In the last class, we were discussing the exec system call. The exec system call creates a new page table to load the target executable. setupkvm:1837 maps all the physical pages in this page table at a virtual address 0x80000000 (2 GB). Let us call the addresses above 2 GB as kernel address space. The first 1 MB of the kernel address space is reserved for devices and bootloader. The kernel text and data are loaded at the physical address 1 MB. Rest of the physical pages (after kernel data), are managed through kalloc and kfree. Because there is a linear mapping between physical addresses and kernel virtual addresses, a kernel virtual address can directly be translated to the physical address by simply subtracting 2 GB (0x80000000) from the kernel virtual address.

After the kernel page table mappings are created, the next step is to load the user executable from disk (file) to the main memory. The compiler divides the executable into different sections and also generates information regarding, wherein the virtual address space the OS should load these sections. allocuvm:1953 allocates page table pages corresponding to the virtual address range (generated by the compiler) for each section. allocuvm:1953 also maps physical pages to these virtual pages. allocuvm:1953 uses kalloc to allocate a kernel virtual page and maps them to the user virtual address. In other words, the same physical page is mapped twice in the user and kernel address space.

How does this logic ensure that every process gets a different physical page?

What would be the virtual to physical address translation for user pages?

loaduvm:1918 loads the contents of a section from the disk to the user virtual address.

Why does loaduvm walk the page table to get the physical address first and then convert to the kernel virtual address?

After loading all the sections, the exec system call allocates two pages for stack and revokes user permission from the first page. The first page acts as a guard page to catch the stack overflow scenarios. The second page is used for stack. Because the stack grows downwards if the stack overflows, in most of the cases, it will access the second page and causes some kind fault (because the user accesses are not allowed).

After loading the executable, exec calls the main routine of the executable (which we have already loaded in the user virtual address space). The main routine takes two parameters: argc and argv. “argc” is the number of arguments passed to “main” routine. “argv” is an array of strings that are passed to main. The exec system call itself takes a null-terminated array of strings that need to be passed to the main routine in the target executable. The exec system call setup the stack in such a way that the main routine can see these arguments on the stack. xv6 requires the main routine to explicitly call the exit system call, that destroys the process (and hence never return). Because of this reason, the exec system call pushes a fake return address on the stack (because main never returns).

xv6 also maintains the next available virtual address (which is unused) in the user address space (also called the process size). The exec system call sets the process size to next available virtual address (just after stack) in the process address space.

The next goal is to setup the interrupt frame in such a way that it jumps to the main after returning from the exec system call. To understand this let us look at the alltraps:3254 routine. alltraps is called by vectors.S (generated by a script). In addition to ss, esp, eflags, cs, eip some exceptions additionally push an error code on the stack. To maintain homogeneity vectors.h pushes a fake error code for interrupts and exceptions who do not push error code on the stack. In addition to the error code, it also pushes interrupt vector number on the stack before jumping to the alltraps routine. alltraps pushes ds, es, fs, and gs on the stack. After that, it executes pusha that pushes all general-purpose registers (eax, ecx, edx, ebx, esp, ebp, esi, edi) on the stack. If you look at trapframe:602 data structure in the xv6 you will find that the at this point (in alltraps) top of the stack contains an instance of struct trapframe. alltraps pass this struct trapframe to the trap:3351 routine, which saves the address of trapframe in “tf” field of “proc” variable. You can think of proc as a process control block (PCB), which is created for each process to store metadata corresponding to that process. Exec rewrites eip and esp fields of “proc->tf” such that after returning from the exec system call, the execution starts from main.

Finally, exec calls the switchuvm to load the new page table. After loading the new page table, the old page table is not needed, and hence freevm is called to free all user pages, page table pages, and the page directory.

To support dynamic memory allocation xv6 provides sbrk system call. sys_sbrk:3701 can be called to adjust the process size. If the user requires more virtual address (due to malloc), then sbrk grow the process size by allocating more user pages and adjusting the size of the process. If the extra space is no more needed (due to free), then sbrk shrink the process size by deallocating user pages and adjusting the size of the process. Interestingly, sys_sbrk always calls switchuvm to flush the TLB entries.

There is a potential race condition in deallocuvm:1987. deallocuvm frees a physical page (using kfree) without flushing the TLB entries. At this point, the free page can be allocated to a different process. However, other threads of the current process (which are running on other processors) can still access the freed physical page because the TLB’s entries on other processors might have cached this VA to PA mapping. The correct way to handle this situation would be to flush the TLB on every processor before doing the kfree. The TLB flush ensures that all the threads of the process will not be able to access the physical page anymore. xv6 is not worried about this case because, all the processes in xv6 are single threaded, even though the kernel is multithreaded.