Assignment 6 - Paging

Vishal Vijay Devadiga (CS21BTECH11061)

Observations

- Process PID=2 calls for copy-on-write page fault in the address `0x19a0` everytime a user process gets finished. It clutters up the the shell, why is why I avoided printing it.
- My implementation works with both demand paging and copy-on-write being simultaneously on.
- My implementation returns `pa=0` for page table entries that were never demand paged into the page table. This messes up with the `deallocuvm` function, which is why I have removed the panic statement when `pa=0` and let the function continue.

Demand Paging

Originally, in xv6, the OS would load all pages regardless of it being used or not. To implement demand paging, I changed these following files:

- `exec.c`: Instead of allocating till `memsz` that is size of code + size of all global data, I allocated till `filesz` that is size of code only. I then modified the size of program memory to be `memsz` without allocating the remaining size. This ensures that the global data will go the right page table entry when needed. The program would then allocate the guard page and the user stack as before.
- `vm.c`: Instead of using `allocuvm` function, I created a new function `dempageuvm` that allocates a single page at the given address. Most of the code, if not all, is based on the `allocuvm` function.
- `def.h`: Defined `dempageuvm` function to be available to any program that imports `def.h`.
- `trap.c`: When a page fault occurs, I obtain the address from the `rcr2` register from the trapframe. I then check whether this address is valid or not by comparing it to size of program memory. I then check whether the page is present or not, if not then I pass the required page table entry address to `dempageuvm` and return if successful.

This implements demand paging to xv6.

Copy-on-write

Originally, in xv6, the OS would copy all pages of a parent process to a child process regardless of it being modifiable or not. To implement copy-on-write, I did:

- `vm.c`: Instead of using `copyuvm`, I created a new function `forkuvm` that creates a page table, and copies the address of the physical page instead of making a new page and sets it as the page table entry.
- `vm.c`: Created a `copypageuvm` function, that checks if the processes referencing it are greater than 1, then copy the page and set it as page table entry, else set the original page as page table entry.
- `kalloc.c`: Modified kmem struct to contain an reference array to the pages. Modified `kfreerange` function to set the intial value of all the possible pages to 0. Created `add_ref`, `get_ref`, `dec_ref` functions that help in number of references to a page. Also modified `kalloc`, `kfree` to set the reference to the required value(Decrease by 1 if possible in case of `kfree` and set it to 1 in case of `kalloc`).
- `def.h`: Defined `forkuvm`, `copypageuvm`, `add_ref`, `get_ref`, `dec_ref` function to be available to any program that imports `def.h`.
- `trap.c`: When a page fault occurs. I obtain the address from the `rcr2` register from the trapframe. I then check whether this address is valid or not by comparing it to size of program memory. I then check whether the page is writable or not, if not then I call the `copypageuvm` function.

This implements copy-on-write to xv6.