David Rohweder

Memory Allocator Project 3

CS473

12/6/2022

Memory Allocator Project

The design of my memory allocator project was simple. To begin, I set the chunk of contiguous memory to no protection, or no accesses allowed. Next, I initialize my frames and register my signal handler for SIGSEGV signals. After this initialization process occurs my program is ready to handle input. In the signal handler, the function is basically the interface for when read and write commands are executed. In the handler interface, I determine the causation of the cause of the error (command), the page on which this occurred, and the page offset or difference that the command was trying to execute. The handler contains a siginfo_t which contains a si_addr of the address in which the error occurred. To get the incident page I had to take the incident address – the start of the VM memory and then integer divide it by the size of the page to get the page number. The page offset was similar but instead of an integer divide I did incident modulus page_size) which works because of how the memory was declared in the main using POSIX alignment. Next, to extract the error I use the context structure also sent to the signal handler. Here, I can analyze the context error register and then see what value was contained inside. In the register a 4 is a read fault, a 6 is a write fault, and a 7 is a write-on read-only fault. After I determine all this event information, I encapsulate the data and send it to the policy handler. In the policy handler, I manage the frames of the given policy.

For the FIFO policy, managing frames is very easy. Before I start any eviction policy, I need to occupy my frames first by entering the page number, whether a write-back is needed if the op is a write, a global counter for when the add took place and then incrementing the global counter. This global counter is critical to compare when the frames were added for the FIFO policy. After all the frames have been occupied, I start to try eviction based on new operations. In the eviction handler for FIFIO, I iterate over every frame and determine first whether that frame contains the target page, and if so update the write back to reflect the new command, and set the frame for when computing the physical address later. If the page is not in any of the frames, I compare the occupied time in the frame to the lowest counter variable. If the page has a lower time, I set the eviction page equal to the current page in the current frame, update the lower time to the current frames time, update the write back to the current frames write back, and set the current frame pointer to the current frame for when we calculate the physical address later. After I have found the lowest time frame, I evict (set protection to none) and then update the frame to reflect the new event's data. After the eviction attempt has run, I update the protection on the frame to reflect the events command if write I do a bitwise on protection read and write, and if the command is read I only set the protection for read. After, I calculate the physical frame address by first declaring an unsigned int. Then I use the free frame found in the eviction function times the size of the frame plus the difference found above. Then, at the end of the function, I use the mm_logger to document the results.

For the TC policy, managing frames became a lot more difficult. The only thing different in this than the FIFO is the eviction policy that is called in the policy handler. First, I iterate over all the frames and try to see if the page already exists within the frames. If the page does exist and R=0 I check to see if it qualifies for a tracked read or write operation. If, after R=0, the frame has a write back and the event is a write (W->W) then the event is an operation 4. Else, if the event is a read, then the event is an operation 3. Also, if there is no writeback and the event is a write then the operation is a 2 (this is because when R=0 you remove protection, so you don't know that an op is a write-on read and looks like just a write when really it is a write-on read). If the page is not in any of the frames we "loop forever" or until the satisfying condition is met. We start the search at the victim frame which is initially set to zero. From here, we loop until we find a frame with the golden condition of either (R=0,M=0) or given third chance = true. Since when we add frames we set R=1, if the first condition is not met, we need to do 2nd chance algorithm, and if R=1 set R -> 0 and evict the protection of the frame. Next, we need to do 3rd chance algorithm and if R=0 and write back = true then we need to set a 3rd chance variable to true. An important corner case arose when you have a frame with 3rd chance enabled that did not get evicted, but gets updated, we need to carry that 3rd chance over. Then, if that frame has R=0 and carryover = true then we set carryover to false and given third to true to replace the frame the next time. We do not want to change the M bit though which is why I set the given third to true.