CS350 Design Questions

--- Yong Dai, Frank Li, Sherry Wang

Question 1: *Briefly describe the data structure(s) that your kernel uses to manage the allocation of physical memory. What information is recorded in this data structure? When your VM system is initialized, how is the information in this data structure initialized?*

We created the following `coremap` structure, and we used an array of `coremap`s (we call it `coremap_table`) to keep track of the allocation status of physical memory:

```
struct coremap {
    vaddr_t vaddr;   // faultadress passed from page fault
    paddr_t paddr;   // calculated value of the array index * PAGE_SIZE + firstpaddr
    char occupied;   // indicate the status of the coremap;
    int length;      // records the number of contiguous blocks that
                     // a file holds in the array
    pid_t t_pid;     // pid of the thread; prepared for fork
    void* addrSpace; // record the address space of the current thread;
                     // prepared for SWAPFILE
};
```

The function `vm_bootstrap()` is called for initialization of the VM system. It first calls `mips_ramsize()` to get the total size of available memory, stored in `ramsize`. Second, `ramsize` and `PAGE_SIZE` are used to determine the number of frames that can actually potentially be used for kernel and user programs uses, and can also determine how many `coremap` structures are in `coremap_table`. `ram_getsize()` is then called to get the upper bound (`start`) and the lower bound (`end`) of the rest of the available physical memory. Last, we update the number of available memory frames (`num_entries`) by subtracting `end` from `start` and divide it by the `PAGE_SIZE`.

Question 2: *When a single physical frame needs to be allocated, how does your kernel use the above data structure to choose a frame to allocate? When a physical frame is freed, how does your kernel update the above data structure to support this?*

As we discussed in Q1, the kernel uses the `coremap_table` (an array of `coremap`s) to record the coremap structures. Every time when `vm_getppages()` is called, it will loop through the array to find the next available block by checking the `occupied` field. We must also ensure that the vacant blocks are contiguous. If we find such n contiguous blocks in `coremap_table`, we update the coremap information (eg. record its `vaddr`, `paddr` and `addrspace`, update its status to `occupied = 1`, etc). Every time when a physical frame is freed, our kernel just looks for the `coremap` structure in `coremap_table` according to the physical frame address and resets each field in it to the default value, indicating this slot in memory is reusable.

Question 3: *Does your physical-memory system have to handle requests to allocate/free multiple (physically) contiguous frames? Under what circumstances? How does your physical-memory manager support this?*

Yes, our system does. This happens when the system allocates/frees the page tables (three arrays of page table entries that keep track of text, data and stack segments separately in the physical memory.), and a swap file table (an array of swap file entry structures) which keeps track of the contents of physical memory frames actually in the SWAPFILE.

First of all, we used two indexes to loop through the coremap table to see whether a consecutive blocks of physical memory frames (lets say `n` frames) can fit into it - we used an index `i` to keep track of the starting point where that blocks of frames could potentially begin to fit; then, we used another index `j` to loop from index `i` to the end of the `coremap_table` to see whether we have `n` empty slots that this chunk of memory frames can fit. If it can, just put it into coremap and return the physical memory of the starting point of this chunk of memory frames; if not, we recursively pick one victim (using round-robin) and swap it into SWAPFILE until enough empty slots are present. And also, the way we keep track of which contiguous blocks of memory is touched by the same kmalloc is that we remember the number of memory frames that are sub-frames of the current frame so that we know how many sub-frames follow this current frame. The information mentioned above are all kept in the `coremap` structure.

Question 4: *Are there any synchronization issues that arise when the above data structures are used? Why or why not?*

Yes, the variable that keeps track of the next victim need to be protected by a lock. If we don't have a lock one thread would override other threads' change. It would make the system become inefficient or unstable.

Question 5: *Briefly describe the data structure(s) that your kernel uses to describe the virtual address space of each process. What information is recorded about each address space?*

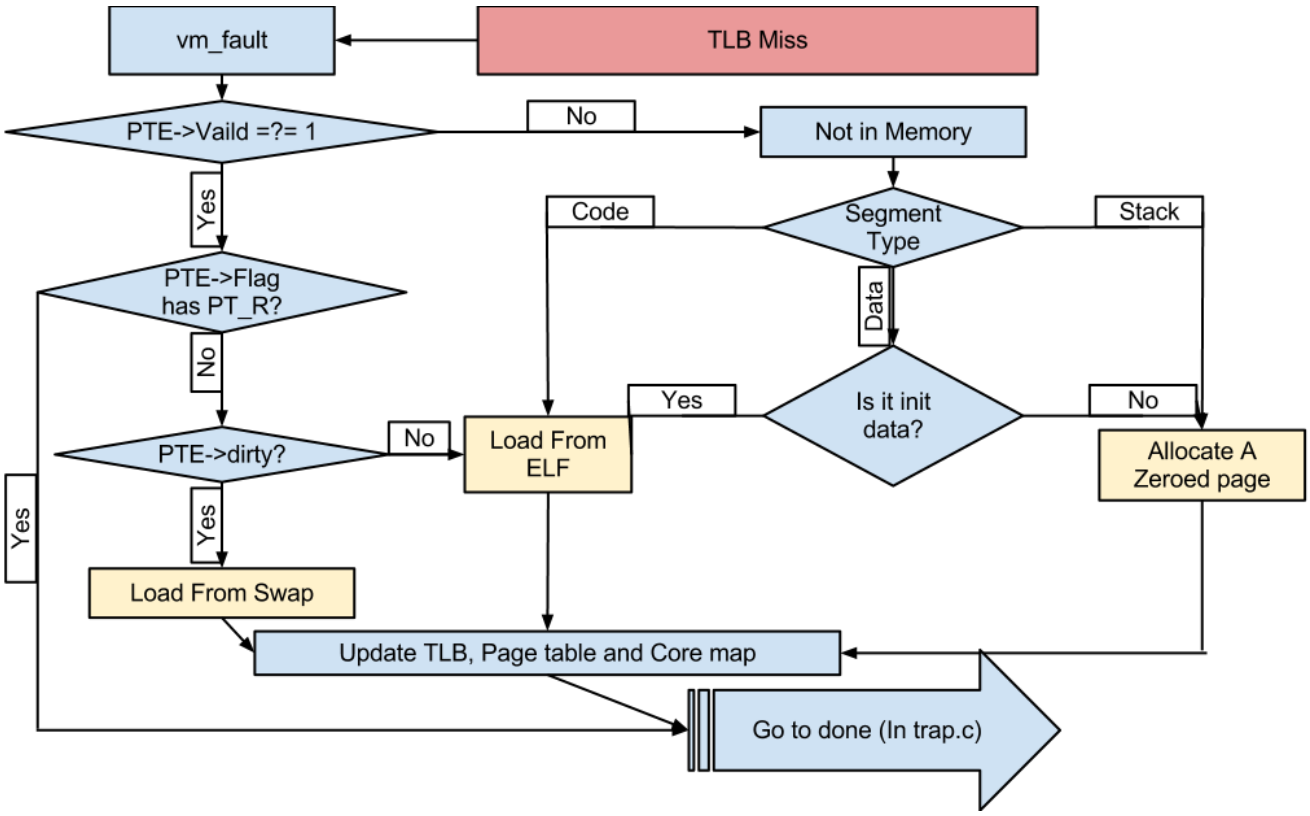
```
struct addrspace {
      vaddr_t as_vbase1;
      size_t as_npages1;
      vaddr_t as_vbase2;
      size_t as_npages2;
      struct vnode *elf_file_vnode;
      struct Pte **pt_code;
      struct Pte **pt_data;
      struct Pte **pt_stack;
      char* progName;
};
```

In our `addrspace` structure, we have `as_vbase1` and `as_vbase2`, which represent the virtual address base of text segment and that of data segment respectively; `as_npages1` and `as_npages2` represent the maximum amount of pages that text segment and data segment can hold respectively; `elf_file_vnode`, as its name suggests, it holds the vnode of ELF file to be loaded when user program runs; `pt_code`, `pt_data` and `pt_stack` are page tables for text, data and stack segment respectively, and they are to be of size `as_npages1`, `as_npages2` and `12`; finally, `progName` holds the name of the running user program, which is useful for `as_copy()`.

Question 6: *When your kernel handles a TLB miss, how does it determine whether the required page is already loaded into memory?*

When kernel has a TLB miss it would trap into `vm_fault`. `vm_fault` will calculate which segment the fault address belongs to and check the page table segment's `valid` flag. If the `pte->valid` flag is `1` and `pte->flag` has `PT_R` flag turned on, this indicates that the page is already loaded into the memory.

Question 7



Question 8: *How does your kernel ensure that read-only pages are not modified?*

Our kernel uses a TLB and 3 page tables for code data and stack segment. When a page is read-only, the kernel will set the flag in corresponding page table entry to disable `PF_W` bit (disable writable bit) and set only the `TLBLO_VALID` flag in the corresponding TLB entry. When a user program tries to write to a read-only page, it would trap into kernel and kernel will handle it (in `vm_fault`).

Question 9: *Briefly describe the data structure(s) that your kernel uses to manage the swap file. What information is recorded and why? Are there any synchronization issues that need to be handled? If so what are they and how were they handled?*

`swapLookupTable` is an array of `swapEntry`.

```
struct swapEntry {
    vaddr_t addr;                          // virtual address.
    int offset;                            // offset into the file
    struct addrspace* belongToAddrsapce;   // the address space the virtual
                                           // address belongs to
};
```

`swapFile` is a vnode that points to a file which stores memory to the disk (i.e., SWAPFILE).

The swap operation uses an array to track where the memory is located and uses file operations to read and write memory into file.

Both `addrSpace` and `vaddr` are use to determine which swap file entry it belongs to. and the offset is saved to determine where the memory locates in file.

Both lookup array and swap file is guarded by locks to ensure threading safety.

Question 10: *What page replacement algorithm did you implement? Why did you choose this algorithm? Is this a good choice, why or why not? What were some of the issues you encountered when trying to design and implement this algorithm?*

We used the given round-robin algorithm to find the page victim which needs to be replaced. We chose this algorithm because it is straightforward and easy to handle (easy to implement and debug). However, on the other hand, it is inefficient. There is no much trouble with this algorithm but it indeed took us a significant amount of time to debug the process of correctly copying the physical memory content to physical memory from the SWAPFILE.