Copy-on-Write Fork

OS Lab UG TAs

December 15, 2024

1 Overview

There are three main parts to this assignment.

1. Implementation of the `getNumFreePages` syscall.
2. Making changes to the `fork` syscall so that the child and parent share pages.
3. Trap handling so that when either the child or parent accesses the shared pages, process isolation is maintained by allocating them new pages.

2 `getNumFreePages` syscall

The `getNumFreePages` returns an integer, the number of free physical frames, and takes no argument. The changes in `user.h, usys.s, syscall.h, syscall.c, defs.h`[1] and `sysproc.c` are very similar to the changes for the `numpp()` syscall implemented in *Part-A*, so it will not be elaborated on here. The idea behind the algorithm is pretty simple, there is a structure called `kmem.freelist` which consists of the free physical frames currently in the system. So all we have to do is find the length of this structure. It is kept as a linked list, so we simply perform a linked list traversal and return the length of the list traversed. The structure is only accessible in `kalloc.c` so we must define our function in `defs.h` and implement it in `kalloc.c`.

3 Changes to the `fork` sycall

Pretty much all of our changes to `fork` syscall will occur in the `copyuvm` function in `vm.c`. In order to make changes here it is crucial to understand how the function works.

3.1 Understanding `copyuvm`

`copyuvm` takes as argument the page directory of the parent process, and an integer that denotes the size upto which pages have to be allocated to the child process. It creates the page directory for the child process (using `pde_t* d = setupkvm()`) and returns it. Then for every page that it has to copy from parent to child it:

1. Finds the page table entry for the page using `walkpgdir`.
2. Ensures that the page exists and is present using `(*pte & PTE_P) != 0`.
3. Finds the physical address corresponding to that page using `uint pa = PTE_ADDR(*pte)`.
4. Finds the permissions granted to that page using `uint flags = PTE_FLAGS(*pte)`.
5. Allocates a new physical frame for the child process using `char* mem = kalloc()`.
6. Copies the data from the parent process' page to the child process' page using `memmove(mem,P2V(pa),PGSIZE)`.
7. Maps the page with the appropriate permissions to the child process' page table using `mappages(d, (void*)i`[2]`, PGSIZE, V2P(mem), flags)`

[1]Once again, it is theoretically possible to do this entirely in `sysproc.c`, however, it is cleaner to do this in `kalloc.c`
[2]i here is the virtual address

3.2 Changes to `copyuvm`

We need to make the following changes to the function:

1. No new physical frames are allocated for the child process.
2. Each page of the child process points to the same physical frame as the corresponding page for the parent.
3. Each page of the child process is marked as read only.
4. The corresponding pages for the parent process are also marked as read only.

In order to do this, after extracting the page table entry and ensuring it exists and is present, extracting the physical address, we modify the page table entry to take away write-permissions using `*pte = *pte & ∼(PTE_W)`. Then, we extract the flags and then instead of allocating a new frame using `kalloc()`, we simply call `mappages(d,(void*)i,PGSIZE,pa,flags)`. We need to note that we have modified the page table of the parent process and as such need to reinstall the page table and flush out the TLB entries, this is done using `lcr3(V2P(pgdir))`.
With this, our `fork` syscall is now fully modified. However, we are still not done, we need to perform trap handling else after forking the parent and child will be stuck with read-only pages![3]

4 Trap Handling

On a 10000 feet view, in the trap handling function we need to:

1. Correctly identify whether the trap is actually caused due to the changes in the `fork` syscall.
2. If not, we need to announce that it is a process error.
3. Otherwise, we need to check whether this is the only process using that frame.
4. If yes, we need to update the process' permissions appropriately (give it write permission).
5. If no, we need to allocate a new physical frame and copy data from the old frame to the new one.
6. Regardless of whether it is the only process or not, we need to update its page table (either by giving it permission or by changing its mapping).

4.1 Correctly identifying whether the trap is due to the changes in the `fork` syscall

This can be done by adding an indicator to all pages that have been affected by the CoW fork. We define this indicator as a new page table entry bit inside of `mmu.h`, call it `PTE_COW`[4]. We need to modify our `copyuvm`, so that instead of just taking away write permission using `*pte &= ∼ PTE_W`, we also perform `*pte |= PTE_COW`[5].
Now, inside of `trap.c`, under the case of `T_PGFLT`, we do the following:

1. Obtain the faulting virtual address using `char* va = rcr2()`.
2. We now need to obtain the page table entry corresponding to this address, however, since this is not doable in `trap.c`, we defer to a function defined in `defs.h` and implemented in `vm.c`, called `handlePageFault`.
3. Using `pte_t *pte = walkpgdir(myproc()->pgdir,va,0)`, we obtain the page table entry corresponding to the offending address.
4. We ensure that the entry exists and check using `*pte & PTE_COW`, that it is indeed caused due to changes in `fork`. If it isn't, we announce a process error.

[3]What happens in practice if we remove the trap handling part? Can the system boot? Can the system create a shell?

[4]Do we really need to do this? A naive solution could involve assuming that all page faults that occur due to missing write permissions are a result of the CoW fork, but will that work in practice?

[5]Theoretically, we should only be installing this indicator if the page originally had write permission

4.2 Checking whether it is the only process using that frame

This requires us to keep track of the number of processes using a frame. In order to do this we need to create a structure/array of integers inside of `kalloc.c`. The size of the array must be $\frac{\texttt{PHYSTOP}}{\texttt{PGSIZE}}$ as there must be a counter per frame. We initialise the counter to be 0 for each frame, when `kalloc()` is called, we modify the value of the counter for that frame to 1. We will need to modify the `kfree` function to ensure that frames are only freed when they are not in use by any process. So when kfree is called on some virtual address `v`, we will check the value of `refs`$[\frac{\texttt{V2P(v)}}{\texttt{PGSIZE}}]$ and if it is > 1 then we decrease it and return, otherwise, we add the frame to the freelist. We will also need to appropriately update the value of refs. In order to do this, we shall modify `copyuvm`, and invoke a function to increment the refs correspond to a particular physical address[6]. The function takes in `uint pa` as input and all it does is `refs`$[\frac{\texttt{pa}}{\texttt{PGSIZE}}]$`++`.

4.3 Control flow based on number of processes using that frame

We can obtain the number of processes using that frame by checking the value of `refs`$[\frac{\texttt{pa}}{\texttt{PGSIZE}}]$.

4.3.1 `refs`$[\frac{\texttt{pa}}{\texttt{PGSIZE}}]$ `= 1`

In this case, we should not be allocating a new frame to this process, instead, we should simply modify the permissions of the page table entry. To do this we:

1. Obtain the page table entry using `pte_t *pte = walkpgdir(pgdir,va,0)`.
2. Assuming that the trap is indeed due to the CoW fork, perform `*pte |= PTE_W` and `*pte &= ~PTE_COW`.
3. Since we have modified the page table, we must perform `lc3(V2P(pgdir))`, which can also be done by invoking the `switchuvm()` function

4.3.2 `refs`$[\frac{\texttt{pa}}{\texttt{PGSIZE}}]$ $>$ `1`

Here we need to allocate a new physical frame for the process, copy the data to the new frame from the old frame, decrement the count of references to the old frame and final change the page table mappings. To do this we:

1. Obtain the page table entry using `pte_t *pte = walkpgdir(pgdir,va,0)`.
2. Obtain the physical address using `uint pa = PTE_ADDR(*pte)`.
3. Allocate a new frame using `char* mem = kalloc()`.
4. Copy data from the old frame to the new frame using `memmove(mem,P2V(pa),PGSIZE)`.
5. Decrement the references to the old physical frame using `kfree(P2V(pa))`.
6. Obtain the permissions using `uint flags = PTE_FLAGS(*pte)`.
7. Modify the permissions using `flags |= PTE_W` and `flags &= ~PTE_COW`[7].
8. Reassign the page table mapping using `*pte = V2P(mem) | flags`.
9. Once again, since we modified the page table, we need to perform either `lcr3(V2P(pgdir))` or `switchuvm()`[8].

[6]As usual, define this function in `defs.h` and implement it in `kalloc.c`.
[7]What happens if you don't do this? Hint : There will be another page fault, but what then?
[8]Could we use `switchuvm` during fork creation? Why or why not?