Project 2 Overview:

Once again, this project will have 2 parts. In the first part, you will get to implement your own user space memory allocator. You will learn the complexities and details of memory allocation and handling free lists first hand. You will also learn to build a shared dynamic library and you may even think a little bit about the efficiency of your code.

In the second part, you will first become familiar with how xv6 virtualizes memory and then add a couple of additional features to xv6 that many modern operating systems already have.

The projects from here out will most likely take much more time than project1, so **you are welcome to work in teams of up to 3 for the remainder of the semester.** Please do not procrastinate and put the project off until the last minute.

Part A: User space memory allocation

For this part of the project, you will be developing your own user space memory allocation library (which will replace the standard memory allocator….for example: malloc() and free()).

Some resources to get started:
Read Chapter 14 on the memory API to get a grasp on how malloc and free are called and what they return. Also have a look at their man pages.
Read Chapter 17 on free space management for details on how to go about implementing your own allocator. This chapter will help you tremendously on this project.

Note: You will eventually need to add a header to each allocated object much like the one described in Ch. 17. Your header should not exceed 16 bytes in size.

Details:

Memory allocators have a couple of responsibilities. First, the allocator must request memory from the OS for the process's heap. Typically, allocators start with a small heap and ask the OS for more space when needed by the application (this reduces the number of valid page table entries for the given process to only those needed by the process). Allocators may use sbrk or mmap to ask the OS for additional space. We will simplify our allocator by only asking the OS for memory space once. When we are out of heap space, we are simply out of space.

The second task of an allocator is to manage the heap, generally by maintaining a list of unused chunks of memory. When the process asks to allocate some heap space, the allocator chooses a chunk of free memory and returns it to the process via a pointer to it. When the process no longer needs a memory chunk, the allocator adds the freed memory back to the list of unused memory chunks.

The user space memory allocator is generally included as part of the standard library. It is not part of the OS. It runs fully in the virtual address space of the process, with no concept of pages or physical addresses. Address translation and the mapping of virtual pages to physical page frames is handled entirely by the OS.

Classic malloc() and free() are defined as follows:
- void *malloc(size_t size): malloc() allocates size bytes and returns a pointer to the allocated memory. The memory is not cleared.
- void free(void *ptr): free() frees the memory space pointed to by ptr, which must have been returned by a previous call to malloc() (or calloc() or realloc()). Otherwise, or if free(ptr) has already been called before, undefined behaviour occurs. If ptr is NULL, no operation is performed.

Your implementations of **Mem_Alloc(int size)** and **Mem_Free(void *ptr)** should follow what malloc() and free() do; see below for details.
You will also provide a supporting function, **Mem_Dump()**, described below; this routine simply prints which regions are currently free and should be used by you for debugging purposes.

Program Specifications:

For this project, you will be implementing several different routines as part of a shared library. Note that you will not be writing a main() routine for the code that you handin (but you should implement one for your own testing). We have provided the prototypes for these functions in the file **mem.h** (which is available in the project 2 resources on piazza); you should include this header file in your code to ensure that you are adhering to the specification exactly. **You should not change mem.h in any way!** We now define each of these routines more precisely.

- **int Mem_Init(int size):** Mem_Init is called one time by a process using your routines. size is the number of bytes that you should request from the OS using mmap().
- Note that you may need to round up this amount so that you request memory in units of the page size (see the man pages for getpagesize()). Note also that you need to use this allocated memory for your own data structures as well; that is, your infrastructure for tracking the mapping from addresses to memory objects has to be placed in this region as well. You are not allowed to malloc(), or any other related function, in any of your routines! Similarly, you should not allocate global arrays. However, you may allocate a few global variables (e.g., a pointer to the head of your free list.)
- Return 0 on a success (when call to mmap is successful). Otherwise, return -1 and set **m_err** to ERR_BAD_ARGS. Cases where Mem_Init should return a failure: Mem_Init is called more than once; size is less than or equal to 0.
- **void *Mem_Alloc(int size):** Mem_Alloc() is similar to the library function malloc(). Mem_Alloc takes as input the size in bytes of the object to be allocated and returns a pointer to the start of that object. The function returns NULL if there is not enough

contiguous free space within **size** allocated by Mem_Init to satisfy this request (and sets **m_err** to ERR_OUT_OF_SPACE). If Mem_Init has not been called yet, you should set m_err to ERR_MEM_UNINITIALIZED and return NULL.

- You may choose to implement any type of allocator you choose. Those discussed in the book include **Best-Fit, Worst-Fit, First-Fit, Next-Fit, etc.**
- For performance reasons, Mem_Alloc() should return 8-byte aligned chunks of memory. For example if a user allocates 1 byte of memory, your Mem_Alloc() implementation should return 8 bytes of memory so that the next free block will be 8-byte aligned too. To figure out whether you return 8-byte aligned pointers, you could print the pointer this way:

   printf("%p", ptr) . The last digit should be a multiple of 8 (i.e. 0 or 8).

- **int Mem_Free(void *ptr):** Mem_Free() frees the memory object that ptr points to. Just like with the standard free(), if ptr is NULL, then no operation is performed. The function returns 0 on success, and -1 otherwise. On failure, set m_err to ERR_INVALID_PTR prior to returning -1.
- **Coalescing:** Mem_Free() should make sure to **coalesce** free space. Coalescing rejoins neighboring freed blocks into one bigger free chunk, thus ensuring that big chunks remain free for subsequent calls to Mem_Alloc().
- **void Mem_Dump():** This is just a debugging routine for your own use. Have it print the regions of free memory to the screen.

You must provide these routines in a shared library named "libmem.so". Placing the routines in a shared library instead of a simple object file makes it easier for other programmers to link with your code. There are further advantages to shared (dynamic) libraries over static libraries. When you link with a static library, the code for the entire library is merged with your object code to create your executable; if you link to many static libraries, your executable will be enormous. However, when you link to a shared library, the library's code is not merged with your program's object code; instead, a small amount of stub code is inserted into your object code and the stub code finds and invokes the library code when you execute the program. Therefore, shared libraries have two advantages: they lead to smaller executables and they enable users to use the most recent version of the library at run-time. To create a shared library named libmem.so, use the following commands (assuming your library code is in a single file "mem.c"):
gcc -c -fpic mem.c -Wall -Werror
gcc -shared -o libmem.so mem.o

To link with this library, you simply specify the base name of the library with "-lmem" and the path so that the linker can find the library "-L.".
gcc -lmem -L. -o myprogram mymain.c -Wall -Werror

Of course, these commands should be placed in a Makefile. Before you run "myprogram", you will need to set the environment variable, LD_LIBRARY_PATH, so that the system can find your

library at run-time. Assuming you always run myprogram from this same directory, you can use the command:
setenv LD_LIBRARY_PATH ${LD_LIBRARY_PATH}:.

If the setenv command returns an error "LD_LIBRARY_PATH: Undefined variable", do not panic. The error implies that your shell has not defined the environment variable. In this case, you simply need to run:
setenv LD_LIBRARY_PATH .

Note that setenv is what you use in tcsh; if you are using bash, you'll have to figure out the alternative command to set the environment.

Unix Hints:

In this project, you will use mmap to map zero'd pages (i.e., allocate new pages) into the address space of the calling process. Note there are a number of different ways that you can call mmap to achieve this same goal; we give one example here:
// open the /dev/zero device
int fd = open("/dev/zero", O_RDWR);

// size (in bytes) needs to be evenly divisible by the page size
void *ptr = mmap(NULL, size, PROT_READ | PROT_WRITE, MAP_PRIVATE, fd, 0);
if (ptr == MAP_FAILED) { perror("mmap"); exit(1); }

// close the device (don't worry, mapping should be unaffected)
close(fd);
return 0;

Grading:

Your implementation will be graded on functionality first.
However, we will also give extra credit for efficiency. The **2 groups** with the most efficient implementations will be given 5 extra credit points each.

To Turn in:
Create a README file for your project. In it, list each member in your group's name and wustl key. Document which allocation strategy you chose to implement. Include why you chose that strategy, giving details on why you believe it will be more efficient than the others if efficiency was your goal. If your goal was "it seemed easiest to implement", that's fine! **However,** you should put some serious thought (**and document in your README**)into why it seemed easiest and how it compares with the other strategies.

Create a new folder named project2a-turnin. Place any files you created, including your README in the folder and create an archive using *tar* as in project1. **Do not** include any compiled code or files given to you (such as mem.h, libmem, or files ending in .o or .so) in this archive. Email the archive to the course email account (eng-cse422s@email.wustl.edu) with the subject line "CSE422-Project2A". **CC all team members on the email.**

You may turn in parts a and b in the same email, use the subject line "CSE422-Project2-All".

Part B: Exploring memory virtualization in xv6

In this part of the project, you should get familiar with how xv6 handles memory virtualization. You will have two goals:
1. Create a new system call that takes a virtual address and returns the physical address that it maps to.
2. Check for a program dereferencing a null pointer and throw an exception if so. This is standard in pretty much all operating systems today.

The order you finish each of these tasks in does not matter. You may find that handling null pointers first may help you understand the memory virtualization structures enough to implement the system call easily.

Start by reading chapter 1 of the xv6 book located on the resources page in Piazza as a reference to how xv6 allocates new processes.

Here are the details:

Part 1: You will add a system call named **sys_translate** to xv6. Translate takes a virtual address as a parameter (a pointer) and returns the physical address that the virtual address maps to. Much of this implementation should be similar to how you implemented sys_runnable in project 1, however you will need to look through the code and gain some understanding on xv6 the data structures and techniques xv6 uses to handle address translation.

Part 2:   In xv6, the VM system uses a simple two-level page table as discussed in class. As it currently is structured, user code is loaded into the very first part of the address space. Thus, if you dereference a null pointer, you will not see an exception (as you might expect); rather, you will see whatever code is the first bit of code in the program that is running. Try it and see! Thus, the first thing you might do is create a program that dereferences a null pointer. It is simple! See if you can do it. Then run it on Linux as well as xv6, to see the difference.
Your job here will be to figure out how xv6 sets up a page table. Thus, once again, this project is mostly about understanding the code, and not writing very much. Look at how exec() works to

better understand how address spaces get filled with code and in general initialized. That will get you most of the way.

You should also look at fork() , in particular the part where the address space of the child is created by copying the address space of the parent. What needs to change in there?

The rest of your task will be completed by looking through the code to figure out where there are checks or assumptions made about the address space. Think about what happens when you pass a parameter into the kernel, for example; if passing a pointer, the kernel needs to be very careful with it, to ensure you haven't passed it a bad pointer. How does it do this now? Does this code need to change in order to work in your new version of xv6?

One last hint: you'll have to look at the xv6 makefile as well. In there, user programs are compiled so as to set their entry point (where the first instruction is) to 0. If you change xv6 to make the first page invalid, clearly the entry point will have to be somewhere else (e.g., the next page, or 0x1000). Thus, something in the makefile will need to change to reflect this as well. You should be able to demonstrate what happens when user code tries to access a null pointer. If you do this part correctly, xv6 should trap and kill the process without too much trouble on your part.

Turning in project2b:

Create a README file for your project. In it, list each member in your group's name and wustl key. Document all files that you updated in xv6.

Create a new folder named project2b-turnin. Place any files you created, including your README in the folder and create an archive using *tar* as in project1. **Do not** include any compiled code or files you did not modify into this archive. Email the archive to the course email account (eng-cse422s@email.wustl.edu) with the subject line "CSE422-Project2B". **CC all team members on the email.**

You may turn in parts a and b in the same email, use the subject line "CSE422-Project2-All" and remember to CC all team members.