Lecture Notes on Operating Systems

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

Memory Management in xv6

1. Basics

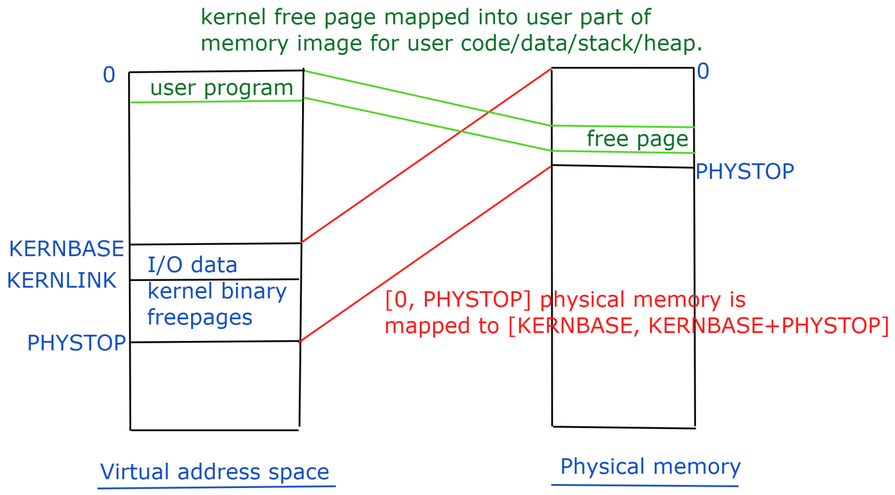
- xv6 uses 32-bit virtual addresses, resulting in a virtual address space of 4GB. xv6 uses paging to manage its memory allocations. However, xv6 does not do demand paging, so there is no concept of virtual memory. That is, all valid pages of a process are always allocated physical pages.

- xv6 uses a page size of 4KB, and a two level page table structure dictated by the underlying x86 hardware. The CPU register CR3 contains a pointer to the outer page directory of the current running process. The translation from virtual to physical addresses is performed by the x86 MMU as follows. The first 10 bits of a 32-bit virtual address are used to index into the page table directory, which provides an address 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. Sheet 8 contains various definitions pertaining to the page table and page table entries. You can find macros to extract specific bits (e.g., index into page table) from a virtual address here, which will be useful when understanding code. You can also find the various flags used to set permissions in the page table entry defined here.

- Sheets 02 and 18 describe the memory layout of xv6. In the virtual address space of every process, the user code+data, heap, stack and other things start from virtual address 0 and extend up to KERNBASE. The kernel is mapped into the address space of every process beginning at KERNBASE. The kernel code and data begin from KERNBASE, and can go up to KERNBASE+PHYSTOP. This virtual address space of [KERNBASE, KERNBASE+PHYSTOP] is mapped to [0,PHYSTOP] in physical memory. In the current xv6 code, KERNBASE is set to 2GB and PHYSTOP is set to 224MB. Because KERNBASE+PHYSTOP can go to a maximum of 4GB in 32-bit architectures, and KERNBASE is 2GB, the maximum possible value of PHYSTOP is 2GB, so xv6 can use a maximum of 2GB physical memory. You can also find here various macros like V2P that translates from a virtual address to a physical addres (by simply subtracting KERNBASE from the virtual address).

- The kernel code doesn't exactly begin at KERNBASE, but a bit later at KERNLINK, to leave some space at the start of memory for I/O devices. Next comes the kernel code+read-only data from the kernel binary. Apart from the memory set aside for kernel code and I/O devices, the remaining memory is in the form of free pages managed by the kernel. When any user process requests for memory to build up its user part of the address space, the kernel allocates memory to the user process from this free space list. That is, most physical memory can be mapped twice, once into the kernel part of the address space of a process, and once into the user part.

- The memory layout described above is illustrated below.


kernel free page mapped into user part of
memory image for user code/data/stack/heap.
0
user program
KERNBASE
KERNLINK
I/O data
kernel binary
freepages
PHYSTOP
0
free page
PHYSTOP
[0, PHYSTOP] physical memory is
mapped to [KERNBASE, KERNBASE+PHYSTOP]
Virtual address space
Physical memory

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 this 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, which are passed onto the memory hardware as-is and used as physical addresses also. Now, the kernel must turn on MMU and start executing at high virtual addresses, in order to make space for user code at low virtual addresses. We will first see how the kernel jumps to high virtual addresses.

- 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. From this point onwards, virtual addresses in the range [KERNBASE, KERNBASE+4MB] are correctly translated by MMU. Now, the entry code creates a stack, and jumps to the `main` function in the kernel's C code (line 1217). This function and the rest of the kernel is linked to run at high virtual addresses, so from this point on, the CPU fetches kernel instructions and data at high virtual addresses. All this C code in high virtual address space 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 still in low virtual address space).

- Remember that once the MMU is turned on, all memory accesses must go through the MMU. So, for any memory to be usable, the kernel must assign a virtual address for that memory and a page table entry to translate that virtual address to a physical address must be present in the page table/MMU. That is, for any physical memory address N to be usable by the kernel, there must exist a page table entry that translates virtual address KERNBASE+N to physical address N. 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 a larger page table to hold more entries. So, the main function of the kernel first creates some free pages in this 4MB in the function `kinit1` (line 3030), and uses these freepages to allocate a bigger page table for itself. This function `kinit1` in turn 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 track a free page. This structure simply stores a pointer to the next free page, and is stored within the page itself. 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. The kernel keeps a pointer to the first page of this free list in the structure `struct kmem` (lines 3018-3022). 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 main function (sheet 12) 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 (by writing the address of the page table into the CR3 register). The address space mappings that are setup by `setupkvm` can be found in the structure `kmap` (lines 1823-1833). `kmap` contains the mappings from virtual to physical address for various virtual address ranges: the small space at the start of memory for I/O devices, followed by kernel code+read-only data (kernel binary), followed by other kernel data, all the way from KERNBASE to KERNBASE+PHYSTOP. Note that the kernel code/data and other free physical memory is already lying around at the specified physical addresses in RAM, but the kernel cannot access it because all of that physical memory has not been mapped into any page tables yet and there are no page table entries that translate to these physical addresses at the MMU.

- The function `setupkvm` works as follows. It first allocates an outer page directory. Then, for each of the virtual to physical address mappings in `kmap`, it calls `mappages`. The function `mappages` (line 1779) is given a virtual address range and a physical address range it should map this to. It then walks over the virtual address range in 4KB page-sized chunks, and for each such logical page, it locates the PTE corresponding to this virtual address using the `walkpgdir` function (line 1754). `walkpgdir` simply emulates the page table walking 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 returned here and used to construct the kernel's page table. Once the inner page table is located (either existing already or newly allocated), walkpgdir uses the next 10 bits to index into it and return the PTE. Once `walkpgdir` returns the PTE, `mappages` writes the appropriate mapping in the PTE using the physical address range given to it. (Sheet 08 has the various macros that are useful in getting the index into page tables from the virtual address.)

- Note that `walkpgdir` simply returns the page table entry corresponding to a virtual address in a page table. That is, it uses the first 10 bits of a virtual address as an index in the page directory to find the inner page table, then uses the next 10 bits as an index in this inner page table to find the actual PTE. What if the inner page table corresponding to the address does not exist? The last argument to the function specifies if a page table should be allocated if one doesn't exist. That is, this function `walkpgdir` serves two purposes. It can simple be used to look up a virtual address in an existing page table and return whatever PTE exists. It can also be used to construct the page table entry if one doesn't exist. In this case, since we are initializing the page table, we use walkpgdir to allocate inner page tables. When it allocates an inner page table, the PTE returned by walkpgdir doesn't hold any valid information as such. The function `mappages` takes this empty PTE returned by `walkpgdir` and writes the correct physical address into it.

- 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. All usable physical memory [0, PHYSTOP] is mapped by `kpgdir` into the virtual address space [KERNBASE, KERNBASE+PHYSTOP], so all memory can be addressed by virtual addresses and translated by MMU. 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 (line 1239).

3. Memory management of 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 first sets up the kernel part of the page table. Then it 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.

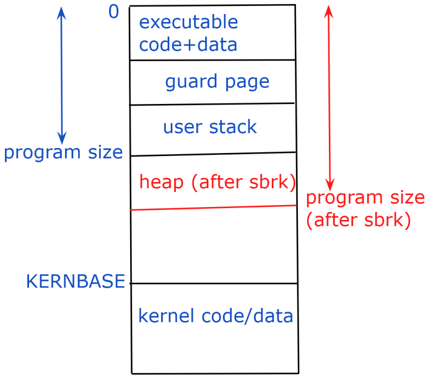
- Next, look at the implementation of the `sbrk` system call (line 3701). This system call can be invoked by a process to grow/shrink the userspace part of the memory image. For example, the heap implementation of `malloc` can use this system call to grow/shrink the heap when needed. This system call invokes the function `growproc` (line 2531), which uses the functions `allocuvm` or `deallocuvm` to grow or skrink the virtual memory image. The function `allocuvm` (line 1953) walks the virtual address space between the old size and new size in page-sized chunks. For each new logical page to be created, it allocates a new free page from the kernel, and adds a mapping from the virtual address to the physical address by calling `mappages`. The function `deallocuvm` (line 1982) looks at all the logical pages from the (bigger) old size of the process to the (smaller) new size, locates the corresponding physical pages, frees them up, and zeroes out the corresponding PTE as well.

- Next, let's understand the exec system call. 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 and the init process is setup, 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. (There is a lot of disk I/O related code here that you can ignore for now.) 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 part of the binary executable (an

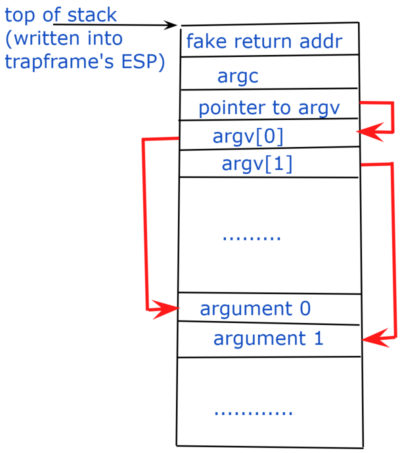
ELF binary is composed of many segments). We have already seen that allocuvm (line 1953) allocates physical pages from the kernel's free pool via calls to kalloc, and sets up page table entries, thereby expanding the virtual/physical address space of the process. loaduvm (line 1918) reads the memory executable from disk into the newly allotted memory page using the disk I/O related readi function. After the end of the loop of calling these two functions for each segment, the complete program executable has been loaded from disk into memory, and page table entries have been setup to point to it. However, exec is still using the old page table, and hasn't switched to this new page table yet. We have only constructed a new memory image and a page table pointing to it so far, but the process that called exec is still executing on the old memory image.

- The new memory image so far has only the code/data present in the executable. Next, exec goes on to build the rest of its new memory image. For example, it allocates two pages for the userspace stack of the process. The second page is the actual stack and the first page serves as a guard page. The guard page's page table entry is modified to make it as inaccessible by user processes, so any access beyond the stack into the guard page will cause a page fault. (Recall that the stack grows upwards towards lower memory addresses, so it will overflow into the page before it.) After the stack in the memory image is where the heap should be located. However, xv6 doesn't allocate any heap memory upfront. The user library code that handles malloc will call sbrk to grow the memory image, and use this new space as the heap, as and when memory is requested by user programs. The program size includes all the memory from address zero until the end of the stack now, and the size will increase to include any memory allocated by sbrk when the currently empty heap expands. The new memory image constructed by exec is shown below.



- After allocating the user stack, the arguments to exec are pushed onto the user stack (lines 6363-6380). The arguments passed to the exec system call are already made available as arguments to the exec function on sheet 63. We must now copy these arguments onto the newly created userspace stack of the process. Why? Because when the main function of the new executable starts, it expects to find arguments argc and argv on the user stack.

- If you are curious, below is an explanation of this process of preparing the user stack (lines 6363-6380) in more detail. Recall the structure of the stack when any C function is called: the top of the stack has the return address, followed by the arguments passed to the function below it. We must now prepare this new userspace in this manner for the main function as well. The top of the user stack has a fake return address (line 6374) because main doesn't really return anywhere. Next, the stack has the number of arguments (argc), followed by a pointer to the argv array. Next on the stack are the actual contents of the argv array (which is of size argc). The element i of the array argv contains a pointer to the i-th argument (which can be any random string). Finally, after the array of pointers to arguments, the actual arguments themselves are also present on the stack. We will now see how this structure is constructed. We start at the bottom of the stack, and start pushing the actual arguments on the stack (lines 6363-6371). In the process, we also remember where the argument was pushed, i.e., we store the pointers to the arguments in the array `ustack`. Finally, we will write out the other things that go above the arguments on the stack: the return PC, argc, pointer to argv, as well as the contents of argv (stored pointers to the arguments). Note that all of these things have to be written into the user stack of the new memory image, not on the current memory image where the process is running. (How does a process access another memory image that is not its own? Note the use of the function `copyout` which simply copies specific content at the specified virtual address of the new memory image defined by the new page table.) This user stack structure is illustrated below.



- 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 image, and does nothing to the kernel part. 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.

- Recall that 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 the new executable 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). It also sets the stack pointer in the trap frame to point to the top of the newly created user stack. Finally, once all these operations succeed, exec switches page tables to start using the new memory image (line 6394). That is, it writes the address of the new page table into the CR3 register, so that it can start accessing the new memory image when it goes back to userspace. Finally, it frees up all the memory pointed at by the old page table, e.g., pages containing the old userspace code, data, stack, heap and so on. At this point, the process that called exec can start executing on the new memory image when it returns from trap. Note that exec waits until the end to do this switch of page tables, because if anything went wrong in the system call, exec returns from trap into the old memory image and prints out an error.