Anne Whitman : alh220
Michael Mazzola : mjm706
(We had a 3rd group partner who may be listed as our partner somewhere on paper, but did not contribute at all to this project and should not receive any credit.)

Quick Important Notes:

Memory Size: 8 Megabytes
Operating System Region Size: 1 Megabyte
Shared Region Size: 4 Pages
User Region Size: 7 Megabytes – 4 Pages
Size of Swapfile: 16MB
THREADEQ is set to 1
LIBREQ is set to 0

Changes/reminders from last project

Max Threads: 32 total, 31 can be created (first one reserved for main method)
Stack Size: 16KB

MetaData Structure:

For OS region and shared region: A struct containing an int to check if used, an int to record size, and a pointer to the next metadata.
For user region: A struct containing a boolean to check if used, an int to record size, an int to record number of pages, and a pointer to the next metadata.

Page Tables:

Internal: each thread has a page table associated with their TCB which holds the page index, the maxsize currently free in that page, an offset in the swap file that defaults to -1 when not being used, and a pointer to the next
External: a 2-dimensional global array, each row holds data about a specific page, the first column holds which page owns it (defaults to -1, when freed set to -2 to guarantee segfault handler), second column holds how many malloc calls are currently using it (so it won't be deleted prematurely).

Basic Logic/Design of user section:

-First our program checks to see how many pages the allocation needs.
-If it needs more than one page, it allocates fresh pages to the thread.
-If it needs less than one page, it first checks to see if there is room in an existing page.
-If there is, it squeezes it into an existing page.
-If there isn't, it creates a new one.
-Once the new space is allocated, we write to the internal page table and external page table

-if it was written to a page that already held something, we increment the counter in external as well so when it's time to free we know that more than one thing is using that page.

Analysis / Things We Tested:

From Main Method (no pthread_create):

1.)Mallocing and freeing 5mb 10, 30, 50 times in a loop

-Outcome: once we start approaching 50 times in a loop, we start getting a stack overflow issue in the OS region.

2.)Mallocing 2MB 10 times and accessing the data in the first call after the last.

-Outcome: first 3 malloc calls are successful, last 7 are not (which is correct, a thread cannot take up more than the entire userspace according to spec). No issue using first malloced space.

3.)One malloc call for 6mb followed by a free and then 1000 malloc calls for 6000 bytes

-Outcome: successfully works for first 894 calls, but then memory fills up and the rest are appropriately returned null pointers.

5.)One malloc call for 6000 bytes followed by a free and then 6 malloc calls for 1000 bytes

-Outcome: successfully frees and then fits the next 4 calls for 1000 bytes into the first page, then moves on to next page for next two calls.

6.)Allocate (pagesize - sizeof(memStruct) - (5 + sizeof(memStruct)), then allocate 5 bytes twice.

-Outcome: First 5 byte call properly consumed the rest of first page, second one properly went into new page.

7.)Allocate (pagesize - sizeof(memStruct) - (5 + sizeof(memStruct)), then allocate 1 byte twice.

-Outcome: First 1 byte call properly consumed the rest of first page, second one properly went into new page.

Single Thread (one call to pthread_create):

1.) Run all of the tests from above (main method).

-Outcome: exact same results.

Multiple Threads:

1.) Allocating 5MB in 6 different threads

-Outcome: As long as you join each thread immediately after creation, runs perfectly. (Runs into some hiccups when threads are not joined immediately after create.

2.) Thread 1 malloc 5MB, thread 2 malloc 5MB, thread 1 frees, make sure thread 2 enters seg fault handler.

-Outcome: works, properly segfaults and accesses correct page

3.) 31 separate thread creates, each calling malloc of 5MB (more than half of the user space) and freeing 200 times.

-Outcome:

4.)31 threads, each calling malloc 200 times on 1 byte, do not free.

-Outcome:

5.)31 threads, each calling malloc 200 times on 1 byte, do free.

-Outcome:

Mallocing from both main method and multiple threads:

1.)Combined results from above sections to do both main and regular threads together

-Outcome: all seemed to work the same as methods above.

Shalloc:

1.)Because this works like a regular malloc (without paging), we just ran multiple cases where we shalloced from one thread and then accessed from other threads.

-Outcome: Successfully accessed between multiple threads.

Methods Definitions, Descriptions, and Return Values:

int findConsecutivePages(int numPages);

This method is used to traverse the page table and search for a number of consecutive pages that are available so they can be allocated to the current thread.

If a group of pages is found successfully, it will return the index of the pages. If the search is unsuccessful it will return -1.

int findSwapIndex();

This method will search for the first index available in the swap file. If an index is found it will be returned.

If an index isn't found, meaning there were no indices available, the method returns -1.

int findEvictIndex(int numPages);

This method will search the page table for an index that is eligible be evicted.

It is given an integer value 'numPages' which is used as a number of consecutive pages that are to be searched for and evicted if found.

If it finds the necessary amount of available consecutive pages it returns the index of the first, otherwise, -1 is returned.

int evictPage(int page);

This method is given an index in the page table that needs to be evicted to allow for its space to be used by a different malloc call.

It first searches the swap file for an available index using the findSwapIndex function, and, if an available index is located, copies the pages data to the swap file.

If an index was successfully located by the findSwapIndex function this method will return the found index, otherwise it will return -1.

int restorePage(int page, int offset);

This method is given a page and an offset. It first takes the given page and attempts to evict it using the evictPage function.

If the swap file is full and the page could not be evicted, the method will return -1.

If a page is found, the method then copies the data from the swap file into the previously evicted page in the page table and returns the offset of the evicted page so we can store it in their page table.

int evictPageIntoBuffer(int page, int offset);

This method will be called when the swap file is full, but we want to restore something from it. There is no space in file to evict, so we temporarily evict into a buffer, restore the page, then we can put the page into the file.

Otherwise works the same as evictPage and restorePage combined.

void segment_fault_handler(int signum, siginfo_t *si, void* unused);

This method is automatically called when a segmentation fault occurs.

It determines if it was called due to a legitimate segmentation fault or if there needs to be a page swap.

If the segmentation fault was legitimate the program will print "real [external/internal] segmentation fault" and exit, otherwise, it will use the 'restorePages' method to retrieve the appropriate page from the swap file and continue execution.

void mallocInit();

This method is called the first time that malloc is called.

It first uses posix_memalign to align the memory array, then builds the swap file of size 16MB and sets all the bytes to zero.

Afterwards, it builds the struct for the segmentation fault handler. Lastly, it sets all the indices in the page table to -1.

spaceBetween(metadata* curr, int size, int bytesFree);

This method is used after an allocation request was granted by using pages that were previously freed.

If there are extra bytes leftover after the request is satisfied, a new metadata structure is appended to the end of the bytes for future reference.

int combineFreeStuff(metaData* iter);

This method is given a metadata structure and will check to see if the metadata following it in the memory buffer is free.

It first stores the given metadata bytes in a variable labeled 'bytesFree' and will continuously add the bytes of the following metadata's until it reaches one that is currently being used or is NULL. Once the search is complete, the method returns 'bytesFree'.

void* myallocate(int size, char* file, int line, int threadId);

This method is used for memory allocation and replaces the malloc call.

A memory region is sectioned into three partitions, the operating system region, the user space region, and the shared memory region.

This method has access to both the operating system and user space section. It is given a requested size, a file and line for reference of where the call originated, and a value of 0 or 1 to determine if the call is pertaining to an operating system or a user process.

If the value of the threadId is 0, the method will handle the call in the operating system region and will return a pointer to chunk of bytes the size that was requested. The metadata for this chunk will be stored directly before the address that is being returned.

If the value of the threadId is 1, the method will handle the request by distributing bytes from the user space.

This region implements paging. When a request is made, the number of pages necessary is determined from the size given and an available location is searched for.

If the space is found it will return a pointer to this location with the metadata stored before it. If a page is not found, a page will be selected for eviction and its data stored in a swap file. A page table is used to keep track of all the pages and their current status as well as assisting in other functions such as determining which pages are eligible to be evicted. If both the memory region and swap file are filled and there is no more allocation room, the method will return a NULL pointer.

void mydeallocate(void* ptr, char* file, int line, int threadid);

This method replaces the “free” call and is used to deallocate memory in the memory region.

It first checks if the value given is NULL, and if so it will return.

It then checks the address of the given pointer to see which region in memory is being referenced.

If the pointer belongs to the operating system region or shared malloc region, it sets its “in use” value to zero and allows for future malloc calls to this region to utilize this space.

If the address belongs to the user space region, it checks if the last page has more than one allocation (by checking the counter in the page table), and if so it will decrement its count and set any other pages count to zero.

If there is only one allocation it will reset the count of all the pages to zero. It also sets the thread ID in the page table to -2 for memory protecting purposes.

It will then set the metadata that is associated with the pointer to no longer in use which will allow for future malloc calls to this region to have access to these bytes.

If the address is pointing somewhere in the memory region but there is no metadata associated with it, the free call will fail, as it is an invalid address.

void* shalloc(int size);

This method is used to distribute memory in the shared memory region.

This region does not use paging and can be accessed by any thread.

It will return a pointer to the beginning of a chunk of data which size is equal to the size requested.

The metadata for this is stored directly in front of the address of the pointer. If there isn’t enough space to satisfy the request, a NULL pointer will be returned.

(Other methods from our pthread library were altered in some places, but only to allow the library to fully function, so we did not include those methods in this README._