Report

Yidan Sun NetID:ys303

Data Structure

Implemented a linked list to store free space node.

A piece of free space contains metadata and the space for actual data space.

A Node points to the start of metadata.

Node contains

- `Node * next`: A pointer to next node
- `size_t datasize`: indicate how big the space is.

Malloc

When malloc, search the linked list at first. If could find a Node in linked list whose datasize is bigger than requested size, we could allocate space out from free space and edit the linked list , otherwise, we need to call `sbrk()` to ask for more space.

First Fit

Stop searching as soon as we find a Node whose datasize is bigger than size.

Best Fit

Search the whole linked list, choose the Node whose datasize is the smallest among Nodes whose datasize is bigger than size (if exist).

Trick

From piazza, ta suggested when best fit, as soon as we search a node whose datasize equals size, we could use this node because this must be the smallest node. By using this trick, we may not have to search through the linked list all the time.

Edit Linked List

If we find a Node whose datasize is bigger than requested size, we need to edit the linked list to show the change. There are two situations here:

- `Node->datasize<=sizeof(Node)+size`

When datasize is smaller than the size of metadata and requested size, we need to delete the Node, because after allocating the space out, the left space will not be enough to store metadata, no need to say actual data.

- `Node->datasize>sizeof(Node)+size`

When datasize is larger than the size of metadata and requested size, we need to spilt this Node because after allocating out the space, the left space will still be large enough to store metadata and actual data.

To split the Node, we need to move `Node*` front for sizeof metadata and size. And change datasize to `datasize-sizeof(Node)-size`.

Call Sbrk

If we need to call sbrk, the argument passed for should be the actual size plus size of metadata.

Return

No matter we allocate space from free space or by calling sbrk, the pointer we got is pointing to the start of metadata. We need to add `sizeof(Node)` to find the actual start place for our data to use.

Free

To free a piece of space, we need to first find the start of metadata. And then find a position in linked list, where the start of metadata is larger than the end of its front node space and the end of this space is smaller than the start of next Node. Then insert this Node into this place.

And we need to compare whether this Node is adjacent to its neighbors. If yes, we need to merge them.

There is no different between first fit free and best fit free.

Study of Performance Policy

Implement of `get_data_segment_size()`

Set a global data called `data_segment_size` and set it to 0 at beginning. Add sizeof(Node) plus size to it when calling `sbrk()`. Then return `data_segment_size` for `get_data_segment_size().`

Implement of `get_data_segment_free_space_size()`

Set a global data called `data_segment_free_size` and set it to 0 at beginning.

When malloc space from free space, minus `sizeof(Node)+Node->datasize` from it if the whole Node is deleted. Or minus `sizeof(Node)+size` from it if we need to split the Node.

And when free space, add `sizeof(Node)+Node->datasize` to it.

Return `data_segment_free_size` for `get_data_segment_size()`.

Analysis of Performance

Results

#NUM_ITERs 100 for equal and small, 50 for large

#NUM_ITEMs 10000 for all sizes

	First Fit(Time/Fragmentation)	Best Fit(Time/Fragmentation)
Equal_size	0.38s / 0.45	0.37s / 0.45
Small_size	20s / 0.074	9s / 0.027
Large_size	140s / 0.093	161s / 0.041

Equal size

There should be no difference between first fit and best fit for equal_size. Since all bunks are in equal size, the best fit should also be the first fit. The fact of equal size bunks also save a lot of time for malloc. Because all free spaces are in same size, the first node in the linked list will be enough for the requested space, we do not need to search through the whole linked list, therefore, the time for equal_size is quite short.

Fragmentation is better for best fit

Since `Fragmentation=free space/total space`, the smaller the fragmentation, the better use of space. Best fit always searches the minimum space that could fit requested size, so that larger spaces are saved for other data to use. However, first fit always give the first fit space out, which tends to split large space into small pieces and these may not be taken used later and cause a waste of space.

Large_size takes more time

Large_size takes more time than small_size because the size ranges from 32 bytes to 64k bytes, it is hard t o find a node in the linked list for a particular size, and we need to search for more time.

While for small_size, the size just ranges from 128 bytes to 512 bytes, and increment by 32 bytes. There are only 13 possibilities for the size of data, so it will be much easier to search for a proper node. Therefore, it takes much shorter time.

Time difference between first fit and best fit

It is easy to assume that best fit will take longer time than first fit because best fit usually search the whole linked list. However since best fit takes good use of space, it may make it easier for other size to search and save time in total. In contrast, even though first fit save time at a single search, it will split the space and in the end makes no space could be used in the linked list and then every size have to search the whole linked list and then ask for new space by calling sbrk.

The good fearture shows better at small number of different bunk size because it is easy to find a node whose datasize is equal to size. And it could explain why best fit takes shorter time at small_size, but longer at large_size.