DSA Assignment 4

Adarsh T. Shah
19473, M.Tech (A.I.)
adarshshah@iisc.ac.in

1 SPheap Implementation

The spheap is implemented in `spheap.h`. The allocation and deallocation methods are defined as follows:

- `void * mmalloc(size_t reqSize)` : to allocate memory of given input size.
- `void free(void * ptr,size_t reqSize)` : to free allocated memory. The function definition is different from `free` of `stdlib.h` as it also takes size as input.

The static allocation of 256 MB is done in `spheap.h` as a character array of size 268435456 bytes. Each memory block allocated has an associated structure stored in the Available Space list. This structure contains TYPE, KVAL and TAG as mentioned in the shen's weighted buddy system[1] along with starting address of the block allocated. The available space list is implemented as 56 sized array of pointers because $56 = 2 * 28$ and 256 MB = 2^{28} so there will not be more than 56 block sizes in the allocation tree. Each entry in ASL stores blocks corresponding to specific block size. `util_print_heap()` is a utility function used to visualized memory allocation. To keep track of internal and external fragmentation, global variables `spaceAllocated` and `spaceRequested` are used with the function `requestAllocation()`. The internal fragmentation is reported as sum of SPheap area allocated for all successful allocation requests minus sum of the request sizes of all successful allocation requests divided by the sum of request sizes of all successful allocation requests. In case of memory overflow, the program exits with error code 1 and displays the information such as total number of requests, internal fragmentation, external fragmentation, the requested memory, total number of splits and combines as mentioned in the assignment.

1.1 Application to Assignment 2

The spheap is implemented in Assignment 2 code by replacing `stdlib.h` with `spheap.h` in the header. The `malloc` and `free` methods are replaced with corresponding `mmalloc` and `ffree` methods respectively. To test it, run `./Assignment2_spheap` after running the `make` command. The Assignment 2 is modified to test automatically. A new function named `testcases()` is introduced which randomly generates test cases. The program is simulated for infinite time until it runs out of memory and stops. The run time is estimated using average clock cycles required to solve one equation. After every 50 test case evaluations, the program prints the statistics such as number of requests, combines, splits, internal fragmentation, external fragmentation and average clock cycles. To calculate average clock cycles, `time.h` and it's `clock()` functionality is used. The following are the main observations.

1. The run time to solve equation increases as the program progresses. This is evident by increase in the average number of clock cycles required to solve the test case.
2. The average clock cycle of spheap is in the order of 10000s whereas onebin takes 11.2 average clock cycles per test case.
3. The following shows the status of the program at a particular stage during execution.
   - Internal Fragmentation : 0.000
   - External Fragmentation : 0.991
   - Total Splits : 319293

- Total Combines : 36614
- Total Requests : 159670
- Avg Clock Cycles : 430268.77
- Total Equations solved : 650

4. Since the program slows down drastically, it becomes infeasible to run this program to the point of insufficient storage.

1.2 Simulation

The simulation is implemented in `test_spheap.c`. To test it, run `./test_spheap` after running the `make` command. The simulation is performed in the same way as performed in the shen's weighted buddy paper. The `Time` is initialized to zero and the `ASL` is initialized with only one memory block corresponding to 256MB memory size.

1. The `Time` is advanced by 1.
2. Generate a memory request of size according to certain distribution.
3. Decide the lifetime of the block requested in the previous step according to certain distribution.
4. Request the block using `mmalloc` and free the blocks whose lifetime ends at the current `Time`.

When `Time` is a multiple of 200, the statistics such as total splits, combines, requests, internal and external fragmentation are displayed. The memory requests are generated from the following two distribution.

- Uniform integer distribution between 100 and 2000
- Exponential integer distribution between 100 and 2000 with mean 1000

The lifetime of each memory block is decided using as uniform integer distribution between 1 and 100. To keep the track the memory block to be freed, an array of points of size 100 is implemented. It holds the details of the requests which are to be freed at particular `Time`. To measure external fragmentation, the memory overflow was activated by ceasing to release blocks after `Time` = 2000. The results of uniform memory request distribution are mentioned as follows:

- Internal Fragmentation : 0.174
- External Fragmentation : 0.218
- Total Splits : 319219
- Total Combines : 3225
- Total Requests : 172138

The results of exponential memory request distribution are mentioned as follows:

- Internal Fragmentation : 0.138
- External Fragmentation : 0.103
- Total Splits : 400125
- Total Combines : 2930
- Total Requests : 245643

The simulation verfies the claim that weighted buddy has better performance in case of exponential requests generation as compared to uniform.

2 One Bin Implementation

The one bin is implemented as `onebin.h`. The 256MB is allocated statically in `onebin.h` as 268435456 sized character array. The memory blocks available are of fixed size. The initialization function `mem_init()` takes fixed allocation block size as input and initializes available memory blocks. The available memory blocks information is maintained as linked list named `available`. The oneBin is similar to spheap with same function definitions but different implementations.

- `void mem_init(size_t reqSize)`
- `void * mmalloc(size_t reqSize)`
- `void free(void * ptr,size_t reqSize)`

2.1 Application to Assignment 2

The onebin is implemented in Assignment 2 code by replacing `stdlib.h` with `onebin.h` in the header. The memory blocks are intialized to the size of the structure `Digit` using `mem_init()`. The `malloc` and `free` methods are replaced with corresponding `mmalloc` and `ffree` methods respectively. The testing is done in the same manner as spheap. To test it, run `./Assignment2_onebin` after running the `make` command. The following are the main observations.

1. The program runs to its entirety and stops at the point of insufficient memory.
2. The execution speed of oneBin is much higher as compared to spheap. This is reflected in the average clocks per test case.
3. The following shows the state of memory at the time of program termination.
   - Internal Fragmentation : 0.000
   - External Fragmentation : 0.000
   - Space Requested: 268358672
   - Space Allocated : 268358672
   - Avg Clock Cycles : 11.02
   - Total Equations solved : 74950
4. There is no internal fragmentation in this case because only the structure `Digit` is used for initialization and allocation. Hence, the memory request size is constant through out the program execution.

2.2 Simulation

The simulation is performed in the same manner as done in spheap implementation. The memory blocks are initialized to 2000 Bytes. The results of uniform memory request distribution are mentioned as follows:

- Internal Fragmentation : 0.952
- External Fragmentation : 0.00

The results of exponential memory request distribution are mentioned as follows:

- Internal Fragmentation : 1.359
- External Fragmentation : 0.00

3 Comparison between SPheap and oneBin

1. **Execution Speed:** The runtime performance of oneBin is much higher as compared to the spheap allocation technique. The `mmalloc` and `ffree` operations in oneBin involves just one linked list deletion and insertion respectively. However, in spheap, `mmalloc` and `ffree` performs insertion and deletion in binary search tree along with `splits` and `combine` operations. Hence, the memory allocation overhead in oneBin involves only **O(1)** operations

whereas spheap involves **O(logn)** operations where n is the number of requests. The execution speed of `test_spheap` slows down with time. This is evident as the number of requests increase, the runtime overhead also increases.

2. **Space Efficiency:** The Simulation results clearly show that spheap's space efficiency is much better as compared to oneBin's. There are 56 possible memory block size options available in spheap whereas only 1 fixed memory block size is available in oneBin.

    - **Internal Fragmentation:** The internal fragmentation will be higher in oneBin as compared to SPheap because if the variation in the memory block request sizes is high during program execution, then most of the memory allocated will be wasted by memory requests with small sizes. In OneBin, the allocation size is initialized to largest possible memory request size. In case of SPheap, smaller block sizes are available for small memory requests. If not available, the larger blocks are split into appropriate smaller blocks reduce internal fragmentation.
    - **External Fragmentation:** There won't be any External Fragmentation in oneBin because when the program runs out of memory, all the fixed size blocks are allocated utilizing the 256MB completely. In case of SPheap, there are blocks of smaller sizes, non-contiguous, unallocated and available when the spheap is not able to fulfill a certain larger request size. This results in external fragmentation in case of SPheap. The simulations also show that external fragmentation is reduced in SPheap in case of exponential requests generation as compare to uniform.

4 Comparision with standard `malloc` and `free`

The automated Assignment 2 simulation with standard `malloc` and `free` is done in `Assignment2.c` same as done with oneBin and spheap implementation. To test it, run `./Assignment2` after `make`. The following are the main observations.

- Average clock cycles : 17.5
- Number of equations : 1688700+.

Clearly, the execution speed of one bin is better as compared to standard implementation but the standard implementation has much higher capacity to solve large number of equations as compared to one bin. The execution speed of spheap is the slowest but has higher capacity to solve large number of equations as compared to standard implementation. The space efficiency of standard implementation is also better than spheap and oneBin. This is because standard implementation provides more number of bin choices as compared to 56 in spheap and 1 in oneBin method.

5 Conclusion

There exists a clear trade-off between execution speed and space efficiency between oneBin and SPheap allocation techniques. OneBin technique provides faster execution but with poor memory allocation management and vice versa for the SPheap.

6 Appendix

6.1 Uniform Integer Distribution Generation

The `rand()` function in `stdlib.h` generates random non-negative integers uniformly between 0 and constant `RAND_MAX`. To generate random numbers uniformly in the interval $[a, b]$, the following is employed.

$$\texttt{X} = \frac{\texttt{rand()}}{\texttt{RAND\_MAX} + 1} * (b - a) + a$$

6.2 Exponential Integer Distribution Generation

To generate random integers between $[a, b]$ distributed exponentially with mean λ, the following is employed.

$$\texttt{X} = \texttt{max}(\texttt{min}(-log(1 - \frac{\texttt{rand()}}{\texttt{RAND\_MAX} + 1}) * \lambda, b), a) \; ; \; \lambda \in [a, b]$$