Lab5 Phase 1

Contributors: Suraj Aralihalli, Tejas Prashanth

November 2022

Contributions

1. Tejas Prashanth
   (a) Design of Kernel data structures for Virtual Memory Allocation and Dellocation
   (b) Design of Kernel data structures for region D and E1
   (c) Identity Mapping Development of NULLPROC
   (d) Development and testing of vmhfreemem
   (e) Dellocation of paging structures when a process gives up acquired pages (Development and Testing)
   (f) Page table allocation logic in pgfhandler (Development and Testing)
   (g) Process creation and TLB Flush
   (h) Pgfdisp testing
2. Suraj Aralihalli
   (a) Design of Kernel data structures for Virtual Memory Allocation and Dellocation
   (b) Design of Kernel data structures for region D and E1
   (c) Identity Mapping Testing
   (d) Development and testing of vmhgetmem
   (e) Dellocation of paging structures during process termination (Development and Testing)
   (f) Frame allocation logic (E1) in pgfhandler (Development and Testing)
   (g) Access violation check in pgfhandler
   (h) Pgfdisp development

5. Enabling, initializing, and managing virtual memory

5.3 Process creation

Per-process kernel structure to track which pages are free in VF

```
// struct to store vmemblk
struct vmemblk {
  uint32 npages;
  struct vmemblk* mnext;
  v32addr_t start_addr;
};
```

We have used a freelist datastructure to track the free pages in Vf for each process. This per process datastructure resides in region C and formed with nodes whose structure is defined above (*vmemblk*). When a process is created, we initialize this kernel structure and save the pointer as the process table field called vmemlist_ptr. In other words, prptr→vmemlist_ptr holds the head of the freelist. The design of this freelist is similar to XINU's memlist. However, the fundamental difference lies in the usage of start_addr to map the virtual address space.

Head node:

1. npages : Indicates the total free pages in the process's virtual space
2. mnext : Stores the pointer to first functional node in the freelist.

Subsequent node:

1. start_addr: Maps to some address in the Vf.
2. npages : Indicates the total free pages available for allocation starting from start_addr.
3. mnext : Stores the pointer to next functional node in the freelist.

A call to vmhgetmem(msize) returns the address in Vf where msize number of pages are available for allocation. This call modifies the freelist according as described in Figure 1. When vmhfreemem(addr, msize) is called the same datastructure is updated to reflect the free pages. The nodes contain only the start_addr of the available pages. For example, when 10 pages where allocated in Figure 1, the start_addr is updated to $start\_addr + (msize * NBPG)$ and npages is updated to $npages - msize$. When the process is terminated, this data structure is gracefully removed from Region C.

The following code is used to set up this kernel datastructure when a process is created the first time.

```
void setup_vmemlist(pid32 pid)
{
    struct procent *prptr = &proctab[pid];

    //initialize the vmemlist
    prptr->vmemlist_ptr = create_vmemblk_node();

    prptr->vmemlist_ptr->npages = prptr->hsize;
    prptr->vmemlist_ptr->start_addr = 0x0;
    prptr->vmemlist_ptr->mnext = create_vmemblk_node();

    prptr->vmemlist_ptr->mnext->npages = prptr->hsize;
    prptr->vmemlist_ptr->mnext->start_addr = REGIONSTART_F * NBPG; // (4096 * 4096)
    prptr->vmemlist_ptr->mnext->mnext = NULL;
}
```

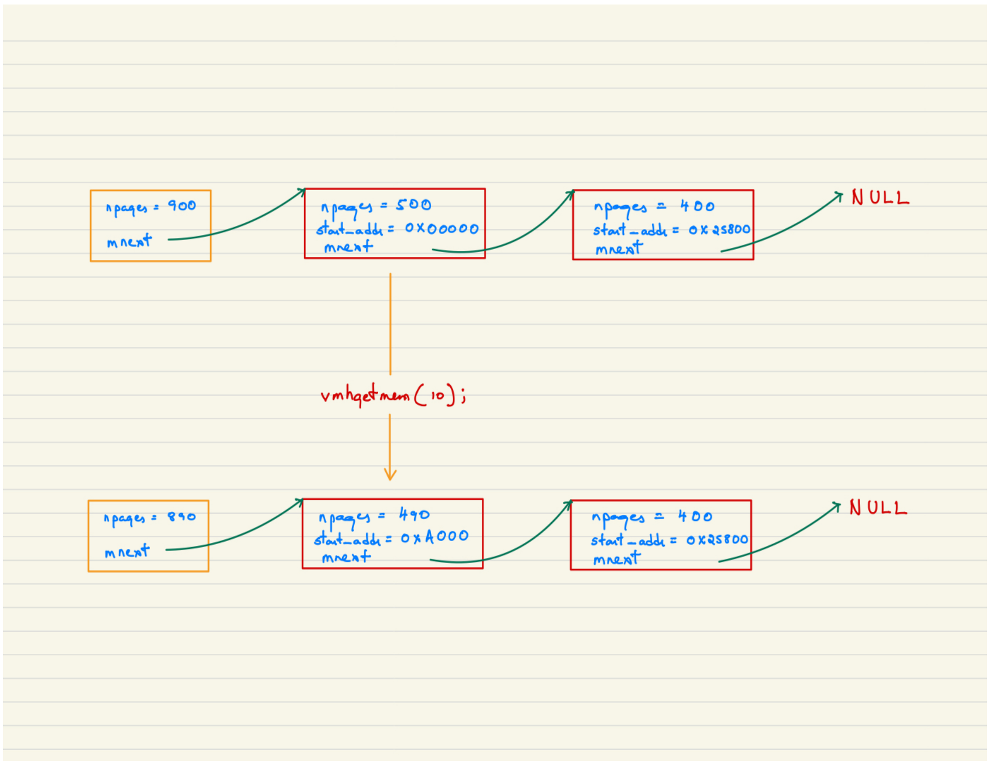

Figure 1: vmemlist per-process datastructure to track free pages in Vf

6. Managing physical memory regions

6.1 Region D: page directories and tables

```
// struct to indicate which frames are free in Region D
struct fholder{
    unsigned int frame_pres : 1;    /* frame is present? */
    pid32  owner_process;           /* Owner process ID */
    v32addr_t vaddr;                /* Virtual address */
    uint16 nentries_allocated;      /* Number of pages allocated */
};

// fHolderListD is an array of size NFRAMES_D
extern struct fholder fHolderListD[];
```

We have used fHolderListD array to track the frames in Region D. This datastructure resides in Region B and formed as an array of struct as defined above (*fholder*). The fundamental purpose of this datastructure is to facilitate ease of frame allocation in region D when a process is created, and deallocation when a process terminates. In addition to that, this data structure supports easy identification of page table when all its page table entries are unused (i.e pt_pres bit is 0) so that the page table can freed proactively (using nentries_allocated mechanism).

```
        // Check if page table has 0 used page table entries
        if(fHolderListD[index_fHolderListD].nentries_allocated == 0) {
            /* Free frame in region D */
            free_frame_in_regionD((v32addr_t) page_table_addr, owner_pid);

            /* Make P bit in page directory entry 0 */
            reset_page_directory_entry(page_dir_entry);
        }
```

The components of this structure include:

1. frame_pres : Indicates if the frame is occupied (could be a page table or a directory) or free.
2. owner_process: Owner id of process that occupies this frame.
3. vaddr: Start (absolute) address of the frame in which the corresponding page table/directory is stored
4. nentries_allocated: Number of frames allocated (number of used page table entries with pt_pres bit 1) in the page table (For page directory, this field is unused)

When a new process is created a free frame is captured from Region D using $get\_empty\_frame\_from\_regionD(pid)$. This frame is used to set up the page directory and the physical address of this frame is updated in the corresponding process table field (prptr→page_dir_addr). New frame can also be acquired while executing the page fault handler if the page table is missing. When a new page is acquired from Vf, bookkeeping is done by adding a new entry to the corresponding page table. Consequently nentries_allocated for the relevant page table is updated in fHolderListD. This field can be used to determine if the page table can be freed, in O(1) time without having to iterate through all the page table entries in the page table for pt_pres bit.

Including the owner_process in the structure gives an advantage at the time of process termination. The same fHolderListD structure acts as an inverted page table avoiding the need to traverse all the page directory entries and page table entries to identify which frames to free. Following mechanism can be used to free the frames

```
void purge_frames_fHolderListD(pid32 pid)
{
    uint32 i;
    for(i = 0; i < NFRAMES_D; i++) {
        if(fHolderListD[i].owner_process == pid) {
            fHolderListD[i].frame_pres = 0;
            fHolderListD[i].owner_process = -1;
            fHolderListD[i].vaddr = 0x0;
            fHolderListD[i].nentries_allocated = 0;
        }
    }
}
```

7. Submission in two phases

7.1 Testing

We test the implementation of the virtual memory management unit in the following manner-

1. We test the identity mapping of regions VA-VE and VG to A-E and G respectively after paging has been enabled by the null process. In order to test this, we use a list of addresses, each one being in a different region. Thereafter, we perform a read operation to that memory location and verify that no page fault is generated due to the read operation. For example, to verify identity mapping between VG and G, we take a random address

2. We test the most basic functionality of virtual memory allocation by allocating a single page in VF using vmhgetmem(). We verify that the call to vmhgetmem() succeeds and attempt to write 100 bytes (i.e as 100 write independent operations) to the page starting from the virtual address returned by vmhgetmem() (say x). We verify that the first write results in a page fault and subsequent write operations complete without a page fault. We read 100 bytes starting from virtual address X, print the value read, and verify that the value read matches the value written. Thereafter, we deallocate the page by invoking vmhfreemem() and verify that all frames in E1 pertaining to this process are free.

3. We test whether pages that are allocated in VF and not explicitly freed using vmhfreemem() are deallocated when the process terminates. In order to test this, we allocate a page using vmhgetmem(), write 100 bytes of data into the page, and terminate the process. During process termination, we verify that all frames that were allocated in E1 pertaining to this process have been freed.

4. We test support for the allocation of multiple pages in VF by a single process. We allocate 2 pages in VF using vmhgetmem() (say it returns virtual address X) and write 5000 bytes starting from X. We verify that this results in 2-page faults. We read the data starting from virtual address X and verify that it matches the data that was written. We deallocate the pages using vmhfreemem().

5. We test if access-after-free is not allowed. We allocate a page using vmhgetmem() in VF (with virtual address X) and free it using vmhfreemem(). Thereafter, we try to read 1 byte starting from X and verify that this is disallowed and results in process termination.

6. We test if the kernel data structure used by vmhfreemem() has been implemented correctly. In order to test this, we allocate 2 pages using vmhgetmem() in VF (which returns a virtual address X). We write 1000 bytes to memory starting from virtual address X. Thereafter, we deallocate pages using 2 separate calls to vmhfreemem() and verify that both succeed.

7. We test support for detecting access to unallocated memory. Note that this is different than 5) in that we verify whether access to *unallocated memory* (not *memory that was previously freed)* is forbidden. We do this by allocating a page in VF using vmhgetmem() (which returns a virtual address X). We write to unallocated memory (X + 4096), which should be forbidden and should result in process termination

8. We verify that a double-free operation is forbidden. We allocate a page using vmhgetmem() and write 100 bytes to the page. We read the data written and deallocate the page using vmhfreemem(). We make a second call to vmhfreemem() and verify that it fails.

9. We test whether multiple processes are able to write to the same *virtual address in VF*. We create 3 processes - process G, process H, process I. Process G allocates 2 pages and writes 5000 bytes to virtual address X, process H allocates 1 page and writes 4000 bytes starting from the same virtual address X and process I allocates 3 pages and writes 9000 bytes starting from virtual address X. In each process, we read the data starting from X and print it on the console. We verify that each process sees only the data that it has written. Moreover, we give each process the same priority to verify that memory allocation, deallocation, and page faults are handled in an atomic manner when processes are interleaved. We deallocate the pages corresponding to each process and verify that all frames pertaining to a given process are free.

Bonus

Please refer code.