Virtual Memory Manager Design and Specs

Done by -

Pritkumar Godhani 19CS10048

Debanjan Saha 19CS30014

Structure of Segment Table:

Maintaining bookkeeping structures for a single variable would mean using more 32-64 bytes for an allocation of 4 bytes which is not very logical or feasible. So, whenever a request for a single variable arrives we allocate a segment with space for 32 variables and use a bitmap to map the location. Then the request for the next 31 variables will be allocated from this segment. After which another new segment could be requested. This may lead to allocating some space for future use but avoid unnecessary and impractical bookkeeping overheads.

We have designed a segment table instead of a page-table to support the following. Whenever a user creates a variable, we either put that variable in a segment which is a container of 32 same type variables, or if we can not find such a segment, we just create a new one considering the memory constraints. For arrays, we don't follow the above, instead we just create one segment for one array with the requested size. This table contains the following fields.

- **prev**: pointer to a previous segment
- **next**: pointer to a next segment
- **baseptr**: base pointer of this segment in the memory
- **size**: size allocated to this segment (32 for variables, array_size for arrays)
- **bitmap:** a 32 bit integer to keep track of 32 variables whether it is occupied in the segment or not.
- **seg_num**: A unique token for extracting the segment when some operation is needed to be done on a variable or array. (For this we keep a segment counter)
- **info:** an 8 bit integer to keep track of the information about this segment. Visualisation is shown below.

```
info : 00000000
                  ^last bit of info denotes if this is an array segment or a variable segment
                 ^reserved for mark and sweep
                ^shows if the segment is present in gc_stack
```

^^the two bits before that specify the var_type of this array or variable segment (INT/BOOL/CHAR/MED_INT)

Additional Data Structures:

- **hole :**

fields:

- **prev**: pointer to previous hole entry
- **next**: pointer to next hole entry
- **baseptr** : base pointer of this hole in the allocated big chunk of memory
- **size**: size of the hole in bytes

Use:

Used to store information about empty spaces/holes in memory.

- **free_list :**

fields:

- **head**: pointer to the starting hole in a list of holes
- **tail**: pointer to the last hole in a list of hole
- **size:** number of holes in the list

use:

Used for managing the list of holes in a linked list which remains sorted by the base pointer of each hole without additional sorting overhead.

- **segment_list :**

fields:

- **var_head**: Pointer to the starting segment in list of segments containing variables
- **var_tail**: Pointer to the last segment in list of segments containing variables
- **arr_head**: Pointer to the starting segment in list of segments containing arrays
- **arr_tail**: Pointer to the last segment in list of segments containing arrays

use:

Used for storing the list of segments

- **books :**

fields:

- **free_list**: Pointer to the structure containing the linked list of holes
- **segment_list**: pointer to the structure containing the linked list of segments
- **seg_counter**: An incremental counter/token for extracting segments from the created variables/arrays

use:
Used for bookkeeping and storing the pointers to the list of holes and segments and maintaining the token system.

- **mmu :**

fields:
- **baseptr**: stores the base pointer of the memory assigned to the user.
- **vmm**: pointer to an instance of book structure for managing memory

use:
Used to manage various operations in the memory space allocated to user

- **gc_stack_node :**

fields:
- **seg**: pointer to a segment
- **next:** pointer to next gc_stack_node in the stack

use:
Management of the variables and arrays used in the mark and sweep algorithm.

- **gc_stack :**

fields:
- **top**: Pointer to the top segment in the stack
- **rbp:** base frame pointer to a temporary stack node

use:
Stores the pointer to segments used by various functions as they are called in a stack.

Impact of Garbage Collector

- We have generated the memory footprint in the log files for demo1.c which will be regenerated every time a user program is run.
- We are running our garbage collector every 250 ms

Compact Function Algorithm:

The Compaction Algorithm we have used is inspired from the LISP2 Mark and Compact. The Algorithm makes three passes of O(n). In the first phase, it computes the new locations of the allocated blocks. In the next phase, it copies the block to their new locations and in the final phase it updates the free_list to denote all holes compacted together.
Compaction is done when the number of holes exceeds a given amount. For instance 1000, in practice this may be set after rigorous,

Justification about the thread lock:

Yes, we are using locks in our library. Our library implementation has two key parts - list of segments(allocated blocks) and list of holes (free space). Now, whenever a user is declaring an array or some variable, a new segment will get created at some point. Now, at that point, if the compaction algorithm runs and we need to have some synchronisation so that no data race occurs due to simultaneous segment creation and deletion.