

Address Spaces, Address Translation, and Paging in C

Asher Chakupa

Fredrick Kairie Njoki

Richard Nana Kweenu Quayson

Department of Computer Science and Information Systems, Ashesi University

CS433_B: Mid-Sem (A Group Mini Project)

Dr Tatenda Kavu

March 01, 2024

Table of Contents

1. DESIGN:.....	3
1.1. Memory sizes:	3
1.2. Page sizes:	3
1.3. Data Structures:.....	4
1.3.1. Physical and Virtual Memory.....	4
1.3.2. Process	5
1.3.3. MasterPageTable	6
1.3.4. SecondaryPageTable	6
1.3.5. PageTableEntry	6
1.4. Functions:.....	7
1.4.1. Address Translation.....	7
1.4.2. Page Allocation and Deallocation	7
1.4.3. Page Faults	9
1.4.4. Free and Allocated Frames.....	11
2. IMPLEMENTATION:	12
2.1. The Package:	12
2.1.1. Processes	12
2.1.2. Process Allocation.....	14
2.1.3. Memory Management	14
2.2. Memory Address Translation:	16
2.3. Page Table:	18
2.4. Error Handling:	19
2.5. Accounting Information (Statistics):	20
3. ANALYSIS AND OPTIMISATION:	23
3.1. System Evaluation:	23

1. DESIGN:

1.1. Memory sizes:

In defining the memory configuration, we specified 256MB for the virtual memory space size and 128MB for the physical memory space size. Given that memory sizes are often represented as bytes in C, with 1 MB representing 2^{20} bytes, the unsigned long long (ULL) data type was used to define the memory space sizes as constants. This was to ensure that the constant is treated as an unsigned 64-bit integer, large enough to represent even 1GB and more.

```
#ifndef MEMORY_CONFIG_H
#define MEMORY_CONFIG_H

#define KB (1024)
#define MB (1024 * KB)
#define GB (1024 * MB)

// virtual memory space size: 256MB
#define VIRTUAL_MEMORY_SIZE (256ULL * MB)

// physical memory space size: 128MB
#define PHYSICAL_MEMORY_SIZE (128ULL * MB)
```

1.2. Page sizes:

The page size defined in our implementation was 4KB. This was to ensure a balance between the granularity of memory management, the overhead of managing many pages and possible internal fragmentation due to large page size. Every page was also divided into 4 different chunks of size 1KB each to account for some degree of offset. The number of pages and frames was thus:

$$PAGE_{SIZE} = \frac{VIRTUAL_MEMORY_SIZE}{PAGE_{SIZE}} \qquad \qquad \qquad FRAME_{SIZE} = \frac{PHYSICAL_MEMORY_SIZE}{PAGE_{SIZE}}$$

```
// page size: 4KB
#define PAGE_SIZE (4 * KB)
```

```
// frame size: 4KB
#define FRAME_SIZE PAGE_SIZE

// number of pages
#define NUM_PAGES (VIRTUAL_MEMORY_SIZE / PAGE_SIZE)

// number of frames
#define NUM_FRAMES (PHYSICAL_MEMORY_SIZE / PAGE_SIZE)

// chunk size: 1KB
#define CHUNK_SIZE (KB)
```

1.3. Data Structures:

In representing the virtual and physical memory, processes, and their corresponding master tables, we used the *typedef struct* feature of C to create structures.

1.3.1. Physical and Virtual Memory

The virtual memory structure has an array of pages that reference a Page structure and an integer `remaining_memory` attribute to keep track of the remaining memory size in the virtual memory after allocating some memory to processes. Each Page had an `id`, `is_allocated` (to determine whether all chunks in the page have been allocated), and `chunks` (a reference to a Chunk structure, each of size 1KB). Consequently, the Chunk structure has an `offset` and an `is_allocated` attribute.

```
#ifndef VIRTUAL_MEMORY_H
#define VIRTUAL_MEMORY_H

// Define the Chunk structure
typedef struct Chunk {
    int offset;           // Offset within the page, from 0 to PAGE_SIZE - 1
    int is_allocated;     // 1 if the chunk is allocated, 0 otherwise
} Chunk;

// Define the Page structure
typedef struct Page {
    int id;               // Page identifier
    Chunk chunks[PAGE_SIZE / KB]; // Array of chunks within the page
    int is_allocated;     // 1 if all chunks in the page are allocated,
                          // 0 otherwise
}
```

```

} Page;

// Define the VirtualMemory structure
typedef struct VirtualMemory {
    Page pages[NUM_PAGES];           // Array of pages in virtual memory
    int remaining_memory;             // Remaining memory in virtual memory
} VirtualMemory;

#endif // VIRTUAL_MEMORY_H

```

Moreover, the physical memory structure followed the same approach in representing the Frames in the PhysicalMemory, and the Chunks in the Frames.

```

#ifndef PHYSICAL_MEMORY_H
#define PHYSICAL_MEMORY_H

// Define the Frame structure
typedef struct Frame {
    int id;                          // Frame identifier
    Chunk chunks[FRAME_SIZE / KB];   // Array of chunks within the frame
    int is_allocated;                 // 1 if all chunks in the frame are
    // allocated, 0 otherwise
} Frame;

// Define the PhysicalMemory structure
typedef struct PhysicalMemory {
    Frame frames[NUM_FRAMES];         // Array of frames in physical memory
    int remaining_memory;             // Remaining memory in physical memory
} PhysicalMemory;

#endif // PHYSICAL_MEMORY_H

```

1.3.2. Process

The process structure has three attributes: an int id representing the id of the process, an int memory_size representing the amount of memory the process requires and a MasterPageTable* mpt pointer representing a reference to a MasterPageTable.

```

typedef struct Process {
    int id;
    int memory_size;                 // Total memory size of the process
    MasterPageTable* mpt;            // Pointer to the MasterPageTable
} Process;

```

1.3.3. MasterPageTable

The page table implemented uses a hierarchical approach such that there is a MasterPageTable which references an array of secondary page tables. The MasterPageTable structure stores an array of SecondaryPageTable pointers, tables and an int count representing the number of SecondaryPageTables.

```
typedef struct MasterPageTable {  
    SecondaryPageTable** tables;    // Array of pointers to SecondaryPageTable  
    int count;                      // Number of SecondaryPageTable pointers  
} MasterPageTable;
```

1.3.4. SecondaryPageTable

Every SecondaryPageTable is of size 4MB and as such, the number of SecondaryPageTables in a MasterPageTable is determined by the memory_size requirement of the process and the size of each SecondaryPageTable (4MB). The SecondaryPageTable has two attributes, an int size of the SecondaryPageTable (4MB or less) and an array of PageTableEntry pointers with the number of PageTableEntry pointers determined by the size of the SecondaryPageTable and the number of Pages needed to represent the process.

```
#define SECONDARY_TABLE_SIZE (4 * MB)  
  
typedef struct SecondaryPageTable {  
    PageTableEntry* entries;    // Dynamic array of PageTableEntry  
    int size;                   // Size of this secondary page table, in bytes  
} SecondaryPageTable;
```

1.3.5. PageTableEntry

The PageTableEntry structure has four attributes: an int page_num which is a reference to a Page's id; an int frame_num which references a Frame's id; a Boolean is_valid which indicates whether a given entry is valid or not and an int array chunks, which represents the ids of the Page chunks used to represent the process.

```
typedef struct PageTableEntry {
    int page_num;           // ID of the page in the virtual memory
    int frame_num;          // ID in a Frame in the physical memory
    bool is_valid;          // Indicates if the entry is valid
    int chunks[PAGE_SIZE / KB]; // List of chunk IDs used to store the process
} PageTableEntry;
```

1.4. Functions:

The implementation of the memory management system required multiple number of functions and the primary functions includes:

1. main.c – to run all the dependencies, and the defined functions within them
2. memory_config.h
3. page_table.c
4. physical_memory.c
5. virtual_memory.c

1.4.1. Address Translation

1.4.2. Page Allocation and Deallocation

The function below handles page allocation. IT iterates through the 'MasterPageTable' fo the process to find the pages needing allocation. It then loops through each 'SecondaryPageTable' and then through each entry in that table.

For each 'PageTableEntry', it checks if 'frame_num' is equal to -1, which indicates that the page is not yet allocated in physical memory. If a page is not allocated, it calls the 'findFreeFrame()' function to find a free frame in physical memory. If a frame is available ('findFreeFrame()' returns a valid frame index), it assigns the free frame to the 'frame_num' of the 'PageTableEntry', marks the frame as allocated in the 'PhysicalMemory' structure, and prints a message indicating the allocation.

```
void allocatePagesToPhysicalMemory(Process* process, PhysicalMemory* pm) {
    // Iterate through the MasterPageTable to find the pages needing allocation
    for (int i = 0; i < process->mpt->count; i++) {
        SecondaryPageTable* spt = process->mpt->tables[i];
        for (int j = 0; j < (spt->size + PAGE_SIZE - 1) / PAGE_SIZE; j++) {
```

```

PageTableEntry* entry = &spt->entries[j];
if (entry->frame_num == -1) { // Page is not yet allocated
    // Find a free frame in physical memory
    int free_frame = findFreeFrame(pm);
    if (free_frame != -1) {
        entry->frame_num = free_frame; // Assign the free frame to the page
        pm->frames[free_frame].is_allocated = 1; // Mark the frame as allocated
        printf("Allocated frame %d for page ID %d in process ID %d.\n", free_frame, entry-
>page_num, process->id);
    } else {
        printf("No free frame available for page ID %d in process ID %d.\n", entry-
>page_num, process->id);
        return; // Return if no free frame is available
    }
}
}
}
}
}
}

```

The function below handles page deallocation.

```

void deallocatePagesFromPhysicalMemory(Process* process, PhysicalMemory* pm) {
    // Iterate through the MasterPageTable to find the pages to deallocate
    for (int i = 0; i < process->mpt->count; i++) {
        SecondaryPageTable* spt = process->mpt->tables[i];
        for (int j = 0; j < (spt->size + PAGE_SIZE - 1) / PAGE_SIZE; j++) {
            PageTableEntry* entry = &spt->entries[j];
            if (entry->frame_num != -1) { // Page is allocated
                int frame_num = entry->frame_num;
                entry->frame_num = -1; // Reset the frame_num of the PageTableEntry
                pm->frames[frame_num].is_allocated = 0; // Mark the frame as deallocated
            }
        }
    }
}

```



```

        printf("Deallocated frame %d for page ID %d in process ID %d.\n", frame_num, entry-
>page_num, process->id);
    }
}
}
}

```

The function iterates through the ‘MasterPageTable’ of the process to find the pages needing deallocation. It loops through each ‘SecondaryPageTable’ and then through each entry in that table. For each ‘PageTableEntry’, it checks if ‘frame_num’ is equal to -1, which indicates that the page is not yet allocated in physical memory. If a page is allocated, it retrieves the frame number from ‘frame_num’ of the ‘PageTableEntry’, resets ‘frame_num’ to -1, marks the frame as deallocated in the ‘PhysicalMemory’ structure.

1.4.3. Page Faults

During memory access, page faults can occur. The function below handles page faults.

// Function to access a process's frame in physical memory

```

int accessMemory(Process* process, int page_id) {
    num_accesses++; // Increment the number of memory access attempts

    // Iterate through the MasterPageTable to find the PageTableEntry for the given page_id
    for (int i = 0; i < process->mpt->count; i++) {
        SecondaryPageTable* spt = process->mpt->tables[i];
        for (int j = 0; j < (spt->size + PAGE_SIZE - 1) / PAGE_SIZE; j++) {
            PageTableEntry* entry = &spt->entries[j];
            if (entry->page_num == page_id) { // Found the corresponding PageTableEntry
                if (entry->frame_num == -1) { // Page fault occurs if frame_num is -1
                    page_faults++; // Increment the global page_faults counter
                    printf("Page fault occurred for page ID %d in process ID %d.\n", page_id, process->id);
                    return -1;
                }
            }
        }
    }
}

```

```
} else {  
    // Successfully accessed the page in physical memory  
    printf("Successfully accessed frame %d for page ID %d in process ID %d.\n", entry->frame_num, page_id,  
        process->id);  
}  
return 0; // Exit after handling the page access  
}  
}  
}  
  
// If the page_id was not found in any PageTableEntry, it's considered an invalid access  
printf("Invalid page ID %d access attempt in process ID %d.\n", page_id, process->id);  
}  
  
printf("\nEnter the Page ID you wish to access: ");  
int pageId;  
scanf("%d", &pageId);  
int accessResult = accessMemory(process, pageId); // This function internally increments num_accesses  
  
// result of access is -1 if page fault occurs,  
// handle it by asking the user if they want to allocate the page to physical memory  
// if yes, call allocatePagesToPhysicalMemory function  
  
if (accessResult == -1) {  
    printf("Do you want to allocate the page to physical memory? (y/n): ");  
    char choice;  
    scanf(" %c", &choice);  
    if (choice == 'y' || choice == 'Y') {  
        allocatePagesToPhysicalMemory(process, pm);  
    }  
}
```

```

} else {
break;
}
} else {
break;
}

// print the physical memory allocation after the handling of page fault
printf("\nPhysical Memory after handling page fault:\n-----");
printAllocatedFrameMemory(pm);

} else {
printf("\nProcess ID %d not found.\n", pid3);
}
break;

```

The ‘accessMemory()’ function is called with the process and the page ID that needs to be accessed. It first iterates through the Master Page Table of the process to find the corresponding Page Table Entry (PTE) for the given page ID. If the PTE for the page ID is found, it checks if the ‘frame_num’ is –1. If it is, it indicates that the page is not currently in physical memory, resulting in a page fault. When the page fault occurs (entry->frame_num = -1), the function increments the global ‘page_faults’ counter. The user is prompted whether they want to allocate the page to physical memory. If they choose to allocate the page to physical memory, the ‘allocatePagesToPhysicalMemory()’ function is called and the page fault gets handled.

1.4.4. Free and Allocated Frames

The free and allocated frames are handled under the page allocation and deallocation.

2. IMPLEMENTATION:

2.1. The Package:

2.1.1. Processes

To initiate the processes, we create the process using **create_process** function which creates a process in a virtual memory environment. It ensures that the virtual memory is sufficient for the process and allocates memory for the process within the virtual memory. The function calculates the number of secondary tables needed based on the memory size of the process and allocates memory for the master page table accordingly. It then initializes page table entries for each page in the process's memory, allocating virtual memory pages as needed. Finally, it returns a pointer to the created process if successful, or NULL if there's an error.

```
// Updated to reflect new allocation logic
Process* create_process(int id, int memory_size, VirtualMemory* vm) {
    if (!vm || memory_size <= 0 || memory_size > vm->remaining_memory) {
        printf("\nInsufficient virtual memory to create process.\n");
        return NULL;
    }

    Process* process = (Process*)malloc(sizeof(Process));
    if (!process) return NULL;

    process->id = id;
    process->memory_size = memory_size;
    vm->remaining_memory -= memory_size; // Deduct the allocated memory from the
    remaining virtual memory

    int numSecondaryTables = (memory_size + SECONDARY_TABLE_SIZE - 1) /
    SECONDARY_TABLE_SIZE;
    process->mpt = (MasterPageTable*)malloc(sizeof(MasterPageTable));
    if (!process->mpt) {
        free(process);
        return NULL;
    }

    process->mpt->tables = (SecondaryPageTable**)malloc(numSecondaryTables *
    sizeof(SecondaryPageTable*));
    process->mpt->count = numSecondaryTables;

    // Allocate memory and initialize PageTableEntries
```

```

    for (int i = 0; i < numSecondaryTables; ++i) {
        int tableSize = (i < numSecondaryTables - 1) ? SECONDARY_TABLE_SIZE :
memory_size - i * SECONDARY_TABLE_SIZE;
        process->mpt->tables[i] = allocateSecondaryPageTable(tableSize);

        // allocate PageTableEntries
        int remaining_memory = memory_size;
        for (int i = 0; i < numSecondaryTables; ++i) {
            int tableSize = (i < numSecondaryTables - 1) ? SECONDARY_TABLE_SIZE :
memory_size - i * SECONDARY_TABLE_SIZE;
            process->mpt->tables[i] = allocateSecondaryPageTable(tableSize);

            int remaining_process_size = tableSize; // Memory remaining to be
allocated in this secondary table
            int allocated_memory = 0; // Track allocated memory in the current
secondary table

            // Calculate the number of pages (PageTableEntries) needed for this
SecondaryPageTable
            int numPagesNeeded = (tableSize + PAGE_SIZE - 1) / PAGE_SIZE;

            for (int pageIndex = 0; pageIndex < numPagesNeeded &&
remaining_process_size > 0; ++pageIndex) {
                int pageID = allocatePage(vm);
                if (pageID == -1) {
                    printf("Failed to allocate enough virtual memory for the
process.\n");

                    // Cleanup code omitted for brevity
                    return NULL;
                }

                // Initialize PageTableEntry for the current page
                PageTableEntry* entry = &process->mpt->tables[i]-
>entries[pageIndex];
                entry->page_num = pageID; // Set the page number
                entry->frame_num = -1; // Assuming no physical frame is allocated
yet
                entry->is_valid = true; // Mark as valid since we're allocating
memory for it

                // Calculate how many chunks are needed for this secondary table
                int chunksNeeded = tableSize / CHUNK_SIZE + (tableSize %
CHUNK_SIZE != 0);

                // Allocate chunks within the allocated page

```

```

        allocateChunksInPage(vm, pageID, chunksNeeded, entry);
        chunksNeeded -= PAGE_SIZE / CHUNK_SIZE; // Update the number of
chunks needed

        // Update remaining_memory
        remaining_memory -= PAGE_SIZE;
        if (remaining_memory <= 0) break; // Stop if we've allocated
enough memory
    }

}

}

printf("\nProcess %d created successfully with %d bytes of memory.\n", id,
memory_size);
return process;
}

```

2.1.2. Process Allocation

Process allocation is handled when a process is created and assigned memory.

2.1.3. Memory Management

In managing the memory, we initially implemented the Frame structure for either the physical or the virtual memory to be implemented. A struct data structure implemented the Frame and with attributes namely id, an array of chunks within the frame and int is_allocated to hold a value integer value 1 if allocated and 0 otherwise.

Frame:

```

// Define the Frame structure
typedef struct Frame {
    int id; // Frame identifier
    Chunk chunks[FRAME_SIZE / KB]; // Array of chunks within the frame
    int is_allocated; // 1 if all chunks in the frame are allocated, 0 otherwise
} Frame;

```

An implementation of the structure of the PhysicalMemory was implemented using a struct data structure allocating the number of frames the processes will run from. It keeps track of the remaining memory when memory is freed or assigned to a process in the Physical Memory.

```

typedef struct PhysicalMemory {
    Frame frames[NUM_FRAMES]; // Array of frames in physical memory
    int remaining_memory; // Remaining memory in physical memory
}

```

```
} PhysicalMemory;
```

In addition, an initialization of both the physical and virtual memory was implemented respectively. The physical memory initialization utilizes malloc to dynamically allocate the memory spaces referenced by vm.

```
// Function to initialize physical memory
PhysicalMemory* initializePhysicalMemory() {
    PhysicalMemory* pm = malloc(sizeof(PhysicalMemory)); // Allocate memory for
the physical memory structure
    if (pm == NULL) {
        // Handle memory allocation failure
        return NULL;
    }

    for (unsigned long long i = 0; i < NUM_FRAMES; i++) {
        pm->frames[i].id = i; // Set frame ID
    }

    // remaining_memory = PHYSICAL_MEMORY_SIZE; // Initialize remaining memory in
physical memory
    pm->remaining_memory = PHYSICAL_MEMORY_SIZE;
    return pm;
}
```

```
// Function to initialize virtual memory
VirtualMemory* initializeVirtualMemory() {
    VirtualMemory* vm = malloc(sizeof(VirtualMemory)); // Allocate memory for the
virtual memory structure
    if (vm == NULL) {
        // Handle memory allocation failure
        return NULL;
    }

    for (unsigned long long i = 0; i < NUM_PAGES; i++) {
        vm->pages[i].id = i; // Set page ID
    }

    // remaining_memory = VIRTUAL_MEMORY_SIZE; // Initialize remaining memory in
virtual memory
    vm->remaining_memory = VIRTUAL_MEMORY_SIZE;
    return vm;
}
```

```
}
```

The memory that has been allocated to processes, when a process is done executing can be freed using the `freeMemory` function which uses the `free` function. The function handles either the physical or virtual memory.

```
// Function to free the allocated memory for virtual and physical memory
void freeMemory(VirtualMemory* vm, PhysicalMemory* pm) {
    if (vm != NULL) {
        free(vm); // Free virtual memory
    }
    if (pm != NULL) {
        free(pm); // Free physical memory
    }
}
```

2.2. Memory Address Translation:

To translate the virtual addresses to physical address the **`translateVirtualToPhysicalAddress`** function translates virtual addresses of a process to physical addresses. It parses the virtual address to extract page ID and offset. Then, it looks up the page in the process's page table to find its frame number. If the page is not in physical memory, it prompts the user to allocate it. After handling the page fault, it prints the physical memory allocation and physical addresses for the pages in the process. If the page is in physical memory, it directly prints the physical address for the given virtual address.

```
// Function to translate all virtual addresses of a process to physical addresses
void translateVirtualToPhysicalAddress(PhysicalMemory* pm, char* virtualAddress,
int processId) {
    int pageId, offset;
    char physicalAddress[20]; // Assuming this is large enough for the
physical address format

    // Validate and parse the virtual address
    if (sscanf(virtualAddress, "0vp%d%s", &pageId, &offset) != 2) {
        printf("Invalid virtual address format.\n");
        return;
    }

    Process* process = findProcessById(processId);
    if (!process) {
```



```

        printf("Process with ID %d not found.\n", processId);
        return;
    }

    // Lookup the page in the process's page table to find its frame number
    int frameNum = -1;
    for (int i = 0; i < process->mpt->count; i++) {
        for (int j = 0; j < (process->mpt->tables[i]->size + PAGE_SIZE - 1) /
PAGE_SIZE; j++) {
            if (process->mpt->tables[i]->entries[j].page_num == pageId) {
                frameNum = process->mpt->tables[i]->entries[j].frame_num;
                break;
            }
        }
        if (frameNum != -1) break;
    }

    // If the frame number is -1, the page is not in physical memory
    if (frameNum == -1) {
        page_faults++;        // Assume entire process loading counts as one page
        fault for simplicity
        printf("Page ID %d not found in physical memory for process ID %d.\n",
pageId, processId);
        printf("Do you want to allocate the page to physical memory? (y/n): ");
        char choice;
        scanf(" %c", &choice);
        if (choice == 'y' || choice == 'Y') {
            allocatePagesToPhysicalMemory(process, pm);
        } else {
            return;
        }
    }

    // Print the physical memory allocation after handling the page fault
    printf("\nPhysical Memory after handling page fault:\n-----
-----");
    printAllocatedFrameMemory(pm);

    printf("\n");        // newline
    // Print the physical address for the pages in the process
    for (int i = 0; i < process->mpt->count; i++) {
        for (int j = 0; j < (process->mpt->tables[i]->size + PAGE_SIZE - 1) /
PAGE_SIZE; j++) {
            if (process->mpt->tables[i]->entries[j].frame_num != -1) {

```

```

        printf("Physical address for virtual address '0vp%s%d' of
process ID %d: 0pf%s%d\n", process->mpt->tables[i]->entries[j].page_num, offset,
processId, process->mpt->tables[i]->entries[j].frame_num, offset);
    }
}
}

} else {
    // Print the physical address for the given virtual address
    printf("Physical address for virtual address '%s' of process ID %d:
0pf%s%d\n", virtualAddress, processId, frameNum, offset);
}
}
}

```

2.3. Page Table:

```

// SecondaryPageTable allocation function
SecondaryPageTable* allocateSecondaryPageTable(int memorySize) {
    int numEntries = (memorySize + PAGE_SIZE - 1) / PAGE_SIZE; // Number of pages
needed
    SecondaryPageTable* spt =
(SecondaryPageTable*)malloc(sizeof(SecondaryPageTable));
    if (!spt) return NULL;

    spt->entries = (PageTableEntry*)malloc(numEntries * sizeof(PageTableEntry));
    if (!spt->entries) {
        free(spt);
        return NULL;
    }

    spt->size = memorySize;
    for (int i = 0; i < numEntries; ++i) {
        spt->entries[i].frame_num = -1; // Initially, no frame is allocated
        spt->entries[i].is_valid = false; // Mark as invalid initially
        for (int j = 0; j < PAGE_SIZE / KB; ++j) {
            spt->entries[i].chunks[j] = -1; // Mark all chunks as unallocated
        }
    }
    return spt;
}

```

2.4. Error Handling:

To ensure that the code does not encounter some loopholes and error withing each error handling was implemented in multiple functions for different checks.

```
// SecondaryPageTable allocation function
SecondaryPageTable* allocateSecondaryPageTable(int memorySize) {
    // Memory allocation for SecondaryPageTable
    SecondaryPageTable* spt =
    (SecondaryPageTable*)malloc(sizeof(SecondaryPageTable));
    if (!spt) return NULL;
```

Checks if memory allocation fails and returns NULL

```
// Function to find and allocate a page in virtual memory, returning the page ID
int allocatePage(VirtualMemory* vm) {
    // Iterates through virtual memory pages
    // ...
    return -1;
}
```

Return -1 if unable to find a free page

```
// Function to translate all virtual addresses of a process to physical addresses
void translateVirtualToPhysicalAddress(PhysicalMemory* pm, char* virtualAddress,
int processId) {
    // ...
    Process* process = findProcessById(processId);
    if (!process) {
        printf("Process with ID %d not found.\n", processId);
        return;
```

Print error message and return if process is not found

// Function to find and allocate a page in virtual memory, returning the page ID

```
int allocatePage(VirtualMemory* vm) {
    // Iterates through virtual memory pages
    // ...
    return -1; // Error handling: Return -1 if unable to find a free page
}
```

2.5. Accounting Information (Statistics):

It is important to keep track of the accounting information and printing out about the statistics of the accounting information, below are functions printing the accounting information about the processes, interaction with memory, the addresses and how they are managing the frames.

printVirtualMemory function prints out “*Virtual Memory is not initialized*” when the virtual memory pointer is not assigned to any virtual memory frame address, it prints whether the page is allocated or not, along with information about each chunk within the page, including its offset and whether it's allocated.

```
// Function to print the values in virtual memory
void printVirtualMemory(const VirtualMemory* vm) {
    if (vm == NULL) {
        printf("\nVirtual Memory is not initialized.\n");
        return;
    }

    printf("Virtual Memory Contents:\n");
    for (unsigned long long i = 0; i < NUM_PAGES; i++) {
        printf("Page %llu (Allocated: %s): ", i, vm->pages[i].is_allocated ?
"Yes" : "No");
        for (int j = 0; j < PAGE_SIZE / KB; j++) {
            printf("Chunk %d (Offset: %d, Allocated: %s), ", j, vm-
>pages[i].chunks[j].offset, vm->pages[i].chunks[j].is_allocated ? "Yes" : "No");
        }
        printf("\n");
    }
}
```

Consequently, the **printPhysicalMemory** function prints out what pages and chunks are being used in our virtual memory. It goes through each page, checking if a page is occupied or not and prints out and any chunks it has. If there are no pages are in use it prints out and request an option for allocation, that is moving a process from virtual memory to the physical memory.

```
// Function to print the values in physical memory
void printPhysicalMemory(const PhysicalMemory* pm) {
    if (pm == NULL) {
        printf("\nPhysical Memory is not initialized.\n");
    }
}
```

```

        return;
    }

    printf("Physical Memory Contents:\n");
    for (unsigned long long i = 0; i < NUM_FRAMES; i++) {
        printf("Frame %llu (Allocated: %s): ", i, pm->frames[i].is_allocated ?
"Yes" : "No");
        for (int j = 0; j < FRAME_SIZE / KB; j++) {
            printf("Chunk %d (Offset: %d, Allocated: %s), ", j, pm-
>frames[i].chunks[j].offset, pm->frames[i].chunks[j].is_allocated ? "Yes" :
"No");
        }
        printf("\n");
    }
}

```

To get information about the allocated virtual memory, **printAllocatedVirtualMemory** took into account of looping through the virtual memory pages and check the allocated chunks. If the allocated virtual memory is found it prints the page and the chunks otherwise it prints out that *No allocated pages in virtual memory*

```

// Function to print allocated pages and their chunks in virtual memory
void printAllocatedVirtualMemory(const VirtualMemory* vm) {
    int allocatedPagesFound = 0;
    for (unsigned long long i = 0; i < NUM_PAGES; i++) {
        if (vm->pages[i].is_allocated) {
            printf("\nAllocated Virtual Page: %d\n", vm->pages[i].id);
            // Print details about allocated chunks within this page
            for (int j = 0; j < PAGE_SIZE / KB; j++) {
                if (vm->pages[i].chunks[j].is_allocated) {
                    printf("\tAllocated Chunk: %d (Offset: %d)\n", j, vm-
>pages[i].chunks[j].offset);
                }
            }
            allocatedPagesFound++;
        }
    }
    if (allocatedPagesFound == 0) {
        printf("\nNo allocated pages in virtual memory.\n");
    }
}

```

On the other hand, the **printAllocatedFrameMemory** deals with the same above aspect but on the physical memory.

```
// Function to print allocated frames and their chunks in physical memory
void printAllocatedFrameMemory(const PhysicalMemory* pm) {
    int allocatedFramesFound = 0;
    for (unsigned long long i = 0; i < NUM_FRAMES; i++) {
        if (pm->frames[i].is_allocated) {
            printf("\nAllocated Physical Frame: %d\n", pm->frames[i].id);
            // Print details about allocated chunks within this frame
            for (int j = 0; j < FRAME_SIZE / KB; j++) {
                if (pm->frames[i].chunks[j].is_allocated) {
                    printf("\tAllocated Chunk: %d (Offset: %d)\n", j, pm->frames[i].chunks[j].offset);
                }
            }
            allocatedFramesFound++;
        }
    }
    if (allocatedFramesFound == 0) {
        printf("\nNo allocated frames in physical memory.\n");
    }
}
```

Finally, to display the comprehensive statistics, the **displayStatistics** function printouts or the necessary statistics displaying about how the memory is managed on the system basing on the processes that are created, executing and idle.

```
// Function to display memory management statistics
void displayStatistics(VirtualMemory* vm, PhysicalMemory* pm) {
    // Calculate hit rate as the ratio of successful accesses to total accesses
    float hitRate = (num_accesses - page_faults) / (float)num_accesses * 100;

    // Calculate the total and remaining memory in both virtual and physical memory spaces
    int totalVirtualMemory = NUM_PAGES * PAGE_SIZE;
    int usedVirtualMemory = totalVirtualMemory - vm->remaining_memory;

    int totalPhysicalMemory = NUM_FRAMES * FRAME_SIZE;
    int usedPhysicalMemory = totalPhysicalMemory - pm->remaining_memory;

    // Display the statistics
    printf("\nMemory Management Statistics:\n");
```

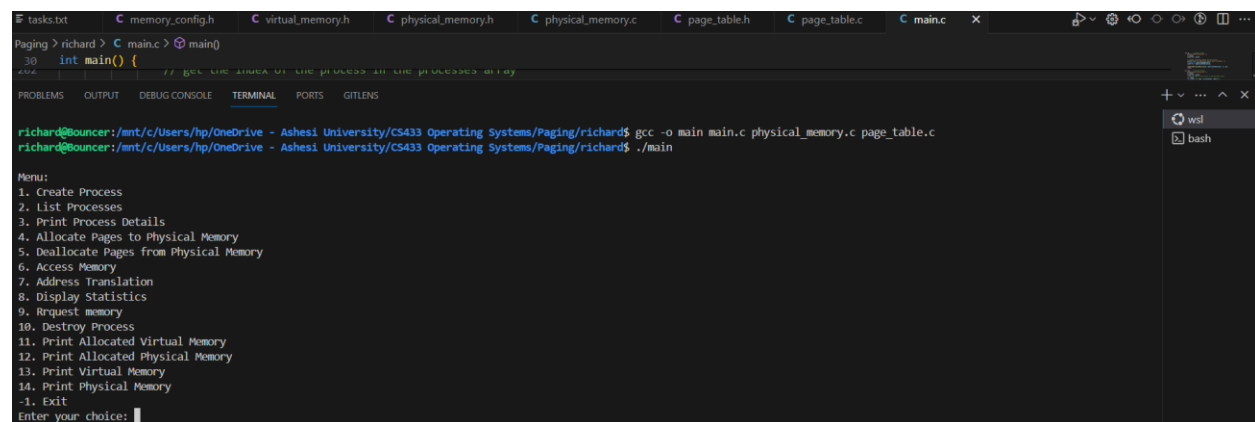
```

    printf("Number of allocated pages in virtual memory: %llu\n", NUM_PAGES -
(vm->remaining_memory / PAGE_SIZE));
    printf("Number of frames in physical memory: %llu\n", NUM_FRAMES - (pm-
>remaining_memory / FRAME_SIZE));
    printf("Number of accesses in physical memory: %d\n", num_accesses);
    printf("Number of page faults: %d\n", page_faults);
    printf("Hit rate: %.2f%%\n", hitRate);
    printf("Total memory used in virtual memory: %d bytes\n", usedVirtualMemory);
    printf("Remaining memory in virtual memory: %d bytes\n", vm-
>remaining_memory);
    printf("Total memory used in physical memory: %d bytes\n",
usedPhysicalMemory);
    printf("Remaining memory in physical memory: %d bytes\n", pm-
>remaining_memory);
}

```

3. ANALYSIS AND OPTIMISATION:

3.1. System Evaluation:



```

tasks.txt  memory_config.h  virtual_memory.h  physical_memory.h  physical_memory.c  page_table.h  page_table.c  main.c
Paging > richard > C main.c > main()
30 int main() {
404 // GET THE ADDRESS OF THE PHYSICAL MEMORY
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS  GITLENS
richard@bouncer: /mnt/c/Users/hp/OneDrive - Ashesi University/CS433 Operating Systems/Paging/richard$ gcc -o main main.c physical_memory.c page_table.c
richard@bouncer: /mnt/c/Users/hp/OneDrive - Ashesi University/CS433 Operating Systems/Paging/richard$ ./main

Menu:
1. Create Process
2. List Processes
3. Print Process Details
4. Allocate Pages to Physical Memory
5. Deallocate Pages from Physical Memory
6. Access Memory
7. Address Translation
8. Display Statistics
9. Request memory
10. Destroy Process
11. Print Allocated Virtual Memory
12. Print Allocated Physical Memory
13. Print Virtual Memory
14. Print Physical Memory
-1. Exit
Enter your choice:

```

```
Enter your choice: 1
Enter process ID: 1
Enter memory size (in bytes): 4096

Process 1 created successfully with 4096 bytes of memory.

Menu:
1. Create Process
2. List Processes
3. Print Process Details
4. Allocate Pages to Physical Memory
5. Deallocate Pages from Physical Memory
6. Access Memory
7. Address Translation
8. Display Statistics
9. Request memory
10. Destroy Process
11. Print Allocated Virtual Memory
12. Print Allocated Physical Memory
13. Print Virtual Memory
14. Print Physical Memory
-1. Exit
Enter your choice: 2

List of processes:
Process ID: 1, Memory Size: 4096
```



```
Enter your choice: 3
Enter process ID: 1
Process {
    id: 1,
    memory_size: 4096 bytes,
    MasterPageTable {
        1: SecondaryPageTable {
            entries: [
                {
                    page_num: 0,
                    frame_num: -1,
                    is_valid: true,
                    chunks: [0, 1, 2, 3]
                }
            ],
            size: 4096 bytes
        },
    },
}
```

```
Enter your choice: 11

Allocated Virtual Page: 0
    Allocated Chunk: 0 (Offset: 0)
    Allocated Chunk: 1 (Offset: 0)
    Allocated Chunk: 2 (Offset: 0)
    Allocated Chunk: 3 (Offset: 0)
```

```
Enter your choice: 12

No allocated frames in physical memory.
```

```
Enter your choice: 4
Enter process ID: 1

Pages allocated to physical memory for process 1.
```

```
Enter your choice: 12
```

```
Allocated Physical Frame: 0
```

```
    Allocated Chunk: 0 (Offset: 0)
```

```
    Allocated Chunk: 1 (Offset: 0)
```

```
    Allocated Chunk: 2 (Offset: 0)
```

```
    Allocated Chunk: 3 (Offset: 0)
```

```
Enter your choice: 5
```

```
Enter process ID: 1
```

```
Pages deallocated from physical memory for process 1.
```

```
Enter your choice: 12
```

```
No allocated frames in physical memory.
```

```
Enter your choice: 6
```

```
Enter Process ID: 1
```

```
Listing all pages for Process ID 1:
```

```
Page ID: 0
```

```
Enter the Page ID you wish to access: 0
```

```
Page fault occurred for page ID 0 in process ID 1.
```

```
Do you want to allocate the page to physical memory? (y/n): 
```

```
Do you want to allocate the page to physical memory? (y/n): y
```

```
Pages allocated to physical memory for process 1.
```

```
Physical Memory after handling page fault:
```

```
-----
```

```
Allocated Physical Frame: 0
```

```
Enter your choice: 7
```

```
Enter Process ID: 1
```

```
Enter the virtual address in the form 0vp<page_id>s<offset> e.g. 0vp01s0 for page 1 and offset 0: 0vp0s0
```

```
Physical address for virtual address '0vp0s0' of process ID 1: 0pf0s0
```

```
Enter your choice: 9
```

```
Enter process ID: 1
```

```
Enter the additional memory size (in bytes): 4096
```

```
Pages deallocated from physical memory for process 1.
```

```
Pages allocated to physical memory for process 1.
```

```
Additional memory allocated to process ID 1. Total memory: 8192 bytes.
```

```
Memory Management Statistics:
```

```
Number of allocated pages in virtual memory: 2
```

```
Number of frames in physical memory: 2
```

```
Number of accesses in physical memory: 3
```

```
Number of page faults: 2
```

```
Hit rate: 33.33%
```

```
Total memory used in virtual memory: 8192 bytes
```

```
Remaining memory in virtual memory: 268427264 bytes
```

```
Total memory used in physical memory: 8192 bytes
```

```
Remaining memory in physical memory: 134209536 bytes
```

```
Enter your choice: 12
```

```
Allocated Physical Frame: 0
```

```
Allocated Physical Frame: 1
```

```
    Allocated Chunk: 0 (Offset: 0)
```

```
    Allocated Chunk: 1 (Offset: 0)
```

```
    Allocated Chunk: 2 (Offset: 0)
```

```
    Allocated Chunk: 3 (Offset: 0)
```

```
Enter your choice: 10
```

```
Enter process ID: 1
```

```
Process ID 1 destroyed and resources freed.
```