A Thread-Safe User-Level Virtual Memory Manager

Akhil Sankar
Department of Electrical & Computer Engineering
Rutgers, The State University of New Jersey

***Abstract*—This is a first-principles implementation of a simulated 32-bit virtual memory management system, employing a multi-level page table, translation lookaside buffer and thread-safe operations. The library manages a 1GB buffer of physical RAM, which is simulated on top of the underlying virtual memory interface provided by the operating system. Further, the library provides a standard 4GB of virtual address space to all consuming processes.**

I. Introduction

Traditional linear page tables for a 32-bit system require 4MB of contiguous memory to map the entire address space for a given process, causing significant internal fragmentation[1] when only a small fraction of memory might be required at a time. This architecture quickly overflows when running more memory-intensive functions. In this project, a matrix-matrix multiplication operation is used to simulate such a function.

To mitigate this issue of fragmentation, a multi-level page table (MLPT) can be used as an alternate data structure [1], providing a mechanism through which virtual pages can be lazily allocated and spread across the physical buffer through an associated mapping [1]. A translation-lookaside buffer (TLB) cache can be used to store the most recently accessed page table entries, significantly improving lookup times by reducing the overall number of scans through the page table.

In order to leverage any concurrency and parallelism provided by the OS, mutual-exclusion (mutex) based locking is also applied to maintain a controlled access to shared resources in memory and to ensure a deterministic order of operations across threads, eliminating the potential for race conditions.

Here, an implementation of a virtual memory manager wrapping a two-level page table is presented. Section II provides an in-depth description of the responsibilities of the virtual memory manager, macro and member function designs and associated flowcharts and diagrams to connect logic to higher level principles. Section III describes the addition of a TLB cache layer on top of the MLPT. Section IV showcases the triple-nested nature of matrix-multiplication and explores the basic benchmarking. Section V provides a discussion of the findings. Section VI provides ideas for future work and Section VII concludes the treatment.

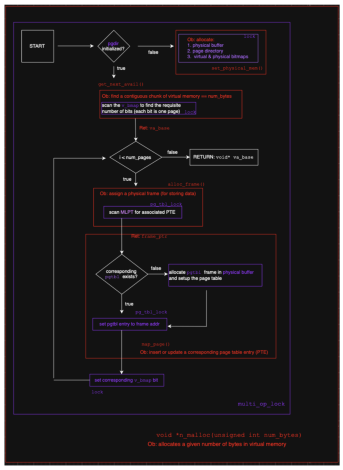

Fig. 1. Logic flow describing thread-safe memory allocation.

II. The Virtual Memory Manager

A. High Level Logical Flows & Multithreading

A set of high level operations are responsible for the interaction between the user application and the memory management library. These operations are i) allocating memory (Fig. 1), ii) de-allocating memory (Fig. 2), and iii) copying a segment of data bidirectionally between an application level buffer and physical memory, through an abstraction layer of virtual memory (Fig. 3). These operations can be run sequentially without any added logic. However, due to a list of shared memory resources, a three-tiered mutual exclusion approach [2] is implemented to enforce controlled sharing of these resources by locking them for access to one thread at a time. Such sections of code are called critical sections (bordered in purple in the given flowcharts Fig. 1, Fig. 2 and Fig. 3) and are executed within the bounds of a given lock (annotated along the border of each critical section in the corresponding figures).

Three distinct locks are used to enforce determinism. First, a low-level lock (annotated `lock` in the associated figures) is designated for low-level operations such as bitmap setter and getter operations, or coupling such bitwise instructions to remove the chance of an interrupted read-after-write (RAW). Next, a page-table level lock (annotated `pg_tbl_lock`) is designated exclusively for accesses to the MLPT. This lock is chosen for added clarity to more easily identify instances of deadlocking during nested calls to functions that use the

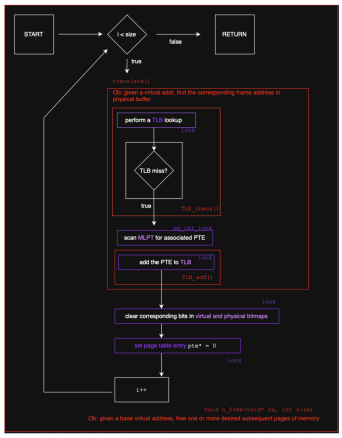

Fig. 2. Logic flow describing thread-safe memory de-allocation.

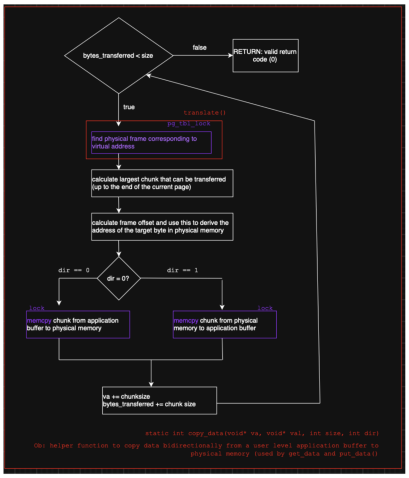

Fig. 3. Logic flow describing a thread-safe copy operation.

lower level lock. Finally, a highest-level multi operation lock (annotated `multi_op_lock`) is used to lock sections with multiple consecutive interactions with memory, forcing thread-safe resource access during the entire sequence. As can be seen in Fig. 1, much of the function is wrapped in purple, showing the potential for multiple different race conditions between shared-resource operations.

B. Supporting Macros for Virtual Addresses

Macros are used to generate bitmasks to retrieve specific positional components of a given virtual address: `PDX` is used to retrieve the page directory offset bits, `PTX` retrieves the page table offset bits and `OFF` retrieves the physical frame offset bits. The offset bits are calculated by taking the log_2`(PGSIZE)`, given by `__builtin_ctz(PGSIZE)`. These are the least significant bits in the virtual address.

A linear page table is highly space complex and needs to be fully allocated all at once, and would use all the remaining bits in the virtual address to identify a page table entry. In order to facilitate the most modular, lazy allocation of entries, smaller, page sized page-tables are preferable. Thus, the remaining bits in the virtual address are split up evenly, such that the page directory entries take up minimal space. This means that a page table can be accessed through one 32-wide element and can be lazily allocated as needed, in dynamic memory. This also reduces fragmentation significantly, as the page table can be located mostly anywhere in free physical memory.

For each of these helper functions, a target bitmap is provided, along with a target index. This index represents the flattened offset of the given page/frame in a linearized memory space. Thus, it can be broken down into coordinates: a byte and bit value that locates the corresponding bit in the bitmap. These functions can be applied the same way to either bitmap, virtual or physical.

C. Initializing Physical Memory, Page Directory & Bitmaps

The `set_physical_mem` function allocates the actual physical memory buffer and other relevant data structures (bitmaps and TLB). The actual buffer is allocated on the machine using `mmap` of `MEMSIZE` (the macro that defines the requisite physical memory). In addition, all entries in the buffer have read and write permissions. Note that this buffer is likely to be fragmented on the machine's actual memory, utilizing the same memory virtualization method described in this paper. Once the buffer is allocated, the `tlb_store` is additionally allocated and zeroed out. `tlb_store` is derived from the predefined struct `tlb` of capacity `TLB_ENTRIES`. Next, each of the bitmaps (virtual memory bitmap and physical frame bitmaps) are similarly allocated and zeroed out, representing free memory. Each bit in these bitmaps represents a single page/frame sized element. Finally, the frame(s) at the very top of the buffer are reserved for the page directory.

It is possible for multiple frames to be required, depending upon the predetermined size of the frames and the number of page tables to be tracked. In addition, the corresponding bits in the physical bitmap are set, ensuring that future allocations cannot overwrite the page directory. Note that by choosing a page directory that fits in one frame, it can be fully scanned using the preset `OFFSET_BITS`, provided in the virtual address. This makes for a much cleaner design, and lookups can be limited to the scope of a single frame.

D. Page Directory & Page Table Entries

The page directory entry `pde_t` is a 32-bit wide value made up of two components: Its more significant component represents the address of the page table as an offset. The less significant component (OFFSET bits wide by design) is space for relevant flags. Similarly, the page table entry (`pte_t`) is also a 32-bit wide value made up of two components:

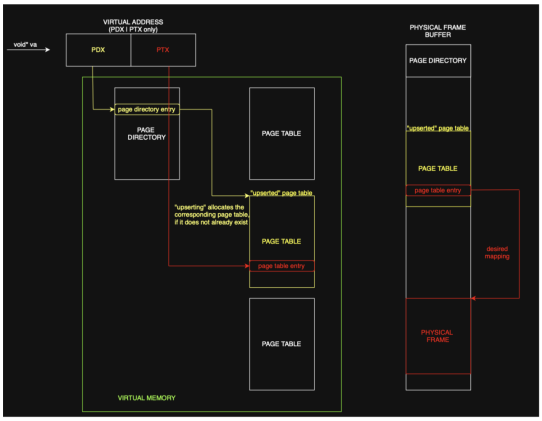

Fig. 4. Creating and storing a mapping of a given virtual address to a corresponding physical frame.

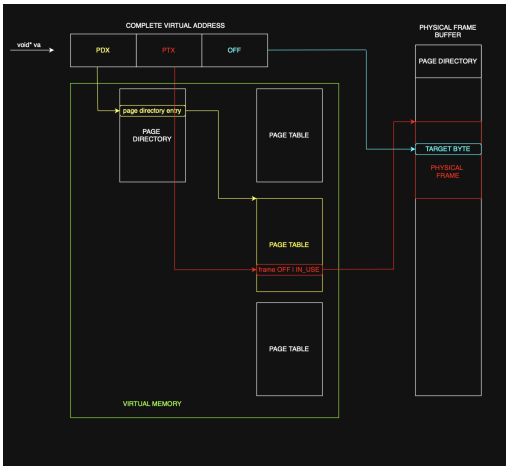

Fig. 5. Parsing a virtual address and scanning through a 2-level page table to locate its corresponding page table entry (and eventual physical byte).

the more significant component represents the address of the physical frame as an offset, while the less significant component (`OFFSET`) bits wide by design) is space for relevant flags.

Both page directory entries and page table entries are "frame addressable" (i.e. they encode a frame addressable offset). This means that the offset increments by `PGSIZE` such that each page table entry points to a discrete physical frame. The `IN_USE` flag marks whether a `pde_t` or `pte_t` is already in use.

E. Identifying Available Virtual Pages

Given a target number of pages to store, the `get_next_avail` function performs a lookup on the virtual memory bitmap, and returns a pointer to the first-found contiguous chunk of virtual pages with the desired capacity (if it exists). A virtual bitmap of equivalent size to the maximum number of virtual pages is used. Specifically, each bit in the bitmap represents whether a corresponding virtual page is already allocated in the address space. For a 4GB virtual address space with 4KB pages, there would be 220 virtual pages, and therefore 220 bits would be stored in the virtual bitmap.

A running `chunk_start` index and `counter` can then be maintained while traversing the bitmap in a loop. The bit is checked using `get_bit`, as discussed earlier. Upon identifying a contiguous sequence of unset bits that match the desired size, the `chunk_start` index can be mapped (frame-addressable) to an actual virtual address:

```
vpage_byte_offset = chunk_start * PGSIZE
```

The advantage is that a caller can to allocate a large contiguous segment of memory. By providing this virtual interface, each page of virtual memory can be mapped to a random fragment of physical memory, reducing fragmentation.

F. Creating Page Table Entries

If the desired virtual-page-to-physical-frame mapping does not already exist, the `map_page` function creates a new page table entry to do so (as shown in Fig. 4), while also updating the associated bitmaps. To create new mapping, the given virtual address can be stripped into its PDX and PTX components. The PDX and page directory base address can then be used to look up a page directory entry (encoding the target page table) in the page directory. If no page directory entry exists, a new page table needs to be allocated (through the `alloc_frame` function), as seen in Fig. 4. The new frame is then zeroed out (`memset`), setting all `IN_USE` flag bits to 0 (i.e. all pages are free to be allocated).

Recall that the advantage of this architecture is that page tables can be lazily allocated as needed. This significantly improves internal fragmentation that might be caused by a linear page table that must be fully allocated all at once. The page directory entry encodes an address (offset from `p_buff`) pointing at the page table in memory. The PTX value represents the specific page table index that stores the physical frame address.

The `alloc_frame` function can used to allocate both new page tables and physical frames. It identifies and returns the first unallocated physical frame found in the physical bitmap. The selected frame is then allocated by setting the corresponding bit in the bitmap. This bit represents the frame index with respect to `p_buff`, the base of physical memory. In addition, since each frame is of size `PGSIZE` (i.e. frame-addressable), a pointer to the actual frame memory is calculated like so:

```
p_buff_offset + (i * PGSIZE)
```

G. Identifying the Corresponding Frame for a Virtual Address

Given a virtual address and a pointer to the page directory, the `translate` function will attempt to walk through the page table and return the page table entry pointing to the corresponding frame in physical memory (if it exists). This is summarized in Fig. 5.

Using the PDX component of the virtual address, it first retrieves the parent page directory entry. This entry encodes the page table that would contain the expected mapping. To retrieve the page table, the lower flags component of the page table entry (discussed in `map_page`) can be stripped away by using a bitwise `AND` with a negated `OFFSET` mask. The resulting value represents an offset from the start of physical memory `p_buff`. The PTX component of the virtual address can then be used to scan for the corresponding page table entry to the given virtual address. This page table entry encodes the address to the physical frame (once again, as an offset from `p_buff`). If either the page directory entry or the page table entry does not exist, the mapping also does not exist and the frame is therefore free to be allocated.

H. Allocating Memory

Given a desired number of bytes, the `n_malloc` function allocates them in virtual memory (see Fig. 1). The input is in terms of bytes, so the first step is to convert this into an equivalent number of virtual pages. This term can be used to `get_next_avail`, yielding a virtual address void pointer pointing to the base of a chunk of available virtual pages that meets the size requirement. For each page increment, the following steps are then performed: the corresponding bit on virtual bitmap is identified and set. The bit index is a mapping of the base virtual address (as a page index), offset by an added frame for each page increment. This represents a virtual page that is now in use.

Next, a physical frame is allocated using `alloc_frame`. A `put_data` call can be used later to populate this frame as needed. In addition, a new page entry must be created through `map_page`, mapping the new virtual page to the address of the physical frame it references.

I. Freeing Memory

The function `n_free` frees one or more pages of memory, starting at a given virtual address (see Fig. 2). Similarly to `n_malloc`, the desired size (bytes) is first converted into units of pages (`vaddr32_t`). For each page increment, the following steps are then performed: the corresponding virtual page address is calculated incrementally with respect to the virtual address base:

```
va = va_base + (i * PGSIZE)
```

In addition, the corresponding bit for the virtual bitmap can be easily calculated using:

```
v_page_bit_idx = va / PGSIZE
```

The `translate` function can then be used to find the corresponding page table entry (ultimately mapping to the physical frame address). Again, it should be noted that the page table entry can be stripped by implementing

```
pa_offset = (*pte & ~OFFMASK)
```

as a bitmask to determine the physical frame offset (from the `p_buff` root address). This address is then used to determine the corresponding bit in the physical frame bitmap:

```
p_frame_bit_idx = pa_offset / PGSIZE
```

In addition, the page table entry pointer is also set to 0. By simply setting this entry and the two bitmap bits to 0, it does not matter what is actually stored on the buffer. This data is now treated as garbage and can be freely overwritten. Functionally, this means that both the virtual page and its frame have been freed from memory.

J. Copying Data

This section covers two API level functions: `put_data` and `get_data`. In general, these essentially represent the same sequence of steps to transfer data in opposite directions: `put_data` transfers data from a user buffer to the physical buffer. `get_data` transfers data from the physical buffer to the desired user buffer.

These two functions are further generalized by a helper function: `copy_data` (see Fig. 3). Similar to other functions, copying is performed in an incremental manner until the number of bytes transferred matches the desired size parameter. During each increment, the corresponding page table entry is first identified (using translate) based on the given virtual address. The difference `PGSIZE - OFF` of the virtual address is then used to determine the largest chunk of data that can be transmitted in one increment. If the desired size is less than or equal to the above difference, the entire chunk can be transferred to/fro in one chunk. Alternatively, if it is larger, the chunk is selected to be the remaining memory in that page from the given frame offset.

Next, the two buffer pointers are calculated using the respective base addresses and derived offsets:

```
void* pa_ptr = (char*)p_buff + pa_offset
void* ext_ptr = val + num_bytes_written
```

These pointers are assigned as the source and destination base addresses, contingent on the value of the `dir` parameter, which defines the direction of transfer. A `memcpy` operation is then used to transfer the chunk of bytes from source to

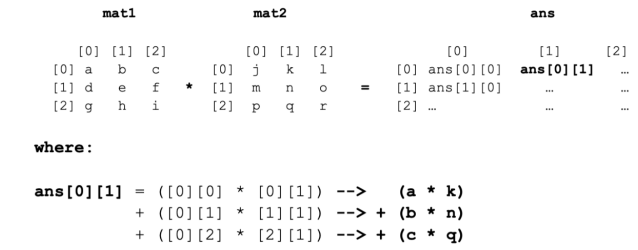

Fig. 6. A simple demonstration of how a single entry of the product matrix is calculated, using two operand values and an accumulator.

destination. Upon transferring the chunk, the base virtual address and `num_bytes_written` are incremented to allow the next increment to continue in the same manner.

III. TLB Caching

The Translation Lookaside Buffer (TLB) cache is used to speed up page table entry lookups. It only requires minimal added logic as it acts as an intermediary between the caller function and memory. The TLB is physically stored as a pre-allocated struct, whose values are arrays that correspond to "rows" in the buffer. Thus, a row can be read/updated using an index, for each of the individual keys. The definition is shown here:

```
#define TLB_ENTRIES   512

struct tlb {
  uint32_t vpn[TLB_ENTRIES];
  pte_t* pte[TLB_ENTRIES];
  bool in_use[TLB_ENTRIES];
  uint32_t last_used[TLB_ENTRIES];
};

extern struct tlb tlb_store;
```

The `TLB_add` function looks up the target virtual address, and "upserts" the corresponding entry. Thus, if an entry already exists, it overwrites that entry. If not, it creates a new entry. The virtual page number and physical address are inputs to this function. A `last_used` field is maintained in case an LRU retention policy should be used later. This value is simply tracked using a counter `tlb_lookups`.

The `TLB_check` function scans through the TLB, attempting to find the corresponding PTE for a given virtual address without needing to perform a main memory lookup, providing a huge time saving (if a cache hit occurs).

The `print_TLB_missrate` function is added for performance tracking: `tlb_misses` and overall `tlb_lookups` metrics are maintained throughout the lifetime of the program. Thus, at the conclusion of benchmark testing, the miss rate can be calculated simply as a ratio of these two metrics.

Single-Threaded Performance:

Matrix Size	Duration (sec)	CPU Usage	TLB Misses	TLB Lookups	TLB Miss Rate
10	0.134	2%	3	2064	0.115%
20	0.129	3%	3	18401	0.016%
30	0.226	41%	30	2060040	0.002%
40	0.134	6%	6	137602	0.0044%

Multi-Threaded Performance

Matrix Size	Duration (sec)	CPU Usage	TLB Misses	TLB Lookups	TLB Miss Rate
10	0.142	6%	45	12248	0.3674%
20	0.380	10%	45	88803	0.0506%
30	0.158	78%	45	289848	0.0155%
40	0.235	152%	45	675248	0.0067%

Fig. 7. Benchmarking results showcasing single-threaded versus multi-threaded performance to exercise the memory manager equipped with a TLB cache layer.

IV. Benchmarking

A provided matrix multiplication function performs the actual work of multiplying two square matrices in an element-twise fashion. Due to the nature of matrix multiplication, the operation requires a triple loop, incrementing over a common index k. A simple example of this behavior is shown in Fig. 6. In this figure, the `0`th index in the first operand and the `1`th index in the second operand is k. In addition, matrices are represented contiguously in memory. Thus, in order to access a specific index for a square matrix, an overall offset is calculated each time:

```
ans =  ans
       + (i * size * sizeof(uint32_t))
       + (j * sizeof(uint32_t));
```

This equation takes the base address of the answer matrix and determines the offset of the target location in memory, scaling row indices by the width of the matrix and incrementally adding in the column values. This same approach is applied for the two operands as well. Once the addresses are calculated, the data stored at these locations is then copied into `a` and and `b`, using `get_data`. An accumulator variable `c` is updated with the product of the two operands elements (see Fig. 6). At the end of the row-column product-summation, this accumulator value is then written to the respective destination position in the answer matrix.

Using the built-in Linux-API provided `time` operation, the duration and CPU usage of the generated benchmark executables are compared when performing a simple square-matrix multiplication operation. Note that the implementation of the matrix multiplication through a triple-nested loop implementation is used to exercise the library. The CPU usage of single-threaded versus multi-threaded (15 threads)

execution is also compared, after verifying that multi-threaded execution generates accurate, deterministic results. The TLB is active for these benchmarks as well, and the number of lookups, TLB misses and derived miss rate is showcased, against an increasing workload based on the size of the matrix operands. The results are showcased in Fig. 7.

V. Discussion

As observed through the collected benchmark metrics in Fig. , duration increases with greater workloading, given by increaasing operand sizes. This trend can be observed in the single-threaded instance. The multi-threaded instance scales CPU usage dramatically, without increasing duration too much. It should be noted that a bottleneck of this approach is the memory allocation for the operands given in the benchmark logic. Changing this value is outside the scope of this treatment.

TLB miss-rates tend to drop significantly with increased operand size, This is likely due to the disproportionate contribution of cold misses, or misses when the associated TLB entry is initially empty before being written to for the first time. As operand size increases, this has a decreasing impact on the overall miss rate.

VI. Future Work

Given the relatively hardcoded approach of the page table architecture, a next iteration would allow for the page table to scale to 4-levels, likely further improving on fragmentation and lazy allocation. This would significantly improve space complexity, while increasing the lookup steps and overall complexity of the code, with more risks of segfaults and memory leaks.

In addition, a more robust thread-safe verification test should be performed, ensuring the expected result is outputted deterministically, over a dynamic range of threads. The allocated memory for the given operands should also be scaled dynamically, allowing for full saturation of CPU execution contexts.

Finally, more experiments can be performed to compare performance based on the changing size of page tables as well as the TLB itself. There may also be hidden edge cases for other such multiplication operations that could cause specific deadlocking issues. The logic presented here is tested only against a square matrix product operation, and is therefore not guaranteed to be thread-safe if adapted for other operations.

VII. Conclusion

A fully functional virtual memory manager using an underlying two-level page table is implemented from first principles. It uses a TLB for improved performance, significantly reducing main-memory accesses. Thread-safety is also applied through the use of mutexes. Matrix-matrix multiplication is used to benchmark performance of the manager.

References

[1] Remzi H. Arpaci-Dusseau, and Andrea C. Arpaci-Dusseau “Operating Systems: Three Easy Steps” Arpaci-Dusseau Books, Nov 2023.

[2] Linux Manual Pages, IEEE/The Open Group: 2017, pthread_mutex_lock(3p)