Joseph Jaeschke (jjj93), Adeeb Kebir (ask171), Crystal Calandra (crystaca)
Tested on the cray1 iLab machine
Asst2: Segmented_Paging_FTW
CS 416

This project was an extension of the "Adventures in Scheduling" project. Because of this we believe the use and functionality of the thread library should not be changed. Also there were no explicit instructions regarding the general structure of the final library. All of our memory manager code is within the my_pthread.c file in between two block comments identifying them as the new functions. Also, additions were made to the my_pthread_t.h file to include macros, struct definitions, and function headers needed for this project. The makefile from the last project will compile this project and no additional steps or changes are needed to use the compiled library.

For project 2, we implemented a memory manager to handle memory requests from our thread library from the first project and for programs running the thread library. This project was implemented in four phases to ease the transition into a full memory manager. The first phase was a simple allocate and deallocate program that returned pointers to memory chunks using a best-fit algorithm. This simple phase allowed for a simpler testing of different algorithms to user when trying to satisfy a request for memory. In addition, it helped determine the kind of meta-data needed to identify memory and locations within the memory array. The second phase added the idea of multiple threads accessing memory. Due to this, a virtual addressing of memory was needed and therefore more meta-data was used to keep track of what thread owns what. Another aspect that was needed was a way to catch a thread trying to access memory and make sure that the memory a thread thought was there was the memory that really was there. In order to achieve this, mprotect was used to protect memory and any segmentation fault that was generated was caught by a signal handler which would place the right memory where it was needed. However mprotect can only protect memory on the granularity of a system page. For this reason, the master memory array was aligned to system pages with the memalign call which gave a pointer to an 8MB array that was aligned with system pages. From this point threads can all have their first memory request start at the same point in the array without interference. The only caveat to this is that a thread will get at least one page even if the request is much smaller. For the third phase, a 16MB swap region was added. Since many threads may want a lot of memory, eventually the 8MB memory array will get filled. If upon a request for memory there is not enough, the swap file will be searched for any free pages. If there are enough in the swap file, those will be marked as owned by the thread and can be copied into the array at their corresponding page. This means that what was at that spot needs to be moved. A very naive eviction algorithm was used which directly swaps out what is in the spot of the memory incoming from swap. Lastly, a shared region of memory was needed. A call to shalloc will return a pointer to a shared region of the master memory array that does not get mprotected. This allows any thread to read or write to this region. This is necessary as a parent thread and a child thread have different address spaces. This means the pointer allocated in the parent thread and passed to the pthread_create function cannot be used in the child thread. If that pointer was allocated with shalloc, both threads can read and write to the data.

All calls to malloc and free are replaced by the calls to our allocate and our deallocate using appropriate macros. Since the thread library knows about the existence of our memory manager, it can make memory requests by calling the functions explicitly. This allows it to provide its own parameters to the functions which identify it as a request coming from the thread library. An explicit call to the allocate and deallocate functions from the library give a fourth parameter as zero. This is different

from the fourth parameter in the macro which is one. This allows requests from the thread library and user code to be distinguished and treated differently.

Meta-data:

There are two main types of meta-data used in the allocator. The first kind is a memory header placed right before a chunk of memory. These have fields that keep track of whether the following chunk is free, a pointer to the next chunk, a pointer to the previous chunk, and a verification number. The first three fields are self-explanatory while the verification may not be. The number used in the verification is a well known number that the deallocator can check to make sure it was given a proper pointer. If the deallocator sees the verification field is not the well known constant after getting to the corresponding memory header, it will reject the request to free the memory chunk.

The other kind of meta-data used was to keep track of the pages. The memory manager needs to know if a page is used or not, what thread owns a page, and what number page number it is for the thread. The combination of this data allows for the signal handler to know that page to look for and where to put the page when it is found. This meta-data structure is used for both the memory array and for the swap file.

Allocation:

There are two types of allocate calls. The one type is if a thread requests memory. If this is the case, then the allocator will check if a thread has any pages yet. If this is the first memory request from this thread, the allocator will determine how many pages the thread will need to satisfy the request. Once that is known, the allocator will look through the meta-data for each page and for the swap file. If there is not enough available pages, the request will be rejected and a NULL pointer will be returned. Otherwise, two memory headers will be made to copy into memory. These two headers break the requested pages into a used half where the request will go and an empty half which can be used to satisfy requests latter from that thread, since this is the first time, the one header will be placed at the start of usr region and the other will be placed at the start of the usr region plus the size of the first header plus the size of the request. If the thread already has pages, it will be treated differently. Since the thread has made allocations before, it is known that there will be a memory header at the start of its memory. This allows all the chunks of memory to be iterated over in order to find a free one. Since a best-fit algorithm is used, all the memory the thread has touched has to be looped over. This takes time, but helps prevent some fragmentation issues. If the request fits in a chunk already in the thread's pages, the free chunk will be split into a part for the request and a part that is still free and a pointer to the start will be returned. If the chunk is too big, more pages must be claimed. If there are not enough pages in memory or in swap, the request will be rejected and allocate will return a NULL pointer. Otherwise, these new pages will have their meta-data set, new memory headers will be made and a pointer to the start will be returned. Since we provide virtual addressing, allocate does not have to move the pages. If the meta-data is set and memory is protected, when the memory headers are copied into memory, a SIGSEGV is triggered and the signal handler will put the right pages in place. This will slow down the allocator, but makes the design and implementation simpler. Since it may be possible that a program will try to allocate memory before it makes a thread, any memory request before the thread library is initialized will be treated the request was from a thread with id zero. Since the main thread gets a thread id of zero upon creation, these pre-thread requests get treated as if the main thread requested memory. The size of the usr region is detailed in the section on shalloc.

If the thread library requests memory, it will be treated differently. Similarly to the thread request, a request from the library will be broken into two types depending on if there was a request made yet. If the library has not made any allocations yet, a new header will be made for the request and another will be made for the leftover space. The header for the request will be placed at the start of memory and the other will be placed at the start plus the requested size. If the library has made a request before, a best-fit algorithm will be used to determine where to put the new request. Although the cases and algorithms are the same between the two, the inner-workings differ slightly. Requests made from the library go into a separate sys region. The size of this is determined ahead of time since the amount of threads that are supported at a time is known. Since 32 threads are supported and each has a stack size of 64KB, it can be determined that 512 pages will be needed just for thread stacks. Adding another 100 pages for return values and other data brings the total up to 612 pages for the sys region. In addition, since the only thing that uses this region is the library, sys region does not have to be protected like usr region. This speeds up access to it as well and less time needs to be taken for the thread library to run. The sys region is the first region of memory with the usr region being the second portion. A global variable is used to tell whether a call has been made or not and if memory has been initialized yet.

Shalloc:

Since each thread has its own address space, it is impossible for two threads to share data between each other. This is especially problematic when parameters need to be sent to threads or return values need to be sent back to a parent thread. To solve this, a third region was added to memory. This third region is the shalloc region which a thread can allocate memory from with a call to shalloc. This region is not protected and any thread that obtains a pointer to this region can dereference the pointer without triggering a SIGSEGV signal. Memory is given out the same way it is in the sys region. Upon the first call, two headers will be made. One of these will be placed at the start and one placed at the end of the request size. If there has been a request, a best-fit algorithm will be used to find a suitable place to fit the request. The size of the shalloc region is only four page. Since the size of the memory array is 8MB and the combined size of the sys region and shalloc region are fixed, the total size of the usr region can be defined. Since there are 2048 pages overall and 616 pages are needed for the library and the shared region, the total size of the usr region is 1432 pages.

It should be noted that a thread cannot allocate more memory than 1432 pages, not counting memory it got through a shalloc call. Since memory is not protected past the usr region , it would be impossible to tell if a thread wanted some of its memory past there since no SIGSEGV signal will be sent. This would also cause an obvious problem since memory past the master memory array is not the memory manager’s to give away and could hold important data for other parts of the program. In addition since the shalloc region is after the usr region, a pointer given out past those 1432 could be in the shared region. This would cause problems since there would be inconsistencies in different threads interpretations of pointers to that region.

Deallocation:

Once memory is allocated, it is necessary to free it when it is not longer needed. This way memory saturation will be less likely since unused memory is able to be claimed by other threads and not hogged by one. The deallocate function is simple and the real work is done in coalesce which will be discussed later. All deallocate needs to do is check if check whether or not the pointer it was given was valid and the mark that memory chunk as free. To verify the given pointer was one handed out form allocate, a verification number was built into each memory header. The pointer given to

deallocate was subtracted by the size of a memory header in order to get the pointer to its header. The verify field of this was checked and if the number matched the well known constant, the deallocation went on. Otherwise the deallocate request was rejected. At this point the memory chunk was marked as free and coalesce was called.

While deallocate did not have to specifically know what region in memory the request was from, coalesce does. In order to determine the endpoints of when to stop coalescing, whether the pointer was to the sys, usr, or shalloc is needed. This parameter was passed from deallocate. Coalesce checks the neighbors of the newly freed pointer. If the left neighbor is free, the next field of the current pointer will be set to the end of the next memory chunk. If the right neighbor is free, the next field of the previous memory header will be set to the end of the current chunk. Of course coalesce needs to check whether or not these neighbors exist. If a coalesce was made, coalesce will call itself with the newly formed free pointer. This process will stop when there are two non-free pointers on both sides of a free chunk, when the free region occupies the entire region between the endpoints, or there is a non-free chunk on one side and an endpoint on the other.

If a thread did not free any of its memory before it exits, there would be a problem in that memory is being claimed while the owner thread is no longer around to use it. For this reason, all the page meta-data for both the memory array and the swap file are looked over when the exiting thread joins with its parent. When this happens, each meta-data structure representing a page is marked as unused meaning any thread after that that needs a page can claim it. Since the entire page is marked as unused, it is not necessary to loop though all the memory headers since the only thread that will be able to access those is no longer around. In this way threads not freeing their memory before they exit is not a problem.

Testing:

In order to test the memory manager, functionality was initially tested not using the thread library. Instead the illusion of a thread library was used. A global variable was used to store an id which could be set to mimic different threads requesting memory. This allowed for a simpler situation and the functionality of the thread library did not have to be considered. Using the memory manager in this way created a way to test how the allocator handled requests from different threads and how it handled requests from what would be the thread library. Shalloc could also be tested this way. Deallocation and coalescing could be tested too. Since there were no interruptions from the scheduler, testing in this way was not representative of how the memory manager would perform inside the thread library. After basic functionality was tested in this manner, the whole program was moved inside the thread scheduler.

At this point the thread library was tested using the same programs from the previous project. Since some of those programs did not have the threads allocating any memory, this was added to them. For the most part, everything worked as intended, except for testing externalCal. This caused our memory manager problems in a way the others did not which was perplexing at first. After careful observation, it was noted that externalCal had a variable of its own with the same name as our master memory array. Once our memory array was renamed and all references to it were changed, externalCal was run again. As expected this fixed the namespace problem as the program ran as it should have.