Lecture Notes for CS347: Operating Systems

Mythili Vutukuru, Department of Computer Science and Engineering, IIT Bombay

8. Memory Management in xv6

8.1 Basics

- xv6 uses 32-bit virtual addresses, resulting in a virtual address space of 4GB. xv6 uses paging to manage its memory allocations. However, xv6 does not do demand paging, so there is no concept of virtual memory.

- xv6 uses a page size of 4KB, and a two level page table structure. The CPU register CR3 contains a pointer to the page table of the current running process. The translation from virtual to physical addresses is performed by the MMU as follows. The first 10 bits of a 32-bit virtual address are used to index into a page table directory, which points to a page of the inner page table. The next 10 bits index into the inner page table to locate the page table entry (PTE). The PTE contains a 20-bit physical frame number and flags. Every page table in xv6 has mappings for user pages as well as kernel pages. The part of the page table dealing with kernel pages is the same across all processes.

- In the virtual address space of every process, the kernel code and data begin from KERNBASE (2GB in the code), and can go up to a size of PHYSTOP (whose maximum value can be 2GB). This virtual address space of [KERNBASE, KERNBASE+PHYSTOP] is mapped to [0,PHYSTOP] in physical memory. The kernel is mapped into the address space of every process, and the kernel has a mapping for all usable physical memory as well, restricting xv6 to using no more than 2GB of physical memory. Sheets 02 and 18 describe the memory layout of xv6.

8.2 Initializing the memory subsystem

- The xv6 bootloader loads the kernel code in low physical memory (starting at 1MB, after leaving the first 1MB for use by I/O devices), and starts executing the kernel at `entry` (line 1040). Initially, there are no page tables or MMU, so virtual addresses must be the same as physical addresses. So the kernel entry code resides in the lower part of the virtual address space, and the CPU generates memory references in the low virtual address space only. The entry code first turns on support for large pages (4MB), and sets up the first page table `entrypgdir` (lines 1311-1315). The second entry in this page table is easier to follow: it maps [KERNBASE, KERNBASE+4MB] to[0, 4MB], to enable the first 4MB of kernel code in the high virtual address space to run after MMU is turned on. The first entry of this page table table maps virtual addresses [0, 4MB] to physical addresses [0,4MB], to enable the entry code that resides in the low virtual address space to run. Once a pointer to this page table is stored in CR3, MMU is turned on, the entry code creates a stack, and jumps to the `main` function in the kernel's C code (line 1217). The C code is located in high virtual address space, and can run because of the second entry in `entrypgdir`. So why was the first page table entry required? To enable the few instructions between turning on MMU and jumping to high address space to run correctly. If the entry page table only mapped high virtual addresses, the code that jumps to high virtual addresses of main would itself not run (because it is in low virtual address space).

- Remember that once the MMU is turned on, for any memory to be usable, the kernel needs a virtual address and a page table entry to refer to that memory location. When main starts, it is still using `entrypgdir` which only has page table mappings for the first 4MB of kernel addresses, so only this 4MB is usable. If the kernel wants to use more than this 4MB, it needs to map all of that memory as free pages into its address space, for which it needs a larger page table. So, main first creates some free pages in this 4MB in the function `kinit1` (line 3030), which eventually calls the functions `freerange` (line 3051) and `kfree` (line 3065). Both these functions together populate a list of free pages for the kernel to start using for various things, including allocating a bigger, nicer page table for itself that addresses the full memory.

- The kernel uses the `struct run` (line 3014) data structure to address a free page. This structure simply stores a pointer to the next free page, and the rest of the page is filled with garbage. That is, the list of free pages are maintained as a linked list, with the pointer to the next page being stored within the page itself. Pages are added to this list upon initialization, or on freeing them up. Note that the kernel code that allocates and frees pages always returns the virtual address of the page in the kernel address space, and not the actual physical address. The V2P macro is used when one needs the physical address of the page, say to put into the page table entry.

- After creating a small list of free pages in the 4MB space, the kernel proceeds to build a bigger page table to map all its address space in the function `kvmalloc` (line 1857). This function in turn calls `setupkvm` (line 1837) to setup the kernel page table, and switches to it. The address space mappings that are setup by `setupkvm` can be found in the structure `kmap` (lines 1823-1833). `kmap` contains the mappings for all kernel code, data, and any free memory that the kernel wishes to use, all the way from KERNBASE to KERNBASE+PHYSTOP. Note that

the kernel code and data is already residing at the specified physical addresses, but the kernel cannot access it because all of that physical memory has not been mapped into any logical pages or page table entries yet.

- The function `setupkvm` works as follows. For each of the virtual to physical address mappings in `kmap`, it calls `mappages` (line 1779). The function `mappages` walks over the entire virtual address space in 4KB page-sized chunks, and for each such logical page, it locates the PTE using the `walkpgdir` function (line 1754). `walkpgdir` simply outputs the translation that the MMU would do. It uses the first 10 bits to index into the page table directory to find the inner page table. If the inner page table does not exist, it requests the kernel for a free page, and initializes the inner page table. Note that the kernel has a small pool of free pages setup by `kinit1` in the first 4MB address space—these free pages are used to construct the kernel's page table. Once `walkpgdir` returns the PTE, `mappages` sets up the appropriate mapping using the physical address it has. (Sheet 08 has the various macros that are useful in getting the index into page tables from the virtual address.)

- After the kernel page table `kpgdir` is setup this way (line 1859), the kernel switches to this page table by storing its address in the CR3 register in `switchkvm` (line 1866). From this point onwards, the kernel can freely address and use its entire address space from KERNBASE to KERNBASE+PHYSTOP.

- Let's return back to main in line 1220. The kernel now proceeds to do various initializations. It also gathers many more free pages into its free page list using `kinit2`, given that it can now address and access a larger piece of memory. At this point, the entire physical memory at the disposal of the kernel [0, PHYSTOP] is mapped by `kpgdir` into the virtual address space [KERNBASE, KERNBASE+PHYSTOP], so all memory can be addressed by virtual addresses in the kernel address space and used for the operation of the system. This memory consists of the kernel code/data that is currently executing on the CPU, and a whole lot of free pages in the kernel's free page list. Now, the kernel is all set to start user processes, starting with the init process.

8.3 Creating user processes

- The function `userinit` (line 2502) creates the first user process. We will examine the memory management in this function. The kernel page table of this process is created using `setupkvm` as always. For the user part of the memory, the function `inituvm` (line 1903) allocates one physical page of memory, copies the init executable into that memory, and sets up a page table entry for the first page of the user virtual address space. When the init process runs, it executes the init executable (sheet 83), whose main function forks a shell and starts listening to the user. Thus, in the case of init, setting up the user part of the memory was straightforward, as the executable fit into one page.

- All other user processes are created by the `fork` system call. In `fork` (line 2554), once a child process is allocated, its memory image is setup as a complete copy of the parent's memory image by a call to `copyuvm` (line 2053). This function walks through the entire address space of the parent in page-sized chunks, gets the physical address of every page of the parent using a call to `walkpgdir`, allocates a new physical page for the child using `kalloc`, copies the contents of the parent's page into the child's page, adds an entry to the child's page table using `mappages`, and returns the child's page table. At this point, the entire memory of the parent has been cloned for the child, and the child's new page table points to its newly allocated physical memory.

- If the child wants to execute a different executable from the parent, it calls `exec` right after `fork`. For example, the init process forks a child and execs the shell in the child. The `exec` system call copies the binary of an executable from the disk to memory and sets up the user part of the address space and its page tables. In fact, after the initial kernel is loaded into memory from disk, all subsequent executables are read from disk into memory via the `exec` system call alone.

- `Exec` (line 6310) first reads the ELF header of the executable from the disk and checks that it is well formed. It then initializes a page table, and sets up the kernel mappings in the new page table via a call to `setupkvm` (line 6334). Then, it proceeds to build the user part of the memory image via calls to `allocuvm` (line 6346) and `loaduvm` (line 6348) for each segment of the binary executable. `allocuvm` (line 1953) allocates physical pages from the kernel's free pool via calls to `kalloc`, and sets up page table entries. `loaduvm` (line 1918) reads the memory executable from disk into the allotted page using the `readi` function. After the end of the loop of calling these two functions for each segment, the program executable has been loaded into memory, and page table entries setup to point to it. However, `exec` hasn't switched to this new page table yet, so it is still executing in the old memory image.

- Next, `exec` goes on to build the rest of its new memory image. For example, it allocates a user stack page, and an extra page as a guard after the stack. The guard page has no physical memory frame allocated to it, so any access beyond the stack into the guard page will cause a page fault. Then, the arguments to `exec` are pushed onto the user stack, so that the `exec` binary can access them when it starts.

- It is important to note that `exec` does not replace/reallocate the kernel stack. The `exec` system call only replaces the user part of the memory. And if you think about it, there is no way the process can replace the kernel stack, because the process is executing in kernel mode on the kernel stack itself, and has important information like the trap frame stored on it. However, it does make small changes to the trap frame on the kernel stack, as described below.

- Now, a process that makes the `exec` system call has moved into kernel mode to service the software interrupt of the system call. Normally, when the process moves back to user mode again (by popping the trap frame on the kernel stack), it is expected to return to the instruction after the system call. However, in the case of `exec`, the process doesn't have to return to the instruction after `exec` when it gets the CPU next, but instead must start executing in the new memory image containing the binary file it just loaded from disk. So, the code in `exec` changes the return address in the trap frame to point to the entry address of the binary (line 6392). Finally, once all these operations succeed, `exec` switches page tables to start using the new memory image, and frees up all the memory pointed at by the old page table. At this point, the process that called `exec` can start executing on the new memory image. Note that `exec` waits until the end to do this switch, because if anything went wrong in the system call, `exec` returns to the old process image and prints out an error.