Assignment-5: Prasad Khedekar and Philip Chakram
Design Document:

TLB Handling:

An important aspect of the assignment involved how TLB faults were handled. As mentioned in the specifications, there are two alternatives on how the state of the TLB is initialized after a context switch. In our VM system, we decided to invalidate all the TLB entries on a context switch. The entries are then re-loaded after the TLB faults as pages are referenced. For accomplishing this task, a linked list of free TLB entries is maintained, which is re-initialized on a context switch. After re-initialization, each of the TLB entries is invalidated after propagating the information held by each entry to the respective page table for maintaining the sanity of the pages in the page table. This information includes whether the page is dirty, valid, if invalid then the location of the page in the disk.

Paging Capability:

Paging is an invaluable feature which helps to mitigate the limitation on size of available physical memory. User programs can make use of the extended virtual memory on top of the limited physical memory due to the concept of paging. In our VM system, we have implemented paging by making use of page table and backing store on the disk. The intricate details about the data structures developed for implementing paging are discussed in the subsequent sections.

When the operating system selects a location in physical memory in which to place the new page, the space may already be occupied. In this case, the operating system must evict that other page from memory. If the page has been modified or does not currently have a copy on disk, then the old page must first be written to disk before the physical page can be reallocated. If the old page has not been modified and already has a copy on disk, then the write to disk can be avoided. The appropriate page table entry must be updated to reflect the fact that the page is no longer in memory.

As with any caching system, performance of a virtual memory system depends on the policy used to decide which things are kept in memory and which are evicted. On a page fault, the kernel must decide which page to replace. Ideally, it will evict a page that will not be needed soon. We made use of a Least-Recently Used (LRU) approximation algorithm: the Second-chance algorithm (Clock algorithm), for choosing a page to evict from the physical memory. If the page is modified, as denoted by the dirty bit in our system, it is written to the swap space on the disk.

Solution Integration to existing OS 161 kernel code:

Code integration for managing virtual address space:

The kernel uses the address space structure to keep track of the virtual address space, which contains the segments array, the main structure keeping track of the space. These structures are described in subsequent sections. Each address space has information about the segments from the elf file, and the stack segment. These segments then contain the page tables, so the address space keeps information about the current status of each virtual page. The address space also contains a vnode for the elf file, allowing for demand paging from the elf file. The number of segments is also stored.

Coremap Integration Effect:

Core map is a sort of reverse page table that is indexed by its physical page number and contains the virtual address and address space identifier for the virtual page currently backed by the page in physical memory. The core map acts as an intermediary and helper function for the implementation of paging. Its integration has the following effects discussed in detail:

1. Handling single physical frame allocation:

When a single physical frame is needed, the kernel will first check the core maps free frame list, and immediately return the first frame it finds, setting kernel use to 1 so that it will not be altered, and removes it from the free list. If no free frames are found, our kernel will run the clock algorithm on the frames to select a victim. This clock algorithm only uses the use bit, and not the modified bit to select its victim. If a kernel page is needed, we just look for the first non-kernel page, and put it there, swapping out that page if it isn't free. This is done to avoid fragmenting kernel pages preventing large kmalloc that require more than 1 page of contiguous memory to work. To avoid constantly paging out user pages to make way for kernel pages, user pages go to the beginning of the free list when freed, and kernel pages go to the end of the free list. Also, the free list starts at the last page and works backwards, while kernel pages start looking at the first page and go forwards. This prevents unnecessary “bullying” by the kernel over the user programs.

2. Handling multiple physically contiguous frame allocation/freeing:

The physical memory system has to handle requests to allocate/free multiple physically contiguous frames. Under the circumstances when kmalloc is called, and the size requested is greater than LARGEST_SUBPAGE_SIZE, then we will try to allocate (size + PAGE_SIZE - 1)/PAGE_SIZE contiguous frames. Our physical manager supports this through the data structure provided by core map. Upon the request of “k” number of pages, the core map will check and see if there are “k” free contiguous pageable frames inside the core map. If such k frames exist, all k frames will be cleared out by either pushing the contents to swap or taking it away from the list of unused frames and marking them as not pageable kernel memory and finally returns the physical memory location of the starting physical frame.

Functionality of implemented solutions:

Handling TLB Miss:

a) When required page is already loaded in memory:

First the kernel checks the address space to find the segment that the BADVADDR belongs to. After finding the segment, it checks the segments page table for the physical frame the page occupies. If this is set to be a non-physical frame number, we know that the frame is not in memory, otherwise we have the frame that the page is loaded in.

b) When required page is not in memory:

First trap.c calls vm_fault, which determines if the address of the fault is a valid address, after the segment is determined from the address space with the given fault address. Finally, pt_page_in(faultaddress, s) is called with the fault address, and the segment that the fault address belongs in. Inside the pt_page_in function we first check to see if the sfn in page_details is not -1 if so, swap in from swap space. Otherwise, we check to see if the page detail is valid, and the physical frame number is not -1, then we just add the page into the TLB from the given physical frame number, and finally, we will load the page from the elf file.

In our VM system, we make sure that our kernel does not modify read-only pages. We have made use of the writable/dirty property, defined inside segments, and page tables for ensuring this. Whenever adding TLB entries for the page, we always use this bit to decide if we set the dirty flag. As such if the page is modified, a read-only fault will be generated and we can kill the process.

Managing Synchronization Issues:
During the design and development of our kernel, we decided that we will use any necessary synchronization mechanisms to ensure critical atomic operations. Because the low levelness of virtual and physical memory, and to accommodate multiple threads, throughout the virtual memory handling codes we've used several forms of synchronization from locals to disabling interrupts to handle situations such as making sure the writing to the swap space is atomic, and allocating and deallocating core map details. We also mark pages as kernel when allocating them, even if they are user pages, then setting kern back to 0 (if it's a user program) through the function cm_finish_paging. This prevents a page from being swapped out while it's being setup since kernel pages are never swapped out.

Descriptions of new data structures:

Our VM system has the following main data structure components:

1. Core map - to manage physical memory
2. Address spaces - to manage virtual memory
3. Segments - to store information about each address space segment and
4. Page tables - to manage page information for a particular segment.

The core map keeps track of information for our page replacement algorithm (a clock pointer, since we use the clock algorithm), how much memory is used for kernel memory that cannot be freed (such as memory core map uses itself) and an entry for every physical page of RAM. For each page of RAM, we keep track of the virtual page that owns the current memory, whether the kernel has marked the memory as not pageable, and pointers to the next and previous free page of memory (if the memory is free). We initialize core map very early in the bootstrap process since it is used by many other systems. To initialize it we calculate how much memory will be unfreeable by using ram_getsize(), and set the other pages to: have no virtual page owning it, not in use by the kernel, and part of the free list.

The address space structure keeps an array of segment information, and a vnode for the elf file so demand paging may take place. This structure is initialized when a thread is created, by reading from the elf file for segment information, and then storing the vnode for later access. A separate stack segment for each address space is also created by the kernel.

The segment structure mainly stores the base virtual address, the size of the segment, whether the segment is write-able, and a page table to store information about each specific page. There are also fields for offset information to find the location of the segments data in the elf-file on a demand load. Segment data is initialized by reading from the elf file when a program is loaded. For the stack segment, special zeroed elf file information is stored, so that when trying to load the page from elf, it will always zero fill.

The page table stores for each page: the base virtual address, the physical frame number the page is loaded in (or not loaded), the swap space location the page is stored in (or not stored), a valid bit, a dirty bit, and a use bit, which is used for page replacement. The page table is initialized during segment initialization, and all pages are initially set to be not loaded, not used, and not stored in physical memory, causing them to be demand paged on a fault.

Page Replacement Algorithm:

We have used the Second-Chance Algorithm (Clock Algorithm) as our page replacement algorithm.
Second-Chance Algorithm: It is basically FIFO algorithm with the reference bit. When a page is selected for replacement, we inspect its reference bit.

- If the reference bit is 0, we directly replace it.
- If the reference bit is 1, we give that page a second chance and move on to select the next FIFO page. However, we set its reference bit to 0.

Pros:

- Conceptually simple
- Better efficiency than FIFO page replacement algorithm
- Interesting and easy to implement

Cons:

- The worst case may take a long time
- Need to maintain a circular queue, which can take a long time to go around when memory is very large.

TLB Replacement Algorithm:

We have used the Round-Robin algorithm as our TLB entry replacement algorithm. Whenever there is a need to replace an entry in the TLB i.e. when it is full, we select an entry which comes in next to the one previously evicted from the TLB.

Pros:

- Easy to implement
- No extra overhead of maintaining any data structures for eviction

Cons:

- Efficiency is low
- A TLB entry which might be required often may be evicted, leading to loss in performance

Answers in malloc.txt:

Q.1.a) There are up to two sbrk() calls during the one time initialization (if __heapbase == 0), once to find the base of the heap and once to align the heap base if necessary. In subsequent calls to malloc(), sbrk() is only called if nothing was found to expand the heap inside call to __malloc_sbrk().

Q.1.b) The relevant piece of code is that the heap is expanded with:

mh = __malloc_sbrk(size + MBLOCKSIZE)

Here the size is 16 (after alignment) and MBLOCKSIZE is defined to be 8, so the program break is incremented in increments of 24 bytes. Since there are 10 iterations, finish-start = 24*10 = 240

Q.2) No, malloc() would not need to call sbrk() to expand the heap even though free() only tries to merge adjacent blocks. free() will merge the 6 freed blocks into 2 sets of 3 * 24 = 72 byte blocks. The block size necessary for malloc(60) after alignment is 64 + 8 = 72, just the right size.

x = start + 24 (the location of res[1])

Q.3) It is conventional for libc internal functions and variables to be prefaced with "__".
They are identifiers reserved to the implementation for future extensions to the C language or POSIX.

Q.4)
```
/* Round size up to an integral number of blocks. */
size = ((size + MBLOCKSIZE - 1) & ~(size_t)(MBLOCKSIZE-1));
```

Notes:

After running the “matmult” command using “p testbin/matmult” on the sys161 kernel, it successfully executes the matmult program printing “Passed” at the end, but the system hangs after executing that command. We did not observe this behavior after executing any other command such as sort, palin, km1, km2, etc. (the scripts for which are included in the submission). We could not get around this issue in time.