OPERATING SYSTEMS – ASSIGNMENT 3

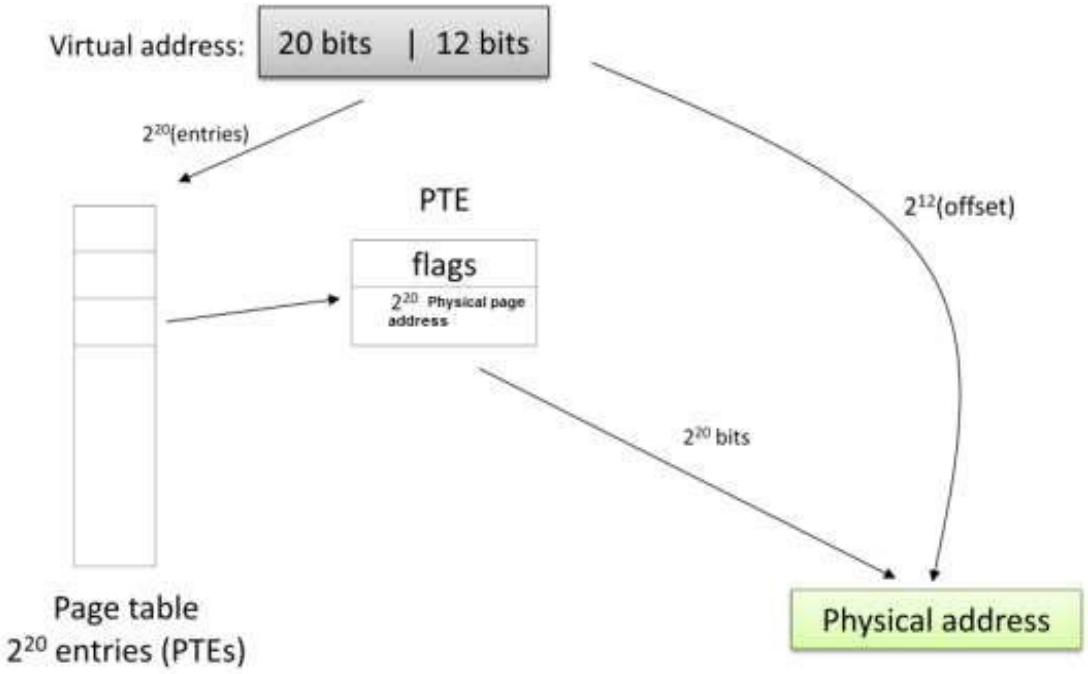
MEMORY MANAGEMENT

Responsible TA's: Omer and Vadim

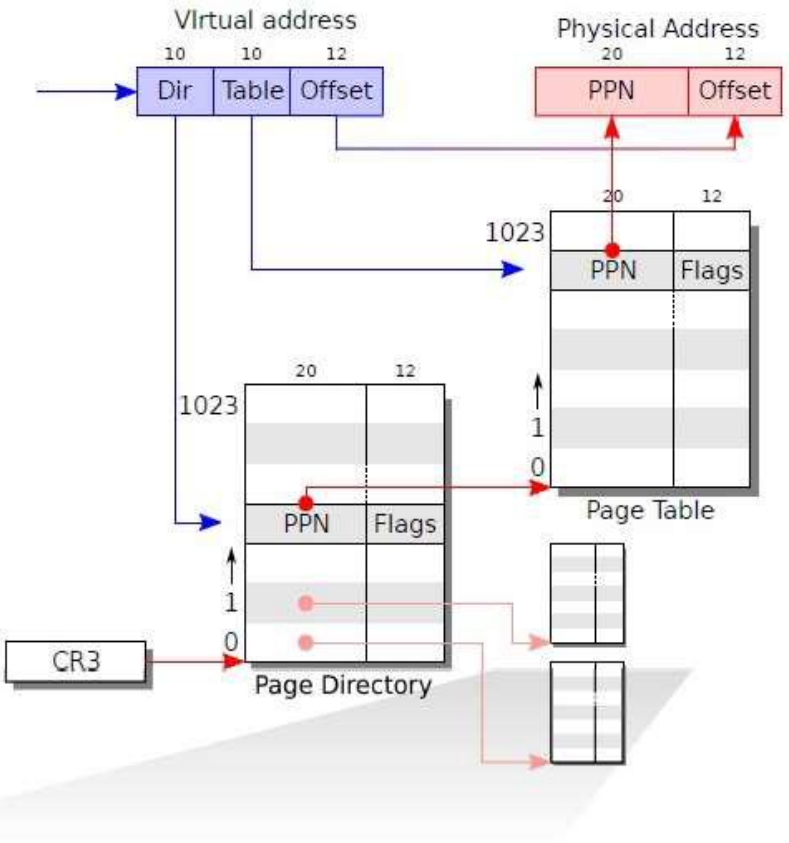
Introduction

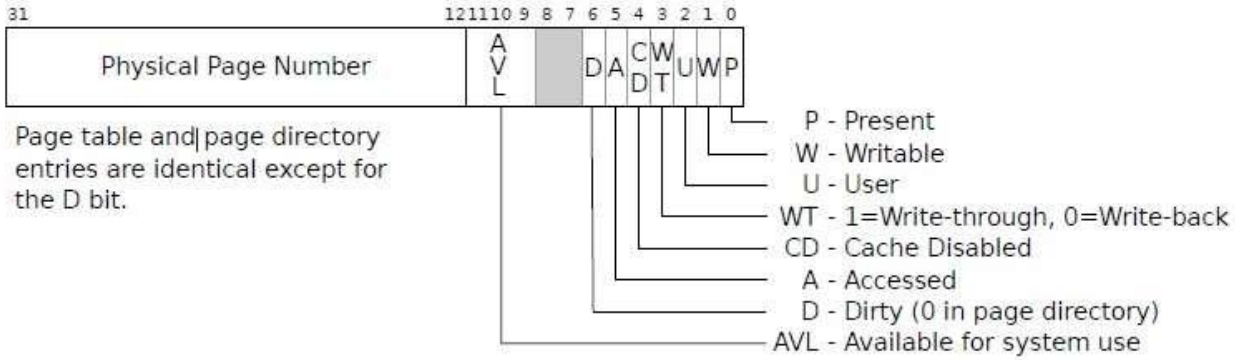
Memory management and memory abstraction are among the most important features of any operating system. In this assignment we will examine how xv6 handles memory and attempt to extend it. To help get you started we will provide a brief overview of the memory management facilities of xv6. We strongly encourage you to read this section while examining the relevant xv6 files (`vm.c, mmu.h, kalloc.c, etc`).

Memory in xv6 is managed in pages (and frames) where each such page is 4096 (=2^{12}) bytes long. Each process has its own page table which is used to translate virtual to physical addresses. The virtual address space can be significantly larger than the physical memory. In xv6, the process address space is 2^{32} bytes long while the physical memory is currently limited to 224 MB only (`see PHYSTOP at memlayout.h`). When a process attempts to access some address in its memory (i.e. provides a 32 bit virtual address) it must first seek out the relevant page in the physical memory. Xv6 uses the first (leftmost) 20 bits to locate the relevant Page Table Entry (PTE) in the page table. The PTE will contain the physical location of the frame – a 20 bits frames address (within the physical memory). These 20 bits will point to the relevant frame within the physical memory. To locate the exact address within the frame, the 12 least significant bits of the virtual address, which represent the in-frame offset, are concatenated to the 20 bits retrieved from the PTE. Roughly speaking, this process can be described by the following illustration:



Maintaining a page table may require a significant amount of memory as well, so a two level page table is used. The following figure describes the process in which a virtual address translates into a physical one:





Each process has a pointer to its page directory (*line 59, proc.h*). This is a single page sized (4096 bytes) directory which contains the page addresses and flags of the second level table(s). This second level table is spanned across multiple pages which are very much like the page directory.

When seeking an address the first 10 bits will be used to locate the correct entry within the page directory (by using the macro PDX(va)). The physical frame address can be found within the correct index of the second level table (accessible via the macro PTX(va)). As explained earlier, the exact address may be found with the aid of the 12 LSB (offset).

At this point you should go over the xv6 documentation on this subject:
http://pdos.csail.mit.edu/6.828/2014/xv6/book-rev8.pdf (chapter 2)

- Tip: before proceeding further we strongly recommend that you go over the code again. Now, attempt to answer questions such as:
  - How does the kernel know which physical pages are used and unused?
  - What data structures are used to answer this question?
  - Where do these reside?
  - Does xv6 memory mechanism limit the number of user processes?
- A very good and detailed account of memory management in the Linux kernel can be found here: http://www.kernel.org/doc/gorman/pdf/understand.pdf
- Another link of interest: "What every programmer should know about memory" http://lwn.net/Articles/250967/

Task 0: running xv6

As always, begin by downloading our revision of xv6, from the git repository:

- Open a shell, and traverse to the desired working directory.
- Execute the following command (in a single line):
  git clone http://www.cs.bgu.ac.il/~os152/xv6_3.git
- Build xv6 by calling: make
- Run xv6 on top of QEMU by calling: make qemu

Task 1: managing software TLB lookups

A translation lookaside buffer (TLB) is a memory cache that stores recent translations of virtual memory to physical addresses for faster retrieval. When a virtual memory address is referenced by a program, it is first searched in the TLB, and only if it is not in the TLB (a TLB miss) a full page table walk is performed to find the required address.

Two schemes for handling TLB misses are commonly found in modern architectures:

1. With hardware TLB management, the CPU automatically walks the page tables (using the CR3 register on x86 for instance) to see if there is a valid page table entry for the specified virtual address. If an entry exists, it is brought into the TLB and the TLB access is retried: this time the access will hit, and the program can proceed normally. If the CPU finds no valid entry for the virtual address in the page tables, it raises a page fault exception, which the operating system must handle. With a hardware-managed TLB, the format of the TLB entries is not visible to software, and can change from CPU to CPU without causing loss of compatibility for the programs.

2. With software-managed TLBs (MIPS architecture, SPARC V9 architecture etc.), a TLB miss generates a "TLB miss" exception, and operating system code is responsible for walking the page tables and performing the translation in software. The operating system then loads the translation into the TLB and restarts the program from the instruction that caused the TLB miss. As with hardware TLB management, if the OS finds no valid translation in the page tables, a page fault has occurred, and the OS must handle it accordingly. Instruction sets of CPUs that have software-managed TLBs have instructions that allow loading entries into any slot in the TLB.

Since xv6 is designed to run on x86 architecture, we will briefly describe how actually virtual addresses are translated to physical. Each time a reference to an (virtual) address is made, the MMU is responsible to translate the virtual address to physical. First the MMU checks if the mapping exists in the TLB. If the mapping exists, the address resolution is done. Otherwise, the MMU will use the page directory (from cr3) to find the physical address (and updates TLB accordingly). If no such mapping exists (both in the TLB and in the page directory), a page fault is raised.

Simulating software TLB

In this task we will simulate software managed TLB. We will use kpgdir as the simulated TLB. In other words we will use low virtual addresses of kpgdir (user space addresses) in order to maintain needed mapping. Since xv6 is based on x86 architecture, the TLB management is implemented at the hardware level. In order to simulate software-managed TLB, we will not provide the page directory of the current running process to the MMU. This can be done by removing instruction lcr3(v2p(p->pgdir)) at function switchuvm (*see vm.c*), where p->pgdir is the page directory of the process p (page directory structure was presented above). As a result the MMU will not be aware of the process' virtual address translation and the CPU will raise a page fault exception for each unmapped user-space memory access. When the kernel catches the page fault exception it will find the page of the virtual address that generated this exception by going through the process page tables, and map this page in the simulated TLB (kernel page tables kpgdir *see vm.c*). The next time this address will be referenced it will already be in the kernel page directory (which is the page directory that the MMU knows), so no page fault will be generated.

- The implementation described above uses the bottom half of the page directory of the kernel as a simulated TLB (understand why the bottom half was originally unused in xv6).
- Pay special attention to process scheduling. Think carefully what must be done and what must happen when a process is scheduled.

Limiting the number of TLB records

After implementing the simulation of software TLB management, we are able to limit the number of TLB records. In our case we will limit it to only 2 entries. In other words, when you need to update the TLB cache, you must ensure that the TLB contains at most one entry, otherwise one of the entries must be removed from the TLB (this can be done by zeroing the appropriate record in the page table). We assume FIFO order in TLB entries update (i.e., the first entry inserted to TLB is the first entry to be removed). When you remove a TLB entry (from pgtab) you must ensure that this pgtab contains at least one mapping, otherwise the page corresponding to this pgtab must be de-allocated and kpgdir updated accordingly.

Task 2: lazy page allocation

In this task you will implement lazy allocation of heap memory. A process can ask the kernel for heap memory by using the sbrk() system call (or more simply by using malloc(), since the implementation of malloc() uses sbrk()). In xv6 the kernel allocates memory immediately when sbrk() is called. There are programs that allocate memory but never use it, for example to implement large sparse arrays. Sophisticated kernels delay allocation of each page of memory until the application tries to use that page - as signaled by a page fault. You'll add this lazy allocation feature to xv6 in this exercise.

The first thing you'll want to do is to remove the allocation of memory from the sbrk() system call. At this point, any user program will fail (you can try to run "echo hi" and see the result). Next you will add the allocation of new pages in the trap handler (add a new case for page fault). Look at the cprintf arguments to see how to find the virtual address that caused the page fault. Follow the procedures called from sbrk() to find where the actual user virtual memory is allocated and mapped, and move the relevant code to the trap handler. Pay attention to only map the single page that caused the fault. When a dynamic page allocation will fail, we will kill the process (this is for simplicity, since we do not want to keep track of the number of pages waiting to be allocated). If you have implemented everything described so far, "echo hi" should work fine.

The lazy allocation we have by now is very naïve, and we have to address the following problems:

- Upon fork, the child process tries to copy all of the data from the parent. Some of the pages may not be present (and that is ok with lazy allocation). Furthermore, when the heap is big enough, some page tables may not be present as well.
- sbrk() may be called for reducing the heap size, by using negative arguments. Make it possible to do so.
- We do not want to allow a process to request virtual addresses that belong to the kernel. Limit the size of a process in sbrk() (growproc() is responsible for that in the normal implementation).
- In this implementation we have caught every page fault and allocated a new page for it. There are two occasions where we want the original handling of the page fault:
    - When the virtual address is bigger than the size of the process.
    - When we have a stack overflow. Stack overflows in xv6 are handled by positioning an empty inaccessible page below the stack. A check should be made if the virtual address is in the page below the stack. Since the stack in xv6 is exactly one page in size, we can find the page below the stack using the stack pointer.

When all of the issues are dealt with "usertests" should be working, although "mem test" will take much longer (make sure you understand why). To completely check that your implementation is working, add a printout whenever lazy allocation is being done, and write a testing user application that does the following:

- malloc() a big chunk of memory (at least 100 pages) and see that it only allocating a small number of pages.
- Write to several pages and make sure that for each write exactly one page is allocated.
- Make sure that you can malloc() very large chunks of memory (enough to use several page directories entries).
- Try to malloc() memory up to the kernel, and make sure that it fails.
- Check that fork() behaves as it should in those situations.
- Make sure that you have stack overflow protection.

Enjoy...