ECE 650 project 1 report

Can Pei (cp357)

Implementation details

In my implementation of allocation policies, the allocated space is divided into segments. Each segment contains two parts: the head of metadata and the storage space. The storage space is used to store the actual data, and the metadata is used to record information on the segment's relative location, size, and availability, which is a struct as follows:

```
typedef struct _Metadata {
  size_t size;
  int used;
  struct _Metadata* freeprev;
  struct _Metadata* freenext;
  struct _Metadata* allprev;
  struct _Metadata* allnext;

} meta;
```

In the struct, the "used" field is used to indicate the segment's availability, which is 0 if the space is free for use and 1 if the space is already used. The 4 pointers are used to show the segment's relative location. I use 2 linked lists to locate a segment. One is free-list, which contains only free segments, and the other is all-list, which contains all created segments. The meta-list struct is as follows:

```
typedef struct _Mlist {
  meta* head;
  meta* tail;
  int type;//0: free, 1: all
} Mlist;
```

A Mlist sturct contains 2 pointers pointing at the head and tail of the list, and "type" field to show whether it is free-list or all-list.

When an arbitrary size of memory is requested, the free-list is searched through to look for any suitable free segments. If none is available, a new segment is created using sbrk() and appended to the tail of all-list. Otherwise, depending on if the chosen segment has enough space, the entire segment or a portion of the segment will be allocated. If the sbrk() is unsuccessful due to the full occupancy of the heap space, a null pointer will be returned.

A segment is defined as "splittable" if it is big enough for the requested data, in addition to an extra metadata block. If the segment is not splittable, the entire segment is allocated for the request. Otherwise, the segment will be split into two, of which the first portion is allocated for request, and the second becomes a new free segment, being appended to the tail of free-list and inserted as the next segment of the original segment in all-list. When a

segment is freed and appended to the free-list, it coalesces with its neighboring blocks if they are physically adjacent. This would defragment the freed memory space for more optimized memory allocation performed later.

Performance

The library is implemented in C, and executed on Ubuntu system in VMware with 4GB of RAM. Three sets of experiments are performed to measure the outcome: equal size alloc, small random size alloc and large random size alloc. The execution times and fragmentation proportions are shown in table below.

	First fit		Best fit	
	Speed(s)	Fragmentation	Speed(s)	Fragmentation
Small rand size	5.274	0.049	3.535	0.024
Large rand size	25.052	0.118	65.362	0.042
Equal size	2.558	0.450	2.676	0.450

Speed

For equal size allocations, both policies show similar speeds. First-fit policy relies on a greedy approach to search for the first available free segment, thus promising a fast execution. While best-fit policy checks for equality between the requested size and the freed segment and allocates immediately if the condition is fulfilled. This allows the library to break out from exhaustive searching, further optimizing its execution.

For small-size random allocations, best-fit policy executes moderately faster than first-fit. In this case, the first-fit policy would split the closest available segment, while the best-fit policy traverses through the free list. If the free list is relatively short, although traversing is of linear complexity, it still has a better performance than the greedy approach, which would spend considerable time splitting and rearranging segments each time upon a request.

Both policies show low speed for large-size random allocations. First-fit policy performs better, as it does not need to travel the entire free list, as opposed to the exhaustive searching strategy, when the free list too exceedingly varying to search for the best fit.

Fragmentation

Both policies perform the same under equal size allocation, as the allocation requests are all identical, and the segment choices would also be identical disregarding different approaches.

For both random size allocations, best-fit policy does better than first-fit policy, as the former has a more optimized segment selection strategy, reducing the need to split a bigger segment and leaving the rest of it too small to be requested. The effect is especially

conspicuous for large random size allocation, where the latter only results in a third of the fragmentation of its former's.

Discussion and Conclusion

Through the test and experiment, we found that there is a tradeoff between the speed of allocation and the integrity of the memory space. First-fit policy is suitable for a series of more varying and unpredictable memory requests, or when speed is more important than memory integrity. By contrast, the best-fit policy is better when the requested sizes are moderately consistent, or when the memory should be preserved as cleanly as possible. However, both allocation policies are too slow to be considered for practical usage.