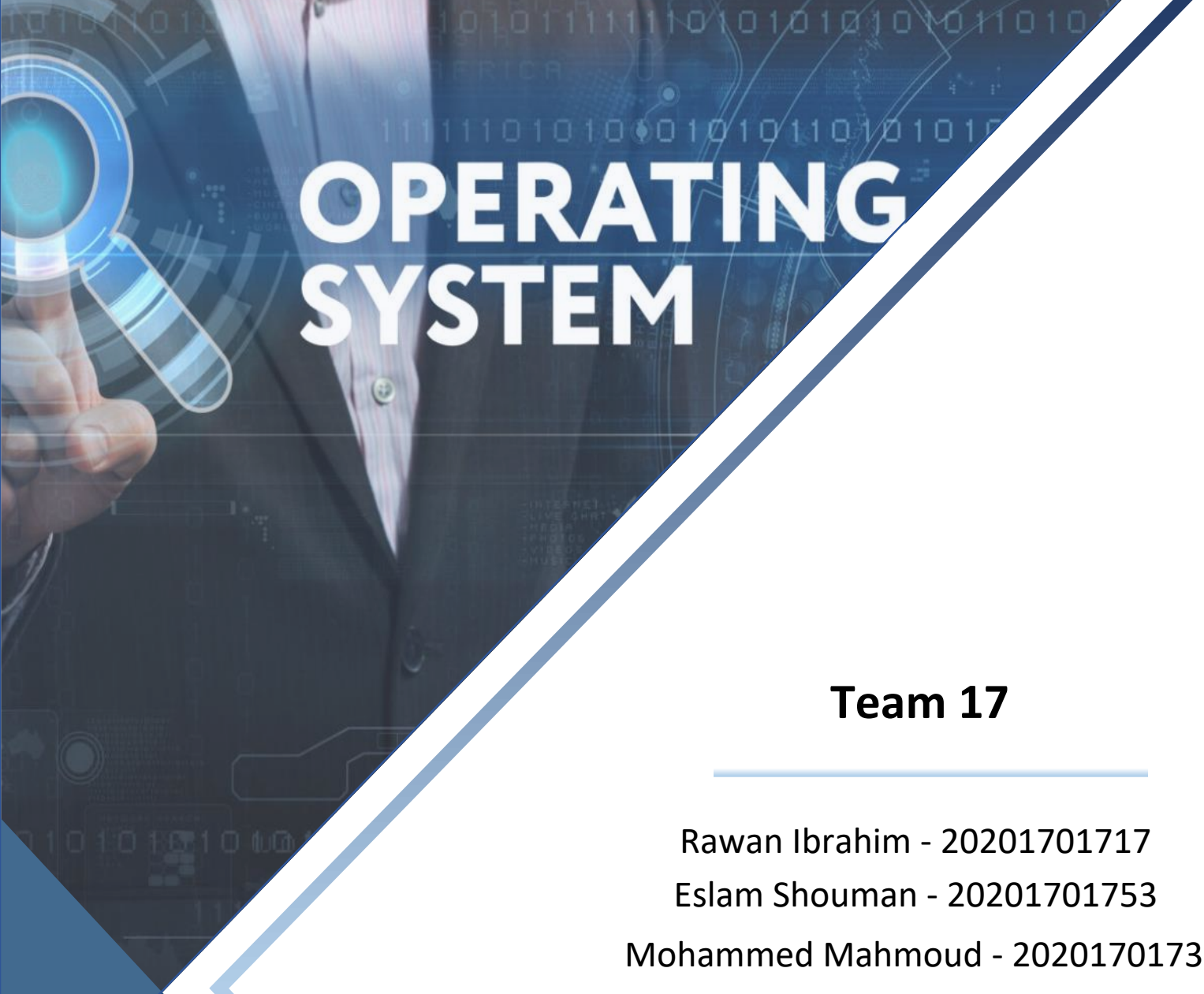


# Team 17

Rawan Ibrahim - 20201701717

Eslam Shouman - 20201701753

Mohammed Mahmoud - 20201701736

**Under the supervision of:**

Dr/Hanan Hindy

Dr/Razan Bayoumi

Dr/Mohammed Raafat

# Contents

# Table of Figures

## Introduction:

The Faculty of Computer and Information Science OS (FOS) project was undertaken with the objective of gaining practical experience and deepening our understanding of operating systems. This report documents the progress and outcomes of the project, which involved the implementation of specific functions within the FOS educational operating system.

The project was divided into two milestones, each focusing on distinct objectives. In the first milestone, our primary goal was to implement essential functionality related to memory management. This involved the development of functions such as "**kmalloc**", "**kfree**", "**kheap_physical_address**", and "**kheap_virtual_address**". These functions enabled the kernel to dynamically allocate and free memory space at runtime, thereby enhancing the efficiency and flexibility of the operating system.

Building upon the achievements of the first milestone, the second milestone aimed to handle page faults during execution and introduce user-level memory management capabilities. Functions implemented during this phase included "**page_fault_handler_with_buffering**", "**malloc**", "**allocate_mem**", "**free**", and "**free_mem_with_buffering**". By applying the MODIFIED CLOCK replacement algorithm, the operating system effectively handled page faults, while user programs were empowered to dynamically allocate and free memory space at runtime.

The implementation of these functions within the FOS operating system provided us with practical insights into operating system concepts and allowed us to apply our theoretical knowledge in a real-world context. Additionally, it contributed to the usability and versatility of the FOS OS, enhancing its educational value for future learners.

In the following sections, we will provide a comprehensive overview of the milestones, detailing the specific functions implemented, the methodologies employed, and the progress achieved in each milestone. By documenting our experiences and accomplishments, this report aims to contribute to the knowledge base in operating system development and serve as a valuable resource for future projects and learning endeavors.

# Overview:

## Milestone 1: Kernel Heap

In the first milestone, our focus was on implementing fundamental memory management functionality within the FOS operating system. The following functions were developed:

1. **Kmalloc:** This function allowed the kernel to dynamically allocate memory space at runtime, providing flexibility for efficient memory management.
2. **Kfree:** With this function, the kernel could release previously allocated memory, freeing up resources and improving memory utilization.
3. **kheap_physical_address:** This function provided the physical address of a given virtual address within the kernel heap, enabling precise memory mapping.
4. **kheap_virtual_address:** By returning the virtual address corresponding to a given physical address within the kernel heap, this function facilitated seamless memory access and manipulation.

The successful implementation of these functions in Milestone 1 established the foundation for advanced memory management capabilities within the FOS operating system.

## Milestone 2: User Heap and Page Fault Handling

In the second milestone, we expanded the functionality of the FOS OS by introducing user-level memory management and robust page fault handling. The following functions were implemented:

1. **page_fault_handler_with_buffering:** This function handled page faults during program execution, utilizing the MODIFIED CLOCK replacement algorithm to efficiently manage memory pages.
2. **Malloc:** Enabling dynamic memory allocation within user programs, this function allowed users to request memory space as needed.
3. **allocate_mem:** With this function, users could allocate memory of a specific size within their programs, promoting flexibility and efficient resource utilization.
4. **Free:** This function provided users with the ability to release allocated memory, freeing up resources and preventing memory leaks.
5. **free_mem_with_buffering:** By incorporating a buffering mechanism, this function enhanced the efficiency of memory de-allocation, improving overall performance.

The successful implementation of these functions in Milestone 2 extended the capabilities of the FOS operating system, enabling user-level memory management and robust handling of page faults.

# Detailed Explanation of kmalloc Function

```
void* kmalloc(unsigned int size)
```

*Figure 1*

The kmalloc function is responsible for dynamically allocating memory space within the kernel heap in the FOS operating system. It takes an unsigned integer size as a parameter, indicating the size of the memory space to be allocated. The function returns a void pointer to the allocated memory space or NULL if the allocation fails.

```
void intialize_allocations()
{
    for (int z = 0; z < MAX_NUM_ALLOCATIONS; z++)
    {
        allocation_info[z].va = 0;
        allocation_info[z].size = 0;
        allocation_info[z].free = 0;
        allocation_info[z].pa=0;
    }
}
```

*Figure 2*

```
if (!are_allocations_intialized)
{
    intialize_allocations();
    are_allocations_intialized = 1;
}
```

*Figure 3*

The function begins by checking if the allocations have been initialized. If not, it calls the "**initialize_allocations**" function to set up the allocation information. The **are_allocations_initialized flag** is then set to 1 to indicate that the allocations have been initialized.

```
size = ROUNDUP(size, PAGE_SIZE);//size of the range needed for the allocation
num_allocations = size / PAGE_SIZE;//number of pages needed for the requested size
```

*Figure 4*

Next, the function performs some calculations to determine the number of pages **(num_allocations)** needed for the requested memory size. It rounds up the size to the nearest multiple of the page size **(PAGE_SIZE)** using the **ROUNDUP** function.

```
int index = 0;//keep track of the allocations

for (int i = inputStart; i < kernelEnd; i += PAGE_SIZE)
```

*Figure 5*

The function then initializes an index variable to keep track of the allocations and enters a loop that iterates over the memory range from **(inputStart)** to **(kernelEnd)**. This range corresponds to the kernel heap area in the FOS operating system.

Within the loop, the function checks for available memory by examining the **allocation_info** array. It looks for contiguous empty pages that can accommodate the requested number of pages. If a suitable range is found, the **found flag** is set to 1, and the loop is exited.

```
struct Frame_Info *ptr_frame_info = NULL;
//uint32 star = index;
if (found)
{
    //start from the first empty page and loop until number of pages needed for the requested size
    for (int x = index; x <(index + num_allocations); x++)
    {
        int return1 = allocate_frame(&ptr_frame_info);
        if(return1!=E_NO_MEM)
        {
            allocation_info[x].pa=to_physical_address(ptr_frame_info);//return pa in case needed
        }
```

*Figure 6*

After finding an available range of pages, the function proceeds to allocate frames and map them to virtual addresses in the kernel heap. It uses the "**allocate_frame"** function to obtain a free frame represented by the **ptr_frame_info** pointer. If successful, the physical address of the frame is stored in the **allocation_info** array.

```
void* va = (void*) ((x * PAGE_SIZE) + KERNEL_HEAP_START);
int return2 = map_frame(ptr_page_directory, ptr_frame_info, (void*) va , PERM_WRITEABLE);
if(return2==E_NO_MEM)
{
    kpanic_into_prompt("Error no Memory!!!");
    return NULL;
}
```

*Figure 7*

The function then maps the frame to a virtual address in the kernel heap using the **"map_frame"** function. If the mapping fails due to insufficient memory, the function handles the error by calling "**kpanic_into_prompt**" and returning NULL.

```
ptr_frame_info->va=(uint32)va;
allocation_info[x].free = 1;
allocation_info[x].va = (uint32) va;
allocation_info[x].size = num_allocations;
```

*Figure 8*

For each allocated page, the function updates the allocation information in the allocation_info array. It sets the free flag to 1, indicating that the page is allocated. It stores the virtual address **(VA)** and the size **(num_allocations)** in the corresponding array entries.

```
    uint32 va_start = (index * PAGE_SIZE) + KERNEL_HEAP_START;
    return (void*) va_start;
} /* if( found ) */
else
{
    return NULL;
}
```

*Figure 9*

Finally, the function calculates the starting virtual address of the allocated memory space based on the index and the kernel heap start address. It returns the void pointer to, indicating the successfully allocated memory space. If no suitable range of contiguous pages is found or if the loop reaches the maximum number of allocations, then it returns NULL, indicating the allocation failure.

The kmalloc function plays a crucial role in enabling dynamic memory allocation within the kernel heap of the FOS operating system, allowing for efficient memory management and utilization.

# Detailed Explanation of kfree Function

```
void kfree(void* virtual_address)
```

*Figure 10*

The kfree function is responsible for freeing the dynamically allocated memory space within the kernel heap in the FOS operating system. It takes a void pointer **virtual_address** as a parameter, which represents the starting virtual address of the memory space to be freed.

```
for (int i = 0; i < MAX_NUM_ALLOCATIONS; i++)
{
    if (allocation_info[i].va == (uint32) virtual_address)
```

*Figure 11*

Within the function, there is a loop that iterates over the **allocation_info** array, which stores information about the allocated memory blocks. It compares the virtual address of each allocation with the provided **virtual_address** parameter to find a matching allocation.

```
uint32 vv = allocation_info[i].va;
int z = allocation_info[i].size;
```

*Figure 12*

Once a matching allocation is found, the function retrieves the virtual address (vv) and the size (z) of the allocation. The size represents the number of pages that were originally allocated for the memory block.

```
for (int j = 0; j < z; j++)
{
    unmap_frame(ptr_page_directory, (void*) vv);
    allocation_info[i].va = 0;
    allocation_info[i].free = 0;
    allocation_info[i].size = 0;
    allocation_info[i].pa=0;
    vv += PAGE_SIZE;
    i++;
```

*Figure 13*

Next, a nested loop is used for freeing each individual page of the allocated memory block. Within each iteration of the nested loop, the **"unmap_frame"** function is called to remove the mapping between the physical memory frame and the virtual address in the kernel heap.

After unmapping the frame, the allocation information for the corresponding entry in the **allocation_info** array is updated and all its info were set to their initial values (0).

The vv variable is incremented by (PAGE_SIZE) to move to the next virtual address within the memory block. The outer loop variable (i) is also incremented to continue searching for any other matching allocations in the allocation_info array.

By completing this process for each page in the allocation, the kfree function successfully frees the memory space within the kernel heap that was previously allocated using kmalloc.

Overall, the kfree function plays a crucial role in the dynamic memory management of the FOS operating system, ensuring that allocated memory can be efficiently freed and reused when no longer needed.

## Detailed Explanation of kheap_virtual_address Function

```
unsigned int kheap_virtual_address(unsigned int physical_address)
```

*Figure 14*

The kheap_virtual_address function is used to retrieve the virtual address corresponding to a given physical address within the kernel heap in the FOS operating system. It takes an unsigned integer **physical_address** as a parameter, representing the physical address for which the virtual address needs to be determined.

```
uint32 v_a;
struct Frame_Info *ptr_frame_info =NULL;
```

*Figure 15*

The function starts by declaring a variable **(v_a)** to store the virtual address. It then declares a pointer **ptr_frame_info** of type **struct Frame_Info** and initializes it to NULL.

```
ptr_frame_info = to_frame_info( physical_address);
v_a = ptr_frame_info->va;
```

*Figure 16*

The **(to_frame_info)** function is called, passing the physical_address as a parameter, to obtain the frame information for the given physical address.

The resulting frame information is assigned to the **ptr_frame_info** pointer. Next, the virtual address is retrieved from the **ptr_frame_info** structure and assigned to the **v_a** variable.

Finally, the function returns the virtual address (v_a).

In summary, the kheap_virtual_address function provides a convenient way to map a physical address to its corresponding virtual address within the kernel heap in the FOS operating system. This can be useful for various purposes, such as debugging or memory management operations.

# Detailed Explanation of kheap_physical_address Function

```
unsigned int kheap_physical_address(unsigned int virtual_address)
```

*Figure 17*

The **`kheap_physical_address`** function is used to retrieve the physical address corresponding to a given virtual address within the kernel heap in the FOS operating system. It takes an unsigned integer **`virtual_address`** as a parameter, representing the virtual address for which the physical address needs to be determined.

```
uint32 *ptr_page_table = NULL;
struct Frame_Info *ptr_frame_info = NULL;
uint32 pa=0;
```

*Figure 18*

The function starts by declaring a pointer **`ptr_page_table`** of type **`uint32`** and initializing it to `NULL`. It also declares a pointer **`ptr_frame_info`** of type **`struct Frame_Info`** and initializes it to `NULL`. Additionally, an unsigned integer **`pa`** is declared to store the physical address.

```
get_page_table(ptr_page_directory, (void *) virtual_address,&ptr_page_table);
```

*Figure 19*

The **`get_page_table`** function is called to retrieve the entry corresponding to the virtual address.

```
if(ptr_page_table==NULL)
    return 0;
ptr_frame_info = get_frame_info(ptr_page_directory,(void *) virtual_address, &ptr_page_table);
```

*Figure 20*

If the **`ptr_page_table`** is `NULL`, indicating that the page table entry could not be found, the function returns `0`, indicating a failure in obtaining the physical address. Else If the **`ptr_page_table`** is not `NULL`, the **`get_frame_info`** function is called to retrieve the frame information corresponding to the virtual address.

```
if(ptr_frame_info==NULL)
    return 0;
pa = to_physical_address(ptr_frame_info);
```

*Figure 21*

If the **`ptr_frame_info`** is `NULL`, indicating that the frame information could not be found, the function returns `0`, indicating a failure in obtaining the physical address. Else If the **`ptr_frame_info`** is not `NULL`, the **`to_physical_address`** function is called, passing the **`ptr_frame_info`** as a parameter. This function retrieves the physical address associated with the frame information.

The resulting physical address is assigned to the **`pa`** variable.

Finally, the function returns the physical address **(`pa`)**.

# Detailed Explanation of page_fault_handler_with_buffering Function

The __page_fault_handler_with_buffering function is a page fault handler in the FOS operating system, specifically for handling page faults with buffering. It consists of two main parts: placement and replacement.

## Placement

```
uint32 Exact_start_fault_va = ROUNDDOWN(fault_va, PAGE_SIZE);
```

*Figure 22*

In the placement part, we begin by rounding down the faulting virtual address (fault_va) to the nearest page boundary using the ROUNDDOWN function. This ensures that we have the exact starting virtual address of the page that caused the fault.

```
uint32 allocated_pages_WS = env_page_ws_get_size(curenv);
if (allocated_pages_WS < curenv->page_WS_max_size)
```

*Figure 23*

Next, the function checks if there is space in the working set of the current environment (curenv) to allocate a new page. The env_page_ws_get_size function is used to determine the number of pages currently loaded in the main memory for the given environment. If the number of allocated pages is less than the maximum allowed size (curenv->page_WS_max_size), the placement case is executed.

```
uint32 permissions = pt_get_page_permissions(curenv,Exact_start_fault_va);
uint32 *ptr_page_table = NULL;
get_page_table(curenv->env_page_directory,(void *) Exact_start_fault_va, &ptr_page_table);
struct Frame_Info *ptr_frame_info = NULL;
if (permissions & PERM_BUFFERED)
{
    pt_set_page_permissions(curenv, Exact_start_fault_va, PERM_PRESENT,PERM_BUFFERED);
    ptr_frame_info = get_frame_info(curenv->env_page_directory,(void*) Exact_start_fault_va, &ptr_page_table);
    ptr_frame_info->isBuffered = 0;
    if (permissions & PERM_MODIFIED)
    {
        bufferlist_remove_page(&modified_frame_list, ptr_frame_info);
    } else
    {
        bufferlist_remove_page(&free_frame_list, ptr_frame_info);
```

*Figure 24*

In the placement case, the function checks the permissions of the faulted page using the pt_get_page_permissions function. It retrieves the corresponding page table entry (ptr_page_table) using the get_page_table function. If the page is buffered (i.e., the PERM_BUFFERED flag is set), it updates the permissions to mark the page as present (PERM_PRESENT) and clears the buffered flag. Additionally, if the page is modified (i.e., the PERM_MODIFIED flag is set), it removes the frame info from the modified_frame_list. If the page is not modified, it is removed from the free_frame_list.

```
int return1 = allocate_frame(&ptr_frame_info);
if (return1 == E_NO_MEM)
{
    kpanic_into_prompt("Error no Memory!!!");
    return;
}
int return2 = map_frame(curenv->env_page_directory, ptr_frame_info,(void*) Exact_start_fault_va,PERM_PRESENT
```

*Figure 25*

If the page is not buffered, it proceeds to allocate a new frame using the allocate_frame function. If there is no available memory, the function panics with an "Error no Memory!!!" message. Once a frame is successfully allocated, it is mapped to the faulted virtual address using the map_frame function, with the permissions set to present (PERM_PRESENT), user accessible, and writable.

```
int found = pf_read_env_page(curenv, (void*) Exact_start_fault_va);
if (found == E_PAGE_NOT_EXIST_IN_PF)//page does not exist on page file
{
    if ((Exact_start_fault_va < USTACKBOTTOM)|| (Exact_start_fault_va >= USTACKTOP))
    {
        panic("ILLEGAL MEMORY ACCESS TEAM 17");
    }
    else
    {
        int return3 = pf_add_empty_env_page(curenv,Exact_start_fault_va, 0);
        if (return3 == E_NO_PAGE_FILE_SPACE)
            panic("ERROR: No enough virtual space on the page file");
    }
```

*Figure 26*

After mapping the frame, the function checks if the page exists in the page file using the pf_read_env_page function. If the page is not found in the page file (E_PAGE_NOT_EXIST_IN_PF), it checks if the faulted virtual address falls within the valid stack range. If it does not, indicating an illegal memory access, the function panics with an "ILLEGAL MEMORY ACCESS TEAM 17" message. Otherwise, it adds an empty page to the page file using the pf_add_empty_env_page function.

```
//update working set
for (int i = 0; i < curenv->page_WS_max_size; i++)
{
    if (curenv->ptr_pageWorkingSet[i].empty)
    {
        env_page_ws_set_entry(curenv, i, Exact_start_fault_va);
        curenv->page_last_WS_index = i;
        if ((curenv->page_last_WS_index + 1)== curenv->page_WS_max_size)
            curenv->page_last_WS_index = 0;
        else
            curenv->page_last_WS_index++;
        break;
    }
}
```

*Figure 27*

Next, the function updates the working set. It searches for an empty slot in the working set (ptr_pageWorkingSet) of the current environment (curenv). Once an empty slot is found, it sets the entry to the exact start of the faulted page (Exact_start_fault_va). It also updates the page_last_WS_index of the environment to point to the newly added page. The page_last_WS_index serves as a pointer to the last accessed page in the working set and is used for the LRU (Least Recently Used) replacement policy.

If the number of allocated pages in the working set reaches the maximum allowed size (curenv->page_WS_max_size), indicating that the working set is full, the replacement case is executed.

## Replacement

### Case 1

```
for (i = curenv->page_last_WS_index; i < length; i++)
{
    victim_va = env_page_ws_get_virtual_address(curenv, i);
    uint32 permissions = pt_get_page_permissions(curenv,victim_va);
    if (((PERM_USED & permissions) == 0)&& ((PERM_MODIFIED & permissions) == 0))
    {
        //unmap_frame(curenv->env_page_directory,(void*)victim_va);
        env_page_ws_clear_entry(curenv, i);
        case_1_found = 1; //vicitim page found
        //curenv->page_last_WS_index++;//update pointer to point the next page to the victim in WS
        break;
```

*Figure 28*

In the replacement part, the function searches for a victim page to be replaced. It first attempts to find a page that is neither used nor modified (case 1). It iterates through the working set from the page_last_WS_index to the end, looking for a page that satisfies the conditions. If a victim page is found, its entry in the working set is cleared, indicating that it is no longer present in memory.

If a victim page satisfying case 1 (neither used nor modified) is found, its entry in the working set is cleared, indicating that it is no longer present in memory. The frame associated with the victim page is then retrieved using the get_frame_info function, and its corresponding permissions are checked.

If the page is dirty (i.e., modified) and the page is also present in the page file, it is written back to the page file using the pf_update_env_page function. This ensures that any modifications made to the page in memory are persisted to the page file before eviction.

After writing the page to the page file, the frame is freed using the free_frame function, making it available for future allocations.

### Case 2

```
for (i = curenv->page_last_WS_index; i < length; i++)
{
    victim_va = env_page_ws_get_virtual_address(curenv, i);
    uint32 permissions = pt_get_page_permissions(curenv,victim_va);
    if (((PERM_USED & permissions) == 0))
    {
        env_page_ws_clear_entry(curenv, i);
        case_2_found = 1;
        break;
    }
    if (((PERM_USED & permissions)))
        pt_set_page_permissions(curenv, victim_va, 0,PERM_USED);
    if (((i == curenv->page_last_WS_index - 1)
            || ((i == 0) && (curenv->page_last_WS_index == 0)))
            && (execute_only_1))
        break; /* for( i < length ) */
    if ((i == length - 1))
    {
        execute_only_1 = 1;
        i = -1;
```

*Figure 29*

If case 1 does not yield a victim page (i.e., all pages are either used or modified), the function proceeds to case 2. It searches for a page that is used but not modified. The process is similar to case 1, but in this case, the modified flag is checked to determine if the page is modified.

If a victim page satisfying case 2 is found, its entry in the working set is cleared, and the corresponding frame is handled in the same way as in case 1: writing it back to the page file if it's dirty and freeing the frame.

If neither case 1 nor case 2 yields a victim page, it means all pages in the working set are both used and modified. In this scenario, the function panics with a "Panic in page fault handler: all pages in the working set are both used and modified" message, indicating a failure in the page replacement process.

Once a victim page is selected and its frame is freed, the function proceeds to allocate a new frame for the faulted page, similar to the placement case. The new frame is mapped to the faulted virtual address, and if necessary, an empty page is added to the page file. The working set is updated to include the faulted page, and the necessary bookkeeping is performed.

Finally, the function returns from the page fault handler.

In summary, the __page_fault_handler_with_buffering function handles page faults by first checking if there is space in the working set to allocate a new page. If space is available, it handles the placement case. If the working set is full, it searches for a victim page to be replaced based on the LRU policy. If a victim page is found, its frame is freed, and a new frame is allocated for the faulted page. The function also takes care of buffering and writing back modified pages to the page file as necessary.

## Page Replacement with Buffering and Frame Lists

```
victim_frame = get_frame_info(curenv->env_page_directory,(void*) victim_va, &ptr_page_table);
//Replace it
victim_frame->isBuffered = 1;
victim_frame->environment = curenv;
victim_frame->va = victim_va;
pt_set_page_permissions(curenv, victim_va,/*1*/PERM_BUFFERED,/*0*/PERM_PRESENT);
uint32 permissions = pt_get_page_permissions(curenv, victim_va);
if ((permissions & PERM_MODIFIED) == 0)
{
    bufferList_add_page(&free_frame_list, victim_frame);
}
else
{
    bufferList_add_page(&modified_frame_list, victim_frame);

    //Get size of modified frame list
    uint32 size = LIST_SIZE(&modified_frame_list);

    if (size == getModifiedBufferLength())
    {
        //Check which modified frame belongs to the current environment

        struct Frame_Info *ptr_fi;
        LIST_FOREACH(ptr_fi, &modified_frame_list)
        {
            int ret = pf_update_env_page(ptr_fi->environment,(void*) ptr_fi->va, ptr_fi);
            pt_set_page_permissions(ptr_fi->environment, ptr_fi->va, 0,PERM_MODIFIED);
            bufferlist_remove_page(&modified_frame_list, ptr_fi);
            bufferList_add_page(&free_frame_list, ptr_fi);
        }
```

*Figure 30*

# Detailed Explanation of malloc Function

The malloc function is responsible for dynamically allocating memory of a specified size and returning a pointer to the allocated memory block. Here's a breakdown of the code:

```
if (sys_isUHeapPlacementStrategyBESTFIT()) {
```

*Figure 31*

The code begins by checking if the best-fit placement strategy is set using the "sys_isUHeapPlacementStrategyBESTFI"T function. This strategy aims to find the smallest available memory block that can accommodate the requested size.

Next, the code initializes the necessary variables and data structures to track the allocation information. The initialize_allocations function is called to set up the allocation_info array, which holds information about the allocated pages.

After that, the size parameter is rounded up to the nearest multiple of the page size (PAGE_SIZE) using the ROUNDUP function. This ensures that the requested size aligns with the page boundaries.

The number of allocations (num_allocations) is calculated by dividing the rounded-up size by the page size. This determines the number of pages needed to fulfill the requested size.

```
for (int i = 0; i < MAX_ALLOCATIONS; i++) {
    if (allocation_info[i].free) {
        num_contiguous_allocations = 0;

        while (i + num_contiguous_allocations < MAX_ALLOCATIONS
                && allocation_info[i + num_contiguous_allocations].free) {
            num_contiguous_allocations++;
        }

        if (num_contiguous_allocations * PAGE_SIZE >= size) {
            uint32 contiguous_size = num_contiguous_allocations * PAGE_SIZE;
            if (contiguous_size < smallest_contiguous_size) {
                smallest_contiguous_size = contiguous_size;
                best_fit_index = i;
            }
        }
        if(num_contiguous_allocations!=0)
        i += num_contiguous_allocations - 1;
    } /* if( allocation_info[i].free ) */
```

*Figure 32*

The code then proceeds to search for the best-fit allocation. It iterates through the allocation_info array, examining each entry for available and contiguous free pages. It keeps track of the smallest contiguous size that can accommodate the requested size and the index of the best-fit allocation.

The virtual address (VA) for each allocated page is calculated based on the index and the user heap's starting address (USER_HEAP_START).

After the allocation process, the code determines the starting virtual address (start_va) of the allocated memory block and calculates the total allocation size (allocation_size).

```
sys_allocateMem(start_va, allocation_size);

return (void*) start_va;
```

*Figure 33*

Finally, the sys_allocateMem function is then called to allocate the memory range starting from start_va with the size of allocation_size. Finally, the function returns the starting virtual address (start_va) as a void pointer, indicating a successful allocation.

If there is not enough memory available or the best-fit index remains -1, indicating no suitable allocation, the function returns NULL to indicate the allocation failure.

## Detailed Explanation of allocateMem Function

```
void allocateMem(struct Env* e, uint32 virtual_address, uint32 size)
{
    //TODO: [PROJECT 2023 - MS2 - [2] User Heap] allocateMem() [Kernel Side]
    // Write your code here, remove the panic and write your code
    //panic("allocateMem() is not implemented yet...!!");

    int number_of_pages = size / PAGE_SIZE;

    for(int i=0 ; i<number_of_pages ; i++)
    {
        pf_add_empty_env_page(e,virtual_address,0);
        virtual_address+=PAGE_SIZE;
    }

}
```

*Figure 34*

The provided code snippet presents an implementation of the allocateMem () function, which is responsible for allocating all pages within a specified range in the page file. Let's examine the code in detail.

The function takes three parameters: a pointer to the environment structure (struct Env* e), the starting virtual address of the range (uint32 virtual_address), and the size of the range (uint32 size).

The code begins by calculating the number of pages needed to cover the specified range. This is done by dividing the size by the page size (PAGE_SIZE).

Next, a loop is executed number_of_pages times to allocate each page in the page file. Within each iteration, the function calls pf_add_empty_env_page () to add an empty page entry for the specified environment (e) at the current virtual_address. The last parameter of this function call is set to 0, indicating that no physical memory should be allocated for these pages in the main memory.

After adding the page entry, the virtual_address is incremented by PAGE_SIZE to move to the next page in the range.

By the end of the loop, all the pages in the specified range have been allocated in the page file, and no memory is allocated in the main memory.

# Detailed Explanation of __free_mem_with_buffering Function

```
void __freeMem_with_buffering(struct Env* e, uint32 virtual_address, uint32 size)
```

*Figure 35*

The free_mem_with_buffering () function is a kernel-side function in the user heap implementation. Its purpose is to de-allocate memory for a given range specified by the starting virtual address (uint32 virtual_address) and the size of the range (uint32 size). The function performs several steps to free the memory:

```
//1. Free ALL pages of the given range from the Page File

uint32 allocation_start = virtual_address ;
uint32 allocation_end = virtual_address + size;

for (uint32 i = allocation_start; i < allocation_end ; i += PAGE_SIZE)
{
    pf_remove_env_page(e, i);
```

*Figure 36*

Free pages from the Page File: It iterates over the range of virtual addresses and calls pf_remove_env_page () to free each page from the page file.

```
//2. Free ONLY pages that are resident in the working set from the memory

for (int i = 0; i < (e->page_WS_max_size); i++)
{
    uint32 va = env_page_ws_get_virtual_address(e, i);

    if ((va >= allocation_start) && (va < allocation_end))
    {
        unmap_frame(e->env_page_directory, (void*)va);
        env_page_ws_clear_entry(e, i);
```

*Figure 37*

Free resident pages from memory: It iterates over the working set of the environment and checks if each virtual address falls within the specified range. If a match is found, it calls unmap_frame unmap the corresponding physical frame from the environment's page directory and clears the entry in the working set.

```
uint32 permissions = pt_get_page_permissions(e, i);
if( permissions & PERM_BUFFERED )
{
    ptr_frame_info = get_frame_info(e->env_page_directory,(void*) free_va, &ptr_page_table);

    if (permissions & PERM_MODIFIED)
    {
        bufferlist_remove_page(&modified_frame_list, ptr_frame_info);
        ptr_frame_info->isBuffered = 0;
        ptr_frame_info->environment = NULL;
        ptr_frame_info->va=0;

        pt_set_page_permissions(e, free_va, 0,PERM_MODIFIED);
        pt_set_page_permissions(e, free_va, 0,PERM_BUFFERED);
        pt_clear_page_table_entry(e,free_va);
        free_frame(ptr_frame_info);
    }
    else
    {
        ptr_frame_info->isBuffered = 0;
        ptr_frame_info->environment = NULL;
        ptr_frame_info->va = 0;

        pt_set_page_permissions(e, free_va, 0, PERM_BUFFERED);
        bufferlist_remove_page(&free_frame_list, ptr_frame_info);
        free_frame(ptr_frame_info);
        pt_clear_page_table_entry(e,free_va);
    }
```

*Figure 38*

Free buffered pages: It iterates over the range of virtual addresses and checks if each page has the PERM_BUFFERED permission set. If so, it retrieves the frame information using get_frame_info () and checks if the page is modified or not. If modified, it removes the page from the modified frame list and updates the frame's properties. Then it clears the permissions, clears the page table entry, and frees the frame. If the page is not modified, it updates the frame's properties, clears the permissions, removes the page from the free frame list, frees the frame, and clears the page table entry.

```
//4. Removes ONLY the empty page tables (i.e. not used) (no pages are mapped in the table)
for (uint32 i = allocation_start; i < allocation_end ; i += PAGE_SIZE)
{
    uint32 *page_table = NULL;

    get_page_table(e->env_page_directory, (void*)i, &page_table);

    if (page_table != NULL)
    {
        int empty_entries = 0;

            for (int j = 0; j < 1024; j++)
            {
                if (page_table[j] == 0)
                {
                    empty_entries++;
                }
                else if (page_table[j] != 0)
                {
                    break;
                }
                if (empty_entries == 1024)
                {
                    kfree((void*)page_table);
                    e->env_page_directory[PDX(i)] = 0;
                    pd_clear_page_dir_entry(e,i);
                }
            } /* for( j < 1024 ) */
```

*Figure 39*

Remove empty page tables: It iterates over the range of virtual addresses and checks if each page has an associated page table. If so, it checks if the page table is empty (all entries are zero). If all entries are zero, it frees the page table, clears the corresponding entry in the environment's page directory, and clears the page directory entry.

The function performs these steps to de-allocate memory from the page file, working set, buffered pages, and empty page tables, ensuring that all associated resources are properly released.

# Detailed Explanation of free Function

```
void free(void* virtual_address) {
```

*Figure 40*

The free () function is a user-side function in the user heap implementation. Its purpose is to free the memory allocated for a given virtual address (void* virtual_address). Here's how the function works:

```
uint32 rounded_va = ROUNDDOWN((uint32 )virtual_address, PAGE_SIZE); //to get the actual start of the given va
```

*Figure 41*

Round down the virtual address: The virtual address is rounded down to the nearest multiple of the page size using the ROUNDDOWN macro. This ensures that the starting virtual address of the allocation is obtained.

```
for (int i = 0; i < MAX_ALLOCATIONS; i++) {
    if (allocation_info[i].va == rounded_va) {
```

*Figure 42*

Search for the allocation: The function iterates over the allocation info array (allocation_info) to find the entry that matches the rounded virtual address. Once found, it identifies the size of the allocation.

```
uint32 free_end = allocation_info[i].size;

for (uint32 j = 0; j < free_end; j++)
{
    allocation_info[i].va = 0;
    allocation_info[i].free = 1;
    allocation_info[i].size = 0;
    i++;
```

*Figure 43*

Free the allocation: The function marks the allocation as free by setting the corresponding fields (VA, free, size) to appropriate values. It also iterates free_end times to clear the allocation info for subsequent pages within the allocation.

```
sys_freeMem(rounded_va, free_end*PAGE_SIZE);
break; /* for( i < MAX_ALLOCATIONS ) */
```

*Figure 44*

*Call sys_freeMem ()*: The function calls sys_freeMem () with the rounded virtual address and the size of the allocation multiplied by the page size. This system call notifies the kernel to free the corresponding memory range.

Overall, the free () function searches for the allocation associated with the given virtual address, updates the allocation info, and notifies the kernel to free the memory range. This allows the memory to be reused for future allocations.

# Conclusion

In conclusion, the successful implementation of dynamic memory allocation and freeing in both the kernel heap and user heap, combined with the integration of the MODIFIED CLOCK replacement algorithm for handling page faults, has greatly enhanced the memory management capabilities of the system.

The introduction of the kernel heap, supported by the kmalloc and kfree functions, has provided the kernel with the ability to allocate and release memory space dynamically at runtime. This has significantly improved the flexibility and efficiency of memory management within the kernel. With the kernel heap, the system can allocate memory as needed, ensuring optimal resource utilization and allowing for more efficient execution of kernel operations.

For user programs, the user heap plays a vital role in enabling dynamic memory allocation. Through the malloc function, user programs can request memory space on-demand, allowing for the creation and manipulation of data structures as required by the application. Conversely, the free function allows previously allocated memory to be released when no longer needed. These features greatly enhance the versatility and efficiency of user programs, contributing to more robust and flexible software solutions.

Moreover, the incorporation of the MODIFIED CLOCK replacement algorithm in the page_fault_handler_with_buffering function has significantly improved the handling of page faults during program execution. By intelligently selecting pages for replacement based on their access history and modification status, the MODIFIED CLOCK algorithm optimizes memory usage. It strikes a balance between performance and resource management, ensuring that frequently accessed and modified pages are retained in memory while allowing less frequently used pages to be replaced, thereby maximizing the system's overall efficiency.

In addition to the core functionality, the allocate_mem and free_mem_with_buffering functions have been introduced to further enhance the user heap. The allocate_mem function allows the allocation of memory in the page file, enabling the system to efficiently manage memory resources between the main memory and the page file. On the other hand, the free_mem_with_buffering function efficiently handles the freeing of buffered pages within the specified range, preventing memory leaks and improving system stability.

In summary, the successful implementation of these functions marks a significant milestone in the development of a robust memory management system. The improved memory allocation and freeing capabilities in both the kernel and user heaps, coupled with the efficient handling of page faults using the MODIFIED CLOCK algorithm, have led to enhanced system performance, stability, and resource utilization. This achievement paves the way for the development of more sophisticated and efficient software solutions, enabling the system to handle increasingly complex tasks and deliver a seamless user experience.