**Segmented Paging FTW!**

by Leo Scarano, Hanson Tran and Vincent Taylor

**Phases**

The following implementation has been created in 4 phases:

* Phase A: Direct-Mapped Memory
  + This part of the project deals with mapping memory request using malloc to a global char array of 8Mb. This effectively acts as our own memory, so we first must align the memory and decide how we are going to store the metadata and handle multiple concurrent request.
* Phase B: Virtual Memory
  + In this stage we must split up the data into pages this way we can mprotect the data. We must decide on a page layout for our memory and how we are going to organize what thread owns what page. Afterwards, we use the malloc in part one to allocate within each page. Additionally, after protecting our memory we must create a signal handler that allows us to load the current thread’s pages in when there is a call to its memory i.e. a Page fault.
* Phase C: Swap File
  + This stage requires us to introduce a swap file of 16mb which we can use as secondary storage to store more pages. This requires us to create a file and have an eviction policy. Furthermore, we must include a way to read, and write to the swap file. After all of this we must be able to find free pages in the swap file and move them into memory giving threads more memory if needed.
* Phase D: Shared Region
  + Finally, this part of the projects tasks us with creating a shared region of memory for all the threads used by the shalloc command. This region acts the same as all the other memory only it is not protected or owned by a single thread.

**Structure**

**Block / metadata**

Size - Size of allocated request

isFree - Determines whether the metadata is free

nextPtr – Pointer to the next block if the data spans multiple blocks

TID – TID of the data owner (Primarily Used in the Swap File)

**Page**

TID - TID of the thread that owns the page

isFree – Determines whether the page is free or not

page - Pointer to the actual page

page\_num - Number of the page (identifier)

next - Pointer to the next page

**Page Table membook**

The page table data structure for going though pages.

**Memory Layout**

[\_Page Table 15 pages | temp region 1 page| Thread Page Counter 1 page |Shared Region 111 page| memory 1920 pages]

In depth look/description

We purposely segment our memory into two chunks of 960 pages this allows us to swap more easily with the swap file which will also be in chunks of 960 pages. There is another rule we added to this, it is that no thread may possess more than 480 pages the reason for this will be explained later in our swap algorithm.

**Swapfile Layout**

[\_swap book 256 pages\_| Swapfile contents 3840 pages] The large gap of swap book pages is due to the segmentation implementation we wanted to implement. Since space is bountiful we decided this was better than a longer runtime for swapping

**Eviction Algorithm**

Swapping Algorithm Structure

All our memory obeys these rules

1. If one of a thread’s pages is in memory, then all of its pages are.
2. If one of a thread’s pages is in the Swapfile then all of its pages are.
3. At the time of swap all a threads pages will be in either the left chunk or the right chunk. (No thread can have pages in each segment of the memory)
4. No Thread may own more than (1/4 of the current pages in memory in our case that is 480)
5. The Swap file can only move segmented chunks between memory and itself

Sample Memory

{[69|69|12|2|12|] [1|4|4|5|1]}

Sample Swap File

{[9|6|6|7|9] [10|0|3|3|3] [11|11|11|11|11] [16|0|14|14|16]}

When we want to shift out a segment of our memory, so we must obey rule 3

We swap the right chunk of memory with the 2nd chunk of the Swapfile

{[69|69|12|2|12|] [10|0|3|3|3]}

{[9|6|6|7|9] [1|4|4|5|1] [11|11|11|11|11] [16|0|14|14|16]}

Now we can call align pages to place the current threads pages into the front (we will use 3 as the TID, 0 is a free block)

memAllign

{[3|3|3|2|12|] [10|0|69|69|12]}

Now if we want to swap we call our re-organize function and make which effectively memAllign the blocks to the left half of the memory

1. reorganize (3)

{[3|3|3|2|12|] [10|0|69|69|12]}

2) reorganize (2)

{[3|3|3|2|12|] [10|0|69|69|12]}

3) reorganize (12)

{[3|3|3|2|12|] [12|0|69|69|10]}

However, we are in between two chunks, so we fail to copy and revert back

{[3|3|3|2|12|] [10|0|69|69|12]}

4) reorganize (10)

{[3|3|3|2|10|] [12|0|69|69|12]}

Now since we have reached the end of the left chunk we can safely swap either or both chunks without worry.

**Important Functions**

**void\* myallocate (int size, char FILE [], int LINE)**

This function acted as our allocation function as well as our initial setup function. We have a global variable called first\_time that allows us to set up our signal handler, initialize our page table, create/clear the Swapfile, and align our memory. Afterwards, we proceed to our findSpace function which returns a page that is now owned by that thread. This page pointer is then casted to a block where we can now read metadata. We then move through the page until we find a free space place metadata and return the pointer to the start of the data.

**void\* mydeallocate ()**

This function is used to free the pointer. We go through the page table and check the pages and compare the two pages pointers by subtraction if the current pointer is greater than the pointer given then we search through the given page to find a pointer that matches the given pointer. Once we find a match we memset the memory to null terminators and set the metadata as free. Unfortunately, we do not have a collocating option but if we did we would use a buddy allocation system and check the next pointer location to see if it is also free and memset the entire region. Without this option we are prone to more internal fragmentation; however, with limited time it is a sacrifice that will only become a problem once several dozen threads are in use.

**void\* findSpace (size\_t size)**

As mentioned previous this function find space within the page and returns a pointer to that space within the block

**int memAllignPages ()**

This is the most crucial function by far within the program. It begins by allocating --

**void fileCreation ()**

creates a swap file and lseeks 16mb in, and writes a null terminator, effectively creating a 16MB Swapfile for use. Upon creation our Swapfile does not require any sorting or setup. The Swapfile will be inherently sorted by the reorganize function.

**int unProtectMem ()**

Unprotects all of memory so that read or writes can occur or allocating/freeing memory works.

**int protectMem ()**

Gets the current thread and locks any/ all pages that belong to it. It searches from the beginning of the page table (which is mem-aligned) and increments a counter to see the page span in memory this effectively tells us the width of a threads memory, so we can lock it. When the width of memory is calculated it, is measured in page sizes so mprotect works well with limited calculation needed.

**void swap\_in\_memory (void \* addr1, int seekBytes)**

This function takes in the address of the memory to swap and the number of bytes into the file in which we must swap. Then using the temp block we will swap the data in memory with the data in the swap file one block at a time. Even with the segmentation each thread will have the ability to access ~2mb of data at max.

**void moveCurABS (int Fd, int dist)**

Move the file pointer an absolute distance from the beginning to the distance specified. Good for progressing quickly through an array without calling multiple functions.

**void moveCur (int Fd, int dist)**

Move the file pointer in relation to where it currently is. The file descriptor can move left (negative), or right (negative) to reset the amount, read in the readPage and the readAmount function.

**void resetFilePointer (int Fd)**

This resets the file pointer back to the offset. This allows us to skip the beginning part of memory which is a victim of segmentation and lack of time. Since we do not want to keep in mind every time we reset the beginning segment we have this function

**void resetABS (int Fd)**

This function resets back to zero just incase there is any reason to implement it in the future other group members have the function abstracted.

**void movePage (int Fd)**

This moves the file descriptor exactly one page length this way we can go through the swap file in aligned segments.

**void readPage (int Fd, void \* address)**

This reads a single page worth of information to an address given. Since the file pointer is moved when reading it also resets the file pointer to the beginning of the page that way our page jump work.

**void readAmount (int Fd, void \* address, int amount)**

This is like read page, but this allows us to read as many or as few bytes as we want. This way we can read each page’s metadata for information. Additionally, this will reset the function so that again move page works accordingly.

**void writePage (int Fd, void\* address)**

This writes whatever is at the address given for an entire page length (useful for swapping or sending things to memory.

**void swap (int Fd, void \* addrRec, void\* addrTemp, int swapFDist)**

This swaps the segments according to the algorithm presented above. The first argument is the file and the next is the address where we are starting to move the information too. The second one is the address to the temporary page and the last one is the distance in pages that we want to start swapping into.

**int findFreeSwap (int Fd, void\* addrTemp)**

Searches the file to find a free block and calculate which chunk the free page falls in so we can swap accordingly.

**int findPageSwap (int Fd, void\* addrTemp, int TID)**

Searches the swap file for the specific chunk that the threads page is in. Afterwards, it will swap accordingly.