Akhil Sankar
Master of Science: Technical Paper
Github: https://github.com/fuzzygreenblurs/nimbus_virtual_memory

A Thread-Safe User-Level Memory Manager for Matrix Multiplication

Abstract

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

Motivation

Traditional linear page tables for a 32-bit system would require 4MB of contiguous memory to map the entire address space, causing significant internal fragmentation when only a small fraction of the pages might be required at a time. A multi-level page table provides a mechanism through which pages can be lazily allocated and spread across the physical buffer through an associated mapping.

The TLB acts as a cache to store the most recently accessed page table entries, significantly improving lookup times by reducing the overall number of scans through the page table.

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

Architecture & Implementation of VM Library

High Level Logic (and MultiThreading)

There are three high level logic flows that are responsible for the interaction between the user application (the actual matrix multiplication function) and the Nimbus memory management library. These operations are:

1. allocating memory (**Fig. 1**)
2. freeing memory (**Fig. 2**)
3. copying a chunk of data bidirectionally (**Fig. 3**) from an application level buffer to physical memory, through the abstraction layer of virtual memory.

These operations can be run sequentially without any added logic. However, due to a list of shared memory resources, a 3-tiered mutual exclusion approach 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, (those bordered in purple in the given flowcharts) and are executed within the bounds of a given lock (annotated along the border).

There are essentially 3 locks used to enforce this:

1. `lock`: This lock 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-affter-write (RAW).

2. `pg_tbl_lock`: This lock is designated for reads or writes performed when accessing the multi-level-page-table (MLPT). This lock name is chosen for added legibility and to more easily identify instances of deadlocking during nested calls to functions that use the lower level `lock`.

3. `multi_op_lock`: Finally, this lock is used to lock the most memory-intensive flows such as malloc, which essentially needs to be run sequentially. As can be seen in Fig. 1, nearly every step is marked in purple, showing the potential for multiple different race conditions.

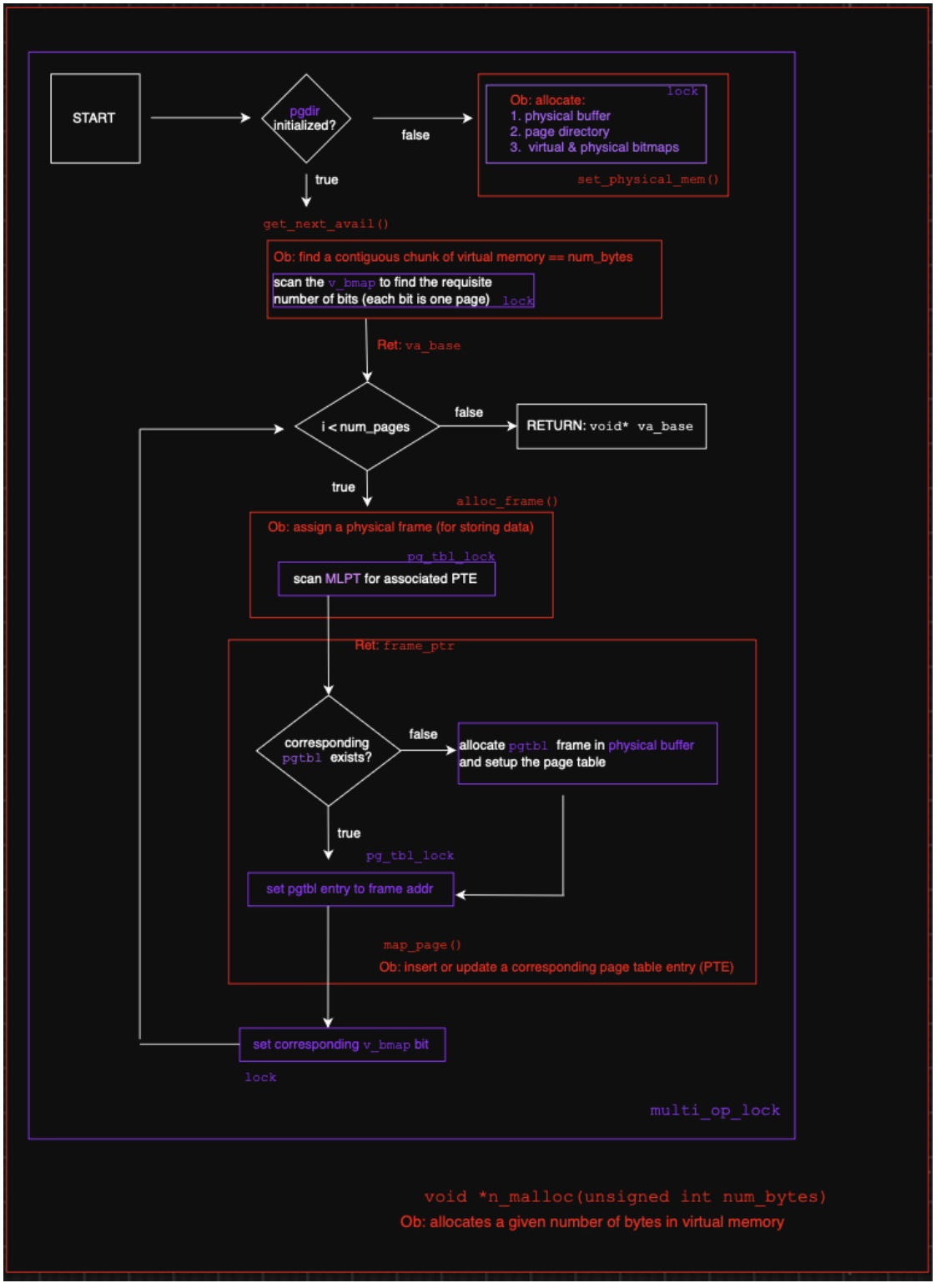

Fig. 1: Logic flow describing thread-safe memory allocation

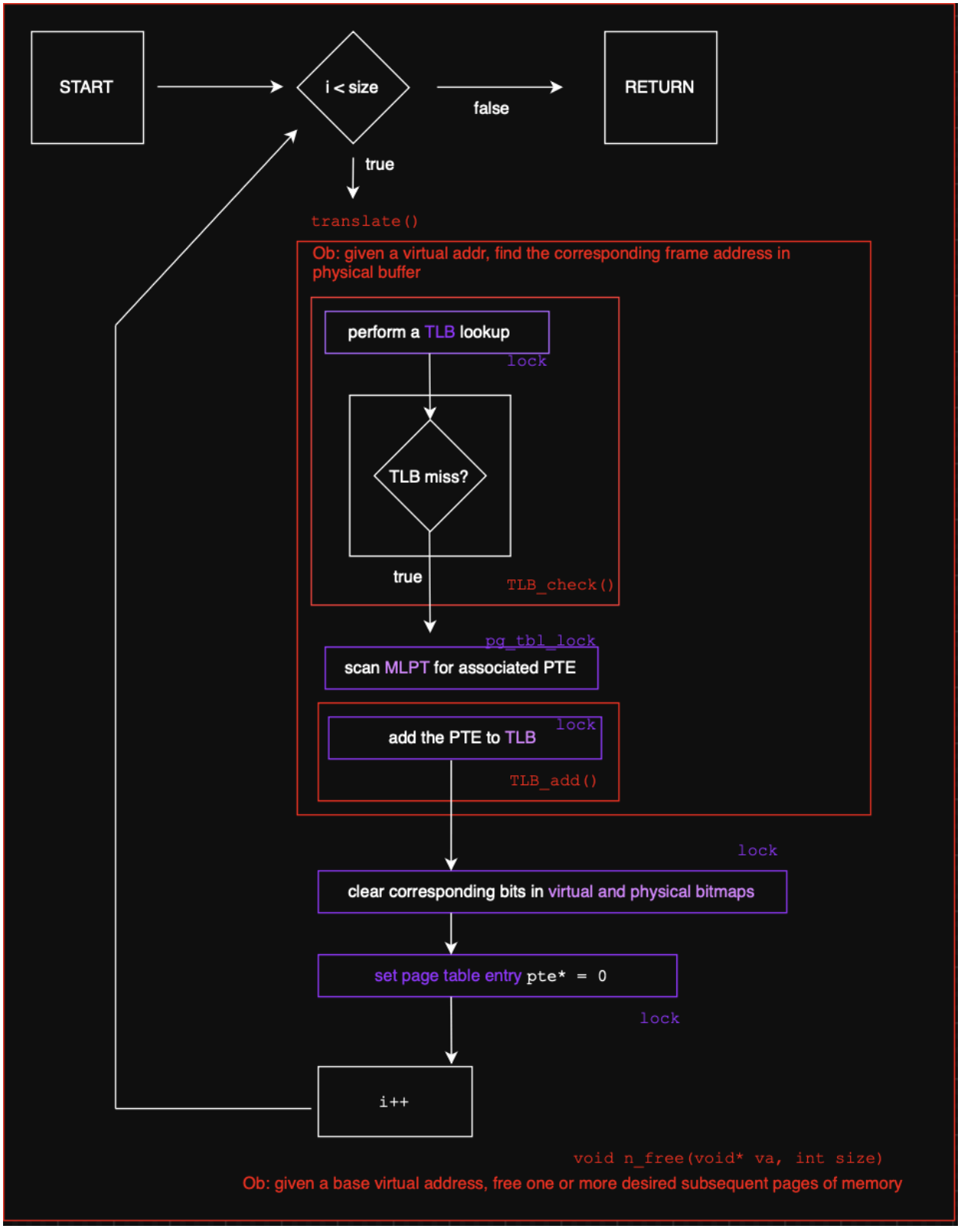

Fig. 2: Logic flow describing thread-safe memory de-allocation

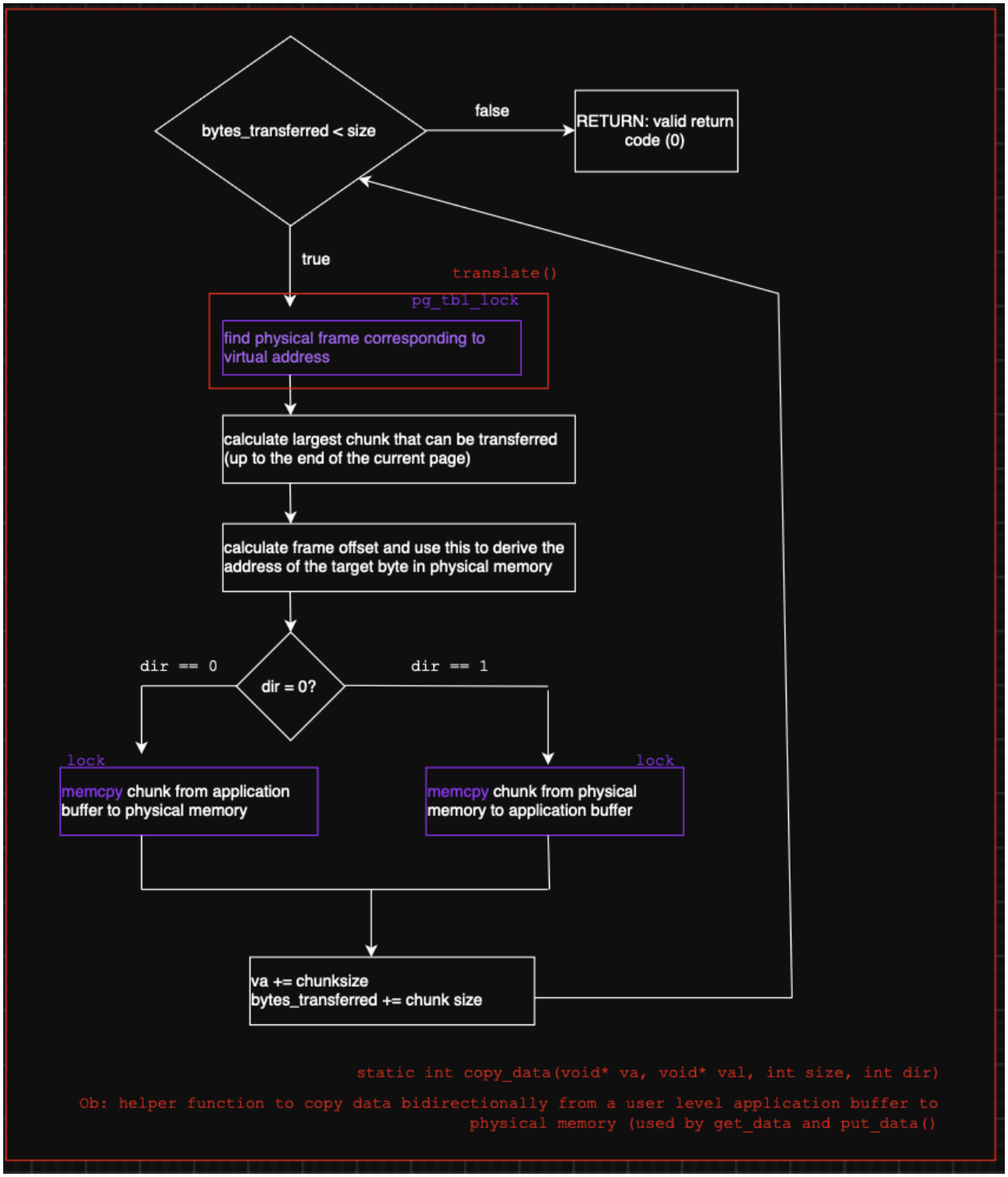

Fig. 3: Logic flow describing a thread-safe copy operation

Macros and Macro-Helper Functions

Objective: These are simple functions that act as masks to retrieve specific positional components of a given virtual address.

1. `PDX: retrieves the page directory offset bits`
2. `PTX: retrieves the page table offset bits`
3. `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 lowest 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.

Bitmap Helper Functions

1. `void set_bit(char* bmap, int idx);`
2. `int get_bit(char* bmap, int idx);`
3. `void clear_bit(char* bmap, int idx);`

For each of these helper functions, a target bitmap is provided, along with a target index. This index represents the flatted 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.

```
uint32_t target_byte = idx / 8;
uint8_t target_bit  = idx % 8;
```

The bit is either set (`|= (mask)`), cleared (`&= ~(mask)`) or read (`&`) accordingly. These functions can be applied the same way to either bitmap (virtual or physical).

Initializing Physical Memory, Page Directory & Bitmaps

```
void set_physical_mem(void);
```

Objective: This 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: this buffer is likely to be fragmented on the machine's actual memory, utilizing this very memory virtualization method described in this paper.

Once the buffer is allocated, the `tlb_store` is additionally allocated and zeroed out. The `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.

Page Directory and Page Table Entries

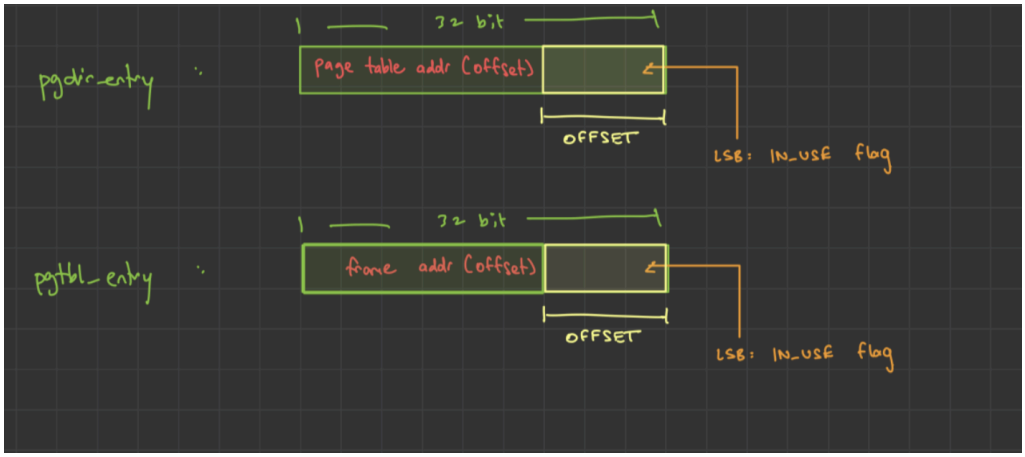

Fig 4: Format of the page directory (upper-level page table) and page table entries

The page directory entry (`pde_t`) is a 32-bit wide value made up of two components:

1. The upper component represents the address of the page table as an offset
2. The lower component (OFFSET bits wide by design) is space for relevant flags

The page table entry (`pte_t`) is also a 32-bit wide value made up of two components:

1. The upper component represents the address of the physical frame as an offset
2. The lower component (OFFSET bits wide by design) is space for relevant flags

Both page directory entries and page table entries are “frame-addressable”: 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.

Identifying Available Virtual Pages

```
void* get_next_avail(int num_pages)
```

Objective: Given a target number of pages to store, perform a lookup on the virtual memory bitmap, and return a pointer to the first-found contiguous chunk of virtual pages with the desired capacity (if it exists).

A virtual bitmap `v_bmap` of equivalent size to the total possible 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 2^{20} virtual pages, and therefore 2^{20} 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:

```
uint32_t vpage_byte_offset = chunk_start * PGSIZE;
```

Note: the advantage here is that a caller might want to allocate a large contiguous segment of memory. By providing this virtual interface, each page of virtual memory can be mapped to a non-contiguous fragment of physical memory, further reducing fragmentation.

Creating Page Table Entries

```
int map_page(pde_t *pgdir, void *va, void *pa)
```

Objective: If the desired virtual-page-to-physical-frame mapping does not already exist, create a new page table entry to do so, and update the associated bitmaps.

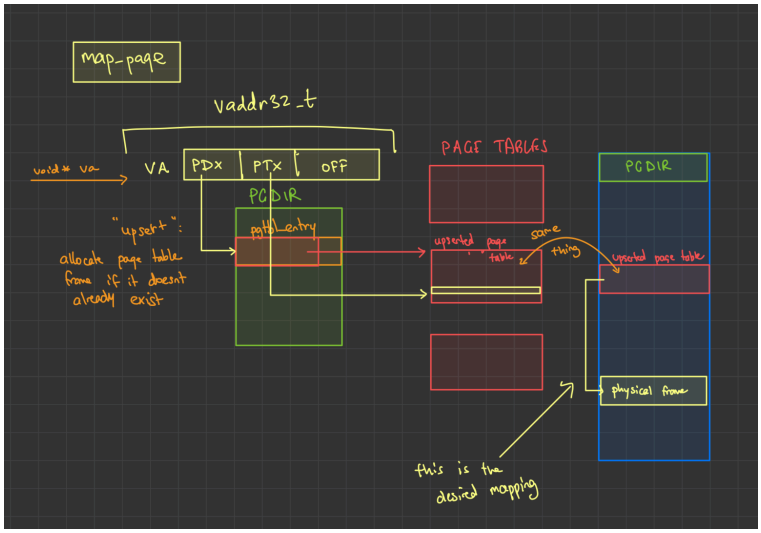

Fig 5: Creating and storing a mapping of a given virtual address to a corresponding physical frame

To create new mapping, the given virtual address can be stripped into its `PDX` and `PTX` components. The `PDX` and `pgdir` 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 (discussed in `alloc_frame`). 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).

The advantage of this architecture is 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.

```
void* alloc_frame()
```

Objective: Identify and return the first unallocated physical frame found in the physical bitmap. Mark this frame as allocated by setting the corresponding bit in the bitmap. `alloc_frame` is used to allocate both new page tables and physical frames.

This bit represents the frame index with respect to `p_buff`, the base of physical memory. In addition, since each frame is of `PGSIZE` size (i.e. frame-addressible), a pointer to the actual frame memory is calculated like so:

```
p_buff + (i * PGSIZE);
```

Identifying the Corresponding Frame for a Given Virtual Address

```
pte_t* translate(pde_t* pgdir, void* va)
```

Objective: Given a virtual address and a pointer to the page directory, this 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).

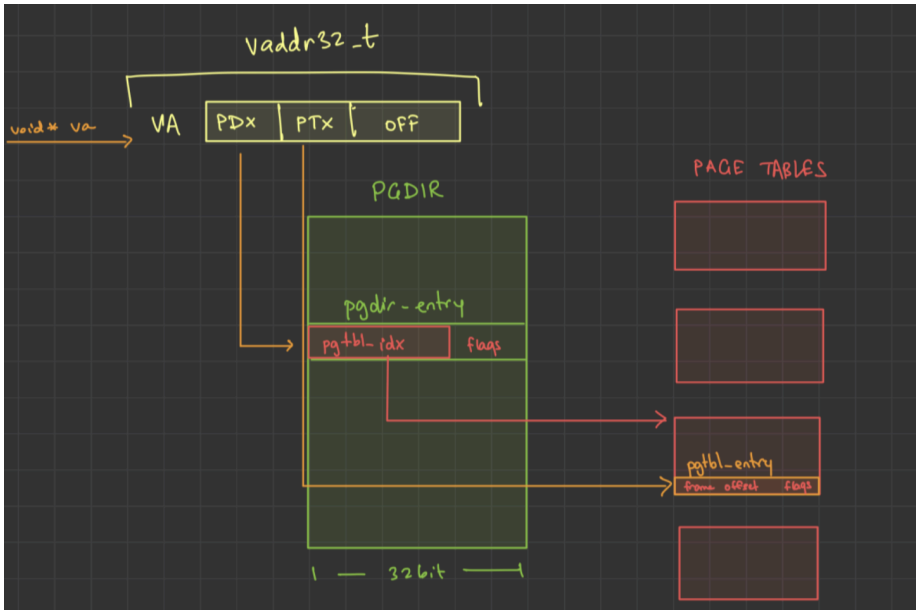

Fig 6: Parsing a virtual address and scanning through a 2-level page table to locate its corresponding page table entry.

Using the `PDX` component of the `vaddr32_t`, retrieve 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 (format discussed in `map_page`) can be stripped away by using a bitwise-`AND` with a negated `OFFSET` mask (`~OFFSET`). The resulting value represents an offset from the start of physical memory `p_buff`.

The `PTX` component of the `vaddr32_t`, 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.

Allocating Memory

```
void *n_malloc(unsigned int num_bytes)
```

Objective: Given a desired number of bytes, this function allocates them in virtual memory.

Given an input in terms of bytes, 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*` 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.

```
uint32_t v_page_bit_idx = (va_base / PGSIZE) + i;
set_bit(v_bmap, v_page_bit_idx);
```

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.

Freeing Memory

```
void n_free(void *va, int size)
```

Objective: This function frees one or more pages of memory, starting at a given virtual address.

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:

```
uint32_t 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 using `pa_offset = (*pte & ~OFFMASK);` 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: `uint32_t 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.

<u>Copying Data</u>

Objective: 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.

1. `put_data`: transfer data from a user buffer (`void* val`) to `p_buff`
2. `get_data`: transfer data from `p_buff` to the desired user buffer `val`

```
int put_data(void *va, void *val, int size);
void get_data(void *va, void *val, int size);
```

These two functions can be generalized by a helper function:
```
static int copy_data(void* va, void* val, int size, int dir);
```

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` component 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 or equal to than 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 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.

Matrix Multiplication

```
void mat_mult(void *mat1, void *mat2, int size, void *answer)
```

Objective: This function performs the actual work of multiplying two square matrices in an elementwise 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 given here:

```
        mat1              mat2                 answer
      [0] [1] [2]       [0] [1] [2]          [0] [1] [2]
  [0] a   b   c     [0]  j   k   l       [0]
  [1] d   e   f     [1]  m   n   o       [1]
  [2] g   h   i     [2]  p   q   r       [2]

ans[0][1] = ([0][0] * [0][1]) --> (a * k)
          + ([0][1] * [1][1]) --> (b * n)
          + ([0][2] * [2][1]) --> (c * q)
```

Here, the 0th index in the 1st operand and the 1st index in the 2nd operand is `k`. In addition, matrices are represented contiguously in memory like so:

```
0x1000: mat[0][0]
0x1004: mat[0][1]
0x1008: mat[0][2]
0x100C: mat[1][0]
```

```
0x1020: mat[2][2]
```

Thus, in order to access a specific index for a square matrix, an overall offset is calculated each time:

```
void* ans =  answer
            + (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 (`op1` and `op2`) as well.

Once the addresses are calculated, the data stored at these locations is then copied into `unint32_t a and b`, using `get_data`.

An accumulator variable `c` is updated with the product of the two operands elements. At the end of the row-column product-summation, this accumulator value is then written to the respective destination position in the `answer` matrix.

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   // Default number of TLB entries

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;
```

Functions:

```
int TLB_add(void *va, void *pa)
```

This 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 provided from the params. A last_used field is maintained in case an LRU retention policy should be used later. This value is simply tracked using tlb_lookups.

```
pte_t *TLB_check(void *va)
```

This function scans through the TLB, attempting to find the corresponding PTE for a given virtual address without needing to scan through the entire page table in memory, providing a huge time saving (if a cache hit occurs).

```
void print_TLB_missrate(void)
```

This function is for performance tracking: tlb_misses and overall tlb_lookups are maintained through the lifetime of the program. Thus, concluding the benchmark tests, the miss rate can be calculated, simply as a ratio of these two metrics.

Benchmarking

Using the `time ./test` and `time ./mtest` operations, the time and CPU usage of the generated executables (based on the provided benchmark tests) is compared when performing a simple square matrix multiplication. Note that the implementation of the matrix multiplication is outside the scope of this project, and the operation through a triple-nested loop implementation is used to exercise the library.

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 matrix operands.

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%

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, 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.