

CS344 - Assignment04

Memory Management

Kartheek Bellamkonda

190101023

Introduction

Youtube link : <https://youtu.be/7PlzVDnjDx8>

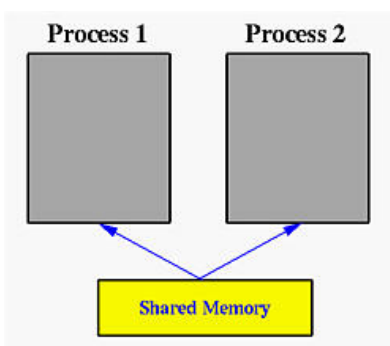
In this report we will compare the Memory management features and aspects present in our operating system in Xv6 & LINUX OS and we will find out some of the missing features in our toy OS with respect to the LINUX and we will try to implement them. This report will give us a insight to those missing features and guide to implement them.

Xv6 is a toy operating system and it lacks so many features compared to the real time operating systems such as Linux. Xv6 clearly lacks the following features: demand paging from disk , shared memory, Copy-on-Write fork , automatically extending stacks , lazy allocation and memory mapped files.


In this report we will see the implementation details of the following three features demand paging , shared memory and copy-on-write fork()

Shared Memory:

Shared memory is one of the missing features in xv6-OS when compared to Linux OS. In xv6 each process has its own virtual address space. Take an example where the same libraries are being used in multiple processes so for each of the processes instead of allocating the same libraries multiple times to reduce the redundancy we can use the concept of shared memory where two processes having two different virtual addresses will still share the same physical memory space. This is one of the major missing feature in xv6 and one of the other interesting usage of shared memory is instead of using pipes for IPC we can also shared memory to enable the interprocess communication so with all these reasons Shared Memory is a good feature that lacks in xv6 and it is better to implement.



The following figure shows two processes and their address spaces. The yellow rectangle is a shared memory attached to both address spaces and both process 1 and process 2 can have access to this shared memory as if the shared memory is part of



its own address space. In some sense, the original address space is "extended" by attaching this shared memory.

The process which creates the shared memory would be created with the help of a key and in general it is known as a server and the other processes can access this shared memory using the key and they are generally called as clients.

Implementation Details


For Implementing the shared memory we would be defining 3 functions namely `shmget()` `shmat()` `shmdt()` in the **file vm.c** the functionalities of these functions are `shmget()` is used to create a shared memory and `shmat()` is used to attach a already existing shared memory to the corresponding process address space and similarly `shdt()` is used to detach the corresponding shared memory from the process's address space.

We will also add a new field in the `proc.h` struct known as `top` and will be initialised at the time of `alloc proc` which is nothing but the end value of the virtual address space .

`int shmget(int key, int num_pages,int flags)`

`shmget` function is used to create a shared memory. We have to implement the following function in `vm.c` file when a process calls the `shmget` it will take the parameters as:

1. **Key:** A key can be used to uniquely identify the shared memory and whenever any other process requires the shared memory it can call the corresponding key and the key can be generated manually or we can have an automated random unique key generation mechanism.
2. **num_pages:** The function will take the number of pages to be allocated to the shared memory and we will assign accordingly.
3. **Flags:** flags can be used to specify the various types of operations that can be performed on the given shared memory like we can enable flags to make the memory `read_only` like that,for this purpose flags are used.



So whenever the function is called we will call the `kalloc()` function and get some memory and allocate the memory and update the page table entry with the `mappages()` function where we assign the `proc->top` as the virtual address and we will decrease the top of the given process by a page size so here we are assigning the shared memory at the top of the available process virtual space and here one more thing is we would maintain a data structure map that would map the given key to a physical address of its memory and similarly we would also maintain the properties of the given shared memory by accessing like **`key[i]->phyaddr`**, **`key[i]->numpages`**, **`key[i]->valid`** this valid bit can be set when a shared memory with the corresponding key is present or not so we would also define a struct named `key` which would contain the physical address `num_pages` and `valid` boolean and we will return a 1 on success and 0 on failure.

`int shmat(int key):`

So whenever a process wants to use an existing shared memory it can use that using the function `shmat()` and it is also implemented in `vm.c` where we can get the physical address of the shared memory with the given key using our data structure map which maintains a key phy addr pair so we can get the corresponding physical address and now we can directly call the `mappages()` with the entry at physical address same as the returned one by our data structure and the virtual address is same as the `proc->top` and we will decrease the `proc->top` by a page size that is we have been allocating the shared memory virtual space at the top of the processes virtual space and update the page table entry using `mappages`. The function would return the virtual address corresponding to the physical address of the shared memory

`int shmdt(int shmaddr):`

Here we want to detach the attached shared memory so we would store the virtual address of this process where the shared memory is attached and we will get it from the return value of the function `shmat()` and we pass it as parameter and now to detach it we will simply update the entry in the process page table corresponding to the given virtual address we will make the corresponding page table entry as invalid by unsetting the present bit present in `xv6` as it is no longer used to point the shared memory.

A general scheme of using shared memory is the following:

- For a server, it should be started before any client. The server should perform the following tasks:
 1. Ask for a shared memory with a memory key and memorize the returned shared memory ID. This is performed by system call **shmget()**.
 2. Initialize the shared memory, if necessary.
 3. Do something and wait for all clients' completion.
 4. Detach the shared memory with system call **shmdt()**.
- For the client part, the procedure is almost the same:
 1. Ask for a shared memory with the same memory key and memorize the returned shared memory ID.
 2. Attach this shared memory to the client's address space.
 3. Use the memory.
 4. Detach all shared memory segments, if necessary.

Here the notation server is the one who created the shared memory and the client is the user of the shared memory.

So we can create and use the shared memory using the above implemented functions i.e create using `shmget()` and use the existing one with `shmat()` and can detach with `shmdt()`. So we have successfully implemented the shared memory.

Demand Paging:

One of the main features lacking in xv6 and present in Linux is Demand paging. Demand paging is a mechanism where we only bring the process pages into main memory when there is a demand for them i.e when there is a instruction to run from that page then only we will bring that page into the main memory , this is often referred as lazy allocation and the missing page will cause a page fault and lazy allocator will do the work. As traditional xv6 brings all the pages of the process whenever it is executing it may not be feasible all the time so what we will do is whenever there is a requirement then only we will bring it to main memory.

Implementation details:

To implement the demand paging mechanism we will be modifying the following files.

sysproc.c:

Here we will modify the `sys_sbrk` function where this function is called by a process to allocate the page a frame and it in turn will call `growproc` present in `proc.c` file which will allocate the memory for our process so instead of calling `growproc` we can modify the size of process to increase by the requested size but not really allocate it.

trap.c:


In `trap.c` file we will handle the page fault case when a page fault occurs we will call the `handler()` function and we will pass the instruction which caused the page fault(see about `rcr2()`) and it would be implemented in the `proc.c` file.

proc.c:

We will here implement the `handler()` which as argument we get the instruction which caused a page fault and we can find out the page where it belongs to and if page is already swapped out we can check it by swap out bit and if that's the case we will bring it from backing store or else we would be bringing it from the disk now and we can call the `kalloc` function to get some physical memory if there is no free frame available then swap out a existing page and free a frame and again now call `kalloc` and assign to it and we can use `mappages()` to update the page table entry.

proc.c:

In `proc.c` we will now implement the swapper function, to choose a victim frame Linux enhanced second chance LRU algorithm so we will also use the same policy. So what we have to do is we should get the outer page table of the process and we will check for the reference bit if it is zero we will get the inside page table and first we will check for the page with both dirty and reference bit unset if not found then again we will check for the page with reference bit unset but dirty bit is set if not found such page then we will check for the pages with reference bit set and dirty bit unset and similarly if we don't find any such pages 'we will finally take a page with both reference bit and dirty bit set and like this we will find the victim frame similar thing can be done for the same for outer page table reference bit set cases and we can do all those things in a for loop and some conditions to check.



Once we got the victim frame we will use one of the free bits for marking this page as swapped out & we will move it to backing store for that we will name a file and store all these files with .swap extension generally in linux there is a special swap space available so we will name the files such that we can uniquely identify them and we will write all the information in page to that file and we can free the page table now by making the page table entry valid bit unset and similarly we can call the kfree function and pass the victim page which will free the frame and add to the free list.

Thus we can implement the demand paging in the following way.

Copy-on-Write (COW)

We know that when an `fork()` system is called in xv6-OS it will completely copies the user space memory of the parent process to the child process. The drawbacks present here are if the user space memory of parent process is high then it will take much time to copy all the contents to child process. Consider the case when a child executes an "exec" system call (which is used to execute any executable file from within a C program) or exits very soon after the `fork()`. When the child is needed just to execute a command for the parent process, there is no need for copying the parent process' pages, since exec replaces the address space of the process which invoked it with the command to be executed.

So we can see that the copying often wastes the memory i.e. in most of the cases neither the parent nor the child modifies a page so that in conclusion what we can do is simply share the same physical memory but on the other hand if either of the parent or child writes to a page then truly a copy of the page is needed.

So the Copy-on-write strategy is been used in Linux as optimisation to save the memory space while forking so here what COW will do is it doesn't assign a separate memory space for child process instead it will just only create the page table with the page table entries which will point out to the parents process pages and whenever there is a write operation is being tried to perform then it will actually creates a copy of the page for the process which is trying to modify and update in the corresponding page table entry and now the write operation is being performed in the latest copy created. As a page would be copied only when a write is performed it is known as Copy-On-Write.

Implementation Details:

As we know we are optimising the `fork()` function using the COW policy. So when someone calls the `fork()` system call it will copy the parent process pages to the child and it will be done by calling the `copyuvm()` function. So to implement the COW mechanism we have to change the `copyuvm()` function first which is present in `vm.c`.

`copyuvm()`:

```
315 pde_t*
316 copyuvm(pde_t *pgdir, uint sz)
317 {
318     pde_t *d;
319     pte_t *pte;
320     uint pa, i, flags;
321     char *mem;
322
323     if((d = setupkvm()) == 0)
324         return 0;
325     for(i = 0; i < sz; i += PGSIZE){
326         if((pte = walkpgdir(pgdir, (void *) i, 0)) == 0)
327             panic("copyuvm: pte should exist");
328         if(!(*pte & PTE_P))
329             panic("copyuvm: page not present");
330         pa = PTE_ADDR(*pte);
331         flags = PTE_FLAGS(*pte);
332         if((mem = kalloc()) == 0)
333             goto bad;
334         memmove(mem, (char*)P2V(pa), PGSIZE);
335         if(mappages(d, (void*)i, PGSIZE, V2P(mem), flags) < 0) {
336             kfree(mem);
337             goto bad;
338         }
339     }
340     return d;
341
342 bad:
343     freevm(d);
344     return 0;
345 }
```

So this is the default `copyuvm()` function present in xv6 here we can see that `kalloc` function has been called and we are using `memmove` to copy the contents from the page to newly allocated memory and finally we are calling the `mappages` to update the corresponding page table entry from pointing it to the latest allocated mem so for implementing cow we will comment out the part of allocating memory and in the `mappages` at the physical address we will pass the physical address of the parent process by this way we are just mapping the pages to parent process pages. And we will have one free bit which is available

for general use in xv6 at 8th position from right we will use it to denote whether the page is cow or not and we will set that bit in pte and similarly we will unset the `PTE_W` bit i.e. the write permission is disabled.



trap.c:

Now whenever a process tries to write in a page but it don't have write permission it will result a trap page fault error so we have to make a copy for the page now and we will implement it in trap.c file we can implement an handler() function which finds out the instruction and gets corresponding page table entry and page number and we will check whether the page fault is due to COW or not by checking whether the PTE_cow bit is set or unset and if it is set then we call the kalloc function and now we will get a frame from the free list and now we can use memmove function and copy the contents of the page into the allocated memory and we have to update the page table entry we will call the mappages function and we will map the page table entry to the physical address which is allocated freshly by kalloc function and we have to set the permissions in the page table to allow writing in this page by setting the PTE_W bit which is previously unsettled and similarly we will unset the PTE_cow bit which is previously set.

Conclusion

So finally We have compared the memory management component in both the xv6 and linux os and try to find some missing features and tried to provide a detailed overview of implementation & we have seen strategies to implement shared memory demand paging and copy on write fork respectively.

■ ■ ■