Your next assignment will be to implement pseudo-virtual memory for your thread library in a series of phases:

Be sure to keep careful documentation of your code as you go. Your design and metadata should be documented.

Phase A: Direct-Mapped Memory

The basic functionality of malloc(size t size) is to return a pointer to a block of the requested size. This memory comes from a memory resource managed by the malloc() and free() functions. To emulate physical memory, you will create a static char array of size 8MB. You will maintain all your bookkeeping state in this array and allocate memory from it as well. Your memory manager will manage memory for your thread library and will presume its existence. Its code should either be part of your thread library, or be in an included library. The use of your own memory manager should be transparent to the source code. You should add a macro definition to transform all calls to malloc() and free() calls to calls to your own functions:

```
    #define malloc(x) myallocate(x, __FILE__, __LINE__, THREADREQ)
    #define free(x) mydeallocate(x, __FILE__, __LINE__, THREADREQ)
```

__FILE__ and __LINE__ are special compiler directives that substitute the name of the file and the line on which a function is called, respectively. This will allow for simpler debugging. THREADREQ should be a library-level constant that lets you know if a memory request is from your library, or from transformed user code. Since your thread library can be aware of your memory manager, it can call your own memory functions directly:

```
int * aptr = (int*)myallocate( 5*sizeof(int), __FILE__, __LINE__, LIBRARYREQ);
```

Since you can call your memory functions directly from the thread library, you can call them with a different library-level constant so that your memory manager can know the request is coming from the thread library itself, rather than from a thread, and can treat it differently.

You should reserve memory for a thread on the granularity of a system page. You can discover the system page size with "sysconf(_SC_PAGE_SIZE)". All malloc requests from the same thread should be served from the same memory page, until they fill that page. At this stage, if a thread requests more than a system page's worth of memory, you should consider its memory full, and return NULL pointers to all subsequent malloc requests. In order to achieve this, your memory functions should communicate with your thread scheduler to determine which thread is currently running, so that it can know who made the request and which memory page to allocate or free memory from.

Phase B: Virtual Memory

Once you can allocate and free memory, you can take the next step and provide for virtual memory allocation. To do this you need to provide a level of abstraction between the pointers your memory allocator returns and the physical locations of pages in memory. If you can do this, you can present an illusion of contiguous allocation. If a thread fills an entire page of system memory with requests and then asks for more memory, if you have a non-contiguous free page, you can swap it into page so that it now is contiguous, and begin allocating memory in that new page for the thread. This should be transparent for the thread. It should not be able to tell that your memory manager is moving pages around in the background.

In order to be sure your library is called whenever a thread tries to touch memory you should mprotect your memory pages. If so, any attempt to read or write that memory will result in SIGSEGV. You can catch that signal and perform the necessary operations in your signal handler:

```
	mprotect( buffer, pagesize, PROT_NONE);  //disallow all accesses of address buffer over length
pagesize
	mprotect( buffer, pagesize, PROT_READ | PROT_WRITE); //allow read and write to address buffer
over length pagesize
```

Memory that you mprotect MUST be page-aligned with the system memory pages. Page-aligned memory can be requested with memalign:

```
	memalign( sysconf(_SC_PAGE_SIZE), sizeofrequest);
```

Your scheduler should communicate with your memory manager. Whenever it decides to swap out the current content, it should mprotect all that context's memory pages. Whenever it decides to swap in a different context, it should unprotect all that that context's pages. This way your memory manager will only get a signal when a thread tries to access a memory location that once held one of its pages, but now holds another thread's data. A thread might try to use any of its pages at any time. If that page has been moved, it will try to access an address that it does not currently own. Your memory manager will need to determine which of the thread's pages correspond with the address it tried to access. In order to do this, your memory manager's signal handler must have the address that caused the SIGSEGV. You can get this address if you use the SIGINFO flag with the sigaction signal handler:

```
static void handler(int sig, siginfo_t *si, void *unused)
{
   printf("Got SIGSEGV at address: 0x%lx\n",(long) si->si_addr);
}

...

struct sigaction sa;
sa.sa_flags = SA_SIGINFO;
sigemptyset(&sa.sa_mask);
sa.sa_sigaction = handler;

if (sigaction(SIGSEGV, &sa, NULL) == -1)
{
   printf("Fatal error setting up signal handler\n");
   exit(EXIT_FAILURE);    //explode!
}
```

Your virtual memory manager should only return NULL on malloc requests if all of its memory pages are full. Any thread can request any amount of memory, and have any number of pages allocated to it.

Phase C: Swap File

Once you can identify and shuffle memory pages the final step is to, when memory fills up, write out memory pages to a swap file. Your memory manager should create a swap file 16MB large when it is first called.

Your threads are sharing the same address space. This means that the first allocation from thread 0 will be mapped to the same address as the first allocation from thread 1. Without a swap file, if one thread allocated all your 8MB, no thread could allocate any memory.

Using your swap file, you can support multiple threads allocating in the same space. If all memory pages are allocated to different threads and there is a new memory allocation request, instead of returning NULL you should pick a page of memory to evict, write it to your swap file and allocate the newly-freed page to whomever made the request. At this stage your eviction policy can be naieve. Any page that needs to be freed can be swapped to the swap file.

You will need to maintain a mapping from thread ID to page location in memory and in the swap file. When a thread tries to access one of this pages that is swapped out, you should pick a new page to evict, swap it out and swap the requested page back into memory.

If a thread consumes all of its address space and allocates the allocatable entire region, it can not allocate any more memory no matter what, and malloc should return NULL.

If both the swap file and memory are full, further malloc requests from all threads should result in NULL.

Phase D: Shared Region

Once the above has been implemented, designate the last 4 contiguous pages from your 8MB to be used as a shared memory region. This region should not be allocated using malloc since its ownership rules are different. If a thread wants to allocate memory in the shared region it should explicitly call the 'shared malloc' function:

```
void* shalloc(size_t size)
```

This function returns a pointer to a block of the requested size from the shared region. The difference in operation of the shared region is that all threads can access all the same pages. This means that any thread can request space in the shared region and that all threads can read and write memory in the shared region. This allows references to be passed to and between threads. This also means the shared region can not be swapable since all threads see the same address space. If the shared region is entirely allocated, your shalloc function should return NULL.