Operating Systems Assignment 2

Fahad Syed, Travis Thiel, Austin Tsai

DESIGN:

1. Our design followed the project specifications. We wrote macro definitions to replace the traditional malloc and free calls. In order to implement our myallocate function, we first had a conditional to check whether myallocate was ever called in the form of our user_first_free variable. If it was 0, it is the first time in the program that we are calling it and we would initialize our metadata in our initmem function. The myallocate function first determines whether the request is coming from our library or a transformed user thread. Then we get the page table and find the first free page. If the page is free or has not been claimed by other threads, then we call a seg_alloc_first function to create metadata for the block and another metadata for the rest of the region in the block.
2. In addition to our myallocate function, we also wrote the mydeallocate function to function as a free. In order to function as a free, we check to see whether the thread owns certain pages inside the page table. In order to implement contiguous virtual pages, we check the next page in my_memory only if the thread owns that page. If it doesn't, then we swap into page to create an illusion of contiguous memory.
3. Our pageswap function contains the details for mprotecting pages. For both mprotecting and unprotecting pages, we use 2 helper methods called protectthreadpages and unprotectthreadpages. When we encounter the page that needs to be swapped out, we traverse the page table by calling on the unprotectthreadpages method, which removes mprotection from any pages belonging to a specified thread. If there was no page to swap in, then we return 0. Otherwise, we return 1.
4. In our initmem function, we initialize our 8 MB char array by allocating it through memalign. We also initialize our page table within my_memory and create a table_ptr to it. We also have a check to determine whether the file we created is the right size. The initmem function also is where we create our swapfile and set up a separate page table for our swap file.
5. The swap file initial setup works the same way as our memory. We set it up with a page table and ensure that it is 16 MB large by defining swap size as 16*1024*1024. The swapfile name is swapFile.bin. In the myallocate function, we determine whether the page table is full or not. If it is, we evict the pages out into the swap file in a FIFO manner. According to the project specifications, we will return NULL in the event that the swap file and memory are both full.
6. For the shalloc function, we had it function similarly to how we created our myallocate. Since we needed to specify the last 4 pages of our 8 MB memory to function as the shared region, we specified it in the shalloc helper function. Similar to how our myallocate works, we also call upon 2 helper functions, shared_seg_alloc_first and shared_seg_alloc. A major difference is that the shared memory is not swappable. It does not get put into the swap table since all the threads have access to the same space. When it comes to freeing shared memory pages, we also call on the same free function for regular memory, mydeallocate function. However, the difference is that we check to see if the memory used is in the shared region of our 8MB memory. If it is, we call a different function, shared_free, to properly free the memory used in the shared region.

DEBUG:

1. When testing our design, we used a pthread_test.c file. We made multiple changes to the file depending on what portion of the project that we were on and actively testing. We also utilized 2 primary debugging methods. We used the printmem and printpage functions. The purpose of the first function was to print out what pages in our page table were allocated and which threads owned them. This would help us verify that our myallocate and mydeallocate functions were working. The second function prints out the segment metadata for a given page. We would be able to tell if the pages were out of order, if they had been swapped out, errors in the size, etc. In addition to these methods, we utilized print statements to test and create benchmarks to help better debug our code due to the difficulties of using gdb in multi-threaded programs. Another problem we encountered was the size of the stack and how it affected the rest of our code. When it came to writing the swap file, we encountered multiple issues. We found that our old stack was too small to contain our swapTable variable, which takes up 64k bytes. In order to compensate, we had to enlarge our stack. After enlarging it, we continuously changed the size of our stack to get it as small as possible while also trying to keep it large enough to support many threads.

DETAILS:

2. The OS region in our program encompasses pages 9-1009. We allocated 1000 pages to our OS region. On the other hand, threads get pages 1010 – 2043. Finally, the last 4 pages, 2044-2047, is allocated for the shalloc function and is designed for the shared memory between all the threads. The size of the stack is 95,000 bytes. It is able to support up to 42 threads. This is due to the size of our swapTable, which takes up 64k bytes by itself. In order to account for this, we had to greatly increase our stack size.