Eric Fiore EJF96
Daniel Parks DAP291
Eric Knorr ERK58

Project 2

Operating Systems (416)

README

Assignment 2

Metadata:

Two separate types of metadata that store information about memory allocations are employed for this library.

1) Page metadata – two different types of data are stored about pages. Both types are listed below. Pages are set up upon first call to the library in the *memory_init( )* function. The number of pages corresponds to the area reserved in memory for general thread use. In the current implementation this is 4,313,088 bytes which is slightly larger than 4 MB. Since each page is 4096 bytes the total amount of pages is 1053 pages.
    a. page_meta_id – This is a char pointer this stores the thread ID that will be using the page
    b. page_meta_index – this is a page number and is used to keep track of the pages position in the memory
2) malloc metadata – two different types of data are stored about malloced memory. Both types are listed below. Initial malloc metadata is set initialized in the *memory_init( )* function. Both items are pointers that are stored in the block of allocating memory.
    a. Size – this stores the amount of space that is available to allocate to requesting thread. Upon initialization the total size of the block of memory minus the area for metadata is set as the size. Since there are three different areas of memory each area has its own initial metadata.
    b. Free – this stores information that will to notify if a block of memory has already been claimed or not been claimed. When set to a zero the block of memory is free and when set to a one the block of memory is occupied. This is a simple short pointer

Memory:

There are four different areas in the total block of memory.

1) Public – this is the area reserved for general thread
2) Private – this area is for scheduler information
3) Shared – for the shalloc function
4) Stack – this area is reserved for thread contexts.

Allocation general information:

There are four possible allocation requests in our implementation. Each type of request is a controlled through a parameter in the *myallocate( )* function.

1) PRIVATE_REQ – this type of request is reserved for the scheduler. That is when the scheduler wishes to save its metadata information the scheduler will call *myallocate( )* with the
2) PUBLIC_REQ – this type of request is reserved for the general thread use. Any memory granted with this flag set will come from the public area
3) SHARED_REQ – this type of request is reserved for a shalloc request. Any memory granted with this flag set will come from the shared area.

4) STACK_ALLOCATION – this type of request is reserved for thread contexts. Any memory granted with this flag set will come from the area reserved for stack allocations

Functions:

void memory_init( )

This function is run when the malloc library is initially called. The total size of the memory is set with a call to *memalign( )*. Each area of the memory (see the memory section of this document for more information) is set and all pages and page metadata are set up to accept incoming requests. Each page is 4096 bytes (4 KB) and the initialization is handled by a for loop. The initial metadata of each page is set to -1 as a placeholder until actual information will be stored. The malloc metadata is also set for the three areas in memory. (private, shared, and stack). The free flag is set to free and the size is the total size of the respective block of memory. Each page is then protected in a for loop with a call to *mprotect( )*. The swap file is created and initialized to the correct size. The swap file is handled through a memory map which extends the area where pages can be stored. Finally, a signal handler is installed to detect when a 'bad" memory reference is made. That is if a thread wants to access its memory, but that memory has been swapped out to another location.

void myallocate(size_t size, char * file, int line, int flag, short TID)

Any time a memory allocation is needed this function will be called. If this is the first call into the library the *memory_init( )* function is called to set up the all the appropriate pointers and metadata. Once the initialization has been done (either if this is the initial call into the malloc library or initialization has already been done) flag is checked to see if the call is for a stack allocation, private request, public request, or a shared request. (see allocation general information for more information) Depending on what is the type of request that is incoming a pointer to the correct spot in memory is set and the maximum space for memory allocations is stored. A check is done to see if the size of the requested allocation is more then the total amount of memory. If this is true NULL is returned and an error message is printed.

void mydeallocate(void * index, char * file, int line, int flag, short TID)

When an allocation is no longer needed the user should call the mydeallocate( ) function in order to release memory blocks and make them available for reuse by other threads.

Upon starting the function flag is checked to see the type of request that is incoming. A pointer to the correct point in the memory block is set depending on the type of request. An if statement checks to see if index (index is a pointer in memory that is being freed) is in the valid pointer. If index is not a valid pointer then an error message is printed, and the function is returned. A pointer is initialized to the beginning of the memory block and then the pointer steps though each block of metadata until metadata block that is being searched for is found. If the metadata block cannot be found an error is printed and the function is returned. If the correct block is found the free flag is set to free and the previous block is checked to see if it is free. If it is free the two blocks are merged.

void vmem_sig_handler(int signo, siginfo_t * info, void * context)

This function is a signal handler that detects if an illegal memory access is detected. This must be done as memory pages are swapped out periodically. This has the result that a thread may look for a page in the

correct position, but that page may not be there. This will cause a segfault which will be caught by our signal handler and resolved by finding the correct page and swapping the bad page with the good page.

Anytime a page is in the swap file the memory map handles that case by extending the addressable memory. A page in the swap file will be swapped out as normal.

When the signal handler is first entered the timer (this is the timer that tracks the how long a thread has been running for the thread scheduler) is paused. The page number of the frame that the calling thread attempted to access is saved. If this page has never been used the metadata for the page is set, the page is unprotected, the timer is resumed, and the signal handler is returned. If the page had been previously used the correct page is searched for if found the page is swapped. If the page is not found a free frame is searched and upon finding a free frame the page is assigned to that block. Just as the signal hander is about to return the timer is resumed.