OS'21: Assignment 3

Contents

Important Instructions

Please read instructions carefully, any mistake or error may result in assignment rejection from the automated grading system:

1. Make sure to start solving assignment3 using the **FOS_Assignment_3_Students_Template.zip** code.
2. Read the document **STEPS TO IMPORT THE ASSIGNMENT** to correctly import the assignment template in your eclipse and adjust the run configurations.
3. During your solution, make sure that you **DON'T CHANGE** any other file rather than kheap.c/.h
4. Your code *MUST* be written inside the given function for each question (as specified below).
   ANY violation to this rule, or change of function names, or deletion of template functions may result in assignment rejection by the automated grading system.
5. Make sure that your code *PASS* the tests Separately with each test in a **FRESH RUN**.
6. Assignments *MUST* be delivered using this online form: https://forms.gle/rJPmoSvuAaoQHHxo7
7. If you face a problem in **RE-submitting** the assignment code, kindly follow these steps [Resubmission ONLY]
8. For any question/problem, don't miss the weekly office hours

Delivery Method & Dates

- **Assignment & Bonus:** through the above online form
  - **Deadline (for all questions):** FRI 10 December @ 23:59
  - **Early Delivery Bonus:** if delivered all the main questions **correctly before FRI 3 December @ 23:59**

Kernel Heap

Problem

The kernel virtual space [KERNEL_BASE, 4 GB) is **one-to-one** mapped to the physical memory [0, 256 MB), as shown in Figure 1. This limits the physical memory area for the kernel to 256 MB only. In other words, the kernel code can't use any physical memory after the 256 MB, as shown in Figure 2.

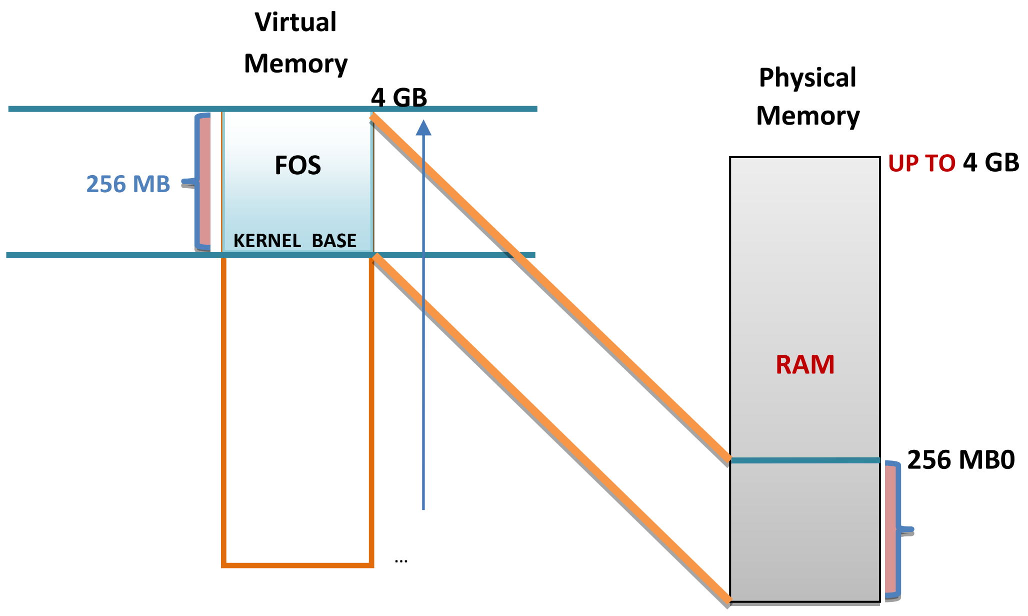

Figure 1: Mapping of the FOS VIRTUAL memory to the PHYSICAL memory [32 bit Mode]

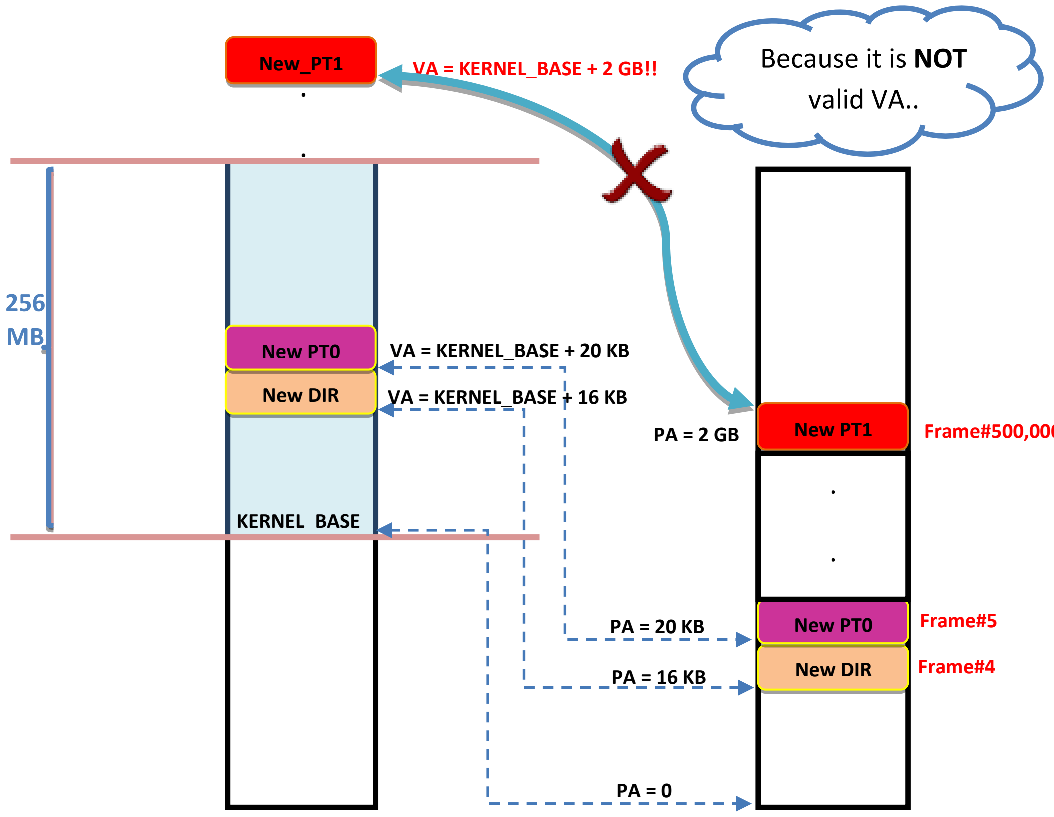

Figure 2: PROBLEM of ONE to ONE mapping's between the FOS VIRTUAL memory and its corresponding frames in the PHYSICAL memory

Solution

Replace the one-to-one mapping to an ordinary mapping as we did in the allocation of the user's space. This allows the kernel to allocate frames anywhere and reach it wherever it is allocated by saving its frame number in the page table of the kernel, as shown in **Figure 3**.

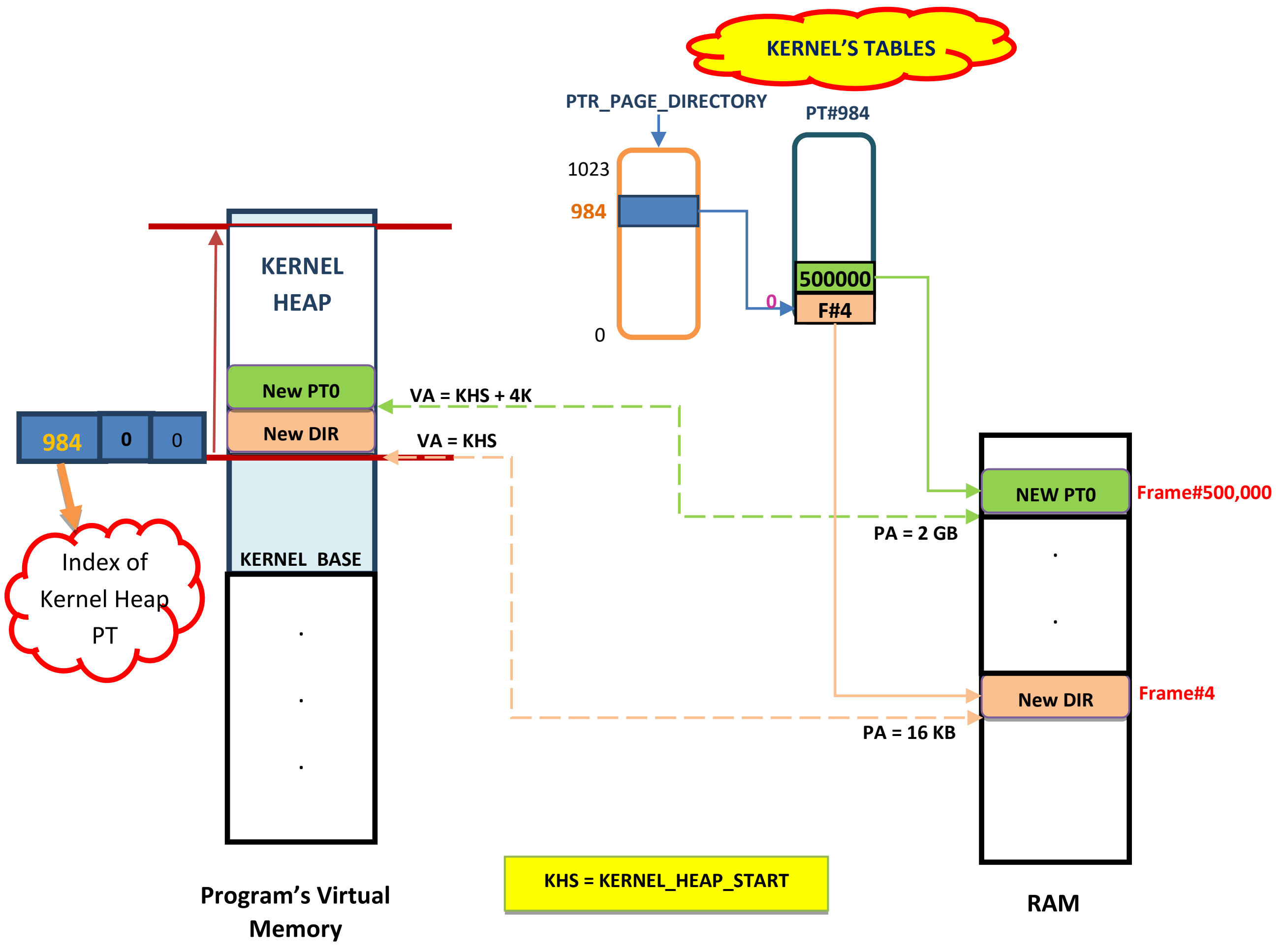

Figure 3: New Model to map kernel's VA of an allocated page in the KERNEL HEAP to a PA

However, since the Kernel virtual space should be shared in the virtual space of all user programs, ALL page tables of the kernel virtual space (from table#960 to table#1023) are already created and linked to the Kernel directory. These tables **SHOULD NOT be removed** for any reason until the FOS is terminated.

You need to fill the following functions (inside kheap.c) that serve the **Kernel's Heap**:

[Main Functions]

1. Kernel Heap Allocation [2.5 marks]

Description

- Dynamically allocates space size CONTINOUSLY starting from KERNEL_HEAP_START and maps it as shown in Figure .
- Pages are allocated on 4 KB granularity (e.g. if the area to be allocated is 6 KB, the function would allocate 8 KB).
- The allocated space mustn't exceed the virtual address "KERNEL_HEAP_MAX". If there is no sufficient space, the function should not allocate any space and it should return NULL.

Function Signature

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

Arguments

- size: size in bytes of the space to be allocated in the kernel heap.

Returns

- The start address of the allocated space (if there is sufficient space in the kernel heap).
- **NULL** (if there is no sufficient space in the kernel heap).

Example

```
void* ptr = kmalloc(6*1024);     //allocate 2 pages (8 KB)

                                 //and return KERNEL_HEAP_START

void* ptr2 = kmalloc(4*1024);    //allocate 1 page (4 KB)

                                 //and return (KERNEL_HEAP_START + 8 KB)
```

2. Kernel Heap Free [2.5 marks]

Description

- Delete a previously allocated space by removing all its pages from the kernel Heap.

- Unmap each page of the previously allocated space at the given address by clearing its entry in the page table and free its frame.
- **DO NOT remove** any table

Function Signature

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

Arguments

- virtual_address: the start virtual address of a previously allocated space

Example

```
void* ptr = kmalloc(6*1024);     // allocate 2 pages (8 KB)

                                 // and return KERNEL_HEAP_START

void* ptr2 = kmalloc(4*1024);    // allocate 1 page (4 KB)

                                 // and return (KERNEL_HEAP_START + 8 KB)

kfree((void*) KERNEL_HEAP_START); // free the previously allocated 2 pages(8 KB)

void* ptr3 = kmalloc(1024);      // allocate 1 page (4 KB)

                                 // and return (KERNEL_HEAP_START + 12 KB)
```

3. Kernel Heap Convert Physical Address to a Virtual Address [3 marks]

Description

- Find kernel virtual address of the given physical one.
- If the physical address wasn't mapped to a virtual address in the kernel heap, return 0

Function Signature

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

Arguments

- `physical_address`: the physical address to find its virtual address in the kernel heap.

Returns

- The kernel heap **virtual address** of the provided physical address.
- **0** (if the physical address wasn't mapped to a kernel heap virtual address)

Example

- In **Figure** , if we call kheap_virtual_address() for physical address 16 KB, it should return KHS (KERNEL_HEAP_START).

4. Kernel Heap Convert Virtual Address to a Physical Address [2 marks]

Description

- Find physical address of the provided kernel heap virtual address.
- If the virtual address isn't mapped to a physical address, return 0

Function Signature

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

Arguments

- `virtual_address`: the kernel heap virtual address to find its physical address.

Returns

- The **physical address** of the provided kernel heap virtual address.
- **0** (if the virtual address wasn't mapped to a physical address)

Example

- In **Figure** , if we call kheap_physical_address() for virtual address KHS + 4 KB, it should return 2 GB.

[Bonus]

5. Kernel Reallocation [3 marks]

Description

- Attempts to resize the allocated space at given "virtual address" to "new size" bytes, possibly moving it in the kernel heap.
- Reallocation should be done on 4 KB granularity (e.g. if the area to be allocated is 6 KB, the function would allocate 8 KB).

Function Signature

```
void *krealloc(void *virtual_address, uint32 new_size)
```

Arguments

- `virtual_address`: the virtual address of a previously allocated space (to resize).
- `new_size:` the new size in bytes

Returns

- The new virtual address: If reallocation was done successfully, in case of moving the old allocation, the old virtual address must no longer be accessed.
- NULL: On failure. the old virtual address remains accessible.

Example

```
void* ptr = kmalloc(6*1024);     // allocate 2 pages (8 KB)

                                 // and return KERNEL_HEAP_START

void* ptr2 = krealloc((void*) KERNEL_HEAP_START, 12*1024);

                          //allocates 1 extra page(the total new size is 12 KB)

                          // and returns KERNEL_HEAP_START

void* ptr3 = kmalloc(5 * 1024);  // allocate 2 page (8 KB)

                                 // and return (KERNEL_HEAP_START + 12 KB)
void* ptr2 = krealloc((void*) KERNEL_HEAP_START, 13*1024);

                          //moves the old allocation to the end of the allocated

                          // space, (the total new size is 16 KB)

                          // and returns KERNEL_HEAP_START + 20 KB
```

Notes

- A call with "virtual address = null" is equivalent to calling kmalloc()
- A call with "new size = zero" is equivalent to calling kfree()
- A call with “new size < old size” should **change nothing** and returns the same virtual address.

Testing

Run every test of the following. If a test succeeds, it will print and success message on the screen, otherwise the test will panic at the error line and display it on the screen.

IMPROTANT NOTES:

1. Run each test in NEW SEPARATE RUN
2. If a certain test failed, then there's a problem in your code
3. Else, this does NOT ensure 100% that this part is totally correct. So, make sure that your logic matches the specified steps exactly

1. *tstkmalloc command:* tests the implementation of **kmalloc()**. It validates return addresses from the kmalloc(),number of allocated frames, accessing the allocated space and permissions
   - **FOS>** `tstkmalloc`
2. *tstkfree command:* tests the implementation of **kfree()**. It validates the number of freed frames by kfree(). It checks the memory access (read & write) of the removed spaces and allocation after free. Also, it ensure that KHEAP tables are not removed.
   - **FOS>** `tstkfree`
3. *tstkvirtaddr command:* tests the implementation of **kheap_virtual_address()**. It validates the returned virtual address of the given physical one for three cases: 1. After kmalloc only, 2. After kmalloc and kfree, 3. For frames that does not belong to KERNEL HEAP (should return 0).
   - **FOS>** `tstkvirtaddr`
4. *tstkphysaddr command:* tests the implementation of **kheap_physical_address()**. It validates the returned physical address of the given virtual one for three cases: 1. after kmalloc only, 2. after kmalloc and kfree, 3. for not allocated area in KERNEL HEAP (should return 0).
   - **FOS>** `tstkphysaddr`

5. tstkrealloc command: tests the implementation of **kheap_realloc()**. It validates the returned virtual address in three cases:1. Test calling krealloc with VA = NULL. It should call malloc, 2. Test krealloc by passing size =0. It should call kfree. 3.Test krealloc allocations with valid and invalid sizes.

- **FOS>** `tstkrealloc`

MAKE SURE that you followed the above instructions carefully. Good Luck isA ☺

Appendix I: Basic and Helper Memory Management Functions

Basic Functions

The basic **memory manager functions** that you may need to use are defined in "`kern/memory_manager.c`" file:

Function Name	Description
`allocate_frame`	Used to allocate a free frame from the free frame list
`free_frame`	Used to free a frame by adding it to free frame list
`map_frame`	Used to map a single page with a given virtual address into a given allocated frame, simply by setting the directory and page table entries
`get_page_table`	Used by "`map_frame`" to get a pointer to the page table if exist
`unmap_frame`	Used to un-map a frame at the given virtual address, simply by clearing the page table entry
`get_frame_info`	Used to get both the page table and the frame of the given virtual address

Helpers Functions

There are some **helper functions** that we may need to use them in the rest of the course:

Function	Description	Defined in…

`LIST_FOREACH (Frame_Info*, Linked_List *)`	Used to traverse a linked list.
	Example: to traverse the modified list
	struct Frame_Info* ptr_frame_info = NULL;
	LIST_FOREACH(ptr_frame_info, &modified_frame_list)
	{
	  ….
	}
		`inc/queue.h`

`to_frame_number (Frame_Info *)`	Return the frame number of the corresponding Frame_Info element	`Kern/memory_manager.h`
`to_physical_address (Frame_Info *)`	Return the start physical address of the frame corresponding to Frame_Info element	`Kern/memory_manager.h`
`to_frame_info (uint32 phys_addr)`	Return a pointer to the Frame_Info corresponding to the given physical address	`Kern/memory_manager.h`

Function	Description	Defined in…
`PDX (uint32 virtual address)`	Gets the page directory index in the given virtual address (10 bits from 22 – 31).	`Inc/mmu.h`
`PTX (uint32 virtual address)`	Gets the page table index in the given virtual address (10 bits from 12 – 21).	`Inc/mmu.h`
`ROUNDUP (uint32 value, uint32 align)`	Rounds a given “value” to the nearest upper value that is divisible by “align”.	`Inc/types.h`
`ROUNDDOWN (uint32 value, uint32 align)`	Rounds a given “value” to the nearest lower value that is divisible by “align”.	`Inc/types.h`
`tlb_invalidate (uint32* page_directory, uint32 virtual address)`	Refresh the cache memory (TLB) to remove the given virtual address from it.	`Kern/helpers.c`
`rcr3()`	Read the physical address of the current page directory which is loaded in CR3	`Inc/x86.h`
`lcr3(uint32 physical address of directory)`	Load the given physical address of the page directory into CR3	`Inc/x86.h`