Dynamic Memory Management

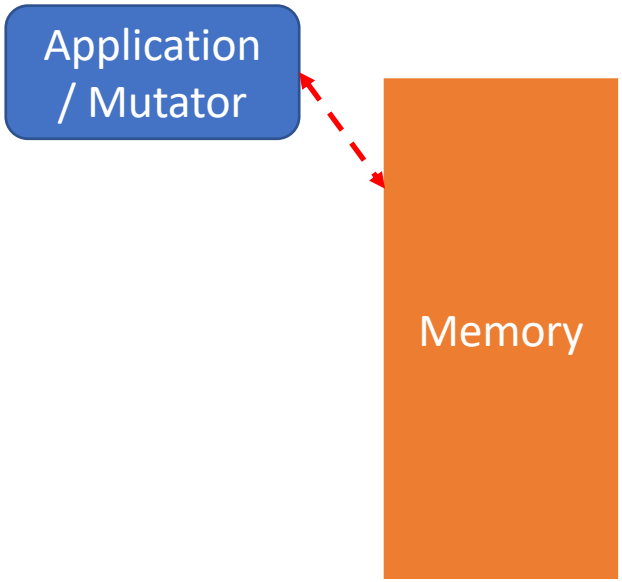
- Dynamic memory management refers to the run-time data structures and algorithms used for allocation and deallocation of objects *of variable size* that may remain alive beyond the lifetime of the method invocation in which they are allocated.
  - Distinct from global and method-static variables, whose allocations are frozen at compile time.
  - Distinct from method-local variables, which are allocated on the stack frame at run time but do not live beyond the method invocation.
  - Objects of variable size are important; if all objects were of a known fixed-size size, then the problem would be much simpler.

Dynamic Memory Management

- Dynamic memory management refers to the run-time data structures and algorithms used for allocation and deallocation of objects *of variable size* that may remain alive beyond the lifetime of the method invocation in which they are allocated.
  - Distinct from global and method-static variables, whose allocations are frozen at compile time.
  - Distinct from method-local variables, which are allocated on the stack frame at run time but do not live beyond the method invocation.
  - Objects of variable size are important; if all objects were of a known fixed-size size, then the problem would be much simpler.
- Two major flavors of dynamic memory management
  - Explicit: The *application* is responsible for freeing allocated memory.
    - E.g., `malloc()`/`free()` in C, `new`/`delete` in C++.
  - Implicit (aka garbage-collected): The *allocator* is responsible for detecting when an allocated block is no longer being used by the application and then freeing the block.
    - E.g., Java, Python.

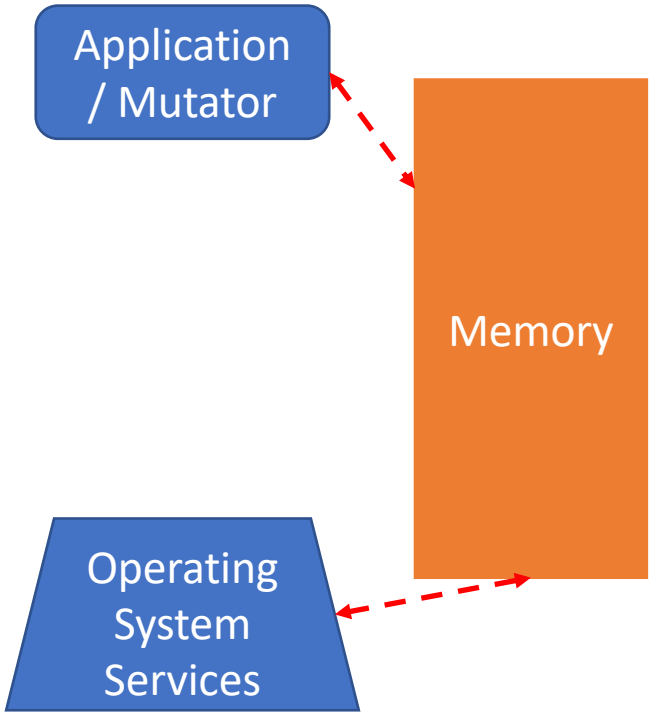
Subsystems and Interactions

- Application (aka mutator).
  - Initiator of object allocation requests.
  - May also initiate object deallocation.



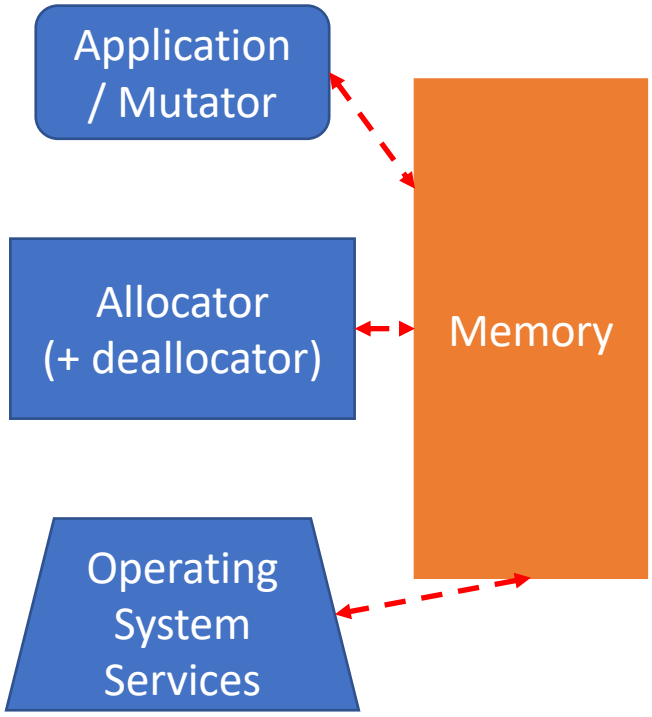
Subsystems and Interactions

- Application (aka mutator).
  - Initiator of object allocation requests.
  - May also initiate object deallocation.
- Operating system services.
  - Supplier of memory pool that allocator uses to satisfy mutator requests.



Subsystems and Interactions

- Application (aka mutator).
  - Initiator of object allocation requests.
  - May also initiate object deallocation.
- Operating system services.
  - Supplier of memory pool that allocator uses to satisfy mutator requests.
- Allocator (+ deallocator).
  - Typically part of language RTS.

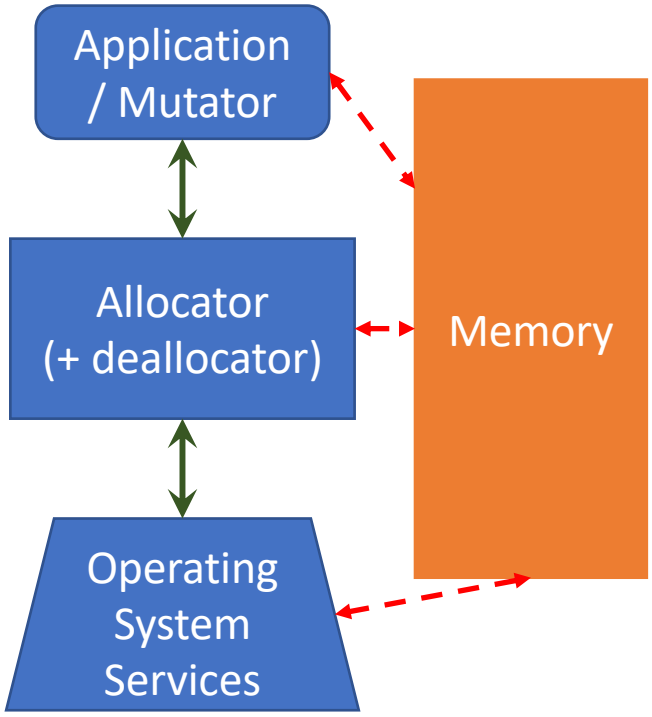


Subsystems and Interactions

- Application (aka mutator).
  - Initiator of object allocation requests.
  - May also initiate object deallocation.
- Operating system services.
  - Supplier of memory pool that allocator uses to satisfy mutator requests.
- Allocator (+ deallocator).
  - Typically part of language RTS.
- Interactions
  - Bidirectional communication: allocator/mutator, and allocator/OS services.
  - [C]

```
#include <stdlib.h>
void *malloc(size_t);
void free(void *);

#include <unistd.h>
void *sbrk(intptr_t);
```



The Allocator-Centric Worldview

- At any instant of time, the allocator possesses some amount of memory that it has obtained from the OS.

The Allocator-Centric Worldview

- At any instant of time, the allocator possesses some amount of memory that it has obtained from the OS.
- This memory pool (“heap”) is partitioned (“fragmented”) into [maximal] contiguous chunks (“blocks”) of two types:

The Allocator-Centric Worldview

- At any instant of time, the allocator possesses some amount of memory that it has obtained from the OS.
- This memory pool ("heap") is partitioned ("fragmented") into [maximal] contiguous chunks ("blocks") of two types:
  - A free block is one that is currently under the allocator's control.
    - The allocator can use such a block to satisfy object allocation requests from the mutator.

The Allocator-Centric Worldview

- At any instant of time, the allocator possesses some amount of memory that it has obtained from the OS.
- This memory pool ("heap") is partitioned ("fragmented") into [maximal] contiguous chunks ("blocks") of two types:
  - A free block is one that is currently under the allocator's control.
    - The allocator can use such a block to satisfy object allocation requests from the mutator.
  - A used block is one that is currently allocated and under the mutator's control.
    - In an explicit DMM scenario, such a block is off-limits to the allocator until the mutator issues a deallocation request on it (and thereby transitions it into a free block).
    - In an implicit DMM scenario:
      - The allocator can *reclaim* the block if it can determine that it is safe to do so. This will necessarily be a conservative approximation.
      - The allocation can *relocate* the block to a different location if it can do so without compromising the mutator's correctness.

Profile of Memory Usage

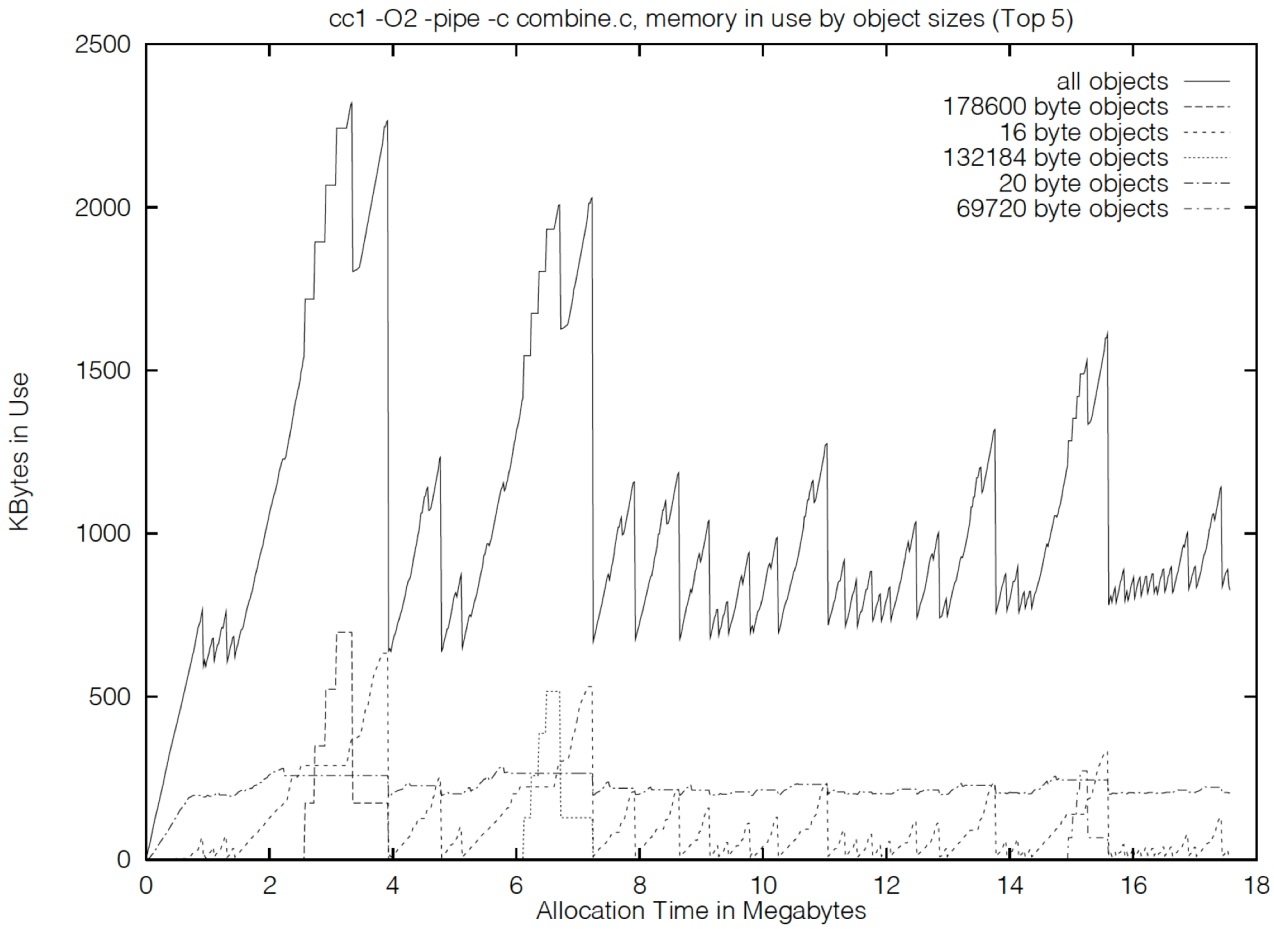


Fig. 1. Profile of memory usage in the GNU C compiler.

See Paul R. Wilson, Mark S. Johnstone, Michael Neely, and David Boles, ``Dynamic Storage Allocation: A Survey and Critical Review'' in *International Workshop on Memory Management*, September 1995

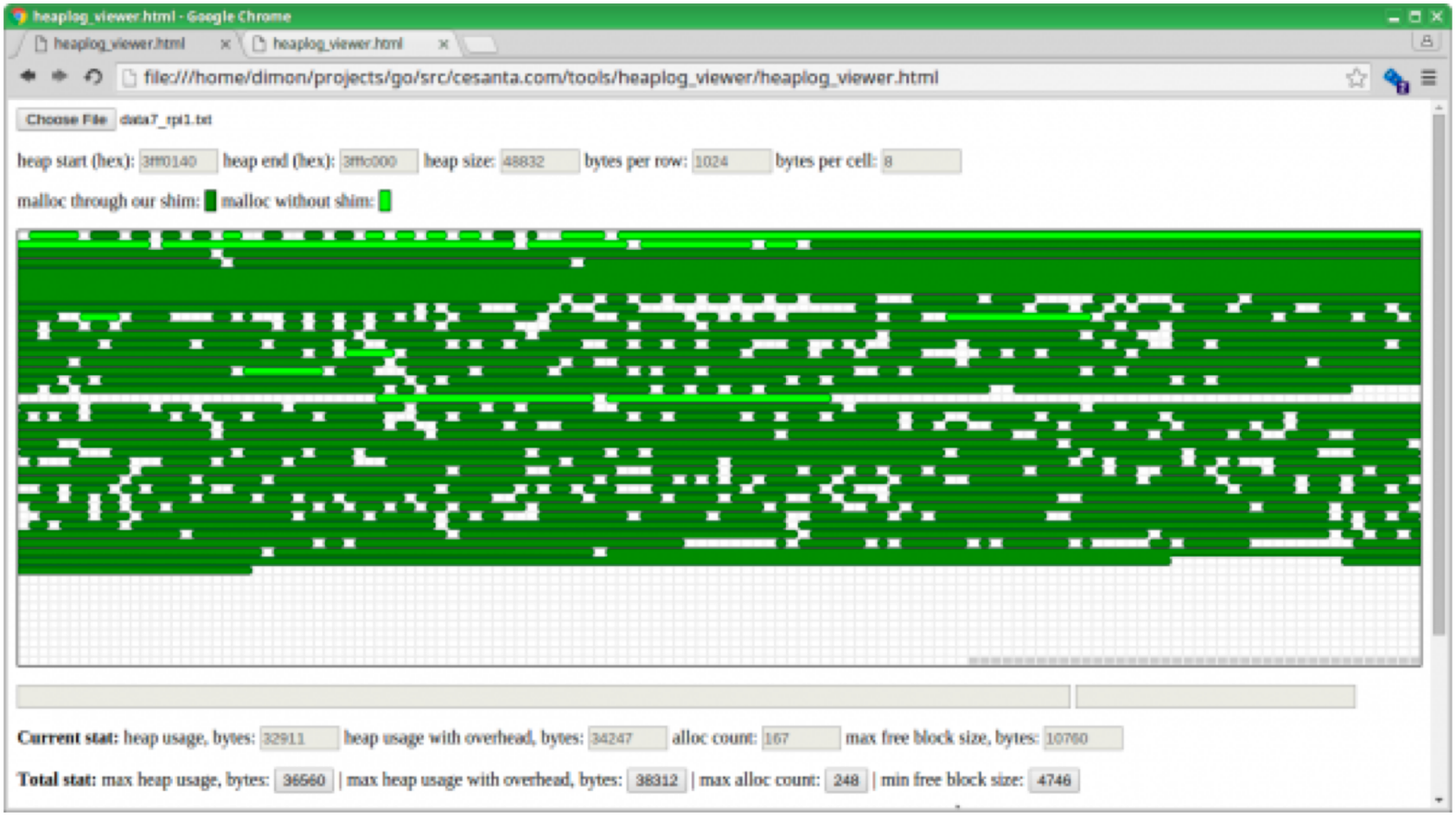
Overheads and Fragmentation

- Management Overheads
  - Each allocated block must carry some hidden *metadata* so that the allocator can figure out how to handle it when deallocating, reclaiming, or relocating the block.
    - At a minimum, this will have the size of the allocated block.
  - An allocated block may also need to incorporate padding in order to satisfy alignment constraints.
  - Let this total overhead add up to v bytes.

Overheads and Fragmentation

- Management Overheads
  - Each allocated block must carry some hidden *metadata* so that the allocator can figure out how to handle it when deallocating, reclaiming, or relocating the block.
    - At a minimum, this will have the size of the allocated block.
  - An allocated block may also need to incorporate padding in order to satisfy alignment constraints.
  - Let this total overhead add up to v bytes.
- Two types of fragmentation
  - Internal fragmentation: An allocation request for n bytes of payload is satisfied with a block of $(n + v + k)$ bytes.
  - External fragmentation: The heap contains free blocks of size n_1 and n_2 bytes, which, had they been adjacent in memory, could be used jointly to satisfy an allocation request for n bytes of payload.

A Snapshot of A Heap



Source: https://dmitryfrank.com/articles/heap_on_embedded_devices

Constraints On Allocator

- Correctness
  - Handling arbitrary sequences of requests.
  - Aligning blocks so that they can hold any type of data object.
  - Not impacting mutator correctness.
    - For explicit DMM: No manipulation/change of used blocks.
    - For implicit DMM: Provably (or conservatively) safe reclamation or relocation of used blocks.

Constraints On Allocator

- Correctness
  - Handling arbitrary sequences of requests.
  - Aligning blocks so that they can hold any type of data object.
  - Not impacting mutator correctness.
    - For explicit DMM: No manipulation/change of used blocks.
    - For implicit DMM: Provably (or conservatively) safe reclamation or relocation of used blocks.
- Responsiveness
  - Responding immediately to individual requests, without reordering or batching.

Constraints On Allocator

- Correctness
  - Handling arbitrary sequences of requests.
  - Aligning blocks so that they can hold any type of data object.
  - Not impacting mutator correctness.
    - For explicit DMM: No manipulation/change of used blocks.
    - For implicit DMM: Provably (or conservatively) safe reclamation or relocation of used blocks.
- Responsiveness
  - Responding immediately to individual requests, without reordering or batching.
- Scalability
  - Using only the heap.
    - Any variable-sized non-scalar data structures used by the allocator must themselves be maintained in the heap.
    - In other words, the allocator can’t call itself.