CS416

Abid Azad(aa2177), Ghautham Sambabu(gs878)

Project 3 Report

Logic Behind Virtual Memory Functions

set_physical_mem(): This function manages the allocation of a memory buffer through the utilization of malloc, simulating the appearance of physical memory. To ensure thread safety, a mutex is applied at the onset and conclusion of the function. In terms of functionality, the function distinguishes between physical and virtual pages, initializing bits through the computation of virtual address bits from offset bits. Following this, a two-level page table is constructed, accounting for both inner and outer pages.

translate(): The function processes a virtual address and a page directory starting address, conducting translation to yield the corresponding physical address in collaboration with the two-level pages. The initial step involves managing locks through the use of a mutex. Within this context, the performBitmask helper function is employed to translate the outer page to the inner page. Subsequently, the add_TLB function is referenced to incorporate the TLB entry and retrieve the resultant physical address.

page_map(): The function accepts a page directory address, virtual address, and physical address as parameters, enabling the establishment of a page table entry. It systematically traverses the page directory to verify the existence of a mapping for the provided virtual address. In cases where the virtual address lacks a corresponding entry, a new one is introduced. To maintain thread synchronization, mutex locking and unlocking are consistently implemented at the initiation and conclusion of threads accessing this function. Furthermore, the performBitMask function is employed to generate addresses for mapping to the directory.

t_malloc(): This function allocates bytes and yields a virtual address. Synchronization is ensured by employing mutex locking and unlocking at the beginning and end, respectively, for threads referencing this function. The process includes an assessment of the feasibility of consolidating small allocations into a single page to optimize space utilization. The get_next_avail function is invoked to determine the next available page. Allocation information is then updated, and through the utilization of physical and virtual bitmaps, masks, and the page_map function, the page table undergoes necessary modifications to incorporate the new page.

check_TLB(): This function verifies the TLB for a valid translation and furnishes the physical page address. To ensure synchronization, mutex locking and unlocking are implemented at the outset and conclusion, respectively, for threads referencing this function. Additionally, the function monitors TLB accesses and misses, providing comprehensive tracking within its operational scope.

t_free(): This function deallocates one or more memory pages based on a provided virtual address. Synchronization is maintained through mutex locking and unlocking at the beginning and end, respectively, for threads referencing this function. The process involves updating allocation information and utilizing physical and virtual bitmaps, along with masks and the performBitmask function, to effectuate essential updates to the page table for releasing the specified page.

put_value(): This function duplicates data pointed to by "val" into physical memory pages identified by a virtual address (va). Synchronization is maintained through mutex locking and unlocking at the onset and conclusion, respectively, for threads referring to this function. More specifically, the translate function is employed to facilitate the process, utilizing memcpy to copy the content held by the given value into the specified virtual address within the page directory.

get_value(); This function functions in contrast to put_value(); when provided with a virtual address, it duplicates the contents of the page into the variable "val." Synchronization is ensured through mutex locking and unlocking at the beginning and end, respectively, for threads referencing this function. Concretely, the translate function is employed, utilizing memcpy to extract and copy the contents held by the page to the specified memory location pointed to by the "val" pointer.

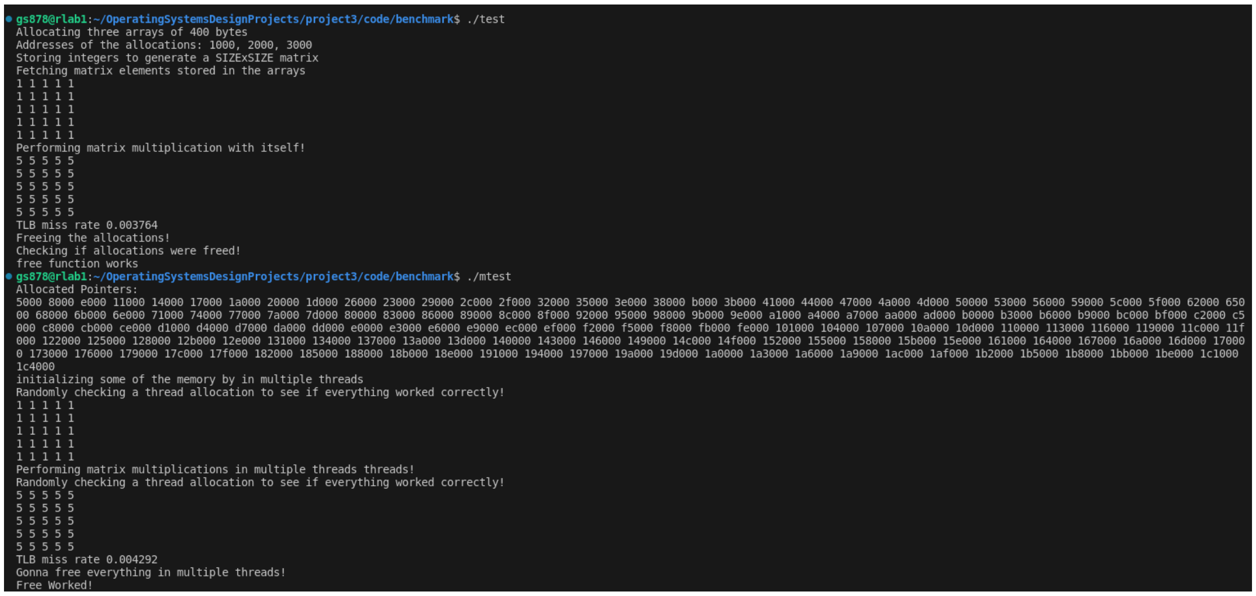
mat_mult(): This function takes two matrices, mat1 and mat2, as input arguments, along with a size parameter indicating the number of rows and columns. Synchronization is maintained through mutex locking and unlocking at the outset and conclusion, respectively, for threads referencing this function. Following matrix multiplication, the result is copied to the answer array, utilizing both get_value and put_value() iteratively. Matrices are indexed using the expression A[i][j] = A[(i * size_of_rows * value_size) + (j * value_size)], facilitating the necessary referencing for matrix operations.

print_TLB_missrate(): This function calculates and outputs the TLB miss rate, determined by the formula miss_rate = misses/accesses.

Benchmark output for Part 1

In our project, we incorporated matrix multiplication benchmarks to thoroughly assess the effectiveness of our virtual memory functions. This benchmark functions as a rigorous test, serving as a reliable indicator of code correctness. Notably, the expected results manifest only upon the integration of our page table and memory management library. Throughout the testing phase, we prioritized thread safety in our code, acknowledging the significance of seamless concurrent execution.

Below are the results for both the test and mtest:



```
● gs878@rlab1:~/OperatingSystemsDesignProjects/project3/code/benchmark$ ./test
  Allocating three arrays of 400 bytes
  Addresses of the allocations: 1000, 2000, 3000
  Storing integers to generate a SIZExSIZE matrix
  Fetching matrix elements stored in the arrays
  1 1 1 1 1
  1 1 1 1 1
  1 1 1 1 1
  1 1 1 1 1
  1 1 1 1 1
  Performing matrix multiplication with itself!
  5 5 5 5 5
  5 5 5 5 5
  5 5 5 5 5
  5 5 5 5 5
  5 5 5 5 5
  TLB miss rate 0.003764
  Freeing the allocations!
  Checking if allocations were freed!
  free function works
● gs878@rlab1:~/OperatingSystemsDesignProjects/project3/code/benchmark$ ./mtest
  Allocated Pointers:
  5000 8000 e000 11000 14000 17000 1a000 20000 1d000 26000 23000 29000 2c000 2f000 32000 35000 3e000 38000 b000 3b000 41000 44000 47000 4a000 4d000 50000 53000 56000 59000 5c000 5f000 62000 650
  00 68000 6b000 6e000 71000 74000 77000 7a000 7d000 80000 83000 86000 89000 8c000 8f000 92000 95000 98000 9b000 9e000 a1000 a4000 a7000 aa000 ad000 b0000 b3000 b6000 b9000 bc000 bf000 c2000 c5
  000 c8000 cb000 ce000 d1000 d4000 d7000 da000 dd000 e0000 e3000 e6000 e9000 ec000 ef000 f2000 f5000 f8000 fb000 fe000 101000 104000 107000 10a000 10d000 110000 113000 116000 119000 11c000 11f
  000 122000 125000 128000 12b000 12e000 131000 134000 137000 13a000 13d000 140000 143000 146000 149000 14c000 14f000 152000 155000 158000 15b000 15e000 161000 164000 167000 16a000 16d000 17000
  0 173000 176000 179000 17c000 17f000 182000 185000 188000 18b000 18e000 191000 194000 197000 19a000 19d000 1a0000 1a3000 1a6000 1a9000 1ac000 1af000 1b2000 1b5000 1b8000 1bb000 1be000 1c1000
  1c4000
  initializing some of the memory by in multiple threads
  Randomly checking a thread allocation to see if everything worked correctly!
  1 1 1 1 1
  1 1 1 1 1
  1 1 1 1 1
  1 1 1 1 1
  1 1 1 1 1
  Performing matrix multiplications in multiple threads threads!
  Randomly checking a thread allocation to see if everything worked correctly!
  5 5 5 5 5
  5 5 5 5 5
  5 5 5 5 5
  5 5 5 5 5
  5 5 5 5 5
  TLB miss rate 0.004292
  Gonna free everything in multiple threads!
  Free Worked!
```

The benchmark output affirms the successful initialization of memory by multiple threads, validating the accuracy of our memory allocation processes. Concurrent matrix multiplications further substantiate the robustness of our implementation. The matrix values in the output align precisely with the expected results, underscoring the precision and reliability of our code. Furthermore, our project addresses the critical consideration of managing TLB miss rates. Our implementation achieves a rate of 0.003764 for three 400-byte operations and a rate of 0.004292 for multi-threading. These values indicate the efficiency of our TLB caching mechanism in mitigating the need for accessing slower main memory during address translation.

The seamless execution of memory deallocation, exemplified by the freeing process in multiple threads, underscores the completeness and dependability of our project, establishing a robust foundation for future developments.

Observed TLB miss rate in Part 2

In our evaluation, the TLB miss rate for the "test" array is 0.003764 or 0.38%, and for multithreading, it is 0.004292 or 0.43%. This implies that, on average, approximately 3.764 and 4.292 TLB misses occur for every 1000 virtual address translations, respectively. It's noteworthy that the average TLB miss rate for any operating system typically falls within the range of 0.01-1%.A lower TLB miss rate is generally considered favorable, as it indicates the effectiveness of the TLB in caching frequently used address translations. This, in turn, reduces the reliance on slower main memory for translation information. Conversely, higher TLB miss rates can lead to increased memory access latency, potentially impacting overall system performance.

Support for different page sizes

In our virtual memory management project, we have methodically introduced a two-level paging table system with adaptable support for page sizes, each being multiples of 4K. This design offers a versatile and effective structuring of virtual memory, addressing various application requirements. A crucial element of our approach involves the meticulous calculation of the offset, determining the necessary number of bits. Through careful consideration of page size and the application of pertinent equations, we guarantee precise mapping of memory addresses to their respective physical locations.

Code issues (if any)

Our project has showcased resilient performance, encountering minimal issues throughout the development phase. The code adeptly fulfills functional requirements, yielding precise outputs and operating reliably across diverse conditions. Our commitment to scalability ensures the software's ability to manage expanding data or user loads. The codebase's maintainability is commendable, marked by well-organized data structures, comprehensive documentation, and adherence to best practices, facilitating ease for developers in comprehending, updating, and debugging..

Collaboration and References

In the process of developing this project, the following resources and collaborations were invaluable:

The Geek Safe Article ”C Thread Safe and Reentrant Function Examples”:

The article "C Thread Safe and Reentrant Function Examples" by Himanshu Arora elucidates the principles of thread safety and re-entrance in C programming through illustrative code examples. Thread safety ensures that a code segment can be executed by multiple threads concurrently without inducing synchronization issues like data corruption or race conditions. Re-entrance signifies that a code segment can be interrupted and reinvoked before the completion of its prior execution, such as by a signal handler or a recursive call.

The article outlines strategies for rendering a function thread-safe, involving the use of mutex locks, and making a function re-entrant by avoiding global or static variables and instead utilizing local variables.

In our project, focused on developing a multi-threaded application for complex calculations, we leveraged this article as a valuable resource. The goal was to ensure that our code could concurrently handle nested calls without errors. Adhering to the guidelines and examples presented in the article, we designed our functions and data structures. Rigorous testing, incorporating diverse scenarios and inputs, was performed. Additionally, we gained insights into utilizing tools and libraries supporting thread safety and re-entrance, such as pthreads and errno. This resource significantly contributed to enhancing the quality and reliability of our code while helping us steer clear of common pitfalls and bugs.

ChatGPT Collaboration for Data Structures:

Engaging with ChatGPT yielded valuable insights and guidance, particularly in the initial development of skeletal template structures for queues and hash map data structures implemented in C. These templates were subsequently expanded upon and tailored to suit the specific requirements of our project.

Community Forums and Programming Communities:

Active engagement with diverse online programming communities and forums, including Stack Overflow and the in-class Piazza programming community, played a pivotal role in troubleshooting and seeking advice from experienced developers and TAs. This participation proved instrumental in resolving coding challenges and gaining insights into best practices.