# OS-II: CS3523
# Programming Assignment 5: xv6 Paging

**SURBHI**
**CS22BTECH11057**

**UNDERSTANDING OF HOW SYSTEM CALL WORKS:-**

When a user-level program needs to execute a privileged operation, such as accessing hardware directly, it requests the operating system's assistance by issuing a system call.Upon receiving a system call request, the processor switches from user mode to kernel mode using a mechanism known as a trap or an interrupt, transferring control from the user-space program to the operating system kernel.The system call number, which identifies the specific operation requested by the user-level program, is passed to the kernel. This number serves as an index into a table or data structure called the system call table.The kernel looks up the corresponding handler function associated with the system call number in the system call table.The kernel executes the handler function, which performs the requested privileged operation on behalf of the user-level program. This operation may involve accessing hardware, managing system resources, or performing other privileged tasks.After completing the requested operation, the kernel returns the result of the operation (e.g., success or failure status) to the user-level program.Control is then passed back to the user-level program, which resumes execution from the point where the system call was invoked, continuing its processing as needed.

## TASK-1 Printing the page table entries

To modify the xv6 kernel to implement a new system call `pgtPrint()`, which prints the valid page table entries (PTEs) for a process with user access permissions, you will need to understand several components of the kernel, specifically how system calls are added and how page table entries are managed. Below I'll provide a step-by-step explanation of the modifications and code required to implement this functionality.

**STEPS TO IMPLEMENT “pgtPrint()” system call**

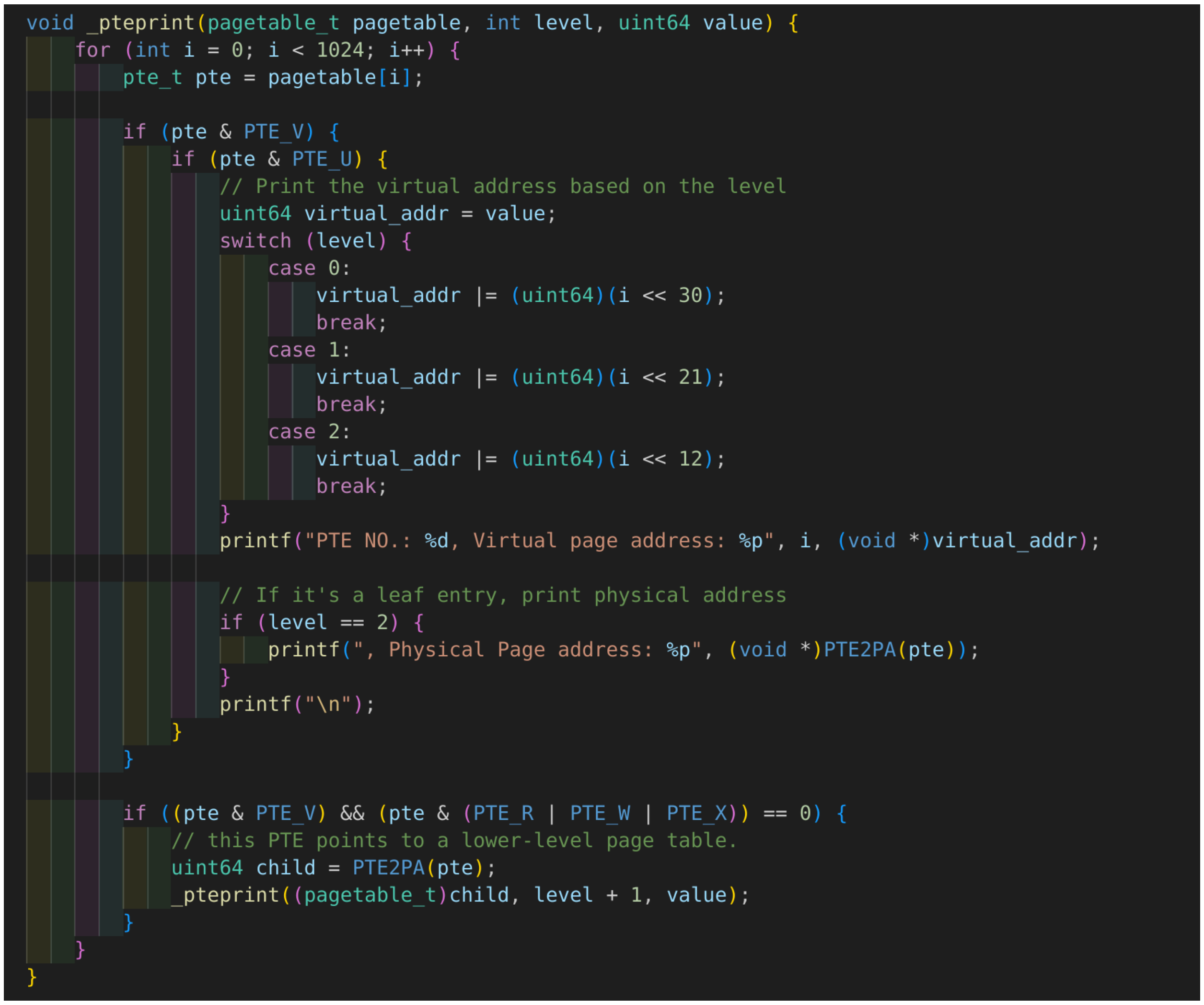
**STEP-1** Add System Call Number in syscall.h as **“#define SYS_pgtPrint 22”**

**STEP-2:** Update System Call Table In ‘syscall.c’ update the system call table to include the new system call. We need to add the prototype of the function in the system call and add entry in the syscalls[ ] array.
In syscall.c we have to add 2 things:-
**- extern uint64 sys_pgtPrint(void);**
**-[SYS_pgtPrint]  sys_pgtPrint,**

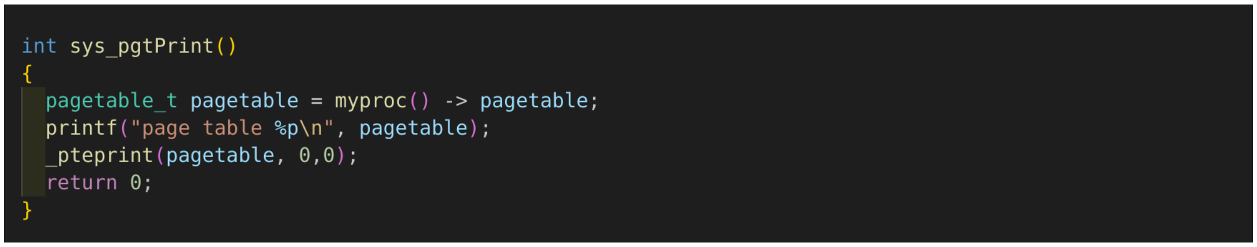
**STEP-3:** Implementing the system call, in sysproc.c.

Explanation of code:-

_pteprint() Function: This is a recursive function used to traverse and print the page table entries. Each entry is checked for validity (PTE_V) and user mode access (PTE_U). If both conditions are met, it prints the page table entry number, virtual page address, and physical page address. pagetable[i] gives the PTE. PTE2PA(pagetable[i]) converts the page table entry to a physical address. sys_pgtPrint() Function: This is the actual system call function that retrieves the current process's page table and prints it by calling _pteprint().



```
void _pteprint(pagetable_t pagetable, int level, uint64 value) {
    for (int i = 0; i < 1024; i++) {
        pte_t pte = pagetable[i];

        if (pte & PTE_V) {
            if (pte & PTE_U) {
                // Print the virtual address based on the level
                uint64 virtual_addr = value;
                switch (level) {
                    case 0:
                        virtual_addr |= (uint64)(i << 30);
                        break;
                    case 1:
                        virtual_addr |= (uint64)(i << 21);
                        break;
                    case 2:
                        virtual_addr |= (uint64)(i << 12);
                        break;
                }
                printf("PTE NO.: %d, Virtual page address: %p", i, (void *)virtual_addr);

                // If it's a leaf entry, print physical address
                if (level == 2) {
                    printf(", Physical Page address: %p", (void *)PTE2PA(pte));
                }
                printf("\n");
            }
        }

        if ((pte & PTE_V) && (pte & (PTE_R | PTE_W | PTE_X)) == 0) {
            // this PTE points to a lower-level page table.
            uint64 child = PTE2PA(pte);
            _pteprint((pagetable_t)child, level + 1, value);
        }
    }
}
```



```c
int sys_pgtPrint()
{
  pagetable_t pagetable = myproc() -> pagetable;
  printf("page table %p\n", pagetable);
  _pteprint(pagetable, 0,0);
  return 0;
}
```

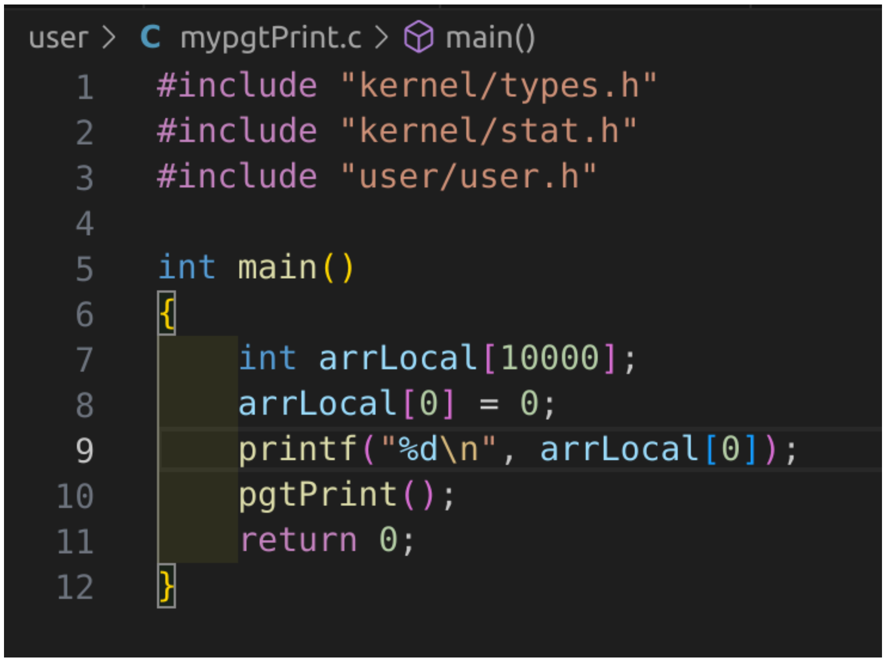
**STEP-4:** Update “usys.pl” file in user: **entry("pgtPrint");**

**STEP-5:** Declare the system call in “user.h”: In the user.h file, declare the function prototype of pgtPrint(). This allows user programs to call this system call as a regular function. Add the following declaration in user.h under the section where other system calls are declared:

**“int pgtPrint(void);”**

**STEP-6:** Create the User Program mypgtPrint.c:-

Create a new file mypgtPrint.c in the user directory of the xv6 source. This program will utilize the pgtPrint() system call to print the page table entries of its own process. Here's how you write the code in a formal and clear manner:

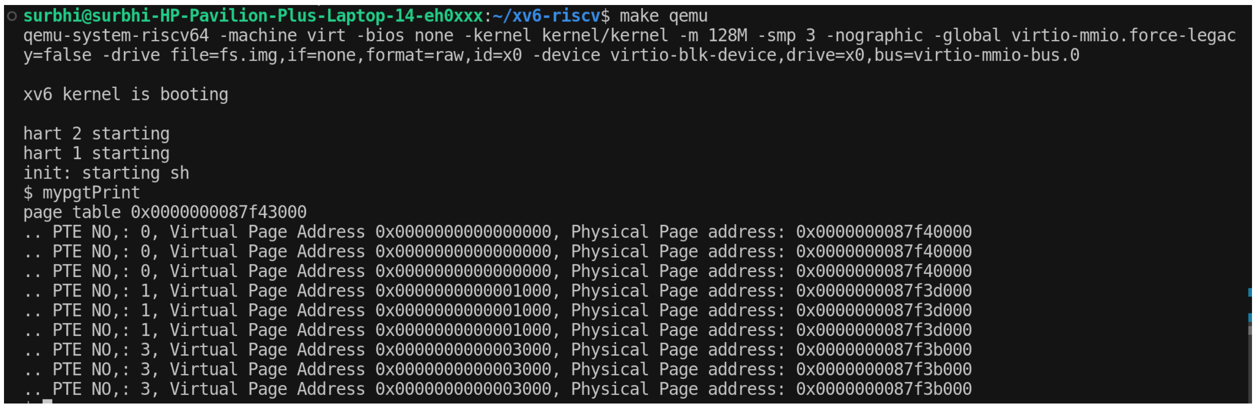


```c
#include "kernel/types.h"
#include "kernel/stat.h"
#include "user/user.h"

int main()
{
    int arrLocal[10000];
    arrLocal[0] = 0;
    printf("%d\n", arrLocal[0]);
    pgtPrint();
    return 0;
}
```

**STEP-7:** Compile and Test the Program**:-** Make sure to include mypgtPrint.c in the Makefile under the UPROGS section to ensure it gets compiled into a user program.

In XV6 kernel we have to run “make qemu” to get the necessary output.

OUTPUT:-



```
surbhi@surbhi-HP-Pavilion-Plus-Laptop-14-eh0xxx:~/xv6-riscv$ make qemu
qemu-system-riscv64 -machine virt -bios none -kernel kernel/kernel -m 128M -smp 3 -nographic -global virtio-mmio.force-legac
y=false -drive file=fs.img,if=none,format=raw,id=x0 -device virtio-blk-device,drive=x0,bus=virtio-mmio-bus.0

xv6 kernel is booting

hart 2 starting
hart 1 starting
init: starting sh
$ mypgtPrint
page table 0x0000000087f43000
.. PTE NO,: 0, Virtual Page Address 0x0000000000000000, Physical Page address: 0x0000000087f40000
.. PTE NO,: 0, Virtual Page Address 0x0000000000000000, Physical Page address: 0x0000000087f40000
.. PTE NO,: 0, Virtual Page Address 0x0000000000000000, Physical Page address: 0x0000000087f40000
.. PTE NO,: 1, Virtual Page Address 0x0000000000001000, Physical Page address: 0x0000000087f3d000
.. PTE NO,: 1, Virtual Page Address 0x0000000000001000, Physical Page address: 0x0000000087f3d000
.. PTE NO,: 1, Virtual Page Address 0x0000000000001000, Physical Page address: 0x0000000087f3d000
.. PTE NO,: 3, Virtual Page Address 0x0000000000003000, Physical Page address: 0x0000000087f3b000
.. PTE NO,: 3, Virtual Page Address 0x0000000000003000, Physical Page address: 0x0000000087f3b000
.. PTE NO,: 3, Virtual Page Address 0x0000000000003000, Physical Page address: 0x0000000087f3b000
```

***CONCLUSION OF TASK-1***

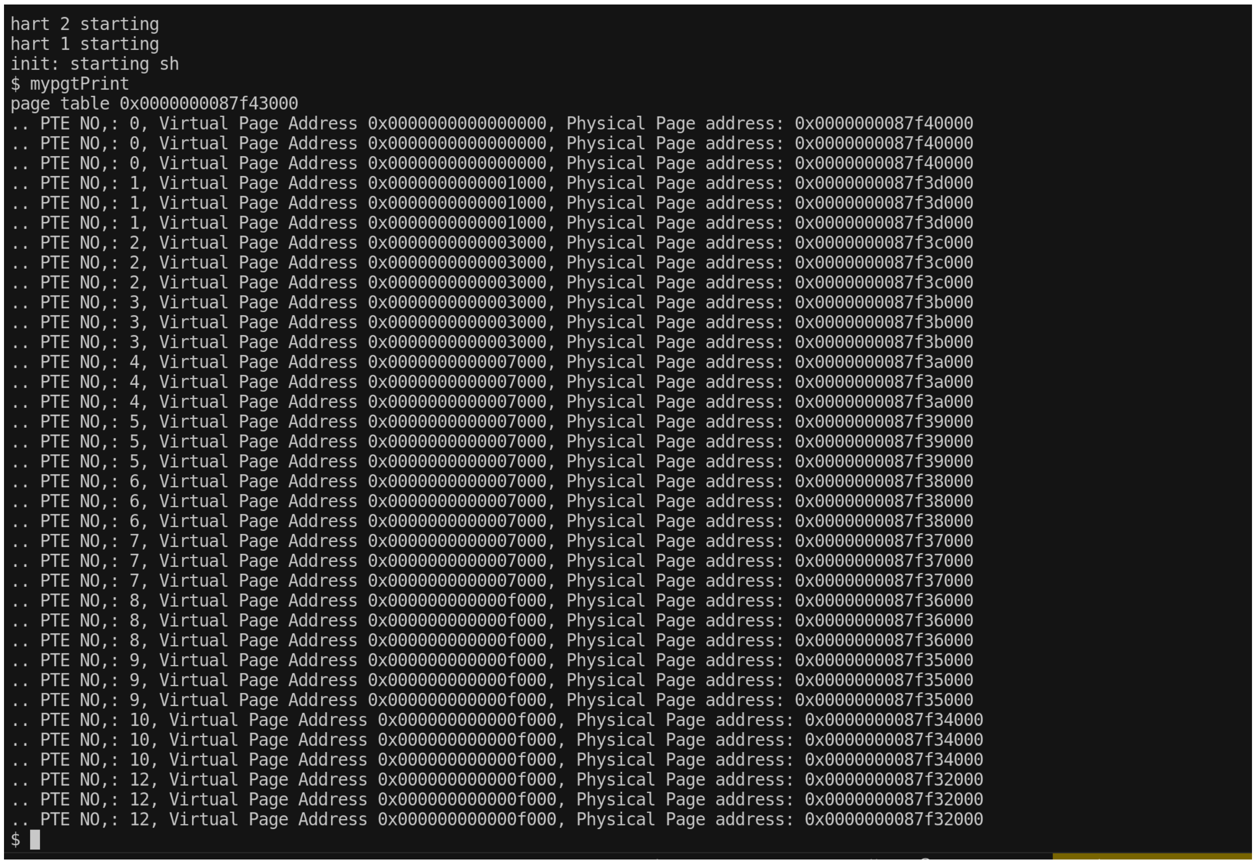
**Page Table Structure and Entry Management:** Understanding how entries are structured within the page table and how they map virtual addresses to physical addresses is crucial for any system-level programmer. This task provided a clear demonstration of this mapping.

System Call Implementation: Adding a system call in xv6 involves multiple steps—defining the call, updating assembly stubs, linking user-space libraries, and handling kernel-side logic. This workflow is typical in many operating systems, though the specifics can vary.

**Kernel and User-space Interaction:** The process helped illustrate the separation and interaction between user space and kernel space, especially how system calls are invoked and handled.

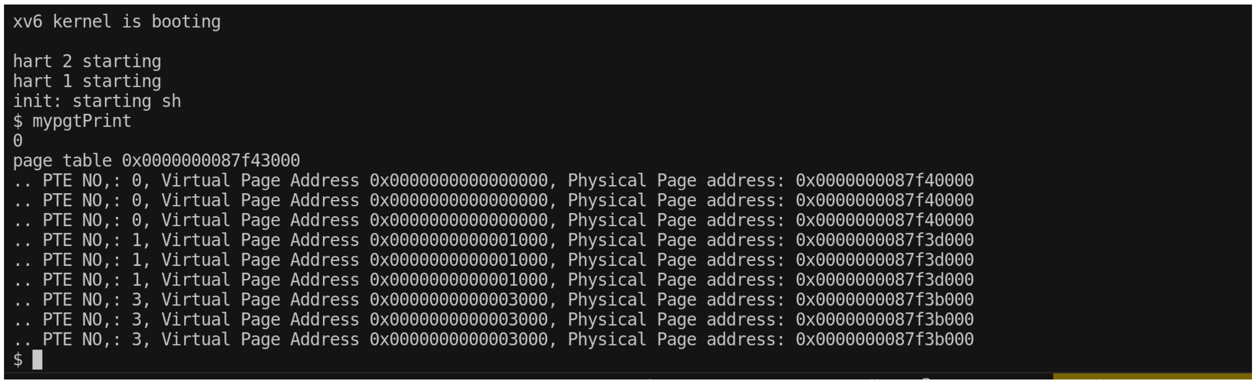
**Observations and Reasoning from Experiments with Different Size Arrays**

**Global Arrays:** Declaring a large size global array (int arrGlobal[10000];) increases the number of valid page table entries. This is because global variables are stored in the data segment, which is allocated space when the process is created. The size of this segment will increase with larger global arrays, thereby necessitating more page table entries to map the increased memory.



```
hart 2 starting
hart 1 starting
init: starting sh
$ mypgtPrint
page table 0x0000000087f43000
.. PTE NO,: 0, Virtual Page Address 0x0000000000000000, Physical Page address: 0x0000000087f40000
.. PTE NO,: 0, Virtual Page Address 0x0000000000000000, Physical Page address: 0x0000000087f40000
.. PTE NO,: 0, Virtual Page Address 0x0000000000000000, Physical Page address: 0x0000000087f40000
.. PTE NO,: 1, Virtual Page Address 0x0000000000001000, Physical Page address: 0x0000000087f3d000
.. PTE NO,: 1, Virtual Page Address 0x0000000000001000, Physical Page address: 0x0000000087f3d000
.. PTE NO,: 1, Virtual Page Address 0x0000000000001000, Physical Page address: 0x0000000087f3d000
.. PTE NO,: 2, Virtual Page Address 0x0000000000003000, Physical Page address: 0x0000000087f3c000
.. PTE NO,: 2, Virtual Page Address 0x0000000000003000, Physical Page address: 0x0000000087f3c000
.. PTE NO,: 2, Virtual Page Address 0x0000000000003000, Physical Page address: 0x0000000087f3c000
.. PTE NO,: 3, Virtual Page Address 0x0000000000003000, Physical Page address: 0x0000000087f3b000
.. PTE NO,: 3, Virtual Page Address 0x0000000000003000, Physical Page address: 0x0000000087f3b000
.. PTE NO,: 3, Virtual Page Address 0x0000000000003000, Physical Page address: 0x0000000087f3b000
.. PTE NO,: 4, Virtual Page Address 0x0000000000007000, Physical Page address: 0x0000000087f3a000
.. PTE NO,: 4, Virtual Page Address 0x0000000000007000, Physical Page address: 0x0000000087f3a000
.. PTE NO,: 4, Virtual Page Address 0x0000000000007000, Physical Page address: 0x0000000087f3a000
.. PTE NO,: 5, Virtual Page Address 0x0000000000007000, Physical Page address: 0x0000000087f39000
.. PTE NO,: 5, Virtual Page Address 0x0000000000007000, Physical Page address: 0x0000000087f39000
.. PTE NO,: 5, Virtual Page Address 0x0000000000007000, Physical Page address: 0x0000000087f39000
.. PTE NO,: 6, Virtual Page Address 0x0000000000007000, Physical Page address: 0x0000000087f38000
.. PTE NO,: 6, Virtual Page Address 0x0000000000007000, Physical Page address: 0x0000000087f38000
.. PTE NO,: 6, Virtual Page Address 0x0000000000007000, Physical Page address: 0x0000000087f38000
.. PTE NO,: 7, Virtual Page Address 0x0000000000007000, Physical Page address: 0x0000000087f37000
.. PTE NO,: 7, Virtual Page Address 0x0000000000007000, Physical Page address: 0x0000000087f37000
.. PTE NO,: 7, Virtual Page Address 0x0000000000007000, Physical Page address: 0x0000000087f37000
.. PTE NO,: 8, Virtual Page Address 0x000000000000f000, Physical Page address: 0x0000000087f36000
.. PTE NO,: 8, Virtual Page Address 0x000000000000f000, Physical Page address: 0x0000000087f36000
.. PTE NO,: 8, Virtual Page Address 0x000000000000f000, Physical Page address: 0x0000000087f36000
.. PTE NO,: 9, Virtual Page Address 0x000000000000f000, Physical Page address: 0x0000000087f35000
.. PTE NO,: 9, Virtual Page Address 0x000000000000f000, Physical Page address: 0x0000000087f35000
.. PTE NO,: 9, Virtual Page Address 0x000000000000f000, Physical Page address: 0x0000000087f35000
.. PTE NO,: 10, Virtual Page Address 0x000000000000f000, Physical Page address: 0x0000000087f34000
.. PTE NO,: 10, Virtual Page Address 0x000000000000f000, Physical Page address: 0x0000000087f34000
.. PTE NO,: 10, Virtual Page Address 0x000000000000f000, Physical Page address: 0x0000000087f34000
.. PTE NO,: 12, Virtual Page Address 0x000000000000f000, Physical Page address: 0x0000000087f32000
.. PTE NO,: 12, Virtual Page Address 0x000000000000f000, Physical Page address: 0x0000000087f32000
.. PTE NO,: 12, Virtual Page Address 0x000000000000f000, Physical Page address: 0x0000000087f32000
$
```

**Local Arrays**: Large local arrays (int arrLocal[10000];) are allocated on the stack. The stack's behavior is dynamic, and its size can change during runtime as functions push local data onto it. However, unless the stack size reaches a new page boundary, the number of page table entries might not change significantly because existing pages can accommodate the new local variables.



```
xv6 kernel is booting

hart 2 starting
hart 1 starting
init: starting sh
$ mypgtPrint
0
page table 0x0000000087f43000
.. PTE NO,: 0, Virtual Page Address 0x0000000000000000, Physical Page address: 0x0000000087f40000
.. PTE NO,: 0, Virtual Page Address 0x0000000000000000, Physical Page address: 0x0000000087f40000
.. PTE NO,: 0, Virtual Page Address 0x0000000000000000, Physical Page address: 0x0000000087f40000
.. PTE NO,: 1, Virtual Page Address 0x0000000000001000, Physical Page address: 0x0000000087f3d000
.. PTE NO,: 1, Virtual Page Address 0x0000000000001000, Physical Page address: 0x0000000087f3d000
.. PTE NO,: 1, Virtual Page Address 0x0000000000001000, Physical Page address: 0x0000000087f3d000
.. PTE NO,: 3, Virtual Page Address 0x0000000000003000, Physical Page address: 0x0000000087f3b000
.. PTE NO,: 3, Virtual Page Address 0x0000000000003000, Physical Page address: 0x0000000087f3b000
.. PTE NO,: 3, Virtual Page Address 0x0000000000003000, Physical Page address: 0x0000000087f3b000
$
```

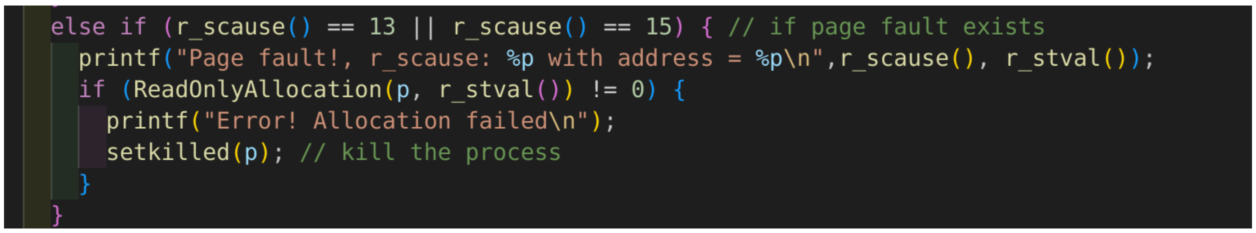
**Repeated Execution:** On repeated execution of the program, the virtual and physical addresses might change due to mechanisms like Address Space Layout Randomization (ASLR), which modern operating systems use to prevent certain types of security attacks. Despite these address changes, the number of page table entries should remain fairly consistent unless there is a change in memory allocation behavior or program structure.

## TASK-2 Implement demand paging

In this task we have to implement demand paging in the xv6 operating system. Demand paging is a memory management scheme where memory pages are loaded into RAM only as they are needed, not at process creation.

**STEP-1: Modify "trap.c" in void user trap(void),**

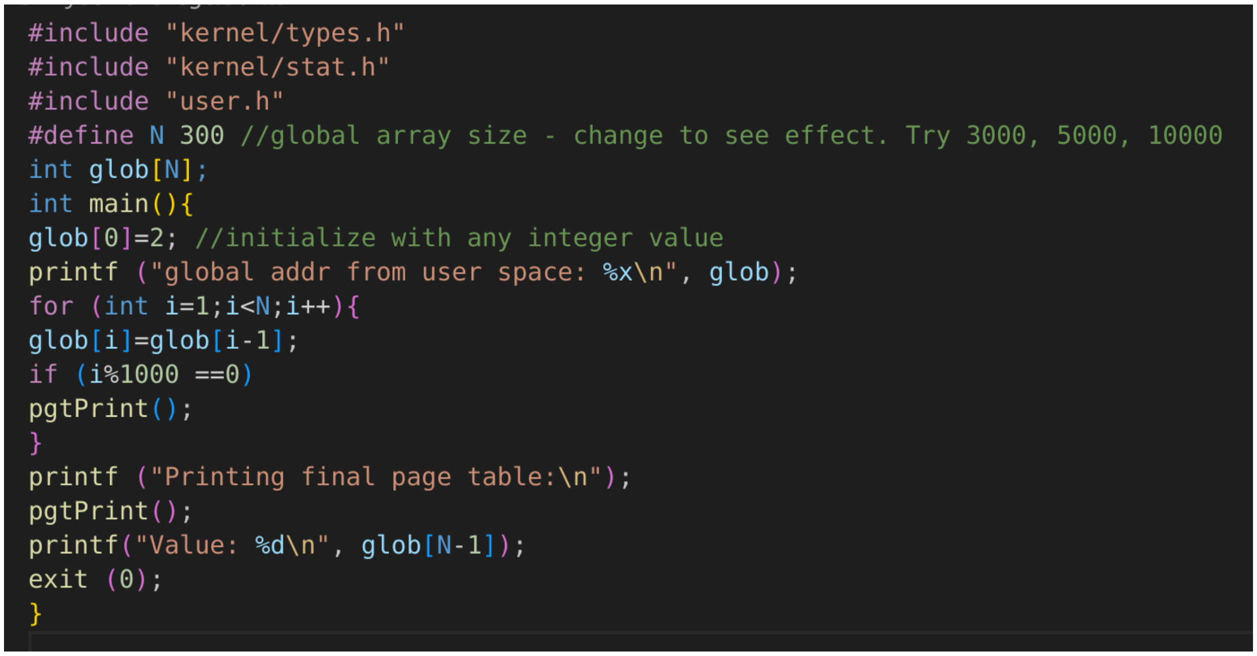
we have to edit else if part to get the correct allocation,



```
else if (r_scause() == 13 || r_scause() == 15) { // if page fault exists
  printf("Page fault!, r_scause: %p with address = %p\n",r_scause(), r_stval());
  if (ReadOnlyAllocation(p, r_stval()) != 0) {
    printf("Error! Allocation failed\n");
    setkilled(p); // kill the process
  }
}
```

Code explanation:- specifically checking for certain types (13 and 15) which typically indicate a missing page. When such a fault occurs, it logs the fault details (cause and the faulting address). It then attempts to handle the fault by allocating the required page lazily (on-demand) through the ReadOnlyAllocation function. If this allocation fails, it logs an error message and terminates the affected process by calling setkilled(p);

**STEP-2: Create a user program “mydemandPage.c”**



```
#include "kernel/types.h"
#include "kernel/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 ("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 ("Printing final page table:\n");
pgtPrint();
printf("Value: %d\n", glob[N-1]);
exit (0);
}
```

Code explanation:-

#define N 300 defines a preprocessor macro N that specifies the size of the global array glob. The comment suggests varying this size to test how the system handles larger arrays.

int glob[N]; declares an array glob of integers with size N. This array is global and will be used to trigger demand paging.

glob[0] = 2; initializes the first element of the global array with the value 2.

printf ("global addr from user space: %x\n", glob); prints the memory address of the global array, which helps in understanding where it is located in the virtual address space.

The for loop iterates from 1 to N and each element of glob is set to the value of its preceding element. This operation accesses multiple parts of the array and is likely to trigger demand paging if the array is large enough:

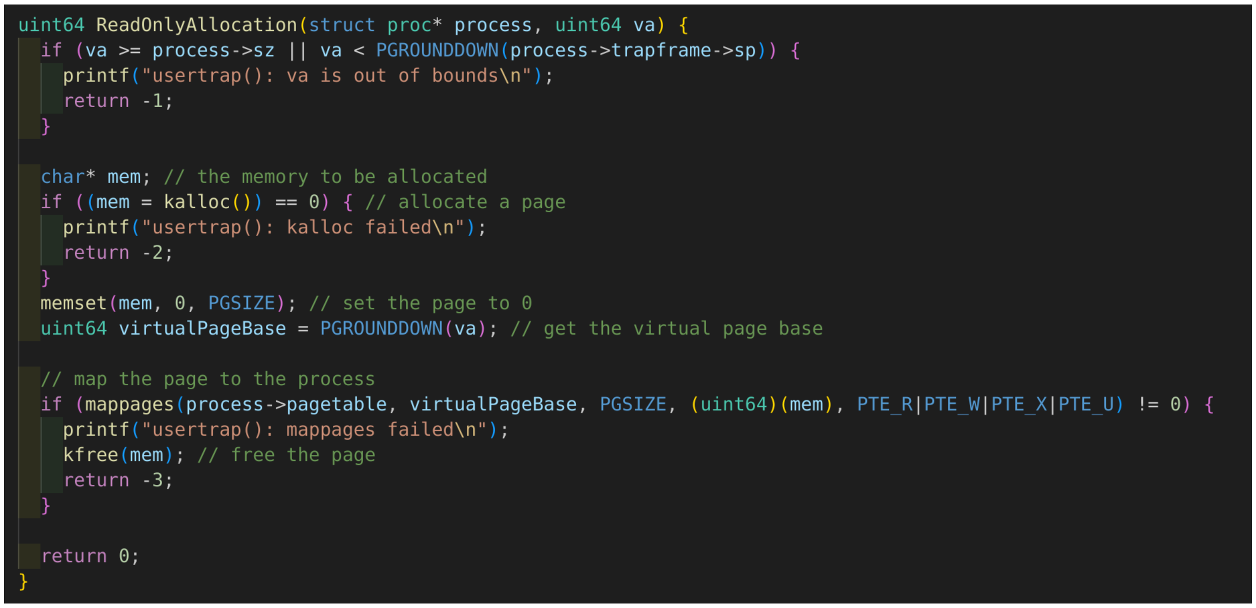
if (i % 1000 == 0) pgtPrint(); calls a function pgtPrint() every 1000 iterations. This function is assumed to print the current state of the page table, which helps in observing how virtual memory pages are being allocated on demand.

After the loop, it prints the state of the page table again to show the final mappings.

printf("Value: %d\n", glob[N-1]); prints the last element of the array to verify the integrity of data after potential page faults and demand paging.

exit(0); terminates the program normally.

**STEP-3 We have to modify and write the lazy allocation code in “vm.c”**



```c
uint64 ReadOnlyAllocation(struct proc* process, uint64 va) {
  if (va >= process->sz || va < PGROUNDDOWN(process->trapframe->sp)) {
    printf("usertrap(): va is out of bounds\n");
    return -1;
  }

  char* mem; // the memory to be allocated
  if ((mem = kalloc()) == 0) { // allocate a page
    printf("usertrap(): kalloc failed\n");
    return -2;
  }
  memset(mem, 0, PGSIZE); // set the page to 0
  uint64 virtualPageBase = PGROUNDDOWN(va); // get the virtual page base

  // map the page to the process
  if (mappages(process->pagetable, virtualPageBase, PGSIZE, (uint64)(mem), PTE_R|PTE_W|PTE_X|PTE_U) != 0) {
    printf("usertrap(): mappages failed\n");
    kfree(mem); // free the page
    return -3;
  }

  return 0;
}
```

**Code explanation:-**

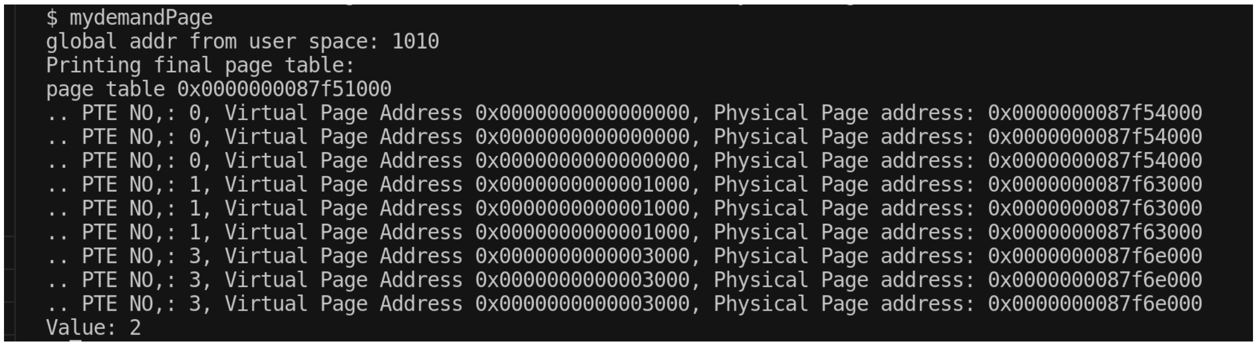
The function ReadOnlyAllocation in xv6 OS handles read only memory allocation for a given process. It validates the virtual address, allocates a physical memory page, initializes it, and maps it to the process's page table.Check if the virtual address is within the valid range of the process's address space. If not, print an error message and return a negative value.Allocate a new physical memory page using kalloc(). If allocation fails, print an error message and return a negative value.Initialize the allocated page by setting all its contents to zero using memset().Determine the base virtual address of the page by rounding down the given virtual address to the nearest page boundary.Map the allocated physical page to the process's page table using mappages(), ensuring it is accessible for reading, writing, and executing.If mapping fails, print an error message, free the allocated memory page using kfree(), and return a negative value.Otherwise, return 0 to indicate successful lazy allocation of the memory page.

**STEP-4 Compile and Test the Program**

Make sure to include mydemandPage.c in the Makefile under the UPROGS section to ensure it gets compiled into a user program.

In XV6 kernel we have to run “make qemu” to get the necessary output.

OUTPUT:-



```
$ mydemandPage
global addr from user space: 1010
Printing final page table:
page table 0x0000000087f51000
.. PTE NO,: 0, Virtual Page Address 0x0000000000000000, Physical Page address: 0x0000000087f54000
.. PTE NO,: 0, Virtual Page Address 0x0000000000000000, Physical Page address: 0x0000000087f54000
.. PTE NO,: 0, Virtual Page Address 0x0000000000000000, Physical Page address: 0x0000000087f54000
.. PTE NO,: 1, Virtual Page Address 0x0000000000001000, Physical Page address: 0x0000000087f63000
.. PTE NO,: 1, Virtual Page Address 0x0000000000001000, Physical Page address: 0x0000000087f63000
.. PTE NO,: 1, Virtual Page Address 0x0000000000001000, Physical Page address: 0x0000000087f63000
.. PTE NO,: 3, Virtual Page Address 0x0000000000003000, Physical Page address: 0x0000000087f6e000
.. PTE NO,: 3, Virtual Page Address 0x0000000000003000, Physical Page address: 0x0000000087f6e000
.. PTE NO,: 3, Virtual Page Address 0x0000000000003000, Physical Page address: 0x0000000087f6e000
Value: 2
```

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

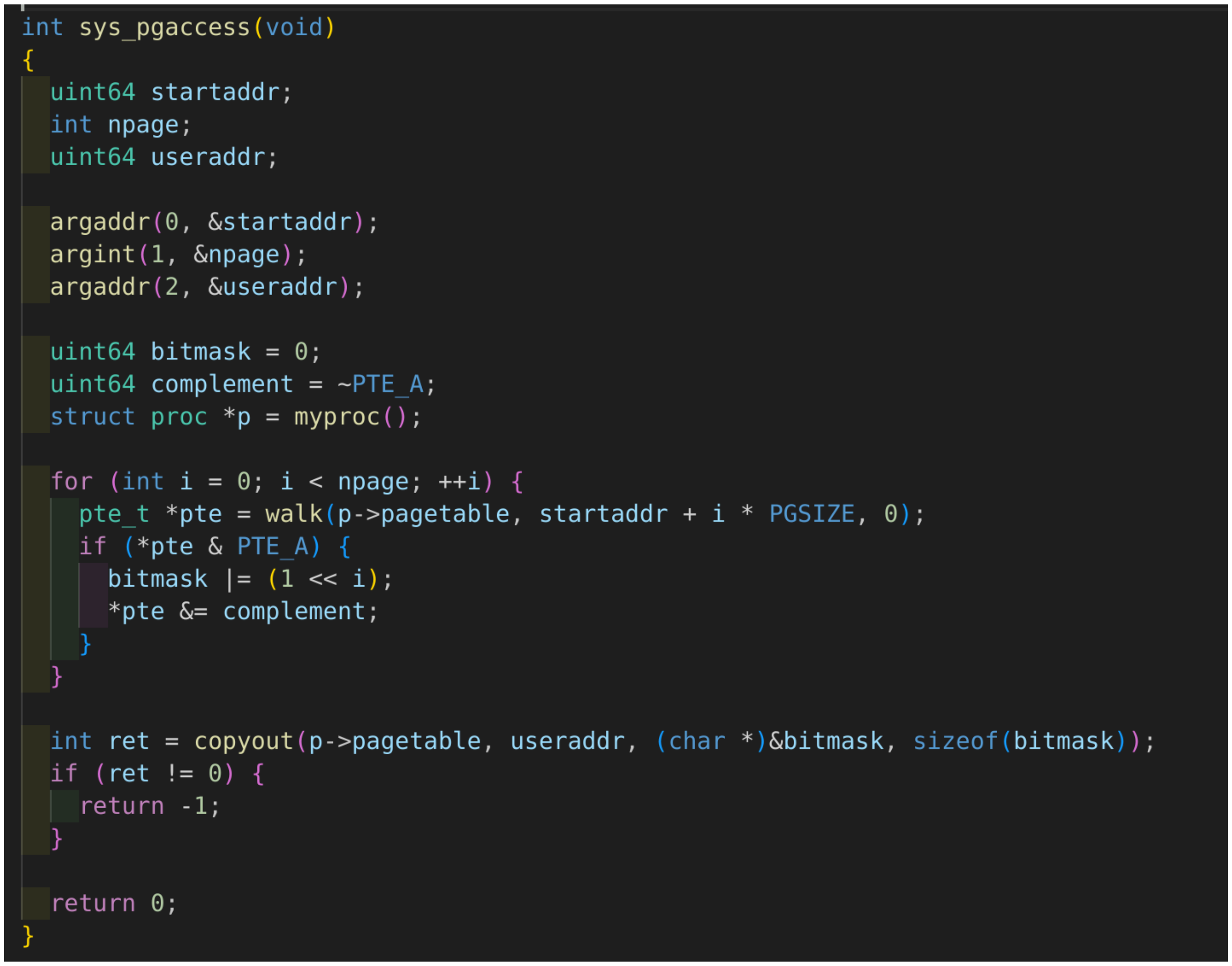
In this task we have to extend the functionality of the xv6 operating system by implementing a new system call called pgaccess(). This system call will allow user space programs to detect and report which pages have been accessed and/or modified (dirty).

**STEPS TO IMPLEMENT “pgaccess” system call**

**STEP-1** Add **“#define SYS_pgaccess 23”** in syscall.h

**STEP-2** Add **“extern uint64 sys_pgaccess(void);”** in prototypes function and **“[SYS_pgaccess] sys_pgaccess”** in array mapping system call in syscall.c file.

**STEP-3** Implement system call in sysproc.c for pg_access

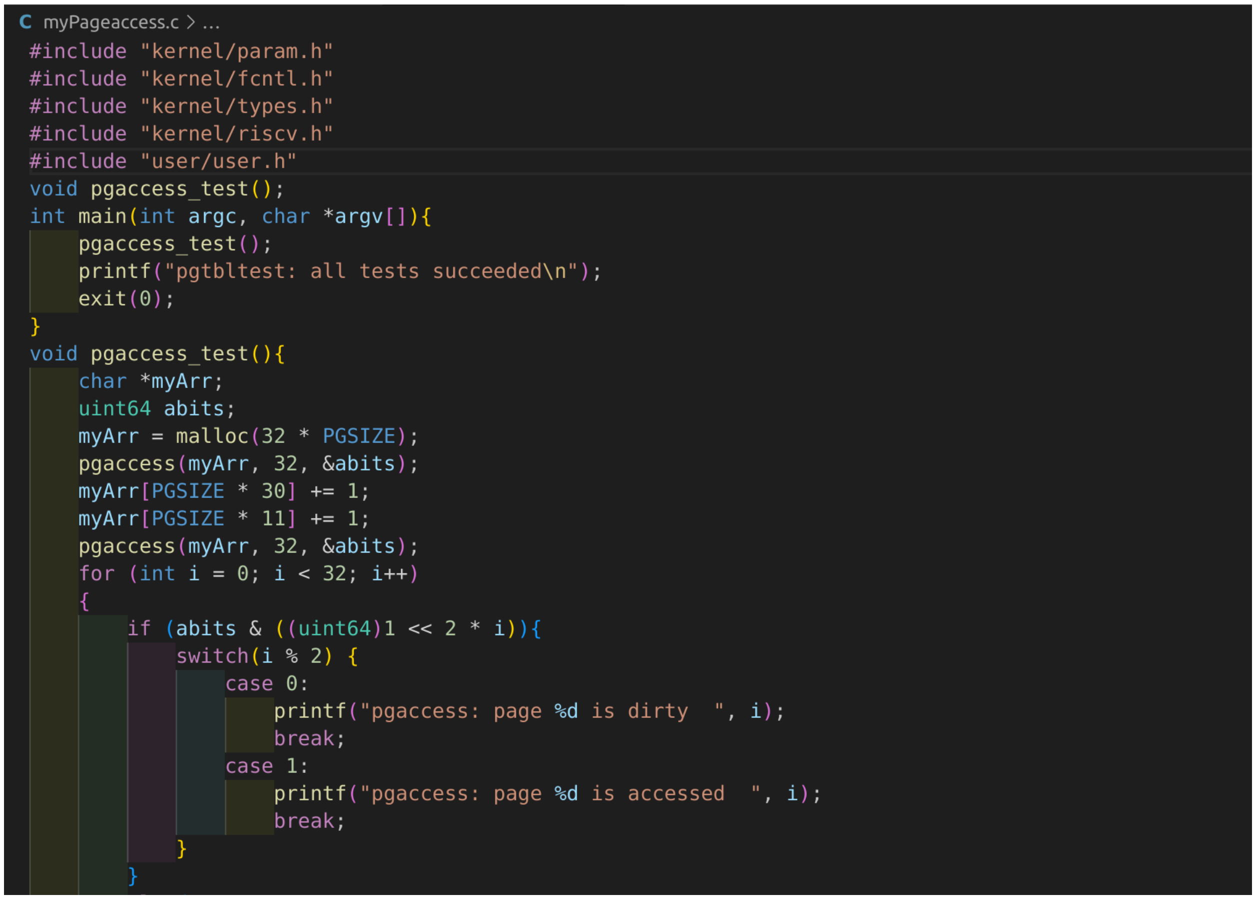


```c
int sys_pgaccess(void)
{
  uint64 startaddr;
  int npage;
  uint64 useraddr;

  argaddr(0, &startaddr);
  argint(1, &npage);
  argaddr(2, &useraddr);

  uint64 bitmask = 0;
  uint64 complement = ~PTE_A;
  struct proc *p = myproc();

  for (int i = 0; i < npage; ++i) {
    pte_t *pte = walk(p->pagetable, startaddr + i * PGSIZE, 0);
    if (*pte & PTE_A) {
      bitmask |= (1 << i);
      *pte &= complement;
    }
  }

  int ret = copyout(p->pagetable, useraddr, (char *)&bitmask, sizeof(bitmask));
  if (ret != 0) {
    return -1;
  }

  return 0;
}
```

**STEP-4** Add “**entry(“pgaccess”)**” in usys.pl file.

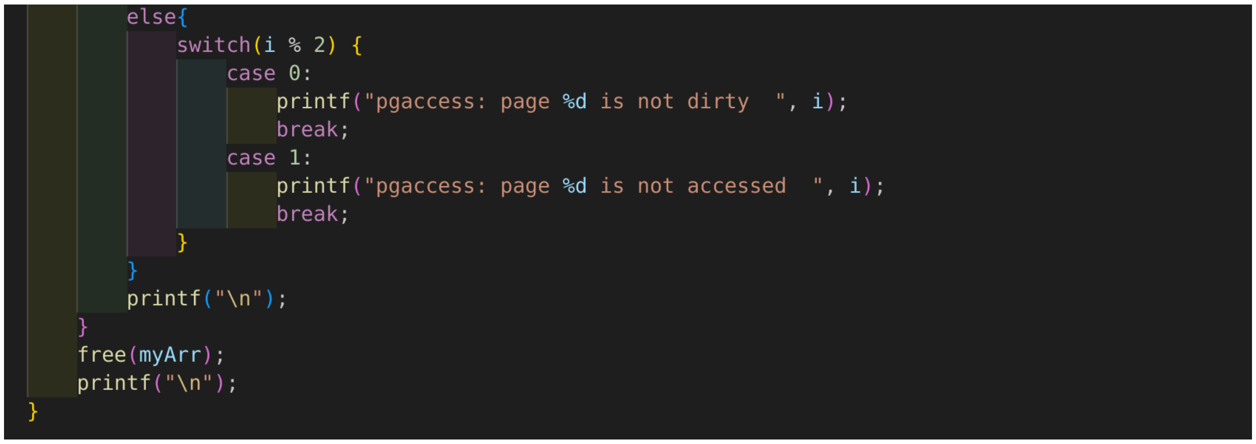
**STEP-5** Add “**int pgaccess(void* startaddr, int npages, void* useraddr);**” in user.h file.

**STEP-6** Create a user program named “myPageaccess.c”.



myPageaccess.c > ...

```c
#include "kernel/param.h"
#include "kernel/fcntl.h"
#include "kernel/types.h"
#include "kernel/riscv.h"
#include "user/user.h"
void pgaccess_test();
int main(int argc, char *argv[]){
    pgaccess_test();
    printf("pgtbltest: all tests succeeded\n");
    exit(0);
}
void pgaccess_test(){
    char *myArr;
    uint64 abits;
    myArr = malloc(32 * PGSIZE);
    pgaccess(myArr, 32, &abits);
    myArr[PGSIZE * 30] += 1;
    myArr[PGSIZE * 11] += 1;
    pgaccess(myArr, 32, &abits);
    for (int i = 0; i < 32; i++)
    {
        if (abits & ((uint64)1 << 2 * i)){
            switch(i % 2) {
                case 0:
                    printf("pgaccess: page %d is dirty  ", i);
                    break;
                case 1:
                    printf("pgaccess: page %d is accessed  ", i);
                    break;
            }
        }
```



```c
        else{
            switch(i % 2) {
                case 0:
                    printf("pgaccess: page %d is not dirty  ", i);
                    break;
                case 1:
                    printf("pgaccess: page %d is not accessed  ", i);
                    break;
            }
        }
        printf("\n");
    }
    free(myArr);
    printf("\n");
}
```

Code explanation:-

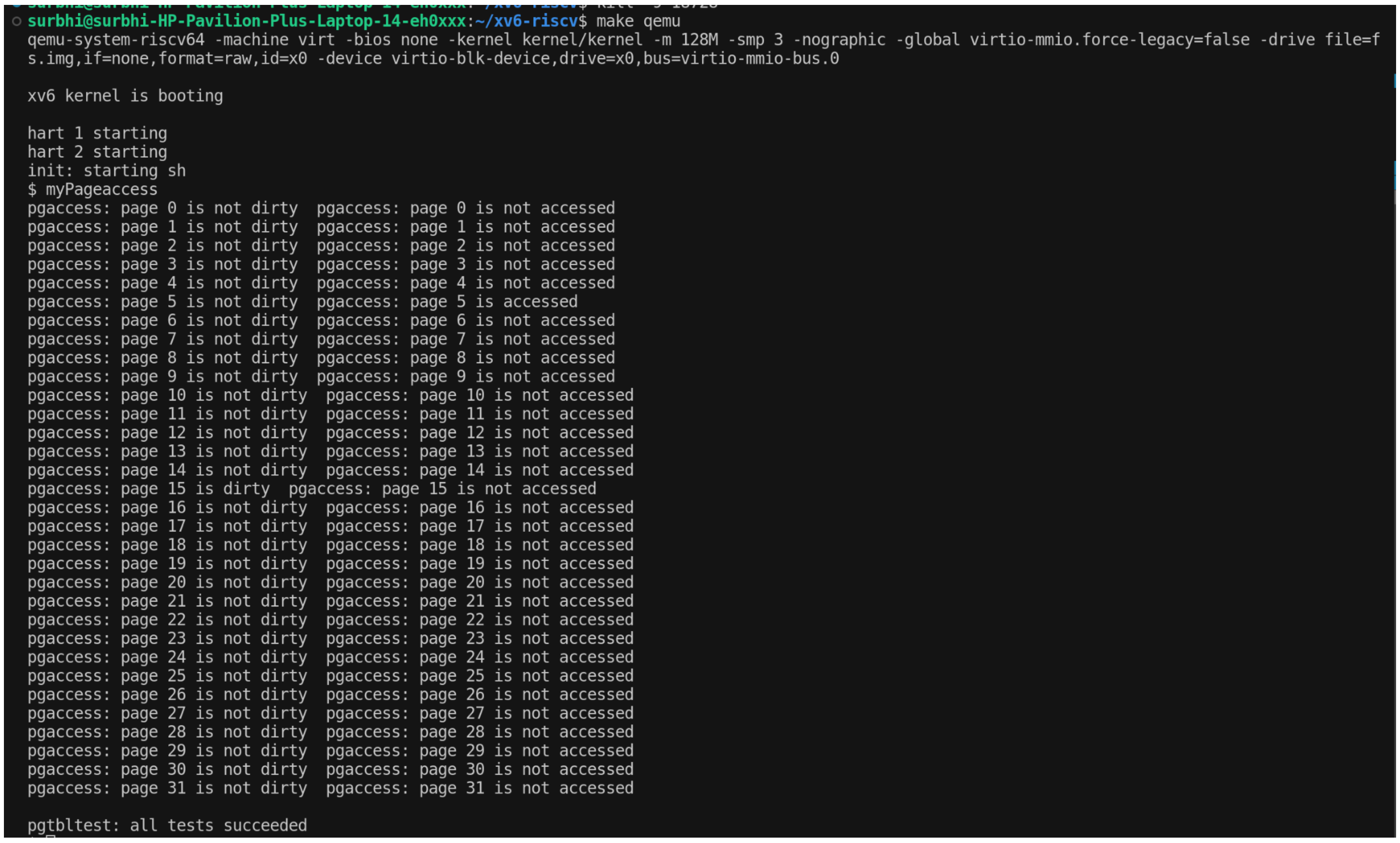
Inside pgaccess_test, memory is dynamically allocated for a character array named myArr with a size of 32 pages, where PGSIZE seems to be the size of a page in the system. The pgaccess function is then called to check the access and dirty status of each page in myArr. After that, the code increments the value at the 30th and 11th page of myArr, simulating write access to these pages. Then, pgaccess is called again to check the status of each page. Finally, a loop iterates through each page, printing whether each page is dirty (has been written to) or accessed (has been read from) based on the values stored in the a bits variable, which seems to represent a bit field indicating the status of each page.

**STEP-7: Compile and Test the Program**

Make sure to include mypgtPrint.c in the Makefile under the UPROGS section to ensure it gets compiled into a user program.

In XV6 kernel we have to run "make qemu" to get the necessary output.

OUTPUT:-



```
surbhi@surbhi-HP-Pavilion-Plus-Laptop-14-eh0xxx:~/xv6-riscv$ make qemu
qemu-system-riscv64 -machine virt -bios none -kernel kernel/kernel -m 128M -smp 3 -nographic -global virtio-mmio.force-legacy=false -drive file=f
s.img,if=none,format=raw,id=x0 -device virtio-blk-device,drive=x0,bus=virtio-mmio-bus.0

xv6 kernel is booting

hart 1 starting
hart 2 starting
init: starting sh
$ myPageaccess
pgaccess: page 0 is not dirty  pgaccess: page 0 is not accessed
pgaccess: page 1 is not dirty  pgaccess: page 1 is not accessed
pgaccess: page 2 is not dirty  pgaccess: page 2 is not accessed
pgaccess: page 3 is not dirty  pgaccess: page 3 is not accessed
pgaccess: page 4 is not dirty  pgaccess: page 4 is not accessed
pgaccess: page 5 is not dirty  pgaccess: page 5 is accessed
pgaccess: page 6 is not dirty  pgaccess: page 6 is not accessed
pgaccess: page 7 is not dirty  pgaccess: page 7 is not accessed
pgaccess: page 8 is not dirty  pgaccess: page 8 is not accessed
pgaccess: page 9 is not dirty  pgaccess: page 9 is not accessed
pgaccess: page 10 is not dirty  pgaccess: page 10 is not accessed
pgaccess: page 11 is not dirty  pgaccess: page 11 is not accessed
pgaccess: page 12 is not dirty  pgaccess: page 12 is not accessed
pgaccess: page 13 is not dirty  pgaccess: page 13 is not accessed
pgaccess: page 14 is not dirty  pgaccess: page 14 is not accessed
pgaccess: page 15 is dirty  pgaccess: page 15 is not accessed
pgaccess: page 16 is not dirty  pgaccess: page 16 is not accessed
pgaccess: page 17 is not dirty  pgaccess: page 17 is not accessed
pgaccess: page 18 is not dirty  pgaccess: page 18 is not accessed
pgaccess: page 19 is not dirty  pgaccess: page 19 is not accessed
pgaccess: page 20 is not dirty  pgaccess: page 20 is not accessed
pgaccess: page 21 is not dirty  pgaccess: page 21 is not accessed
pgaccess: page 22 is not dirty  pgaccess: page 22 is not accessed
pgaccess: page 23 is not dirty  pgaccess: page 23 is not accessed
pgaccess: page 24 is not dirty  pgaccess: page 24 is not accessed
pgaccess: page 25 is not dirty  pgaccess: page 25 is not accessed
pgaccess: page 26 is not dirty  pgaccess: page 26 is not accessed
pgaccess: page 27 is not dirty  pgaccess: page 27 is not accessed
pgaccess: page 28 is not dirty  pgaccess: page 28 is not accessed
pgaccess: page 29 is not dirty  pgaccess: page 29 is not accessed
pgaccess: page 30 is not dirty  pgaccess: page 30 is not accessed
pgaccess: page 31 is not dirty  pgaccess: page 31 is not accessed

pgtbltest: all tests succeeded
```