CS 518 - Project 3: User-level Memory Management

Submitted by Anmol Arora (aa2640) and Raunak Negi (rn444)

Introduction:

The main aim of the project is to generate a user-level page table that can translate virtual addresses to physical addresses by building a multi-level page table. We also designed and implemented Transition Lookaside Buffer(TLB) to reduce the translation cost of translating a virtual address to a physical address using the multi-level page table structure.

Data Structures :

- typedef struct **page_table**:
  Data structures used to store the page table. It has two members one of which is a pointer variable of type void and stores the physical address of the page table. The other stores the index of the page_table.

- typedef struct **tlb_node**{}:
  We used tlb_node struct to represent a node in a Translation Lookaside Buffer (TLB). It contains two members:

    1. unsigned long tag: This member represents the tag associated with a virtual address in the TLB. Tags are used to identify entries in the TLB and are typically derived from certain bits of the virtual address.

    2. void *paddr: This member represents the physical address corresponding to the virtual address specified by the tag. It holds the location in physical memory to which the virtual address is mapped.

Part 1: VIRTUAL MEMORY SYSTEM(70 Points)

1.1 Initialisation

void set_physical_mem():

- The function is responsible for allocating a memory buffer that simulates physical memory malloc. It initializes various data structures, including virtual and physical bitmaps, a page directory, and a Translation Lookaside Buffer (TLB).
- It reserves a contiguous block of physical memory using malloc and calculates addressing bits for page offset, page table, and page directory. The function initializes virtual and physical bitmaps to track page usage, creates a page directory (pg_dir) and initializes it to NULL, and sets up a Translation Lookaside Buffer (TLB).
- Mutex locks ensure thread safety, and the function clears and initializes the bitmaps and page directory.

1.2 Assigning a Virtual address

void *t_malloc(unsigned int num_bytes):

- The function is designed to allocate a specified number of bytes within a 32-bit virtual address space. It utilizes a page-based allocation strategy, where one or more pages are allocated based on the size of the requested memory. The function ensures the initialization of physical and page directory memory structures and employs bitmap-based mechanisms to find available contiguous virtual and physical pages.
- The function attempts to find a contiguous block of free virtual pages using "**get_next_avail()**". If a contiguous block is not available, it returns NULL indicating insufficient space.
- It uses "**get_next_available_physical()"** to find the corresponding number of free physical pages. The corresponding available physical pages need not be contiguous. If the required number of physical pages is not available, it returns NULL.
- Finally, it maps the virtual pages to corresponding physical pages using the "**page_map()"** and returns the virtual address of the first allocated page.

int *get_next_avail(int num_pages):

- This function is responsible for finding the next available page(s) in the virtual address space. It takes the number of pages (num_pages) as an argument and returns a pointer variable. The returned pointer stores the virtual index(es) of the next available page(s).

- The function utilizes a virtual bitmap to track page availability where each bit represents a page. A page is identified using the index of the bit which is equivalent to the index of the page in the virtual address space.
- It locks the virtual bitmap to prevent changes during the search for available pages.
- The function iterates through the virtual pages and identifies contiguous free pages.
- If a single page is requested, the function returns a pointer to an array containing the virtual index.
- For multiple pages, it returns a pointer to an array with each element storing a virtual index.
- The virtual bitmap is updated accordingly to mark the allocated pages as used.
- It returns NULL in case the contiguous blocks of free pages are not available.

int *get_next_available_physical(int num_pages):

- This function is designed to find the next available page(s) in physical memory.
- It takes the number of pages (num_pages) as an argument and returns a pointer variable. The returned pointer stores the physical index(es) of the next available page(s).
- The function uses a physical bitmap to track page availability in physical memory. A page is identified using the index of the bit which is equivalent to the index of the page in the physical memory.
- It locks the physical bitmap to prevent changes during the search for available pages.
- The function iterates through the physical pages and identifies free pages. The function returns a pointer to an array with each element storing a physical index.
- The physical bitmap is updated to mark the allocated pages as used.
- It handles cases where contiguous blocks of free pages are not available and returns NULL.

int page_map(pde_t pgdir, void *va, void *pa):

- The function is designed to set a page table entry for a given virtual address in the context of a page directory. It checks if there is an existing mapping for the virtual address in the page directory.
- The function begins by obtaining the page directory and page table indices using **get_pd_bits()** and **get_pt_bits()** functions.
- It locks the page directory using a mutex to ensure exclusive access during the mapping operation
- If no mapping exists **(pgdir[pg_dir_entry_index] == NULL)**, it allocates a new page table, sets the mapping, and adds a TLB entry.
- If a mapping already exists, it simply creates a new page table entry and updates the TLB.
- The function returns 0 upon successful mapping. It returns -1 if an error occurs, although the comments indicate that this case may not occur.
- The function unlocks the page directory mutex after completing the mapping operation.

unsigned int createMask(unsigned start_index, unsigned end_index):

- The function generates a bitmask based on the specified start and end indices. It iterates through the bit positions from start_index to end_index - 1 and sets the corresponding bits in the bitmask. The resulting bitmask is then returned.

int get_offset_bits(void *virt_addr):

- The function calculates the offset bits for a given virtual address. It uses the createMask function to create a bitmask covering the offset bits. The function then performs a bitwise AND operation between the virtual address and the bitmask to extract the offset bits. The result is returned as an integer representing the offset.

int get_pt_bits(void *virt_addr):

- The function retrieves the internal bits for page table entries (i.e., the index inside the table) from a given virtual address. It uses **createMask()** to generate a bitmask covering the internal bits, performs a bitwise AND operation, and shifts the result to obtain the desired bits. The function returns the internal bits as an integer.

int get_pd_bits(void * virt_addr):

- The function obtains the external bits for page directory entries (i.e., the index inside the directory) from a specified virtual address. It utilizes **createMask()** to create a bitmask covering the external bits, performs a bitwise **AND** operation, and shifts the result to extract the desired bits. The function returns the external bits as an integer.

void put_value(void *va, void *val, int size):

- The function is designed to copy data from a source pointer (val) to physical memory pages using a given virtual address (va). The function handles both small and large data sizes, taking into account cases where the data to be copied exceeds the size of a single page.
- If the size is less than the page size (**PGSIZE**), the function uses the **translate()** function to find the physical page corresponding to the virtual address and then uses **memcpy** to copy the contents of **val** to the physical page.
- If the size is greater than or equal to the page size, the function iteratively finds multiple pages using the **translate()** function and copies the data in chunks to cover the entire specified size.
- The function returns 0 if the operation is successful and -1 otherwise.

void get_value(void *va, void *val, int size):

- The function is responsible for retrieving data from physical memory pages and copying it to a destination pointer (val) using a given virtual address (va). Similar to put_value, this function handles both small and large data sizes.
- If the size is less than the page size (**PGSIZE**), the function uses the **translate()** function to find the physical page corresponding to the virtual address and then uses **memcpy** to copy the contents from the physical page to **val**.
- If the size is greater than or equal to the page size, the function iteratively finds multiple pages using the **translate()** function and copies the data in chunks to cover the entire specified size.

1.3 Retrieving a Physical Address

void *translate(pde_t pgdir, void *va):

- The function performs the translation of a virtual address to its corresponding physical address using a two-level page table.
- It checks the TLB first for a cached translation and returns the physical address if found. If the TLB lookup fails, it proceeds to access the page directory and page table entries. The function checks for the existence of mappings and calculates the physical address by combining the page directory index, page table index, and offset.
- The calculated physical address is then added to the TLB for future use. The function incorporates mutex locks to ensure thread safety during virtual and physical memory access.
- Additionally, it handles cases where the page directory, page table, or the specified virtual address is invalid.

void* get_virtual_address(int virtual_idx):

- The function is used to calculate the virtual address of a page based on its index in the virtual space. It takes a virtual index as input and returns the corresponding virtual address. The calculation involves multiplying the virtual index by the page size (**PGSIZE**) and adding the starting address of the virtual memory space.

void* get_physical_address(int physical_index):

- The function calculates the physical address of a page given its index in physical memory. It takes a physical index as input, multiplies it by the page size (**PGSIZE**), and

adds the starting address of the physical memory space (**phys_mem**). The resulting physical address is then returned.

int get_virtual_index(void *virt_addr):

- The function is responsible for calculating the virtual index of a page given its virtual address in the virtual space. It takes a virtual address as input, subtracts the starting address of the virtual memory space (**PGSIZE**), and divides the result by the page size (PGSIZE). The function returns the virtual index.

int get_physical_index(void *physical_address):

- The function calculates the physical index of a page given its physical address in physical memory. It takes a physical address as input, subtracts the starting address of the physical memory space (**phys_mem**), and divides the result by the page size (**PGSIZE**). The function then returns the physical index.

1.4 Free the memory

int t_free(void *va, int size):

- The function is responsible for releasing one or more memory pages starting from a given virtual address (va). It also takes a size parameter indicating the total size of memory to be freed.
- The function checks the validity of the memory pages, frees corresponding page table entries, updates the virtual and physical bitmaps, and removes translations from the TLB.
- If all pages in the specified range are valid, the function returns 1; otherwise, it returns 0.

int all_pages_valid(int start_index, int end_index):

- The function checks if all virtual pages in a specified range (from **start_index** to **end_index**) are already occupied and have valid mappings.
- It uses the virtual bitmap and the **is_valid()** function to perform this check. The function returns 0 if any of the pages are not valid, and 1 if all pages are valid.

int is_valid(void *va):

- The function checks whether a given virtual address (va) has a valid mapping to a physical address. It verifies the existence of both the page directory entry (PDE) and the page table entry (PTE) for the provided virtual address. The function returns 1 if the mapping is valid and 0 otherwise.

void free_page(void *va):

- The free_page function is responsible for releasing a memory page or a range of pages identified by a virtual address (va).
- Determine the directory and table entry indices corresponding to the provided virtual address.
- Retrieve the physical address associated with the virtual address.
- Derive the physical index from the physical address.
- Clear the corresponding bit in the physical bitmap to mark the physical page as free.
- Clear the page table entry associated with the virtual address.

Part 2: TLB (20 Points)

int add_TLB(void *virtual_addr, void *physical_addr):

- The C function add_TLB is responsible for adding an entry to the Translation Lookaside Buffer (TLB) for a given virtual address and its corresponding physical address.
- The function begins by locking the TLB using a mutex (lock_tlb) to ensure thread safety during the update.
- It calculates the TLB index and tag for the provided virtual address using the get_tlb_index and get_tlb_tag functions, respectively.
- The function then updates the TLB entry at the calculated index with the computed tag and the provided physical address.
- After the TLB update, the TLB lock is released using pthread_mutex_unlock. The function returns 0, indicating a successful addition to the TLB.

void* check_TLB(void *va):

- The C function check_TLB performs a Translation Lookaside Buffer (TLB) lookup for a given virtual address (va). It starts by acquiring a lock on the TLB using a mutex (lock_tlb).
- The TLB index and tag for the provided virtual address are then obtained using the get_tlb_index and get_tlb_tag functions, respectively.
- The function checks if the TLB entry at the calculated index matches the calculated tag. If there is a match, it means that a TLB hit occurred, and the corresponding physical address (pa) is retrieved from the TLB entry.
- The TLB lock is then released, and the physical address is returned.
- If no match is found in the TLB, the TLB lock is released, and the function returns NULL, indicating that no mapping was found in the TLB.

void print_TLB_missrate():

- The C function print_TLB_missrate calculates and prints the TLB miss rate.
- It first initializes a variable miss_rate to zero. The function then prints the count of TLB misses (count_miss_tlb) and the total number of translation calls (count_trans_call).

- Subsequently, it calculates the miss rate by dividing the count of TLB misses by the total translation call count.
- Finally, it prints the TLB miss rate as a percentage using printf to the standard error stream.

int get_tlb_index(void *virt_addr):

- This function calculates the Translation Lookaside Buffer (TLB) index for a given virtual address.
- The TLB index is derived by applying a bitmask to the specified bits within the virtual address.
- The bitmask is created using the function createMask, which takes two parameters (off_bits and tlb_index_bits) to determine the range of bits for the mask.
- The function then performs a bitwise AND operation between the virtual address and the created bitmask. The result is right-shifted by the value of off_bits to obtain the final TLB index.

unsigned long get_tlb_tag(void *virt_addr):

- This function is designed to extract the TLB tag from a given virtual address. It uses a bitmask (tlb_tag_mask) created by the createMask function to isolate the relevant bits for the TLB tag.
- The bitmask is defined by the range from off_bits + tlb_index_bits to the 31st bit (32 bits in total).
- The function then performs a bitwise AND operation between the virtual address and the bitmask.
- The result is right-shifted by the sum of off_bits and tlb_index_bits to obtain the final TLB tag.

Part 3: Results and Analysis

Benchmark results of our **test.c** and **multi_test.c** file with different configurations of PGSIZE and num_threads.

Page_Size (PGSIZE)	Miss Rate (%)
4096	0.750000
8192	0.750000
12288	0.750000

test.c

Page_Size (PGSIZE)	Threads(num_threads)	Miss Rate (%)
4096	15	0.837989
4096	30	0.874126
4096	45	0.885478
8192	15	0.840336
8192	30	0.878735
8192	45	0.889504

multi_test.c

Part 4: Collaboration and References

1. Recitations
2. Piazza Class discussion
3. Links in the write up