OS-II: CS3523
Programming Assignment 5: xv6 Paging

Task-1: Printing the page table entries

Modify the xv6 kernel to implement a new system call named pgtPrint() which will print the page table entries for the current process. Since the total number of page table entries can be very large, the system call should print the entries only if it is valid and the page is allowed access in user mode.

Hints:

1. Use the PTE_P (present bit) to check for valid pages and PTE_U (user) bit to check for user mode access.
2. The pointer to the page table can be obtained using myproc()->pagatable.

The system call should print something of the following format:
PTE No: 0, Virtual page address: 0x0000000000000000, Physical page address: 0x000000000dee0000
PTE No:1, Virtual page address: 0x0000000000001000, Physical page address: 0x000000000de20000
and so on.

Implement a user program (use filename as mypgtPrint.c) to invoke the newly added system call. After a successful implementation, typing mypgtPrint on xv6 shell should print the page table of your user-level program.

Perform the following experiments on this user program and document your observations along with reasoning in the report being submitted.
1. Declare a large size global array (int arrGlobal[10000]) and check if the number of valid entries changes or not.
2. Declare a large size local array (int arrLocal[10000]) within the main function and check if the number of valid entries change or remain the same.
3. Repeat the execution of the user program and check if the number of entries remains the same or not. Also, check if the virtual and physical addresses change or not across multiple executions

Task-2: 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 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 create a large integer array of size 10000 on heap using malloc() within the main function of the user program and check if the number of valid entries change or remain the same. If not, modify your program's logic to force the kernel to provide you more valid page frames for this dynamic array. You may need to invoke pgtPrint system call (implemented in Task-1) intermittently to check that the page table is expanding as per demand paging. A template for the user program with globals 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 address from user space: B00
page fault occurred, doing demand paging for address: 0x1000
PTE No: 0, Virtual address: 0, Physical address: dee2000
PTE No: 1, Virtual address: 1000, Physical address: dfbc000
PTE No: 5, Virtual address: 5000, Physical address: dedf000
page fault occurred, doing demand paging for address: 0x2000
PTE No: 0, Virtual address: 0, Physical address: dee2000
PTE No: 2, Virtual address: 2000, Physical address: df76000
```

Task-3: Implement logic to detect which pages have been accessed and/or dirty

In this task, you extend the code base of Task-2 and add a new feature to xv6 that detects and reports which pages have been accessed and/or modified (dirty) to userspace by inspecting the access and modified bits in the RISC-V page table. The RISC-V hardware page walker marks these bits in the PTE whenever it resolves a TLB miss.

Your job is to implement pgaccess(), a new system call that reports which pages have been accessed and/or dirty. The system call takes three arguments. First, it takes the starting virtual address of the first user page to check. Second, it takes the number of pages to check. Finally, it takes a user address to a buffer to store the results into a bitmask/bitmap (a data structure that uses two bits per page and where the first page corresponds to the least significant bit).

Hints:

- Start by implementing sys_pgaccess() in kernel/sysproc.c.
- You'll need to parse arguments using argaddr() and argint().
- For the output bitmask/bitmap, it's easier to store a temporary buffer in the kernel and copy it to the user (via copyout()) after filling it with the right bits.
- It's okay to set an upper limit on the number of pages that can be scanned.
- walk() in kernel/vm.c is very useful for finding the right PTEs.
- You'll need to define PTE_A, the access bit, in kernel/riscv.h. Consult the RISC-V privileged architecture manual to determine its value.
- Use `PTE_D` to check for dirty (modified) pages
- Be sure to clear PTE_A after checking if it is set. Otherwise, it won't be possible to determine if the page was accessed since the last time pgaccess() was called (i.e., the bit will be set forever).
- vmprint() function (Lab Assignment-3) or pgtPrint() system call (Task-1 of this assignment) may come in handy to debug page tables.

Submission Instruction.
Just bundle the modified files with good documentation as per below instructions.
1. Go inside the xv6 directory
2. make clean
3. tar -zcvf xv6.tar.gz * --exclude .git
4. Create a small report (report.pdf) of explaining
    a. Your understanding of how system call works and what you learnt from this assignment. - 1 page
    b. Your observations along with reasoning for various experiments of different size arrays
    c. **screenshots of outputs**
5. Create a zip file containing xv6.tar.gz and report.pdf and upload it on the google classroom. The zip file should be named as: Assgn5-<RollNo>.zip

Marks Breakup:
1. Task-1: 20%
2. Task-2: 30%
3. Task-3: 30%
4. Report: 20%

Due by April 29th, 11:59 PM. Late submissions attract a penalty of 20% per day!

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.

References:

- https://pdos.csail.mit.edu/6.828/2023/labs/syscall.html
- https://pdos.csail.mit.edu/6.828/2023/labs/pgtbl.html
- https://pdos.csail.mit.edu/6.828/2023/xv6/book-riscv-rev3.pdf