Programming Assignment 6: Paging

Satpute Aniket Tukaram : CS21BTECH11056

April 20, 2023

I have implemented demand paging and copy on write in different files
demand paging : xv6_dp.tar.gz
copy on write : xv6_cow.tar.gz

Part-1: Implement demand paging

- For implementing demand paging we first have to ensure no global data or heap is allocated at time of creating process
- This is done by modifying the exec.c file which allocates memory to process when called the exec() function
- In this function we have to modify allocuvm call such that the memory/pages is only allocated for program part and then there is gap in page table, which will contain global data which has all present bits mapped to 0 after which there is stack which is allocated fix number of pages (here 1 page) and the heap is after stack, all pages after stack are marked invalid
- When a page fault occurs it occur as a trap to system these traps are handled through trap() function in trap.c
- We have to modify this fuction so that when page fault occurs we have to check if the fault is due to the present bit else it is invalid access and trap occurs
- If the fault is due to present bit we check which page is accessed and we use a frame to bring that data from disk to memory and map that frame to the page table entry and mark it valid
- After which the intruction is restored and execution become as it was before

Part-2: Implement copy-on-write

- For copy on write we first create an array of unsigned int for each physical frame,the value of this array with respect a physical address will give number of processes associated with that frame,this array is defined in kalloc.c in struct kmem

- Initially all the array elements are initialized to zero
- when we invoke kalloc is called anywhere i.e allocate memory to a process the associated reference count is increamented by one
- So to implement copy on write when we call the system call fork we call a function copyuvm which copies the content from parent process to child process and maps the pages to copied memory
- Instead copying data to new frame we just map the page table entry to that frame and increase the reference count of the given frame and the frame entry is marked as read only in both parent and child
- If any child or parent tries to write to those pages then the system incur a trap
- We check if the trap is caused by write bit then:
- If that frame as reference count greater than 1 then the process is allocated with new frame and data is copied to it and the page table entry of that process is marked a writable
- If reference count is 1 then just update the write bit of that entry and exit

Observations

- the data resides between code of process and stack, data is fixed and size does not change over course of process
- Different function like loaduvm, allocuvm load and allocate data to given process mapping page table entries to physical frame created bys using kalloc()
- Different processes do fork in the course of their existence
- If we create an int array of size 10000 the n thousandth array exist in nth frame
- We can add structure to system data structures to implement features like copy on write
- We require locks to update such data stuctures