SE 350 (OSwow) Documentation Part 1

Memory (memory.c)

The memory is modeled as two doubly-linked-lists representing free memory and allocated memory. Each memory block is a node in one of these linked lists and contains the following properties as a header:

- A pointer to the next Memory Block
- A pointer to the previous Memory Block
- A pointer to the beginning of the usable data
- An unsigned integer of padding set to 0xABAD1DEA

When a memory block is requested, the data pointer is returned as the beginning of the usable memory. The reason a doubly-linked-list is used is to ensure constant operations for deletion (e.g. moving a block from the free memory list to the allocated memory list). We can also remove blocks from the middle of the allocated list in constant time with this approach.

The memory table is another portion of the OS that was implemented in this phase. The memory table assigns and tracks process ownership of memory which provides read and write protection, preventing a process from using the memory of another. A kernel pid was also implemented to allow the kernel to take possession of memory when allocating or deallocating memory. Otherwise, the processor would assign the memory to the current running process, preventing memory from being moved between memory lists.

Process(process.c)

The process was defined in a process table. The table included the process id (pid), the stack size which was defined as 0x100, and the function pointer of where the process PC starts. The priorities are defined from HIGHEST = 0, HIGH, MEDIUM, LOW, and LOWEST. The LOWEST priority is reserved for the NULL process.

There are 6 user processes defined, along with a NULL process. The 6 user processes define different test scenarios that are run, each polling a result. The NULL process is a simple process that continually runs while calling the release_processor() function.

The first user process just tests if the process gets run, and continually loops. After some iterations, it calls release_processor(). The second user process tries to assign the requested blocks to bad hex values such as 0xDEADC0DE and 0xDEAD10CC. These memory blocks then get verified of their values, and then are tested of if the return values

are correct. The third process we try to release an unallocated memory block, and see if our memory management handles bad calls properly. The fifth user process attempts to change the priorities of itself to see if they get updated. The final process basically polls the results of the rest of the user processes.

When release_memory_block is called, we need to check the block's owner's pid with the current process pid, otherwise, greedy users can just modify the owner pid of every block in the heap to itself, call free on all of them, and then request every block for itself.

Preemption occurs in two specific instances. First, when a memory block is released and there is a process with higher priority that is waiting for it. This blocked process will then continue running as it has acquired the needed resources. The second instance of preemption is when a process increases another process' priority above its own. This other process must be in the ready state, but it will be activated immediately if the conditions are satisfied.

What We Learned:

- Oh, we don't actually need dynamic memory; we can just stick the pointer for next* and prev* inside the blocks that we allocate! Since we're given a pointer the block that we want to free, we can just modify the next* and prev* pointers and do it in O(1) time instead of O(n) time to allocate and release memory.
- We wondered why our memory allocator was returning blocks of 0x800 instead of 0x100. Turns out it is because of pointer arithmetic. Since our doubly linked list has multiple variables and we're adding a constant to a node pointer, the compiler automatically multiplies it by the size of the object.
- Got invalid memory access when trying to normalize (set it to a constant) memory. Fixed it by adding extra padding to the end of the for loop so that it ends 8 bytes before the actual end of RAM.
- We write in big endian but the memory inspector window displays the 4 bytes as little endian so 0xDEADBEEF becomes EF BE AD DE. Created a macro to handle this so we can make our memory nice and readable.
- We had to determine who owned a memory block. We chose to create a table that keeps track of who owns memory. Similar to a hash table but much more secure than storing it in the block itself.
- When k_release_memory is called by either a system call from the user or directly from the kernel, it first checks if the given block's owner's pid matches the current process (stored in a global pcb pointer). However, if the kernel is requesting memory when pushing to the queue (implemented as a linked list), we don't want to use the user process's pid that originally made the system call. We made a work

around by creating a dummy kernel PCB to override the current PCB variable and then restore it once the kernel request finishes.

- The IDE is very bad at telling you why a linker error is occuring. We spent over an hour trying to figure out what was wrong. Turns out we were missing a file.
- Had major issues with files being included, the extern keyword and the IDE actually keeping track of what has to be included in the project. Fixed by slowing pulling it apart and putting it back together… Dependencies are hard
- Found a compiler bug. Apparently 0x7 * 0x4 = 0xA, but only when the 0x7 is in a define macro. When we looked at the disassembly, the compiler loaded the literal #6 to the register containing 0x4 and then moving it to the return register. This makes no sense. Fixed by hardcoding 28.