 - For example, parameter passing during function calls.
- ◆ User-allocated variables
 - Lifetime = until the user deletes it (or until it is garbage-collected)
 - Advantage: permits creation of dynamic structures like trees, linked lists etc.
 - **Heap Allocation.**
 - For example, linked lists.

Memory Management of a Process

- ◆ The memory of a process is divided into the following parts
 - A space for the code of the program
 - A space for the data (global variables)
 - The stack, for the local variables
 - The heap, for the dynamic variables

Logical Memory Components



Why isn't static allocation sufficient?

- ◆ Recursive procedures
 - ◆ Complex data structures
 - If all storage must be reserved in advance (statically), then it will be used **inefficiently** (enough will be reserved to handle the worst possible case).
- OS cannot predict
- ◆ how many jobs there will be or
 - ◆ which programs will be run or
 - ◆ when a process will come and ask for storage space!!!!!!

Dynamic Memory Management

- ◆ Dynamic memory allocation is performed by the operating system when the program is executing and the program (user) sends a request for additional memory.
- ◆ Three issues need to be addressed by the operating system:
 - **allocating** variable-length dynamic storage when needed
 - **freeing** up storage when requested
 - **managing** the storage (e.g. reclaiming freed up memory for later use)

Dynamic Storage Allocation methods

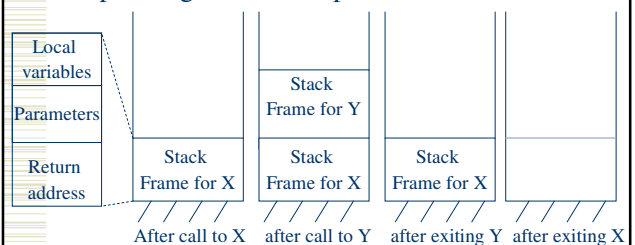
- ◆ **Stack Allocation**
 - Used to allocate local variables.
 - Grown and shrunk on procedure calls and returns.
- ◆ **Heap Allocation**
 - Used to allocate dynamic objects.
 - Heap objects are accessed with pointers.

Stack-Based Allocation

- ◆ Memory allocation and freeing are partially **predictable**.
- ◆ Restricted but **simple and efficient**.
- ◆ Allocation is hierarchical: Memory freed in opposite order of allocation.
 - If alloc(A) then alloc(B) then alloc(C), then it must be free(C) then free(B) then free(A).
- ◆ Procedure call:
 - Program calls X, which calls Y. Each call pushes another stack frame on top of the stack. Each stack frame has space for variable, parameters, and return addresses.

Stack-Based Allocation: Example

- ◆ A stack-based organization keeps all the free space together in one place.

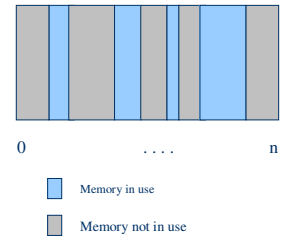


Heap Organization

- ◆ Allocation and release are unpredictable.
- ◆ Heaps are used for arbitrary list structures, complex data organizations.
- ◆ More general, less efficient.
- ◆ Example: payroll system.
 - Do not know when employees will join and leave the company.
 - Must be able to keep track of all of them using the least possible amount of storage.
 - What if we implement it using stack?

Heap Organization

- ◆ Memory consists of allocated areas and free areas (holes).
- ◆ Goal
 - Reuse the spaces in holes.
 - Keep the no. of holes small.
 - Keep the size of holes large.
- ◆ Problem: Fragmentation!!
 - Holes may be too small to be useful for any allocation.
 - Inefficient use of memory.



Fragmentation and Compaction

- ◆ External fragmentation
 - Total memory space exists to satisfy a request, but it is not contiguous.
- ◆ Internal fragmentation
 - Allocated memory may be slightly larger than requested memory; holes in the memory block allocated.
- ◆ Compaction
 - Reduce external fragmentation.
 - Shuffle memory contents to place all free memory together in one large block.

Heap Allocation Methods

- ◆ Typically, heap allocation schemes use a *free list* to keep track of the storage that is not in use.
- ◆ Algorithms differ in how they manage the free list
- ◆ Best Fit
 - Keep linked list of free blocks.
 - Search the whole list on each allocation.
 - Choose block that comes closest to matching the needs of allocation.
 - Save excess for later.
 - Merge adjacent free blocks during release operations.
- ◆ First Fit
 - Keep linked list of free blocks.
 - Scan the list for the first block that comes closest to matching the needs of allocation.
 - Merge adjacent free blocks during release operations.

Example: Heap Allocation Methods

Request for Memory

35



Memory in use

Memory not in use

Example: Best Fit

Request for Memory

35



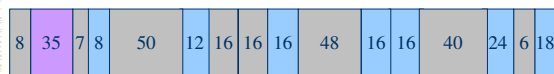
Memory in use

Memory not in use

Example: First Fit

Request for Memory

35



Memory in use

Memory not in use

Comparing the BF and FF

- ♦ Is BF always better than FF?
 - It depends
- ♦ Try the following scenario
 - Memory contains 2 free blocks of size 20 and 15
 - Suppose the allocation requests are 10 then 20. Which will win?
 - Suppose the requests are 8, 12, then 12. Which will win?

Other Heap Allocation Methods

◆ Worst Fit

- Keep linked list of free blocks.
- Search the whole list on each allocation.
- Choose block that worst matches the request.
- Save excess for later.
- Merge adjacent free blocks during release operations.

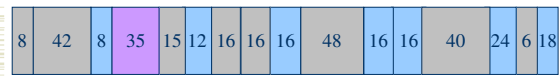
◆ Next Fit

- Keep linked list of free blocks.
- Start where the last search left off.
- Scan the list for the first block that comes closest to matching the needs of allocation.
- Merge adjacent free blocks during release operations.

Example: Worst Fit

Request for Memory

35



Memory in use

Memory not in use

Example: Next Fit

Request for Memory

35

This is where the last search left off



Memory in use

Memory not in use

Bit Map

◆ Bit Map

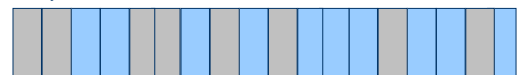
- Used when allocation comes in fixed-size chunks (e. g. disk blocks, or 128-byte chunks).
- Keep a large array of bits, one for every chunk
 - If bit is 0, chunk is in use.
 - If bit is 1, chunk is free.
- No need to merge later operations.

◆ Problem: Internal Fragmentation!!!

Bit Map:

1 1 0 0 1 1 0 1 0 1 0 0 0 1 0 0 1 0

Memory:

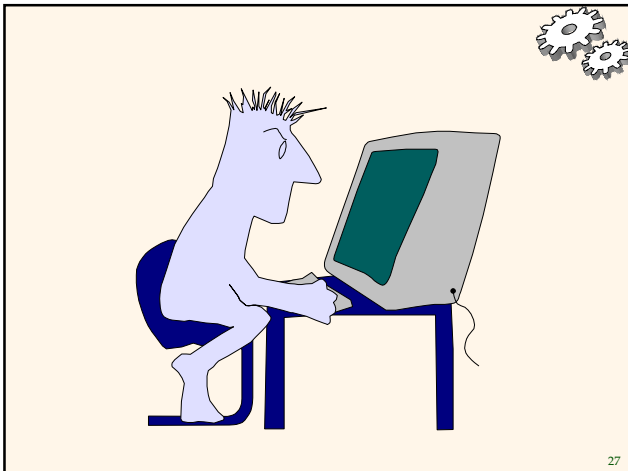
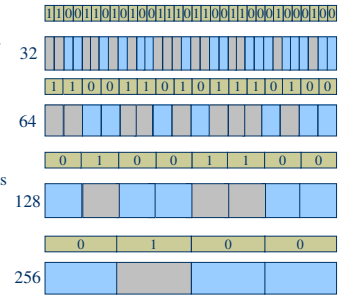


Problems with BF, FF, and Bitmap

- ◆ Linear search
 - Extra processing to maintain and find the free space.
- ◆ Fragmentation
 - External fragmentation for BF, FF
 - Internal fragmentation for bitmap

Pools

- ◆ Pools
 - Multiple free list, one for each size
 - Allocation fast.
 - Reduces Internal fragmentation.
- ◆ Any problems?
 - Inefficiency if some pools run out of space while others have a lot of free space.
- ◆ Solution?
 - Shuffle between pools.



Reclamation Methods

- ◆ How do we know when dynamically-allocated memory can be freed?
 - Easy when chunk only used in one place (one pointer per object).
 - Harder when information is shared (multiple pointers); it can't be recycled until all the sharers are finished.
- ◆ Problems
 - Dangling Pointers: when freed too early
 - Memory Leaks: when you forget to free
- ◆ Goal: **don't free up too soon, but don't forget to free up eventually**
- ◆ If memory leaks 100KB per hour, how long it will take to exhaust 24MB?
 - 10 days
 - When you run out of memory, there's a magic button to get more memory

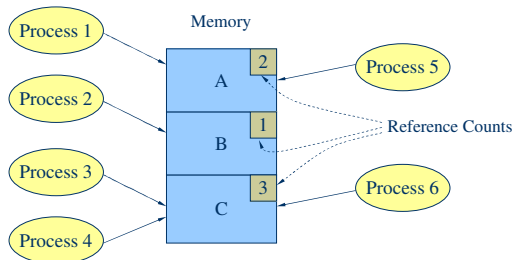
Reclamation Approaches

- ◆ Two general solutions
 - Reference Counts
 - Reference Counts keep track of the number of outstanding pointers for each chunk of memory.
 - Works well for hierarchical structures; must be managed carefully by the system
 - Garbage Collection
 - No explicit free operations; makes life easier for application programmers
 - Examples: LISP, Java

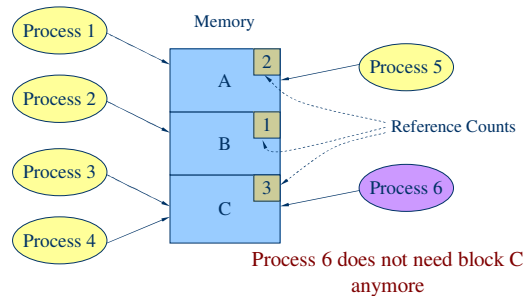
Reference Counter Method

- ◆ Reference Counts keep track of the number of outstanding pointers for each chunk of memory.
- ◆ The reference counts must be managed automatically (by the system) so no mistakes are made in incrementing and decrementing them.
 - Each time a new pointer refers to the block of memory, the reference count of the block must be **increased** by 1.
 - Each time a pointer that refers the block of memory is changed so it no longer points to that sublist, the reference count of the block must be **decreased** by 1.
 - When reference count becomes zero, free the memory.
- ◆ Example: File descriptors in Unix.

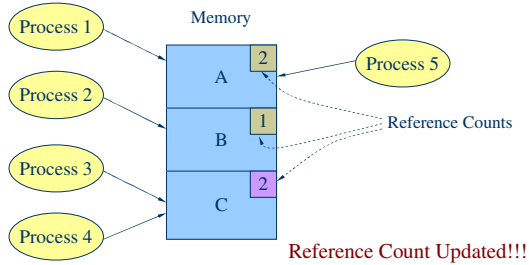
Reference Counter: Example



Reference Counter: Example



Reference Counter: Example



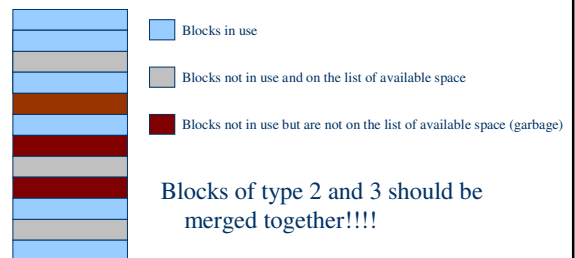
Reference Counter Method

- ♦ **Advantages:**
 - Unused blocks of memory are returned to the available list as soon as they are unused.
 - Time spent for reclamation is being undertaken continually (unlike garbage collection which happens irregularly and takes more execution time when it does happen).
- ♦ **Disadvantages:**
 - Requires additional overheads (storage of the reference count for each block, reference count updating algorithm etc.
 - It is still time consuming for reference count updating etc.. (but more dynamic than garbage collection).
 - **Circular structure:** example scenario?

Garbage Collection

- ♦ At all times, the heap may contain three kinds of dynamic data structures (or memory blocks):
 - Blocks that are **in use**.
 - Blocks that are **not in use** and are on the current list of available memory space.
 - Blocks that are **not in use** but are **not** on the list of available memory space (**garbage**).
- ♦ The management (reclaiming and recycling) of garbage blocks is called **garbage collection**.

Garbage Collection



Garbage Collection

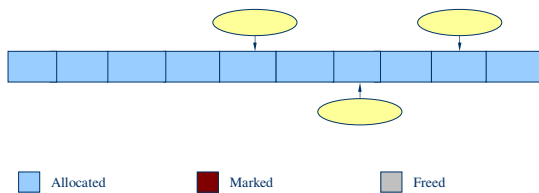
- ◆ Garbage Collection
 - No explicit free operation by the user.
 - When system needs storage, it searches through all the pointers and collects things that aren't used.
 - Only way in circular structures.
 - Incredibly difficult to program and debug.
- ◆ Must be able to find all pointers to objects.
- ◆ Must be able to find all objects.

Garbage Collection Mechanism

- ◆ Initialization:
 - Initialize all memory blocks to be free.
- ◆ Pass 1: Mark
 - Go through all statically-allocated and procedure-level variables, looking for pointers.
 - Mark each object pointed to, and recursively mark all objects it points to.
- ◆ Pass 2: Sweep
 - Go through all objects, free up those that aren't marked.
- ◆ Problem: Expensive!!!

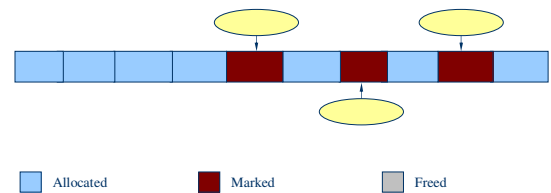
Garbage Collection: Example

Initial State of Memory



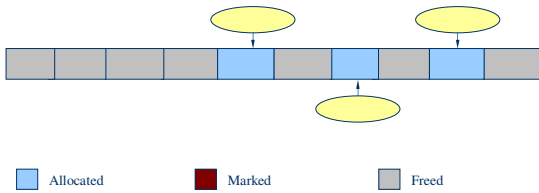
Garbage Collection: Example

Memory after Marking



Garbage Collection: Example

Memory after Garbage Collection



Cost of Garbage Collection

- ◆ Significant amount of CPU time to use it
 - Stop every once in a while
 - A missile could cross the state of VA in 10 seconds
 - Not applicable to certain applications
- ◆ Any better way?
 - Incremental garbage collection
 - Parallel garbage collection

