CS518: OS Design
Assignment 2: Memory-Manager + Scheduler

Members

Roshan Patel	`rrp78`
Daniel Hidalgo	`dh624`
Saad Ismail	`sni7`

Run Instructions

Run the following set of commands:

```
> make
> ./my_pthread_t [NUM_THREADS <= 8]
```

Files Included

makefile	=> makefile to compile the assignment
my_pthread_t.c	=> contains our main and definitions for functions of our structs
my_pthread_t.h	=> header file for scheduler, `my_pthread_t`, `thread_unit` `thread_unit_list`
thread_unit_lib.c	=> library of `thread_unit` helper functions
thread_unit_lib.h	=> header file for `thread_unit` and `thread_unit_list` helper functions
memory-manager.c	=> library of functions for memory abstraction
Memory-manager.h	=> header file for memory-manager

Overview

Our memory abstraction design is a form of segmented paging.

At the single page level, our memory allocator will hand out pointers to segments of memory. These segments are led by a 32-bit header that define metadata for that allocation. To reduce the memory footprint of myallocate() metadata, these headers are not structs. Their information is accessed via bitmasks and bitshifts on base types.

The entire memory block (8MB) will be divided into PAGE_SIZE blocks. Pages are allocated to a thread upon request and new pages are allocated to a thread when they fill up their current page or their new allocation request demands a size greater than a page.

Since threads are blind to their TID in our scheduler implementation, they call a function called myallocate_proxy(int size, __FILE__, __LINE__). This function is handled by the scheduler. The scheduler is able to collect the TID of the currently running thread, so it can simply return myallocate(size, file, line, currently_running->thread->threadID).

Our scheduler makes memory allocations for ucontext_t's, thread stacks, and internal structs that support the scheduler. These memory allocations are made using a special function named scheduler_malloc(). The last 2 pages of memory will be reserved for these structures and they will not be mprotected. Thus, all threads' ucontexts' and the scheduler's structs can be accessed more efficiently and easily. This is at a cost of protection. Since these pages are not protected, it is possible for a thread to alter the stack of another or alter the information inside the scheduler structs.

When a thread's page must be evicted from memory to be replaced by another, it is written out to a reserved location in a swap file (16MB). The swap file is created on disk and is accessed via a file pointer. We create an array of structs that tell us whose page is at any offset in the swap file. In other words, the first index of the array will give us the information for the page in the beginning of the swap file while the 5th index will lseek 5*PAGE_SIZE into the swap file to get that page while the struct will contain the page owner's information.

Due to the strategy we used for allocations made by the scheduler, there is a limit to the number of threads that our memory manager is able to support. This function will support up to 9 threads (including main) while allocating enough space for all of the structures associated with the scheduler. Attempting to run the program with an input that is greater than 8 will fail and exit.

Relevant Structures

memEntry

```
//for memEntry
// +------1-------+-------1------+------1------+----6----+------23-----------+
// |     valid    |     isfree    |   right_dep | GARBAGE |  request_size      |
// |              |               |             |         |                    |
// +--------------+---------------+-------------+---------+--------------------+
//
// 23 request_size  : max request size is 8388608 (8MB)
```

The above is the 32-bit sequence is how we are interpreting a memEntry. It is a total of 4 bytes and appears at the beginning of every memory allocation. The regions following the 23 bit request size is the space reserved for the user's allocation. The maximum allocation size allowed to the user is 8MB (2^23).

```
typedef struct PTEntry_{

    unsigned int    used:1;                 // Is the page currently used
    unsigned int    resident:1;             // Is the page resident in memory
    unsigned int    left_dependent:1;       // Do we need to load the next page
    unsigned int    right_dependent:1;
    unsigned int    dirty:1;                // Indicates if the page has been written to (i.e needs to be written back to memory when evicted)
    // unsigned int    UNUSED:4;
    unsigned int    largest_available:12;   // Size of largest fragment available inside the page
    unsigned int    mem_page_number:15;     // Index of page in memory (if it is loaded) (2048 pages for default page size)
    unsigned int    swap_page_number: 12;    // Index of page in swap where it is resident

}PTEntry;


/*
    A set of 128 PTEntries. Initially only 1 is allocated per thread.
    Up to 32 can be made for a thread.  (128*32) = 4096 total possible pages
*/
typedef struct PTBlock_{
    struct PTEntry_ ptentries[128];         // The block of PTEntries
    unsigned int    TID:8;                  // TID of owning thread
    unsigned int    blockID;                // ID number of block (0 indexed)
}PTBlock;



typedef struct SuperPTArray_{
    unsigned int    array[32];              // Max PTBlocks for a thread is 32  TODO can convert this to one int
    unsigned int    saturated[32];          // Is the PTBlock at index i full?  TODO """
    unsigned int    TID:8;                  // TID of thread that owns this SuperPTA
}SuperPTArray;


typedef struct ThrInfo_{
    unsigned int    TID;                    // TID of thread
    unsigned int    num_blocks;             // Number of PTBlocks it owns
    unsigned int    num_pages;              // Number of PTEntries it uses
    struct SuperPTArray_* SPTArray;         // Pointer to thread's SuperPTArray
    struct PTBlock_**   blocks;             // Array of blocks pointers
}ThrInfo;
```

PTEntry, PTBlock, SuperPTArray, ThrInfo

PTEntry is a page table entry structure. It contains all the meta-data associated with a page that helps us manage it. A PTBlock is an array of 128 page table entries. To save space, threads initially only create an instance of a block of page table entries (they are allowed up to 32 blocks). When a block is saturated, another block is added and 128 more entries are available to the thread. This helps provide the illusion of having a lot of space without actually allocating it.

The SuperPTArray helps us reduce the checktime for an address or a new allocation request. The SuperPTArray is two arrays of integers that indicate whether a block at index i is valid and/or saturated. This way we do not need to directly access the PTBlock unless we know it exists and is not full.
ThrInfo struct contains meta data associated to a thread. It holds pointers to a thread's SuperPTArray and its blocks.

```
typedef struct MemBook_{
    unsigned int    isfree:1;
    int             TID:8;
    int             thread_page_number:12;
    PTEntry*        entry;
}MemBook;


typedef struct SwapUnit_{
    unsigned int    valid:1;
    unsigned int    TID:8;
    PTEntry*        ptentry;
}SwapUnit;
```

MemBook, SwapUnit
A MemBook structure keeps a record of the contents of an index in memory. Who's page is currently resident in memory index i along with a PTEntry* which helps update meta data when swapping out the page. Our memory-manager has a global array of MemBooks[NUMBER_OF_PAGES_IN_MEMORY]. Each index of this array contains a MemBook with information about the contents of that index in memory. A SwapUnit is a structure that keeps a record of pages that have been evicted into swap_file. A pointer to its PTEntry is provided to help bring it back into memory if need be. All new assigned pages are given a reserved spot in swap_file upon initialization.

Functions and Heuristics

//myallocate()
This function returns a pointer to a location in memory with enough reserved space for the request size. When an allocation request is received from a thread, a page location in memory and swap_file is reserved for that thread. If this page is evicted from memory for any reason, it will go to its assigned location in swap. If it is brought back into memory, it be brought back into its assigned index in memory.
For requests that are larger than a page size, the memory manager will determine the number of contiguous pages required for the request. It will then search from the end of memory (excluding the 2 pages reserved for the scheduler) for a location to place these pages. These pages are instanced in the same way as above and are noted to be dependent on one another. If the user makes a reference to a pointer to a page that is dependent on other pages, all the dependent pages will be loaded into memory into their assigned locations.

//scheduler_malloc()
This is the subroutine that handles allocations specific to the scheduler. These allocations are made to the last 2 pages in memory and are not mprotected.

//**mprotect**
When a context switch occurs, all memory regions are protected from read/write. When the thread attempts to reference memory or request an allocation, a signal handler is fired and regions of memory in which this thread's pages are resident will be unprotected. Thus accsess is granted to this thread to use those pages, but not any others. When that thread is context switched out, all of memory is protected once again. The regions for the scheduler are never protected, allowing all threads to access their ucontexts and stacks.

//**deallocate**
The mydeallocate function takes in a pointer to free the space originally allocated by a thread. Since we cant assume the pointer is the same one orginally passed by myallocate, we page align the pointer respective to memory to find its membook index. We also find the currently running threads TID from our scheduler struct. If the threads page is currently loaded then we dont have to swap in the page otherwise we have to need to find the PTE corresponding to that memory block so that we can find its swap page number to load in the page to memory. Using the membook index and the TID we can find the PTE for that index from the ThreadInfo struct for that TID by searching the PTB. If there were any page dependencies(left or right) we start our iteration through from the leftmost page. We loop through the memEntries by adding to the current memory location the size of the mementry keeping track of the previous incase we have to merge multiple blocks. Once we found the entry that we want to free we updated the isfree to 1 and merge it when the left or right as appropriate.