**Project #4 Design**

**Physical memory layout**

Draw the physical memory layout, indicating the areas of memory devoted to code (text area), data, page directories and page tables, BSS, etc. For each memory area, indicate its start address and its size. Note that different teams may use different layouts.



**Initialization of page directories and page tables**

Indicate how page directories and page tables should be initialized. You don’t need to show the exact content of page directories and page tables (i.e., the content of each entry). But, you need to indicate how many page directory/table entries you need for each memory area that should be mapped to the processes’ virtual space. If different types of processes require different mappings, show them all.

In designing the memory layout and defining the initial content of page directories and page tables, you need to consider the following.

* Before paging is enabled, the OS uses only physical addresses. However, after paging is enabled, all memory accesses are through the virtual address space. You need to map the static segments (TEXT, DATA, etc.) in the virtual address space of all processes.
* User processes cannot map the whole FFS area: they can map only the portion of it that they use (that is, FFS area will be mapped only when vmalloc is invoked)
* **ECE565 students:** User processes cannot map the area devoted to page directories and page tables.

For all processes, the page directories and resultant page tables will be identical. They will start with 8 PDEs pointing to full pages of PTEs. This means there are 8K pages since each directory page is 4KB and each entry is 4B resulting in 1K entries. These 8K pages are the XINU\_AREA and are needed to be accessible by any and all processes. From there, system processes will map the PT\_AREA and FFS\_AREA given that they need access to both areas. User processes on the other hand will start by only seeing the XINU\_AREA when initialized. Everything else (the frames of the FFS\_AREA that are allocated) are added dynamically upon allocation.

**System Initialization**

Where is paging enabled and how? (see hints in specs)

* Paging is enabled at the end of the *sysinit()* function in *initialize.c.* This is done by first calling *initialize\_page\_table()* to initialize the page table so that the area of memory that was available prior is available after enabling. The function enable\_paging() function is then called, which sets a bit in a special purpose register to enable paging. After enabled, all memory accesses occur through the page table.

**Process Creation**

How do you need to modify process creation to support paging?

* vcreate is implemented to create user processes with a virtual heap. This heap is made private and exists in the virtual memory space of the process created. As far as system created processes that are made using the *create* function, they will share a page table to support paging. This page table will include 8192 page table entries as well as the areas (including PT\_AREA and FFS\_AREA). All processes also need a CR3 field in the PCB.

**Process Termination**

How do you need to modify process termination to support paging?

* When a process is terminated, the page table of the process is released. This also includes any used free frame spaces that are no longer required. Physical memory used by page directory and second-level page tables are also released upon termination.

**Context Switch**

What should be done at context switch to support paging?

* At a context switch, the CR3 register should be modified to reflect the saved CR3 of the new process.

**Heap allocation, deallocation and access**

What should be done at heap allocation, deallocation and when the heap is accessed? Remember that you need to implement lazy allocation in your code.

* At allocation of heap, we will be using the first-fit policy to find the first free virtual page(s) in the virtual address space to fit the number of bytes desired of the process calling. When the virtual pages are allocated, page table entries are added to the corresponding page table. In deallocation, the frames of the free frame space that are used by the heap being deallocated must be released. This also means the page table and, if necessary, the page directory should be updated to reflect the deallocation. Due to lazy allocation, when the heap is first accessed is the point when physical space is reserved for the heap. This is done by acquiring Free Frame Space that can hold the heap. If the heap has not been allocated before this access, a segmentation fault occurs.

**Page Fault Handler Design**

* In which circumstances will the hardware raise a page fault?
* Page faults will be raised when trying to access a page table entry that is not currently present in memory.
* What operations should be performed by the page fault handler depending on the circumstances under which it is invoked?
* The page fault handler should allocate a physical frame and set the present bit to 1. The CR3 register should be first set to point to the system page table in the handler, then reset it the CR3 to point again to the PDBR of the process that invoked the page fault. The page fault handler will be invoked whether the valid bit is set or not, so it will also distinguish between a situation where the will be an allocation and a situation that should kill the process due to a segmentation fault.

**Page Replacement and Swapping Design (optional)**

Since this aspect is optional, no need to specify your plan in this design document. If you end up deciding to implement swapping, you can describe your design in the final report.