Sakib Rasul (sar370) | Sarah Squillace (ses333)

01:198:416 Operating Systems | Prof. Badri Nath

November 23, 2020

Project 3: User-Level Memory Management

Introduction

For this project, we built a multi-level page table to facilitate thread-safe memory virtualization. Our implementations of `malloc`, `free`, and methods for copying to/reading from virtual memory support a 32-bit virtual address space and a variety of physical memory and page sizes (with the requirement that both sizes be powers of two).

Methods

`SetPhysicalMem` is responsible for allocating the physical memory we want to virtualize. It begins with a call to `init_bitmaps`, a helper function that initializes physical and virtual bitmaps that each contain a 0th "failure indicator" bit (set to true upon memory failures) and bits that indicate whether or not physical and virtual pages are occupied with application data. It then initializes a mutex lock, allocates physical memory, and allocates a page directory. The size of the page directory depends on the number of bits available in virtual memory addresses after setting aside the rightmost $log_2(\text{page size})$ bits for the page table entries themselves. We define a page directory as a table whose entries are pointers to page table entries, which themselves are `void` pointers to physical pages.

`Translate` takes a pointer to the start of a page directory and a virtual address as input and returns the corresponding physical address. If the `USE_TLB` flag is set,

then the method begins by searching for the virtual address's translation in the translation look-aside buffer (TLB). If the flag is not set or if no TLB entries are found, the method continues by deriving page directory, page table, and memory offset indices from the virtual address. Then the address at the given location in the page table is returned, and the translation is loaded onto the TLB if needed.

`PageMap` takes a pointer to the start of a page directory, a virtual address, and a physical address as input and adds a page table entry for the address pair if one doesn't already exist. It begins by decoding the virtual address into a page table entry, initializing a page table if needed, and populating the entry and returning `1` if it is empty, or returning `-1` if the entry is already occupied.

`get_next_avail` checks the virtual bitmap to see if a page block of a given size `num_pages` is available for allocation. It begins by checking if the pages between bitmap indices `start_ptr = 1` and `num_pages` are free, and continues traversing along the bitmap until either an available page block is found or `start_ptr` exceeds the largest index of the bitmap. If a block is not found, `NULL` is returned. If one is found, then it is flagged as "in use" in the bitmap and a pointer to its start is returned.

`get_next_avail_phys` returns a pointer to the next available physical page as reported by the physical bitmap, or `NULL` if none exist. The method works simply by checking the bits of the bitmap in order until either a free page is found or the index being checked exceeds the size of the bitmap. If a page is found, we are careful to offset its page size-multiplied address by the starting address of physical memory in order to return to the application the correct starting address of the found page. Otherwise, all pages are occupied and we return `NULL`.

`myalloc` is an application-facing method responsible for reserving a given number of bytes `num_bytes` in virtual and physical memory. It begins by acquiring the mutex lock, calling `SetPhysicalMem` if it has yet to be called by the application, deriving the number of pages `num_pages` to be allocated from `num_bytes` and the size of a page, and calling `get_next_avail(num_pages)` to retrieve a new virtual address and update the virtual bitmap. If this call returns `NULL`, then the lock is released and the method returns `NULL`, as virtual memory is full. Otherwise, for every virtual page, we call `get_next_avail_phys()` to retrieve a new physical page address and update the physical bitmap and `PageMap` to load the new virtual/physical address pair onto the page directory. The method ends by releasing the lock and returning a pointer to the newly allocated virtual memory block.

`myfree` is an application-facing method responsible for freeing previously reserved page blocks in virtual and physical memory. It begins by acquiring the mutex lock and searching the page directory for every page in the block. If any of the pages are not found, then some or all of the input is not reserved and so cannot be freed. The lock is released and `-1` is returned. If every page is found, then each one is made `NULL` in the directory, and the corresponding bits in the physical and virtual bitmaps are set to `false` to indicate freedom. Last, the lock is released and `1` is returned.

`PutVal` is an application-facing method which copies data pointed to by `void *val` and of size `size` to a given virtual address `void *va`. It begins by acquiring the mutex lock, translating the virtual address to a physical address by calling `Translate`, and using the offset of `va` and `size` to calculate the number of pages the user wants to write. If only one page needs to be written, `memcpy` is used to write to memory, the

lock is released, and we exit the method. Otherwise, we use `Translate` and `memcpy` to continually write parts of `val` to offsets of `va` until all pages that need to be written are written to. We end by releasing the lock.

`GetVal` is an application-facing method which copies the contents of a given page block to a an address `void *val`. It uses much of the same logic as `PutVal`, with one small difference. The main distinction between `GetVal` and `PutVal` is that `PutVal` copies some values to physical memory while `GetVal` copies from physical memory. However, the rest of the algorithm including going page by page to copy the values is the same as `PutVal`. The only distinction is that `memcpy` is called where `val` is the destination to be copied to and the physical memory is where the data is copied from.

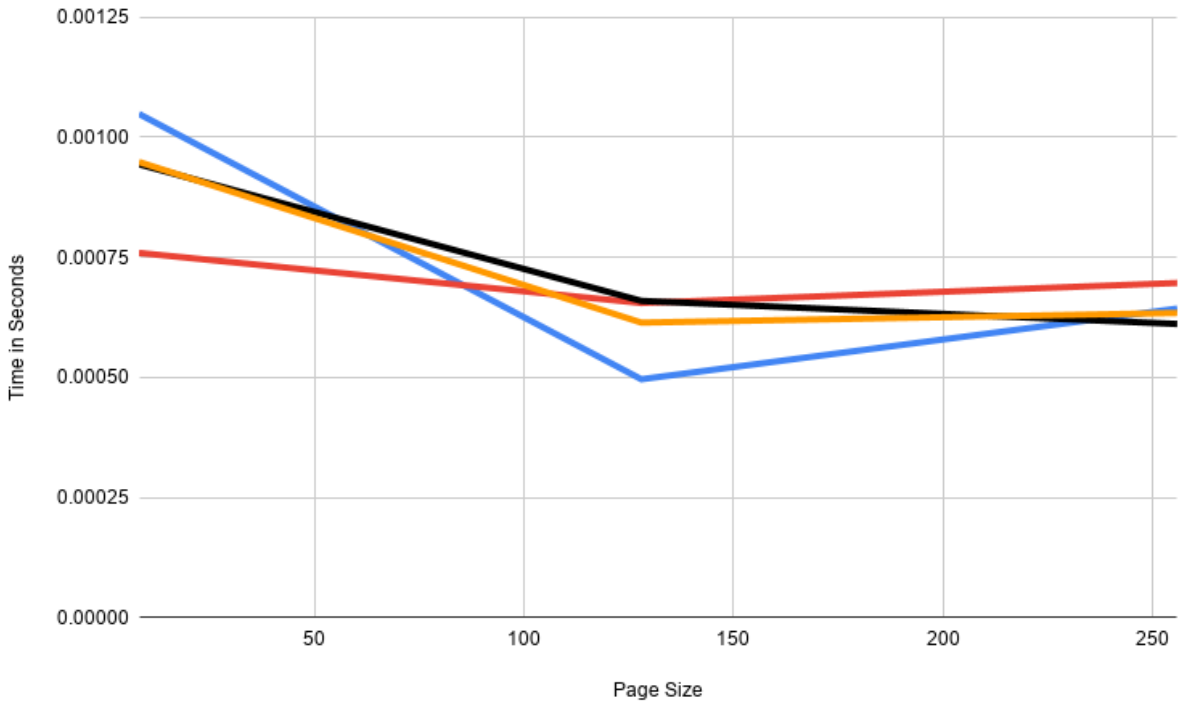
`put_in_tlb` first checks if the TLB is at maximum capacity; if it is, it calls `pop_from_back` to get rid of the oldest entry in the TLB. Then, a new `struct tlb` is created, and the virtual page number and the physical page number it maps to is set. The virtual page number is attained from dividing the virtual address by `PGSIZE`. The physical page number is attained from subtracting `phys_mem` and then dividing by `PGSIZE`. The new TLB entry is then added to the front of the queue.

`check_in_tlb` looks through the TLB until a virtual page number that matches that page number of the virtual address sent in as an argument is found. If an entry is found, it increments the hit counter and returns the physical address. If an entry is not found, it increments the miss counter and returns `NULL`.

`print_TLB_missrate` computes the quotient of the miss counter over the sum of the miss counter and the hit counter. It then prints the result of this computation.

Results

What we have found is that there were some impacts on speed in regards to page size. We tested with 5x5 matrix multiplication for each execution and took the average of the completion time of 10 trials. We found that there was very little difference between 128 byte and 256 byte pages, however both of these pages were much faster than 8 byte pages. We believe the reason why there was no difference is because, beyond 100 bytes, each matrix allocation was able to fit in a single page. A 5x5 matrix of integers will have 25 integers total. If each integer is 4 bytes, then each matrix took up 100 bytes of space. That is why the large page sizes eventually came to a plateau in their speed. Large pages are not advantageous when the user only uses a small portion of the page they allocate.



We also found that the TLB showed no significant increase or decrease in speed when compared to the results of running it without the TLB. We believe this to be one of the problems inherent to our code. The TLB is usually implemented with hardware, and this hardware is generally faster than memory. It is difficult to duplicate this effect on a software level.

Legend

Blue	TLBSIZE = 3
Red	TLBSIZE = 20
Black	TLBSIZE = 120
Yellow	No TLB

PGSIZE	TLBSIZE	AVG (SECONDS)	MISS RATE
8	3	0.0010484	72%
128	3	0.0004964	0.75%
256	3	0.0006434	0.75%
8	20	0.0007592	30.50%
128	20	0.000655	0.75%
256	20	0.0006966	0.75%
8	120	0.0009436	9.75%
128	120	0.0006594	0.75%
256	120	0.0006114	0.75%
8	N/A	0.0009484	N/A
128	N/A	0.0006142	N/A
256	N/A	0.00063425	N/A

Reflections

One way in which we could improve is to change our algorithm for `get_next_avail`. Using a sliding door approach could reduce the time complexity to O(n). We

also could have optimized our code a bit better by putting the code in `PutVal` into a separate function that could be used again in `GetVal`.

The most difficult parts of the project were deciding how to organize our data structures and coming up with computations to get the correct values for the addresses. It was difficult to keep track of these computations, especially in regards to pointer arithmetic. This made debugging a bit more difficult, since the origins of some of the segmentation faults we encountered were not immediately recognizable. Overall, this project required a strong understanding of pointers and the concept of memory virtualization as a whole.