Operating Systems–2: CS3523
January 2023

Programming Assignment 6: Paging

Submission Date: 20th April 2022 (Thursday), **9:00 pm**

You can continue on this assignment using the same xv6 repo where you implemented the system call in Assignment-5.

Part-1: Implement demand paging

We discussed demand paging in our lectures where pages are not allocated on process creation but based on demand. The base implementation of xv6 does not implement demand paging and our task in this assignment is to implement demand paging. We would implement a simpler version of demand paging where the read-only code associated with the process is mapped during the process creation, but the memory required for heap and globals is not assigned pages during process creation but allocated on demand. Also, our demand paging would be simpler to assume that sufficient memory is available and we do not need to replace/evict any page during the demand paging process.

Hint-1: modify exec() in exec.c to prevent allocating memory space for dynamic variables but allocate for the read-only code/data. How to do it? Read about elf file format. A short summary is given below.

ELF (Executable and Linkable Format) is used to represent executable and object files. The first few bytes contain an ELF header. The ELF header has a fixed *magic* number at a defined location and is checked by the loader to confirm that it is an ELF file. The ELF file also has a pointer to various *program headers (ph)* which store information about different program segments. ELF header also contains the information about the total number of program headers. Each program header represents a program segment (e.g., code, data, etc.). Each program header contains the following information:

- type: the type of the segment (Loadable segments are of our interest).
- offset: the file offset at which the program segment is in the file (or on the disk).
- filesz: the size of the program segment in the file.
- paddr: the physical address at which the segment should be loaded.
- vaddr: the virtual address at which the segment should be loaded.

- memsz: the size of the program segment in memory (includes read-only and dynamic data).

memsz must be greater than filesz else the ELF file is not valid. If memsz is not equal to filesz, the remaining bytes (memsz - filesz) are initialized to zero by the loader.

You can print the program headers for a given executable file using the following unix command:

```
objdump -p <executable file name>
```

On Linux, type man elf to get full details about the ELF format.

Hint-2: Inside the trap.c file, the current trap function exits with an error. Modify it to check if the trap corresponds to page fault and if yes, then implement a handler to implement demand paging. The handler should check for the faulting address and if it is a valid address in the virtual memory range of the process. If yes, assign a page and map it to the page directory/page table. If it is not a valid page, then generate errors as was happening earlier.
Do make sure that you zero out the physical page before assigning to the process.

Implement a user program (mydemandPage.c) to exercise this condition and see that it works. Your user program should use a large sized global array write/read from this array to check for functioning of demand paging. Also, invoke pgtPrint system call intermittently to check that the page table is expanding as per demand paging. A template for the user program is given below.

```
#include "types.h"
#include "stat.h"
#include "user.h"

#define N 300           //global array size - change to see effect. Try 3000, 5000, 10000
int glob[N];
int main(){
        glob[0]=2;      //initialize with any integer value
        printf (1, "global addr from user space: %x\n", glob);
        for (int i=1;i<N;i++){
                glob[i]=glob[i-1];
                if (i%1000 ==0)
                        pgtPrint();
        }
        printf (1, "Printing final page table:\n");
        pgtPrint();
        printf(1, "Value: %d\n", glob[N-1]);

        exit();
}
```

On executing this program, the following kind of messages should be displayed (N=3000 in this example):

```
global addr from user space: B00
page fault occurred, doing demand paging for address: 0x1000
  pgdir entry num:0, Pgt entry num: 0, Virtual addr: 0, Physical addr: dee2000
  pgdir entry num:0, Pgt entry num: 1, Virtual addr: 1000, Physical addr: dfbc000
  pgdir entry num:0, Pgt entry num: 5, Virtual addr: 5000, Physical addr: dedf000
page fault occurred, doing demand paging for address: 0x2000
  pgdir entry num:0, Pgt entry num: 0, Virtual addr: 0, Physical addr: dee2000
  pgdir entry num:0, Pgt entry num: 1, Virtual addr: 1000, Physical addr: dfbc000
  pgdir entry num:0, Pgt entry num: 2, Virtual addr: 2000, Physical addr: df76000
  pgdir entry num:0, Pgt entry num: 5, Virtual addr: 5000, Physical addr: dedf000
page fault occurred, doing demand paging for address: 0x3000
Printing final page table:
  pgdir entry num:0, Pgt entry num: 0, Virtual addr: 0, Physical addr: dee2000
  pgdir entry num:0, Pgt entry num: 1, Virtual addr: 1000, Physical addr: dfbc000
  pgdir entry num:0, Pgt entry num: 2, Virtual addr: 2000, Physical addr: df76000
  pgdir entry num:0, Pgt entry num: 3, Virtual addr: 3000, Physical addr: dfbf000
  pgdir entry num:0, Pgt entry num: 5, Virtual addr: 5000, Physical addr: dedf000
Value: 2
```

Part-2: Implement copy-on-write

Similar to Part-1, we now add a capability for copy-on-write (COW) optimization into our xv6 kernel updated in Part-1. When fork() creates a child process, it copies the entire memory space of the parent process for the child. Our task is to avoid this copy and do it only when the parent/child wants to update one of these pages.
The updated fork() just creates a pagetable for the child and the PTEs for user memory point to the parent's physical pages. COW fork() marks all the user PTEs in both parent and child as read-only. When either process tries to write one of these COW pages, the system sees a page fault. The page-fault handler now checks and allocates a page of physical memory for the faulting process, copies the original page into the new page, and modifies the relevant PTE in the faulting process to refer to the new page and allows write permissions.

Implement a user program (myCOW.c) to exercise this condition and check its functioning. Your user program should allocate a large sized memory to a process, and then invoke the fork() system call. Also, invoke pgtPrint system call intermittently to check that the page table is getting updated according to the COW. Modify the page table printing function to print the status of Write bit as well.

```
pgdir entry num:0, Pgt entry num: 0, Virtual addr: 0, Physical addr: dee2000, W-bit: 0
pgdir entry num:0, Pgt entry num: 1, Virtual addr: 1000, Physical addr: dfbc000, W-bit: 1
```

Submission Instructions

Submission is to be done at the appropriate link. Just bundle the modified files as per below instructions.

1. Go inside the xv6 directory
2. make clean
3. tar -zcvf xv6.tar.gz * --exclude .git
4. A small report (report.pdf) with max. 2 pages explaining your implementation in brief, observations for various experiments, and your learning from this assignment.
5. Create a zip file containing xv6.tar.gz and report.pdf and upload it. The zip file should follow the name: Assgn6-<RollNo>.zip

Grading Policy

1. Part-1 working: 35%
2. Part-2 working: 40%
3. Report: 25%

Note: We would run plagiarism checks on the submissions and copy cases would be appropriately dealt with. If needed, we might conduct a viva to confirm the same.