ECE 592: Operating Systems Design
Project 4
REPORT

SAURABH LABDE UNITY ID: sslabde
SHRINATH CHERIYANA UNITY ID: scheriy

CHANGES MADE TO THE DESIGN:

Physical Memory Layout same as design document

Basic setup:

The basic setup described for paging is almost similar to the ones mentioned in the design document. The changes are documented below:

Three arrays to model every frame in PDPT region, FFS region and DSS space
PDPT_tab [MAX_PT_SIZE]
FSS_TAB[MAX_FSS_SIZE]
DSS_TAB[MAX_SWAP_SIZE]
These arrays are of structure type frame_t with the following fields:

1) pid – Stores the PID of the process using the frame.
2) Virtual_pg_no – Stores the virtual page number of this frame in the address space of the process
3) Status – To indicate whether this frame is mapped or unmapped.
4) Is_dirty – To indicate whether this page is dirty or not.

Macros are defined in paging.h to identify the start and end of each region of memory. These are kept dynamic so that they can scale as the max sizes for each of the region's changes.
Other helpful macros are also defined in paging.h to make the code verbose.

A global variable to keep track of the available heap space is used as collective virtual memory of all the processes cannot exceed maximum heap size. This variable is initially set to MAX_SWAP_SIZE;
No global memory list is used as indicated in the design document. We choose to implement a per process virtual memory list of type vmemblk_t with a structure similar to memblk.

The steps required to enable paging and initialize the system are the same as mentioned in the design document.

The steps involved in create.c are the same as mentioned in the design document.

In vcreate.c – all the steps are same with some minor changes as described below:

1) We are not reserving space for the virtual heap of the process on process creation. We do that in vmalloc when a process requests memory.
2) A per process virtual memlist is initialized with the start address as 8192 * PAGE as this is the first possible address of the heap and its length field is set to hsize* PAGE_SIZE
3) We are not writing the value of PDBR of the newly created process into cr3 register on process creation as mentioned in the design document. This is taken care of by the context switch whenever the newly created process is scheduled.

Process termination is the same as mentioned in the design doc with the following changes:

1) We do not add the hsize to the maximum heap size as mentioned in the design document.
2) The virtual memory requested by the process with vmalloc is freed in vfree. But in cases where vfree is not called and a process is killed, we track the memory allocated by every vmalloc call and if these are not freed when kill is call, we add it to the avail_heap_space as the process no longer exists.

Context switch is the same as mentioned in the design document.

The heap allocation and deallocation differ from the design document.

Heap allocation:

Round the number of nbytes to the nearest multiple of page size as we operate on the granularity of pages
Check if the requested memory is less than the virtual heap size requested at creation and less than the available heap space in the system. If not, then throw an error.
Once the above checks have passed, search the freelist and find a fitting block and subtract it from the freelist to indicate that it has been allocated
Subtract the number of bytes allocated from the available heap space.
Add the number of bytes allocated to the allocated space field of proctab of the current process.
Return the virtual address to the first free address allocated.

Heap deallocation:

Round the number of nbytes to the nearest multiple of page size as we operate on the granularity of pages
Switch to kernel mode and write the PDBR of system processes to the cr3 register
Check of the address provided maps in the virtual heap of the process. If not, throw an error.
Iterate the freelist and add the bytes to be deallocated to the freelist at appropriate place.
Check for block overlap so that if vfree is called twice or a non-allocated block is tried to be deallocated, it will display an error.
Add the bytes freed to the available heap space.
Subtract the bytes freed from the allocated space.
Set the present bit of the page table entries of the deallocated pages to 0.
Switch back to user mode and write the PDBR of the current process to the cr3 register.

Heap access

If the present bit of the page table entry of the corresponding address is 1, access is successful. If the present bit is 0, pagefault_handler_disp.S is called which pops the error code and calls the high level pagefault_handler.c which services the interrupt by mapping a page to the physical memory.

Page Fault Handler Design:

- We switch to kernel mode and write the PDBR of NULLPROC to CR3. Next, we get the fault address by reading the CR2 register and check if the address is present in the freelist. If, YES then we return a SEGMENTATION FAULT.
- Next, we locate the page directory entry and the page table entry using the fault address.
- We first check if the page directory entry exists, if NO then this means that the address was never mapped. So, we first get a free frame from PD/PT region to create a page table. Then get a frame from FFS region and update the page table and the page directory entry.
- If the page directory entry exists, then there can be two cases:
    1. The present bit of the page table entry is 0 and the dirty bit of the page table entry is 0. In this case, we get a free frame from the FFS region and update the page table entry.
    2. The present bit of the page table entry is 0 and the dirty bit of the page table entry is 1. This means the frame is in the SWAP space. In this case, we first retrieve the frame from the SWAP space and then get a free frame from the FFS region. We then copy the contents of the frame retrieved from SWAP space into the free frame acquired from FFS region. We then update the page table entry.

Swapping Design:

When FFS region is full, the following procedure is followed:

- A frame is selected at random from the FFS region, we call this victim.
- Next, we locate this frame in the ffs_tab that we have maintained. From this ffs_tab entry we get the pid of the process that currently has this frame and its virtual page number.
- We get the base address of the page directory of this process using PDBR field of the procent structure.
- Next, using the virtual page number left shifted by 12 we get the page directory and page table offset. So, now we can locate the page table entry that points to this frame.
- Next, we check if the victim is present in the SWAP space. There can be two possibilities:
    1. **The victim is not in SWAP. So, now we will have to write it to the SWAP space. So, we will first try to get a free frame from swap. Now there can be two cases:**
        i. *We get a free frame from the SWAP space.*
            - In this case, we will write the contents of the victim to the frame acquired from swap.
            - Then, in the page table entry that points to victim frame we will change the present bit to 0 which indicates the frame is no longer in the FFS region and we will update the base address to point to the frame that was acquired from swap.

- We also update the dss_tab entry of the swap frame with the pid and the virtual page number values of the victim page.
- Next, we return the victim frame, which can now be safely used as a free frame.

ii. *We don't get a frame from the SWAP space. This means the SWAP space is full.*

- In this case, we must select a frame to be evicted from swap, we call this swap_victim. The swap_victim is the frame that is present in FFS and SWAP and is clean.
- After selecting the swap victim, we follow the same steps of (i)

2. **The victim is present SWAP. So, we have a frame in swap that is same as the victim frame, we call this swap_frame. Now there can be two cases:**

i. *The victim selected from the FFS region is dirty. This means the frame has been updated and thus, we will have to write it to the SWAP space. We follow the following steps.*

- Since, the victim is present in SWAP space, we know its location in SWAP space (i.e. swap_frame). So, we will write the contents of the victim to the swap_frame.
- Then, in the page table entry that points to victim frame we will change the present bit to 0 which indicates the frame is no longer in the FFS region and we will update the base address to point to the swap_frame.
- We also update the dss_tab entry of the swap_frame with the pid and the virtual page number values of the victim page.
- Next, we return the victim frame, which can now be safely used as a free frame.

ii. *The victim selected from FFS region is not dirty. In this case we skip the first step of (i) and follow the remaining steps.*

Files edited/added:
Header files – process.h – added fields in the proctab
Paging.h – defined macros, structures and tabs
Xinu.h , prototypes.h
System files:
Directory.c – contains helper functions for managing page directory
Page_tables.c – Contains helper functions for managing the page tables.
Frame_management.c – Contains helper functions for managing the frames
Frame_tables.c – contains functions for managing the frame tabs.
Create.c, vcreate.c, vmalloc.c, vfree.c, initialize.c , pagefault_handler_disp.S, pagefault_handler.c, resched.c , kill.c

This project successfully implements all the functionality mentioned in the spec file. All the specifications are considered and tested for.