

Figure 1: GDB steps

From the GDB tracing we find that malloc returns to sbrk. For the purpose of the exercise, the parsing portion will be ignored to focus on the concepts of malloc.

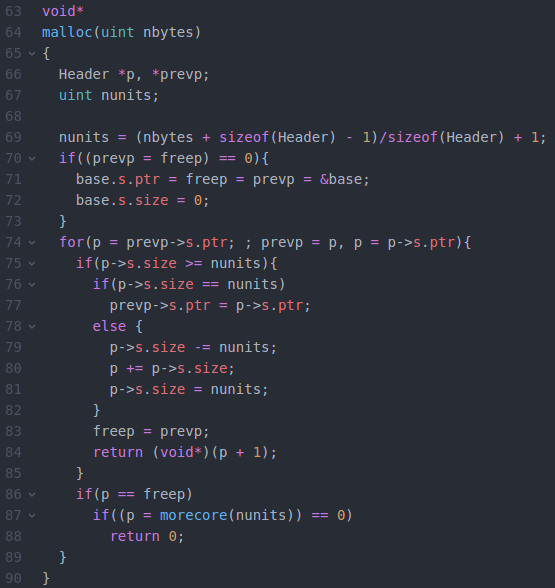


Figure 2: umalloc.c malloc

To being, the user calls on the function malloc from umalloc.c. It is evident that given valid conditionsit will adjust the pointers and size of the header. Once adjusted and the new header is valid, it will call on morecore.

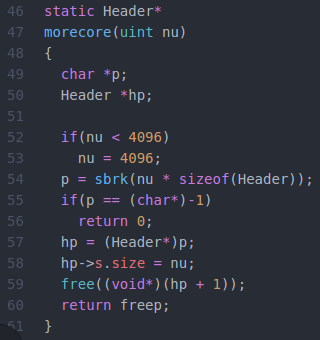
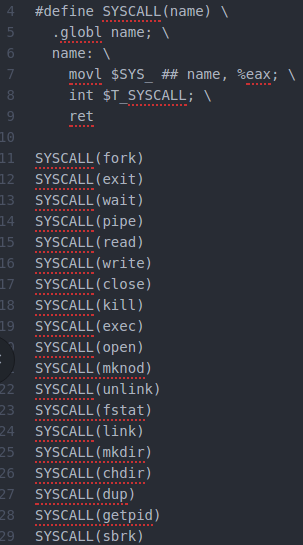


Figure 3: umalloc.c morecore

morecore then gives the new unit size (minimum 4K for page size) to sbrk.



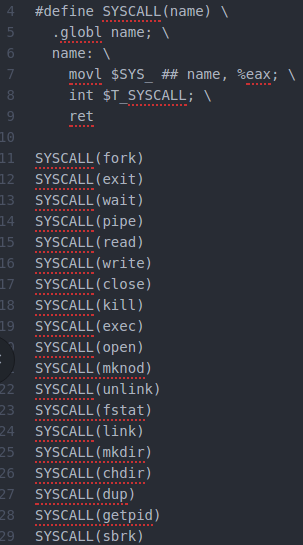
 

Figure 4: usys.S user sbrk Figure 5: syscall.h SYS\_sbrk define

The function sbrk can be found from the usys.S defined library of functions. This utilizes syscall.h to return the syscall number, resulting in the appropriate arguments and registers before performing the trap.

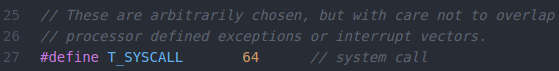
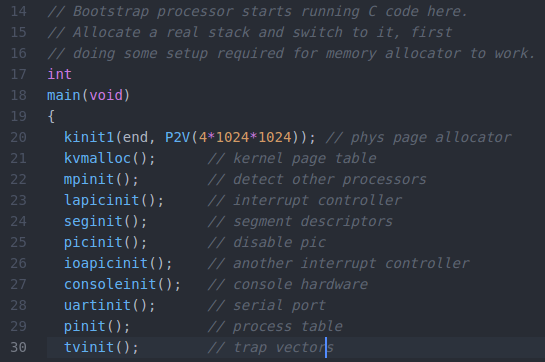


Figure 6: traps.h T\_SYSCALL define

The pushed arguments are then compared to traps.h definitions to determine which type it is, in this case a T\_SYSCALL. At this point privilege/level is changed as the trap function changes from user to kernel.



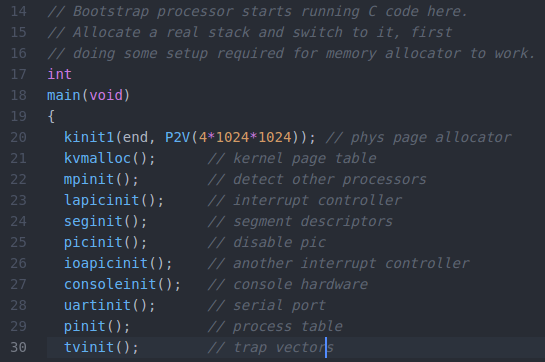


Figure 7: main.c main

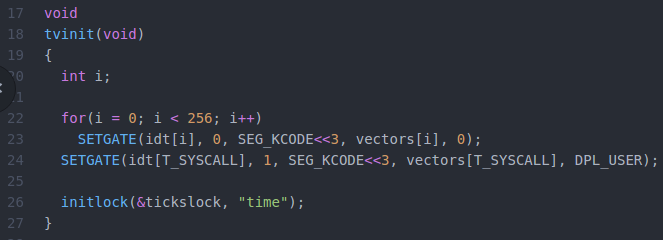


Figure 8: trap.c tvinit

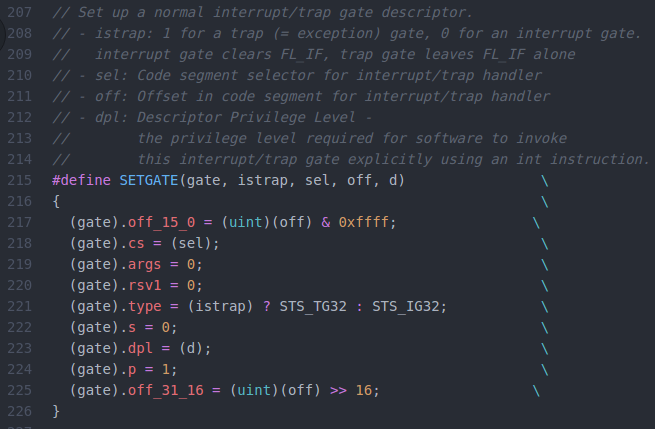


Figure 9: mmu.h SETGATE

It is important to note that the trap table has been initialized during the boot process of the system, which allows it so that any point after boot to execute the defined trap functions. This is done with defining the offsets of the trap vectors.

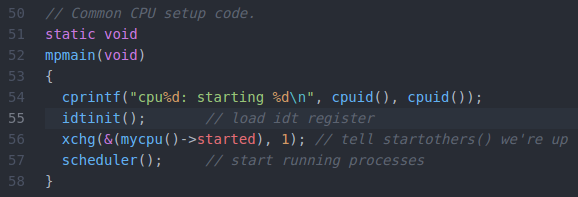


Figure 10: main.c mpmain

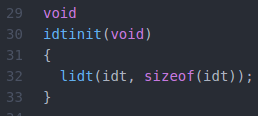


Figure 11: trap.c idtinit

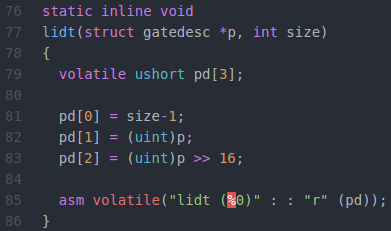


Figure 12: x86.h lidt

The previous set of figures lists how the processor itself (hardware) accesses the table from memory, which in turn allows for some work to be done at the hardware level. While this sequence of trap table tracing was not necessary, it explains priori the access of the trap from memory (and sequentially this occurs before user call as it is set during boot). At this point the trap table is set up, and the tracing of sbrk (and malloc) may continue.

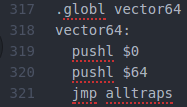


Figure 13: vectors.S global vector

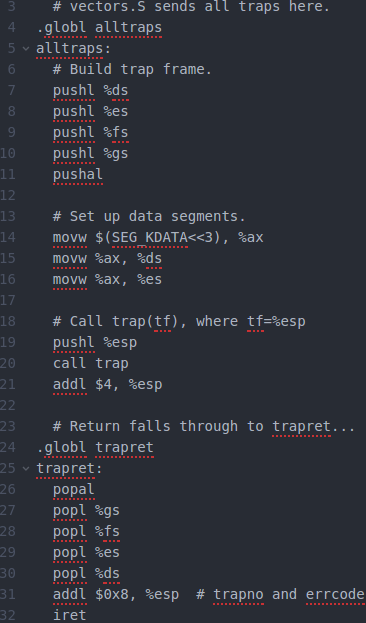


Figure 14: trapasm.S alltraps

Continuing from sbrk when the trap is called, the first thing performed is a push of the trap number followed by a saving of registers to the stack. Afterwards, these registers are adjusted to access kernel memory for trap handling.

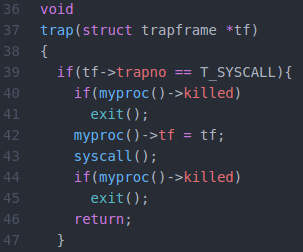


Figure 15: trap.c trap

The trap number read 64, or T\_SYSCALL, thus focusing on that segment of the trap function. It is at this point where the table of system calls is referred to for the statements to perform.

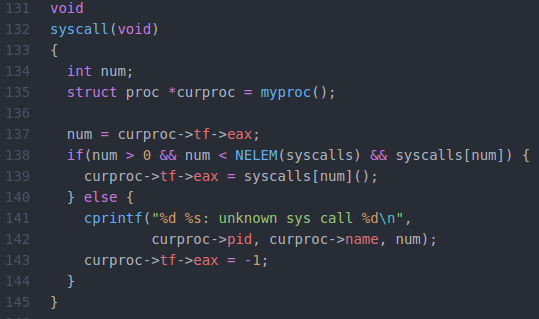
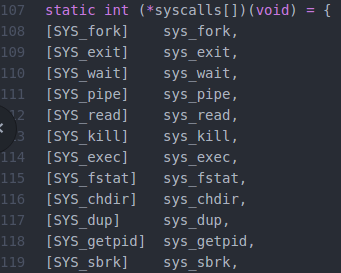


Figure 16: syscall.c syscall



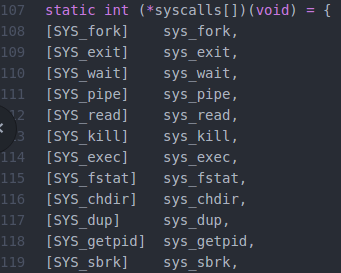


Figure 17: syscall.c syscalls[]

In the function it is found that eax is referred to, better known as the syscall number (12). In this case, sys\_sbrk is called and its return value is set back to eax.

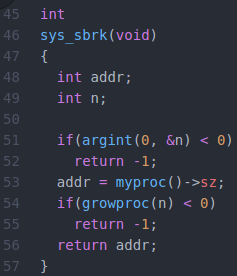


Figure 18: sysproc.c sys\_sbrk

Now beginning with the syscall for sbrk, the sub-function argomt is done to validate the argument for malloc.

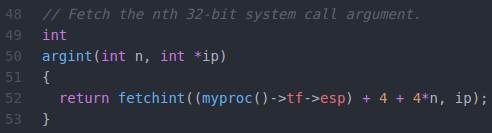


Figure 19: syscall.c argint

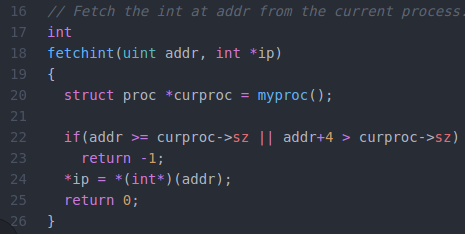


Figure 20: syscall.c fetchint

As provided with in-code documentation, it pulls the process state (myproc is the structure) and fetches the nth 32-bit syscall argument. In fetchint, it validates the address for bounds and size before returning (rewriting the pointer). Now exploring the myproc:

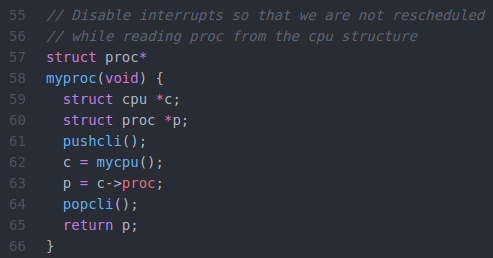


Figure 21: proc.c myproc

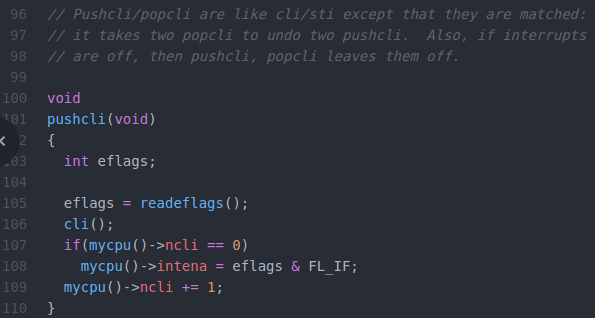


Figure 22: spinlock.c pushcli

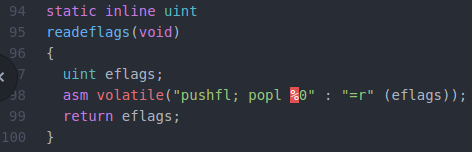


Figure 23: x86.h readeflags

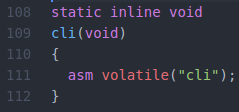


Figure 24: x86.h cli

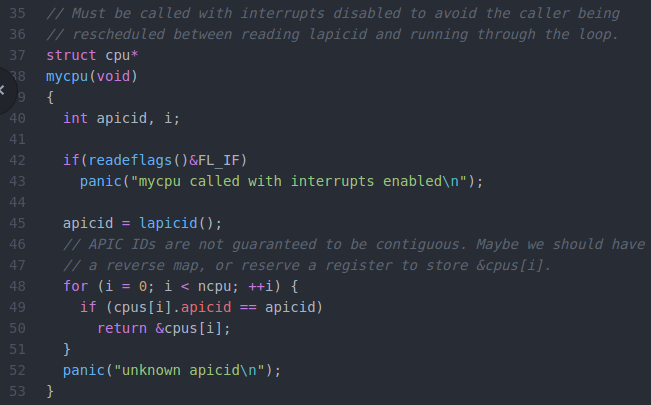


Figure 25: proc.h cpu

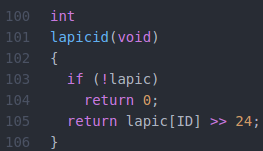


Figure 26: lapic.c lapicid

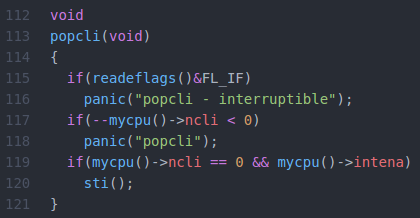


Figure 27: spinlock.c popcli

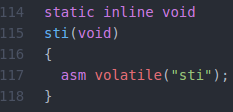


Figure 28: x86.h sti

When calling myproc, pushcli disables interrupts to allow mycpu to retrieve the local APIC, which in turn retrieves the desired process. popcli ends this set of actions by removing the paired pushcli.

Returning to sys\_sbrk (Figure 18), the next performed function is growproc.

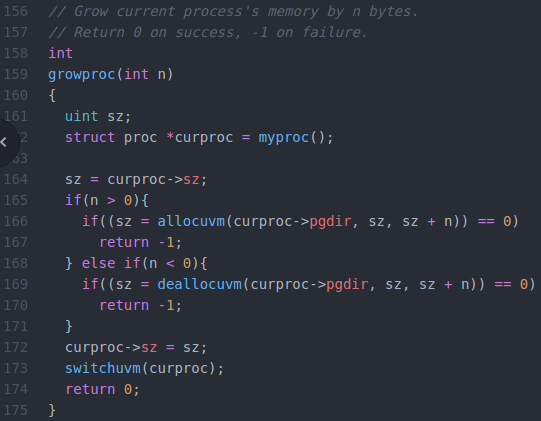


Figure 29: proc.c growproc

As visible, it will either allocate or deallocate n bytes from the process through the two sub-functions.

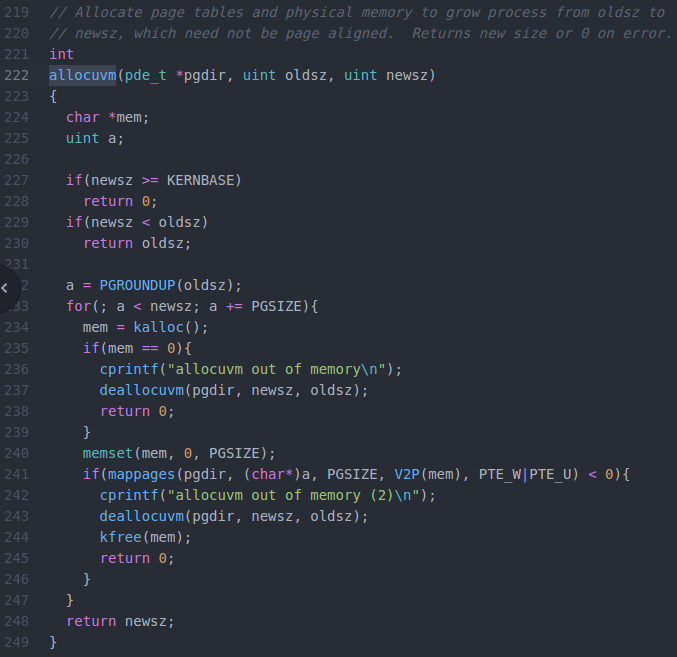


Figure 30: vm.c allocuvm



Figure 31: mmu.h PGROUNDUP

Simple rounds up the page table given an argument.

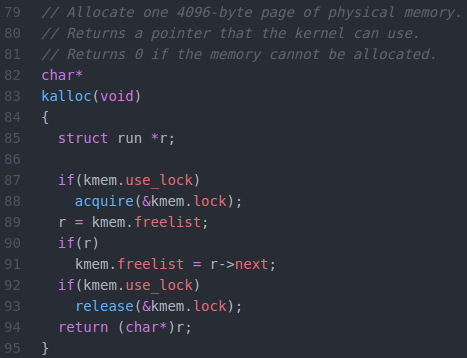


Figure 32: kalloc.c kalloc

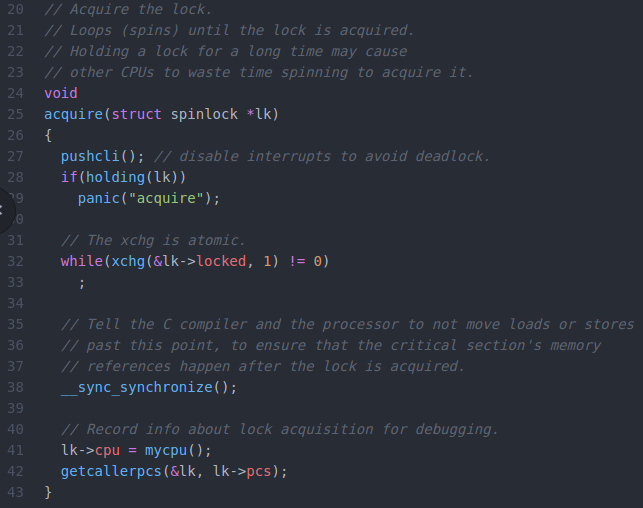


Figure 33: spinlock.c acquire

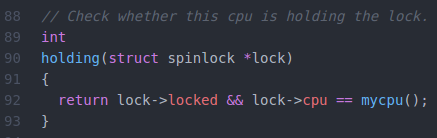


Figure 34: spinlock.c holding

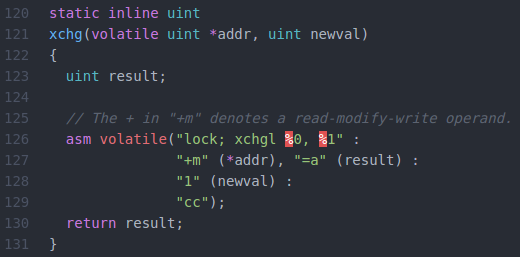


Figure 35: x86.h xchg

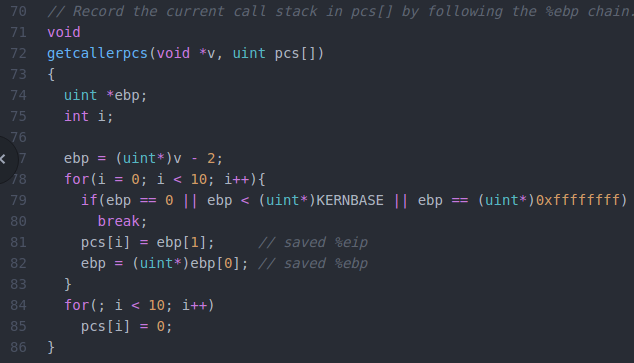


Figure 36: spinlock.c getcallerpcs

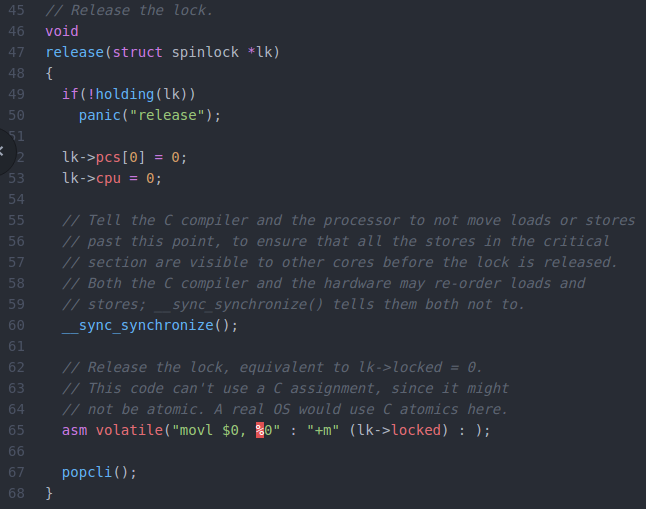


Figure 37: spinlock.c release

The process of kalloc is extensive, in which the process itself will allocate a page of memory and return the pointer. The sub-function acquire will wait for a lock (checks through holding) before continuing to manipulate the memory through moving the free pointer to next. The lock is then released and pointer is returned. In short, kalloc retrieves the free-list pointer, moves the head of said pointer to next and returns the pointer for usage.

Continuing in allocuvm (Figure 30), assuming valid memory it will allocate memory through memset:

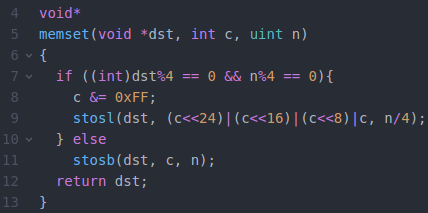


Figure 38: string.c memset

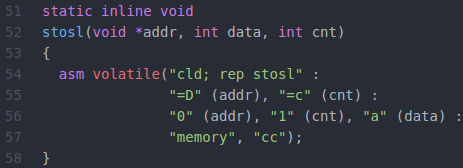


Figure 39: x86.h stosl

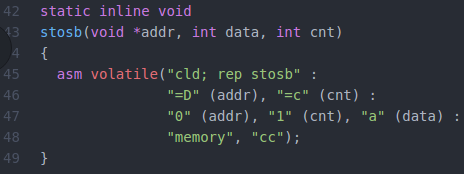


Figure 40: x86.h stosb

memset validates the destination against size, in which depending on condition will store the appropriate string (long vs byte).

Returning to allocuvm (Figure 30) are conditions to check allocuvm through sub-functions mappages and V2P.

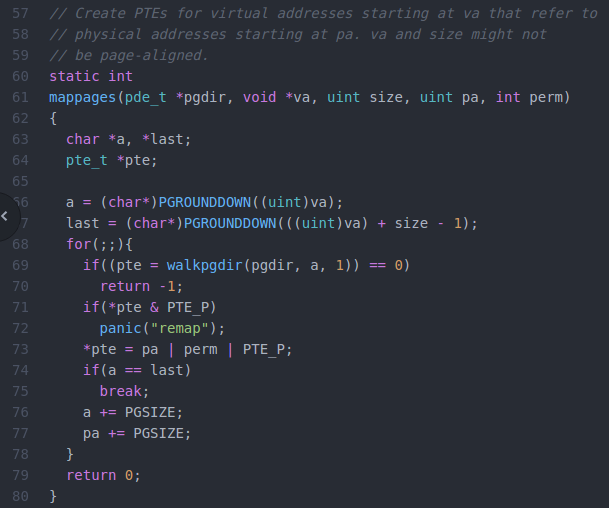


Figure 41: vm.c mappages

The function creates page table entries by walking the table by increasing the page size every iteration.

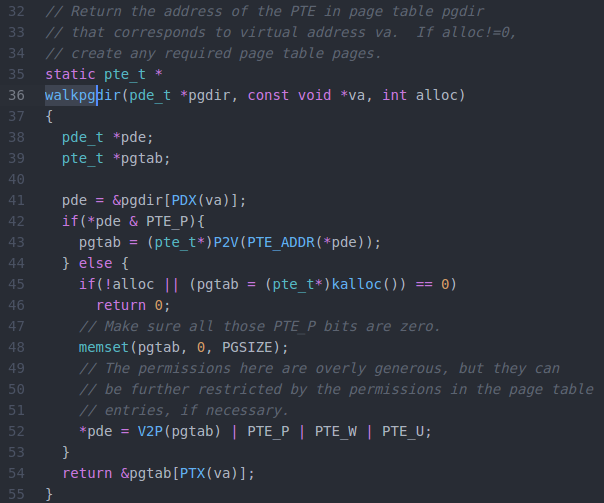


Figure 42: vm.c walkpgdir

Returns valid addresses of the page table corresponding to virtual addresses through validation of the page indices.



Figure 43: mmu.h PDX



Figure 44: memlayout.h V2P and P2V



Figure 45: mmu.h PTE\_ADDR

The three functions simply return the page index (PDX given virtual address), or the desired address given the opposite. The address itself is returned through PTE\_ADDR.

Returning to allocuvm (Figure 30), in the event that the condition is true or growproc was given a negative argument, the function proceeds to deallocuvm.

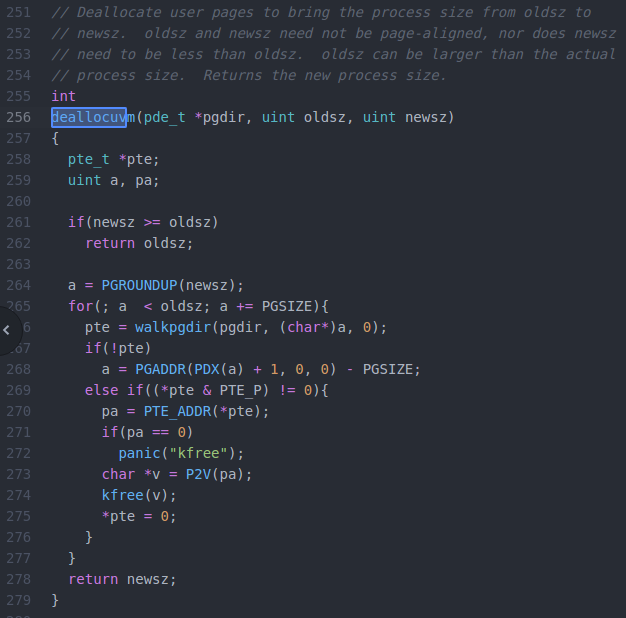


Figure 46: vm.c deallocuvm

Similar to allocuvm with walking, with the exception of kfree instead of increasing size.

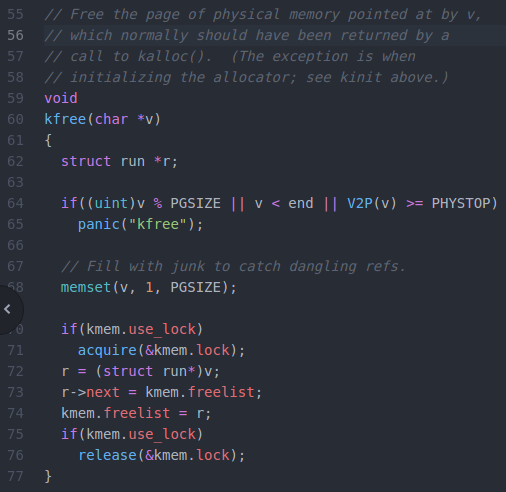


Figure 47: kalloc.c kfree

The opposite of kalloc, freeing the memory and changing the freelist accordingly.

Assuming that de/allocuvm was successful, it will return the new size to growproc (Figure 29), which will set the new size and execute switchuvm.

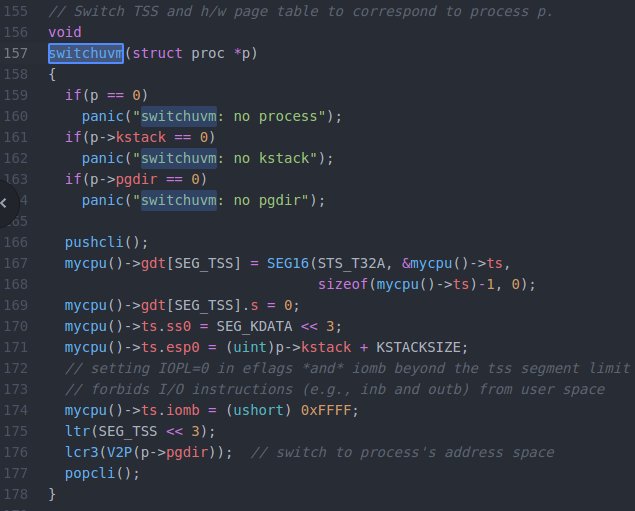


Figure 48: vm.c switchuvm

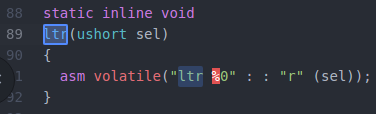


Figure 49: x86.h ltr

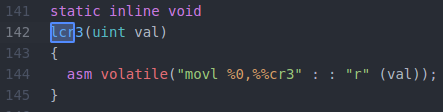


Figure 50: x86.h lcr3

Per documentation, switches the tables so to correspond to correspond to the process, done by manipulating mycpu structure.

Afterwards, growproc returns and begins recursively returning back to user. The returned integer (0 success, 1 fail) is passed as the eax variable and returned to trap.

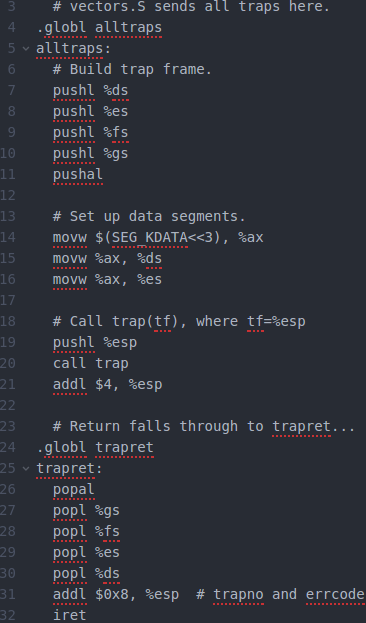


Figure 51: trapasm.S trapret

Everything from the stack prior to the trap call is popped, the privilege/level is set back to user, and the instruction returns back to main. The process results in either a changed process size or a returned error which can be handled accordingly. This returns up to morecore (Figure 3) and validates this pointer and returns 0 or the header freep accordingly.