# Optimal Xinu Paging

Lab 6

Vikrant Gajria
CS503
vgajria@purdue.edu

# Notes: Caveats in Testing!!!

1. Freelist is implemented in virtual space so if a process needs one memory page, 2 frames are needed - one for the page and other for memblk node. So please make sure to **free 3 memory locations** if you want 1 memory page allocated. **Refer my test case 5 below for explanation**.

2. VF -> E2 **mapping is in the page table itself**. So, if a frame is swapped, the page table entry will still exist!

# Addressing problems from Lab 5 submission

1. **vmhfreemem: does not sanity check arguments**

Since my implementation re-uses freemem.c with virtual addresses, I had a sanity check for arguments. But, it did not check whether the given pages were allocated by the process or not. I have added the second sanity check.

```
// No memory to free, or invalid address
if ((nbytes == 0) || ((uint32) blkaddr < MINVHEAP)
        || ((uint32) blkaddr > MAXVHEAP)) {
   restore(mask);
   return SYSERR;
}

// Check if all msize pages are allocated
uint16 i;
for(i=0 ; i<msize ; i++) {
   uint32 addr = ((uint32) blkaddr) + (i * NBPG);
   if(prptr->pralloc[VHNUM(addr)] == 0) {
```

```
            log_mem("vmhfreemem - page %d is not allocated \n", VHNUM(addr));
            restore(mask);
            return SYSERR;
        }
    }
```

### 2. vmhfreemem: a frame holding a page table is not freed after all related pages for region F are freed

I did not add a reference counting mechanism in lab 5. I have added this in lab 6. I didn't get time to work on this, I do have some functions in place but couldn't figure out where to increment the reference count of PT.

After a naive implementation, this was difficult to test because there always exists memblk of freelist in the vheap, therefore the reference count never goes down to 0...

# Space-optimal VF -> E2 Mapping

# by re-using Multi-level Page Table itself

**Space sub-optimal approach** An approach to store VF -> E2 mapping is to keep a per-process map of page number -> frame number in E2. But wait, we already have a data structure that maps VF -> E1. It is the Multi-level Page Tables in region D. Why not just re-use the multi-level page table to map VF -> E2?

**Multi-level page table for swapped pages** By using 1 of the **pt_avail** bits in struct pt_t, I mark a given frame as "swapped". This indicates that the **pt_base** in the PTE points to a frame in E2. A swapped page's PTE has **pt_pres = 0** to cause a page fault on access, which would be handled again by restoration of the frame or swapping the frame with a frame in E1.

**Advantages of the approach**

1. Re-using page table and its multi-level nature's space optimality
2. Easy way to retrieve VF->E2 mapping

**Disadvantages of the approach**

1. Works well in this lab because E2 is in memory
2. If we want to use this with a disk store, then we can only have almost 2^20 frames on disk because of struct pt_t's pt_base being 20 bits wide.
3. **Takes up space in Region D instead of Region B (PLEASE NOTE THIS FOR TESTING)**

# O(1) Time-optimal Eviction FIFO Queue

**Doubly linked-list Queue** The FIFO eviction queue for occupied E1 frames in implemented as a doubly linked list in the inverted page table of Lab 5. Every cell in the inverted page table now has an **fr_next** and **fr_prev** field for linking to next and previous cells.

**Array storage** This FIFO queue's list nodes are stored in the inverted page table itself, which is a random-access array. Therefore, **if any frame is evicted or freed, its list node entry in the FIFO queue can be accessed in O(1) time and deleted efficiently**.

**LRU:** An implementation for clock-hand eviction algorithm, as discussed in class, is given in Bonus section but not included in the current code.

# Modularized Page-fault Handler Logic

The following is the state to action mapping of page fault handler

| Is E2 full? | Is E1 full? | Is Page swapped? | Action |
|---|---|---|---|
| X | No | No | AssignNew |
| X | No | Yes | RestoreOld |
| No | Yes | No | EvictFrame then AssignNew |
| No | Yes | Yes | EvictFrame then RestoreOld |
| Yes | Yes | No | FrameBlock |
| Yes | Yes | Yes | SwapFrames |

Note the function names and their actions:

**AssignNew** = Assign new E1 frame to faulting page

**RestoreOld** = Restore old data frame in E2 to new E1 frame

**EvictFrame** = Evict a used E1 frame to E2 in FIFO manner

**FrameBlock** = Block faulting process till frames in E1 or E2 get free

**SwapFrames** = Swap a used frame in E1 with a old data frame in E2

This state and action mapping is implemented in **pgfhandler** as simple while if-else block as shown below.

```
if(swapped) {
    // Old frame is in E2
    if(e1full && e2full) {
        // Swap it into E1
        swapframe();
    } else {
        if(e1full) {
            // Make space in E1
            evictframe();
        }

        // Move old frame into E2
```

```
            restoreframe();
        }
    } else {
        // Fresh frame in E1 required
        while(e1full && e2full) {
            // Block until space in
            // E1 or E2
            frameblock();
        }

        if(e1full) {
            // Make space in E1
            evictframe();
        }

        // Assign a new frame
        mapfreeframe();
    }
```

The system is extremely **modular**.

# Testing and Logging all changes

I designed one test case that incrementally tests all states of the PGF handler. Here is a step by step explanation of the test cases.

**Case 1.** E1 has space and process tries to vmhgetmem() a new page.

In my implementation, **two frames will be occupied** (2024 and 2025). Frame 2024 is used as the frame for page 4096. Frame 2025 is used for memblk information.

Here is the log for requesting two pages, x of 1 page and y of 1 page. The bolded lines show the change made by PGFhandler to resolve this fault. It assigns a frame to the page. @P is parent process and shows what is happening.

@P Parent process started

$ CASE 1. E1 is free, new page

$ Here 2024th and 2025th frame should be assigned with mapnewframe

@P Requesting one page

vmhinit - Set next=0x01000000 length=4194304

----- PAGE FAULT -----

- PID: 2

- Error code: 0x0

- Errorneous address: 0x1000004

- Allocated: 1

- Page number: 4096

- PDIDX: 4

- PTIDX: 0

- Page Table is absent

- E2Full=0 E1Full=0 Swapped=0

- **Mapping new E1 frame to faulting address**

**- New frame 2024 is now occupied by 2**

**-- Change PTE from 0x00000000 to 0x007e8003**

----- ---------- -----

----- PAGE FAULT -----

- PID: 2

- Error code: 0x0

- Errorneous address: 0x1001000

- Allocated: 1

- Page number: 4097

- PDIDX: 4

- PTIDX: 1

- E2Full=0 E1Full=0 Swapped=0

- **Mapping new E1 frame to faulting address**

**- New frame 2025 is now occupied by 2**

**-- Change PTE from 0x00000000 to 0x007e9003**

----- ---------- -----

@P x = aaaaaaaaaa

@P Requesting one more page

----- PAGE FAULT -----

- PID: 2

- Error code: 0x0

- Errorneous address: 0x1002000

- Allocated: 1

- Page number: 4098

- PDIDX: 4

- PTIDX: 2

- E2Full=0 E1Full=0 Swapped=0

- **Mapping new E1 frame to faulting address**

**- New frame 2026 is now occupied by 2**

**-- Change PTE from 0x00000000 to 0x007ea003**

----- ---------- -----

@P y = bbbbbbbbbb

**Case 2.** E1 is full but E2 has space. Here, FIFO replacement policy should evict frame 2024.

In the same state as above, I manually mark all the frames in E1 as occupied by null process. Now, when process P tries to get another page, the fault tells us that a frame 2024 was evicted.

```
$ CASE 2. E1 is full, new page
$ The 2024th frame should be evicted and re-assigned
@P Setting all frames in E1 to occupied by null
@P Requesting one more page, should cause eviction
----- PAGE FAULT -----
- PID: 2
- Error code: 0x0
- Errorneous address: 0x1003000
- Allocated: 1
- Page number: 4099
- PDIDX: 4
- PTIDX: 3
- E2Full=1 E1Full=0 Swapped=0
- Evicting an E1 frame to E2
-- Selected victim 2024 --> destination 3048
-- Copied destination 0x00be8000 <-- victim 0x007e8000
-- Changed PTE from 0x007e8063 to 0x00be8260
- Mapping new E1 frame to faulting address
- New frame 2024 is now occupied by 2
-- Change PTE from 0x00000000 to 0x007e8003
----- ---------- -----
@P z = ccccccccc
```

**Case 3.** E1 and E2 are full. Let's try accessing page 4096 again, the one that we evicted out to E2. In this case, swapping should occur.

Note that frame 2024 (the one with page 4096) was evicted in Case 2. Now, by FIFO, 2025 should be evicted and 4096 should be mapped to 2025, then the data should be copied back. This is happening below, since 2025 contained 'bbbbbbbbbb' but now it contains 'aaaaaaaaaa'.

```
$ CASE 3. E1 is full, E2 is full, old page
$ The frame with x should be swapped to 2025
@P Setting all frames in E2 to occupied by null
@P Printing all allocated pages
@P x = ----- PAGE FAULT -----
- PID: 2
- Error code: 0x0
- Errorneous address: 0x1000000
- Allocated: 1
- Page number: 4096
- PDIDX: 4
- PTIDX: 0
- E2Full=1 E1Full=1 Swapped=1
- Swap an E1 frame with old E2 frame of faulting address
-- Copied temp <-- source 0x00be8000
- Evicting an E1 frame to E2
-- Selected victim 2025 --> destination 3048
-- Copied destination 0x00be8000 <-- victim 0x007e9000
-- Changed PTE from 0x007e9063 to 0x00be8260
- Mapping new E1 frame to faulting address
- New frame 2025 is now occupied by 2
-- Change PTE from 0x00be8260 to 0x007e9063
-- Copied destination 0x007e9000 <-- temp
----- ---------- -----
aaaaaaaaaa          <——- Data remains the same
```

**Case 4.** More swapping. I also try to access y and z which contain ‘bbbbbbbbbb’ and ‘ccccccccc’. Notice how destination 3048 is the one that contained x and then y from Case 3. This info is from VF -> E2 mapping.

@P y = ----- PAGE FAULT -----

- PID: 2
- Error code: 0x0
- Errorneous address: 0x1001000
- Allocated: 1
- Page number: 4097
- PDIDX: 4
- PTIDX: 1
- E2Full=1 E1Full=1 Swapped=1
- Swap an E1 frame with old E2 frame of faulting address

-- Copied temp <-- source 0x00be8000

- Evicting an E1 frame to E2

**-- Selected victim 2026 --> destination 3048**

-- Copied destination 0x00be8000 <-- victim 0x007ea000

-- Changed PTE from 0x007ea063 to 0x00be8260

- Mapping new E1 frame to faulting address
- **New frame 2026 is now occupied by 2**

-- Change PTE from 0x00be8260 to 0x007ea063

-- Copied destination 0x007ea000 <-- temp

**----- ---------- -----**

**bbbbbbbbbb**

@P z = ----- PAGE FAULT -----

- PID: 2
- Error code: 0x0
- Errorneous address: 0x1002000
- Allocated: 1
- Page number: 4098
- PDIDX: 4
- PTIDX: 2
- E2Full=1 E1Full=1 Swapped=1
- Swap an E1 frame with old E2 frame of faulting address

-- Copied temp <-- source 0x00be8000

- Evicting an E1 frame to E2

**-- Selected victim 2027 --> destination 3048**

-- Copied destination 0x00be8000 <-- victim 0x007eb000

- Mapping new E1 frame to faulting address

**- New frame 2027 is now occupied by 2**

-- Change PTE from 0x00be8260 to 0x007eb063

-- Copied destination 0x007eb000 <-- temp

**----- ---------- -----**

**cccccccccc**

**Case 5.** Blocked when no space in E1 or E2. A new child process is created. From the cases above, E1 and E2 are completely filled up. Now, the child process tries to get a page but it cannot. **It needs 2 frames in my implementation, one for the page itself, the other for memblk.** So, it goes into a custom sleep queue that I created (did not use Xinu's legacy queue because it was faster to unit test a single file/ module. Also my queue uses less storage because its a queue of 16-bit ints).

```
$ CASE 4. Frame block
$ The child processes should be blocked
@C Child process started
@C Requesting one page, this should block
vmhinit - Set next=0x01000000 length=4194304
----- PAGE FAULT -----
- PID: 3
- Error code: 0x0
- Errorneous address: 0x1000004
- Allocated: 1
- Page number: 4096
- PDIDX: 4
- PTIDX: 0
- Page Table is absent
- E2Full=1 E1Full=1 Swapped=0
- Blocking
----- ---------- -----
@P Parent frees frame
----- PAGE FAULT -----
- PID: 3
- Error code: 0x0
- Errorneous address: 0x1000004
- Allocated: 1
- Page number: 4096
- PDIDX: 4
- PTIDX: 0
- Unblocked from frame block!
- Mapping new E1 frame to faulting address
- New frame 2026 is now occupied by 3
-- Change PTE from 0x00000000 to 0x007ea003
----- ---------- -----
----- PAGE FAULT -----
- PID: 3
- Error code: 0x0
```

**- Errorneous address: 0x1001000 <————————- This one is for memblk node of free list**

- Allocated: 1

**- Page number: 4097**

- PDIDX: 4

- PTIDX: 1

**- E2Full=1 E1Full=1 Swapped=0**

**- Blocking**

----- ---------- -----

----- PAGE FAULT -----

- PID: 3

- Error code: 0x0

**- Errorneous address: 0x1001000**

- Allocated: 1

**- Page number: 4097**

- PDIDX: 4

- PTIDX: 1

**- Unblocked from frame block!**

**- Mapping new E1 frame to faulting address**

**- New frame 2027 is now occupied by 3**

-- Change PTE from 0x00000000 to 0x007eb003

----- ---------- -----

**@C Child with pid 3 got page 4096**

kill - deallocating 1033

kill - deallocating 1034

**kill - deallocating 2026**

**kill - deallocating 2027**

# Bonus

### Modularity

Please refer README.2 in the submission folder to gauge the modular approach of my implementation. Everything is in its own file and each data structure is managed by its own function. This is kind of an object oriented approach, except only the encapsulation and abstraction aspect of OOP is used.

The state table mentioned in a section above highlights the minimal coding and maximal re-use of functions.

### Performance

- VF -> E2 mapping re-uses page tables in region D, therefore the mapping can be found in O(1) time and uses minimal space.
- Frame eviction policy's FIFO list is O(1) insertion, random-access deletion, and dequeue by using a doubly-linked list with arrays.

### Attention to detail

- Extreme care in designing the file layout and pgfhandler's reusability.
- Wakeup() makes use of number of processes waiting in queue and number of **free frames in E1 or E2**. This means that, suppose a process with frame in E2 was killed. Then, a waiting process wakes up and identifies that it can evict a frame to E2 then occupy E1. This is done through the if-else ladder explained above.

**Alternative to FIFO eviction:** Second-chance replacement algorithm

While it is mentioned in the Lab handout that default replacement policy is FIFO, I also have an implementation of second-chance/ clock hand replacement policy. The code is mentioned below. This is less efficient in terms of time complexity but this is the LRU approach that we covered in class. My current FIFO eviction is O(1). This is O(n) where n is the number of used frames in E1.

```
/* Second chance page replacement policy */
      frame_t *second_chance(void) {
        frame_t *frame;
        pt_t *pte;

        while (1) {
          /* Start at the head of the linked list */
          frame = fbqhead;

          /* Iterate through the list of frames */
          while (frame != NULL) {
            pte = frame->fr_pte;

            /* Check the reference bit of the page table entry */
            if (pte->pt_acc) {
              /* Clear the reference bit and move to the next frame */
              pte->pt_acc = 0;
              frame = frame->fr_next;
            } else {
              /* This frame can be replaced, so remove it from the list */
              if (frame->fr_prev != NULL) {
                frame->fr_prev->fr_next = frame->fr_next;
              }
              if (frame->fr_next != NULL) {
                frame->fr_next->fr_prev = frame->fr_prev;
              }
              return frame;
            }
          }
        }
      }
```