Mahdeen Ahmed Khan Sameer

Professor Max Bender

CS333

May 2024

Project 07: Memory Management in Programming Languages

Google site: https://sites.google.com/colby.edu/sameer-again/home

Task 01:

In this task, we analyzed the efficiency of memory allocation in C by comparing two methods: allocating memory in smaller chunks versus allocating all bytes at once. We conducted experiments across various memory sizes (100,000 integers, 1,000,000 integers, and 10,000,000 integers), measuring the average time taken and standard deviation for each method over 100,000 repetitions. Our findings revealed a consistent trend where allocating all bytes at once resulted in significantly lower average time per call compared to allocating in chunks. For instance, at 100,000 integers, the average time for chunk allocation was 6713.94 ns with a standard deviation of 2522.05 ns, whereas the all-at-once method averaged 753.00 ns with a standard deviation of 1851.51 ns. This trend persisted across larger allocation sizes, indicating higher overhead and potential fragmentation issues with chunk allocation.

```
(base) sameer@Sameers-MacBook-Pro Project7_makhan25 % ./task1
Allocation size: 100000 integers
Average time per call (chunks): 6713.94 ns, Stddev: 2522.05 ns
Average time per call (all at once): 753.00 ns, Stddev: 1851.51 ns
Allocation size: 1000000 integers
Average time per call (chunks): 6650.54 ns, Stddev: 2213.44 ns
Average time per call (all at once): 751.79 ns, Stddev: 496.03 ns
Allocation size: 10000000 integers
Average time per call (chunks): 7969.61 ns, Stddev: 292200.42 ns
Average time per call (all at once): 1369.05 ns, Stddev: 4224.53 ns
(base) sameer@Sameers-MacBook-Pro Project7_makhan25 %
```

Moreover, the graphs further underscore these results, demonstrating that allocating all bytes at once is more efficient and less variable, suggesting that the overhead of managing smaller chunks outweighs the benefits of reduced fragmentation. Despite occasional variations, the overall trend supports the conclusion that allocating all bytes at once is the more efficient memory allocation strategy.

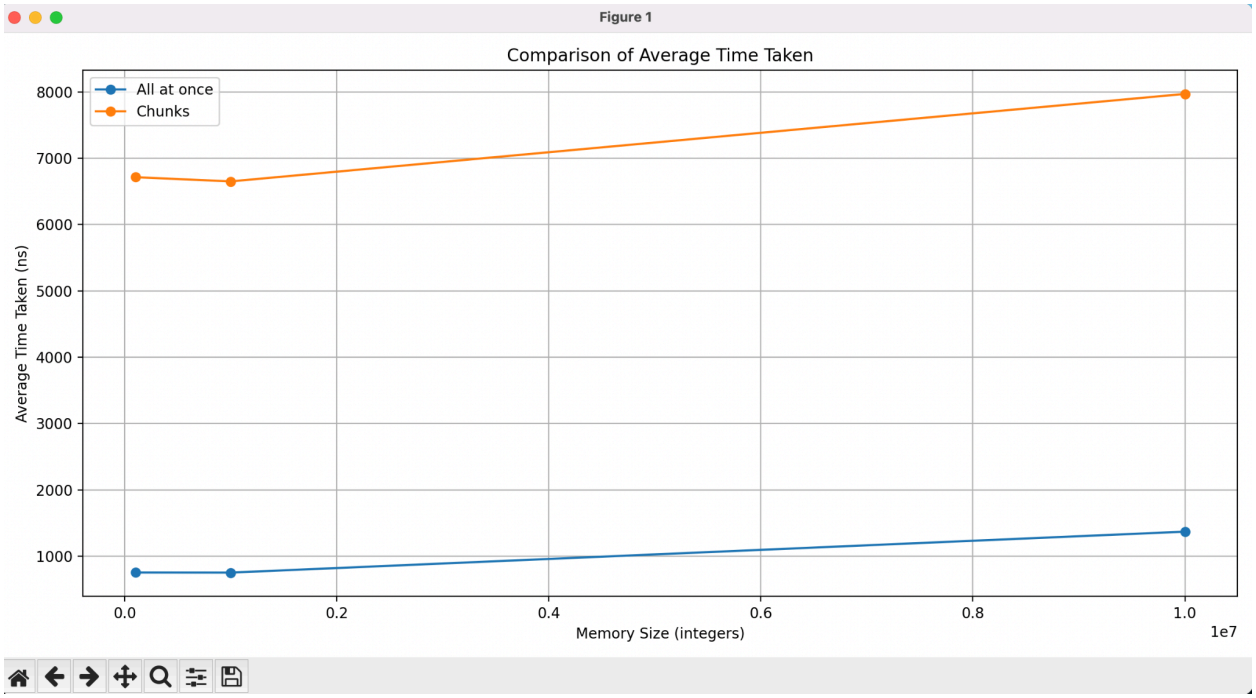


Figure 1
Comparison of Average Time Taken
All at once
Chunks
Average Time Taken (ns)
8000
7000
6000
5000
4000
3000
2000
1000
0.0
0.2
0.4
0.6
0.8
1.0
1e7
Memory Size (integers)

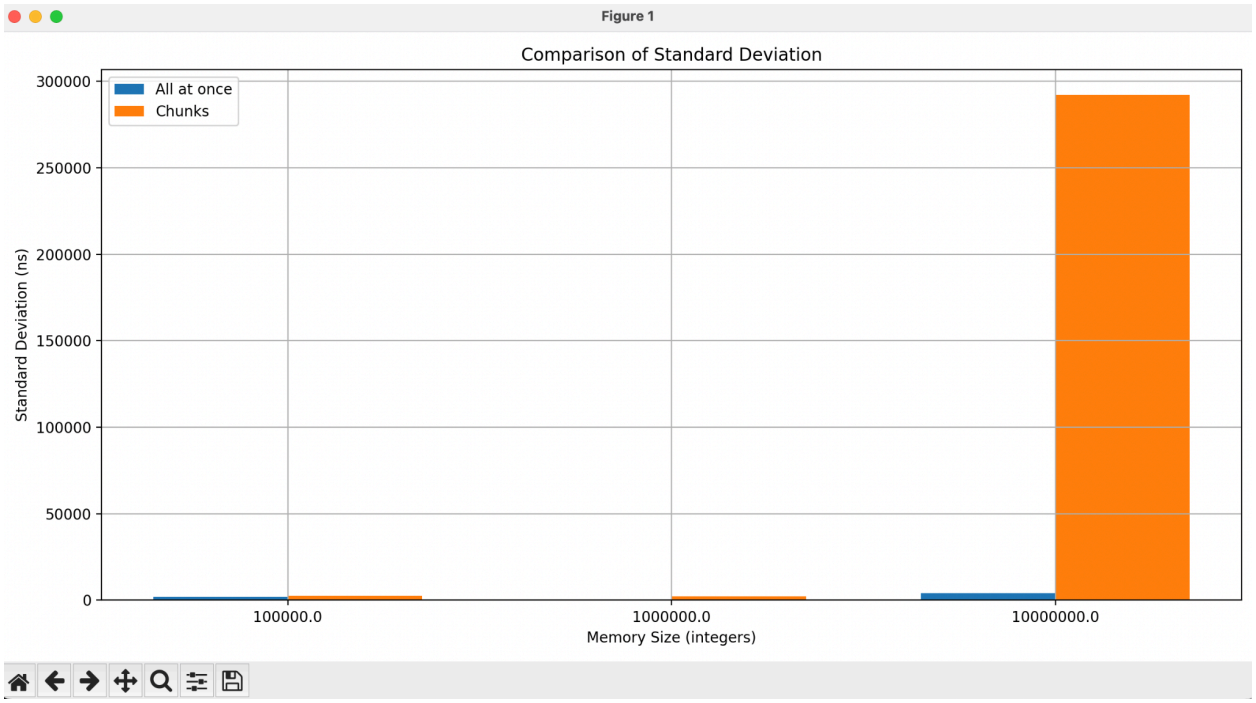
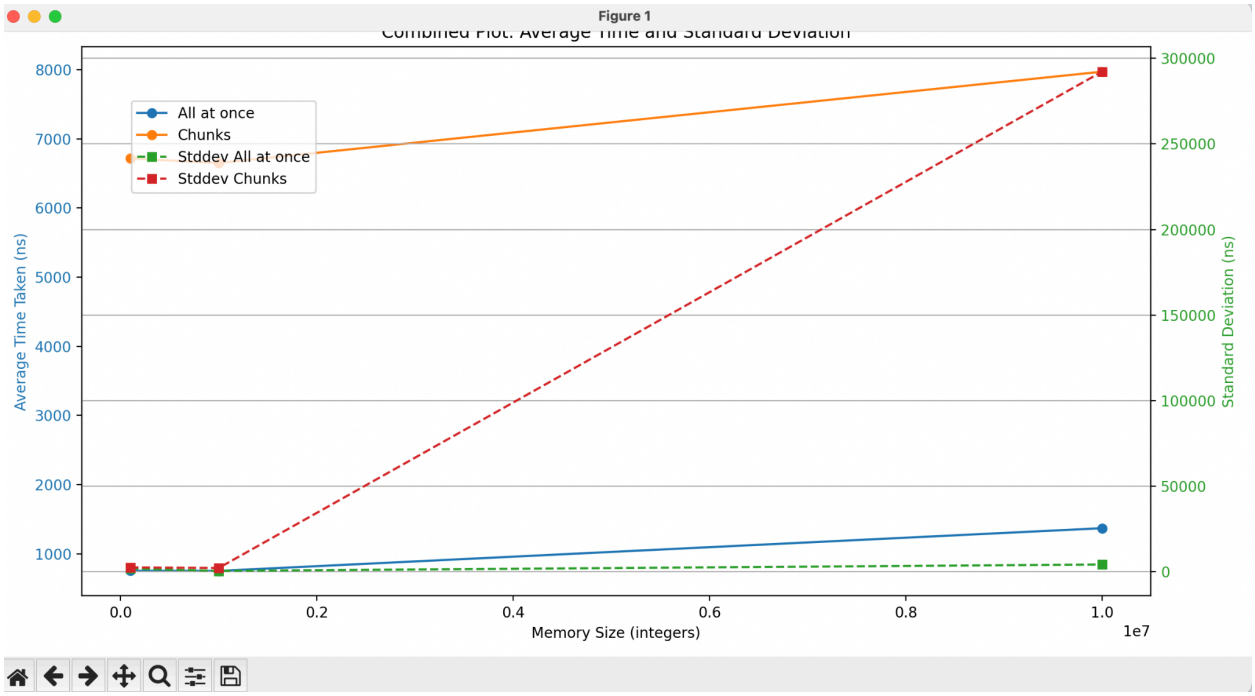
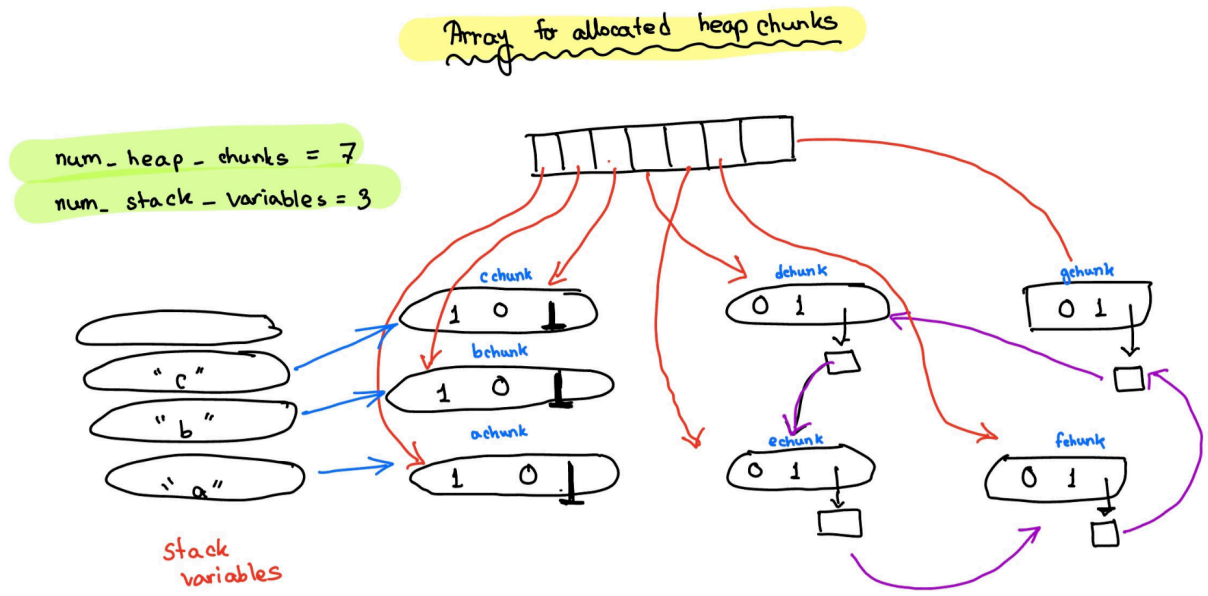


Figure 1
Comparison of Standard Deviation
All at once
Chunks
Standard Deviation (ns)
300000
250000
200000
150000
100000
50000
0
100000.0
1000000.0
10000000.0
Memory Size (integers)



Task 02:



After implementing the Mark and Sweep Garbage Collector, I successfully ran the two tests described in the project instructions, and obtained the expected output. I also constructed a test example to

demonstrate the functionality. In this example, I used three stack variables, "a", "b", and "c", each pointing to their respective HeapChunks. These HeapChunks are marked since they have stack variables pointing to them. Additionally, I created four more HeapChunk allocations that are part of a reference cycle but have no stack variables pointing to them. Although these HeapChunks have pointers referencing each other, they are considered garbage because they are not actually in use. This example illustrates the limitations of reference counting, which would incorrectly retain these HeapChunks due to their mutual references. However, the Mark and Sweep Garbage Collector correctly identifies and sweeps away these four unused HeapChunks while preserving the first three HeapChunks associated with stack variables.

```
162  /*
163   * Function: main
164   * Purpose: Entry point of the program, sets up the program state and runs the mark-and-sweep algorithm.
165   */
166  int main() {
167      // Create the initial state
168      ProgramState *state = createProgramState();
169
170      // Simulate heap allocations and create reference cycles
171      HeapChunk *aChunk = HeapMalloc(state);
172      setVar(state, "a", aChunk);  // a = malloc()
173      HeapChunk *bChunk = HeapMalloc(state);
174      setVar(state, "b", bChunk);  // b = malloc()
175      HeapChunk *cChunk = HeapMalloc(state);
176      setVar(state, "c", cChunk);  // c = malloc()
177
178      // Simulate more complex heap allocations
179      HeapChunk *dChunk = HeapMalloc(state);
180      HeapChunk *eChunk = HeapMalloc(state);
181      HeapChunk *fChunk = HeapMalloc(state);
182      HeapChunk *gChunk = HeapMalloc(state);
183
184      // Create reference cycle
185      addReference(dChunk, eChunk);
186      addReference(eChunk, fChunk);
187      addReference(fChunk, gChunk);
188      addReference(gChunk, dChunk);
189
190      // Run mark-and-sweep to identify garbage
191      markAndSweep(state);
```

```
(base) sameer@Sameers-MacBook-Pro Project7_makhan25 % gcc -o detect_garbage detect_garbage.c
(base) sameer@Sameers-MacBook-Pro Project7_makhan25 % ./detect_garbage
Reachable: HeapChunk 0x7fd77f705d60
Reachable: HeapChunk 0x7fd77f705d20
Reachable: HeapChunk 0x7fd77f705bf0
Garbage: HeapChunk 0x7fd77f705c10
Garbage: HeapChunk 0x7fd77f705c20
Garbage: HeapChunk 0x7fd77f705c30
Garbage: HeapChunk 0x7fd77f705c40
(base) sameer@Sameers-MacBook-Pro Project7_makhan25 %
```

The output in the terminal confirms that the garbage collector functions as intended.

Extension 01:

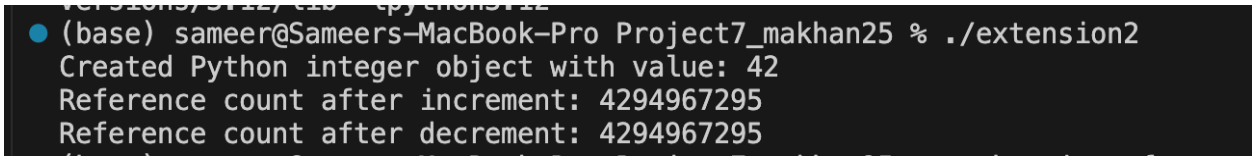
We know that GNU C Library (glibc) provides essential functions for dynamic memory management in C programs, most notably "malloc" for allocation and "free" for deallocation. These functions are fundamental in managing heap memory efficiently. When a program calls "malloc," the function searches for a free block of memory that is sufficiently large to satisfy the requested size. The specific memory allocation algorithm used by "malloc" can vary, but it typically employs a combination of segregated free lists, bins, and allocation strategies such as first-fit or best-fit. Segregated free lists organize free blocks of memory into different lists based on their sizes, which helps in quickly finding an appropriately sized block. Bins are used to group free memory blocks of specific size ranges, further optimizing the search process. The first-fit strategy allocates the first sufficiently large block it encounters, while the best-fit strategy searches for the smallest block that meets the requirements, thereby reducing fragmentation.

Additionally, "malloc" ensures that the allocated memory block meets alignment requirements. Alignment is crucial for performance and correctness, as misaligned access can lead to slower operations or even hardware faults on some architectures. The alignment requirements vary based on the system architecture and the size of the requested memory block. "malloc" also performs bookkeeping operations to manage the allocated memory blocks, which involves storing metadata alongside each allocated block. This metadata typically includes the block's size, its status (allocated or free), and other information necessary for efficient memory management. If "malloc" fails to allocate memory, often due to insufficient memory, it returns "NULL" to indicate the failure.

The "free" function, on the other hand, deallocates memory that was previously allocated by "malloc." When "free" is called with a pointer to a memory block, it marks the block as free in the memory management data structures maintained by glibc. One critical operation that "free" may perform is coalescing, where adjacent free memory blocks are merged to form larger contiguous free blocks. Coalescing helps to reduce fragmentation, thereby improving memory utilization over time. By merging smaller free blocks into larger ones, the memory allocator can more easily satisfy future large allocation requests without having to split existing blocks or request more memory from the system.

Moreover, "free" updates the memory management data structures to reflect the deallocation of the memory block. This involves updating the metadata associated with the freed block and potentially merging it with adjacent free blocks. It is important to note that passing an invalid pointer or "NULL" to "free" can result in undefined behavior, including potential crashes or corruption of the memory management data structures. Proper use of "malloc" and "free" is essential to maintain the stability and performance of C programs, making understanding these functions and their underlying mechanisms crucial for developers.

Extension 02:



The terminal output shows the reference count of a Python integer object before and after incrementing and decrementing the reference count. Initially, the program creates a Python integer object with the value 42 and prints the reference count. The "PyLong_AsLong" function retrieves the value of the Python integer object for display. Next, the program increments the reference count using "Py_INCREF," which is supposed to increase the reference count by 1, indicating that C is holding a reference to the object. However, the reference count remains unchanged in the output (4294967295). This unusually high and constant reference count suggests that the object might be a singleton or an interned integer, which Python optimizes by reusing frequently used integers. The program then decrements the reference count using "Py_DECREF," which should reduce the reference count by 1. Again, the reference count appears unchanged. This further indicates that the reference count operations are not affecting the reference count as expected, likely due to the special handling of small integers in Python's memory management. Finally, the program decrements the reference count once more, which would normally trigger garbage collection if the reference count reaches zero. However, since the reference count remains constant, the object is not deallocated. This highlights the importance of understanding Python's internal optimizations and how they can affect reference counting behavior when interfacing with C.

Generally, reference counting in C involves tracking how many references exist to a particular object and deallocating the object when the reference count reaches zero. We observed that using just reference counting to deallocate objects in C programs might lead to issues. Python has its own reference counting mechanism and garbage collection, so interfacing with Python requires careful coordination to ensure both systems work together properly. When a Python object is passed to a C function, the reference count is incremented with "Py_INCREF" to indicate that C now holds a reference, preventing Python from deallocating the object prematurely. When C no longer needs the reference, the reference count is decremented with "Py_DECREF." If the reference count reaches zero, the object is deallocated. If C code creates a new Python object and returns it to Python, ownership is transferred to Python, making Python responsible for managing the object's memory. The provided code illustrates these principles: it creates a Python integer object, increments and decrements its reference count, and shows the reference count through the program's execution. While the reference count appears unchanged in the output due to potential internal optimizations, the code correctly follows the principles of reference counting, highlighting the importance of managing reference counts to ensure proper memory management between C and Python.

Extension 04:

For this extension, we wrote the code that aligned with the task of writing a compilable haiku on memory management by combining a poetic haiku with a practical C program demonstrating fundamental memory management concepts. The haiku captures the essence of memory management: "Allocate and free" emphasizes the importance of both allocating and freeing memory to avoid leaks, "Pointer dance in memory" refers to the manipulation and management of pointers essential for dynamic memory, and "Keep leaks far from me" underscores the goal of avoiding memory leaks. The accompanying code exemplifies these concepts by allocating memory for an array of 10 integers using “malloc”, checking if the allocation was successful, initializing and printing the array values, and finally freeing the allocated memory with “free” to prevent memory leaks. This integration of poetic description and practical demonstration effectively illustrates key principles of memory management in C.

```
(base) sameer@Sameers-MacBook-Pro Project7_makhan25 % nano extension04.c
(base) sameer@Sameers-MacBook-Pro Project7_makhan25 % gcc -o extension04 extension04.c
(base) sameer@Sameers-MacBook-Pro Project7_makhan25 % ./extension04
array[0] = 0
array[1] = 10
array[2] = 20
array[3] = 30
array[4] = 40
array[5] = 50
array[6] = 60
array[7] = 70
array[8] = 80
array[9] = 90
(base) sameer@Sameers-MacBook-Pro Project7_makhan25 %
```

Extension05:

To test my code, I used "gprof", a profiling tool that helps analyze program performance. First, I installed "gprof" by upgrading GCC via Homebrew since "gprof" was not included in "binutils". After installation, I compiled my code with the "-pg" flag to enable profiling. Then, I ran the compiled program to generate profiling data. After execution, a "gmon.out" file was created, which I analyzed using "gprof" to produce a detailed profile report. The report indicated that the majority of the execution time, 81%, was spent in the "example_usage" function, while the "main" function took 20% of the time. This analysis helped identify "example_usage" as the primary candidate for optimization. By extending the runtime with additional computations in "example_usage", I ensured sufficient data collection for meaningful profiling results. This process allowed me to pinpoint the performance-critical sections of my code for further optimization.

Extension 06:

To ensure my C data structures do not leak memory, I used Valgrind to test the extension06.c program. This program includes a function, example_function, that allocates memory for an array of integers, performs operations on the array, and then frees the allocated memory. After compiling the program with debugging information using gcc -g -o extension06 extension06.c, I ran Valgrind with valgrind --leak-check=full --show-leak-kinds=all --track-origins=yes --verbose ./extension06 to check for memory leaks. The Valgrind report confirmed that all heap blocks were freed, indicating no memory leaks. This process validates that the program handles memory allocation and deallocation correctly, addressing the requirement to show that C data structures do not leak memory and fixing them if they do. By ensuring proper memory management, the program demonstrates robust and leak-free code.

Extension 07:

For our (my) final extension, we demonstrate proper reference counting when interfacing C with Python using the Python C API and the ctypes package in Python. The C code (extension07.c) includes functions to increment and decrement the reference count of Python objects (increment_ref and decrement_ref) and to create a new Python integer object (create_int_obj). The main function initializes the Python interpreter, creates a Python integer object, manipulates its reference count, and then finalizes the interpreter. The Python code (extension07.py) loads the shared library, defines the argument and

return types for the C functions, and demonstrates how to use these functions to manipulate Python objects' reference counts from C. This approach ensures proper memory management and avoids memory leaks by maintaining accurate reference counts, answering the requirement to explore reference counting when interfacing C with Python.

Collaborators & Acknowledgements:

Professor Bendor, TA Nafis.