CSC10007 – OPERATING SYSTEM
PROJECT 3– PAGE TABLE

No.	Name	Student ID	Contribute
1.	Le Quang Khai	22120148	50%
2.	Nguyen Quang Thang	22120333	50%

No.	Exercise	Person in charge
1.	Inspect a user-process page table	Nguyen Quang Thang
2.	Speed up system calls	Nguyen Quang Thang
3.	Print a page table	Nguyen Quang Thang
4.	Detect which pages have been accessed	Le Quang Khai

Table of Contents

1. Overview

Switch from branch `syscall` to branch `pgtbl`.

This lab will explore page tables and modify them to implement common OS features.

Important files:

- `kernel/memlayout.h`, which captures the layout of memory.
- `kernel/vm.c`, which contains most virtual memory (VM) code.
- `kernel/kalloc.c`, which contains code for allocating and freeing physical memory.

2. Inspect a user-process page table (easy)

For every page table entry in the print_pgtbl output, explain what it logically contains and what its permission bits are. Figure 3.4 in the xv6 book might be helpful, although *note that the figure might have a slightly different set of pages than process that's being inspected here. Note that xv6 doesn't place the virtual pages consecutively in physical memory.*

Page table has following output format:

`va 0xFFFFF000 pte 0x20001C4B pa 0x80007000 perm 0x4B`

- va: virtual address. 39 bits are used, 27 of which are for indexing page table entries, and 12 of which are for the offset.
- pte: page table entries. There are 2^{27} rows, corresponding to 27 bits of the virtual address. Each row uses 44 bits for mapping to the physical address, along with some bits for flags.
- pa: physical address. There are 56 bits for the address, 44 of which are for the page address (chunk), and 12 of which are for the offset, copied from the virtual address.
- perm: permission.

3. Speed up system calls (easy)

When each process is created, map one read-only page at USYSCALL (a virtual address defined in memlayout.h). At the start of this page, store a struct usyscall (also defined in memlayout.h), and initialize it to store the PID of the current process. For this lab, ugetpid() has been provided on the userspace side and will automatically use the USYSCALL mapping. You will receive full credit for this part of the lab if the ugetpid test case passes when running pgtbltest.

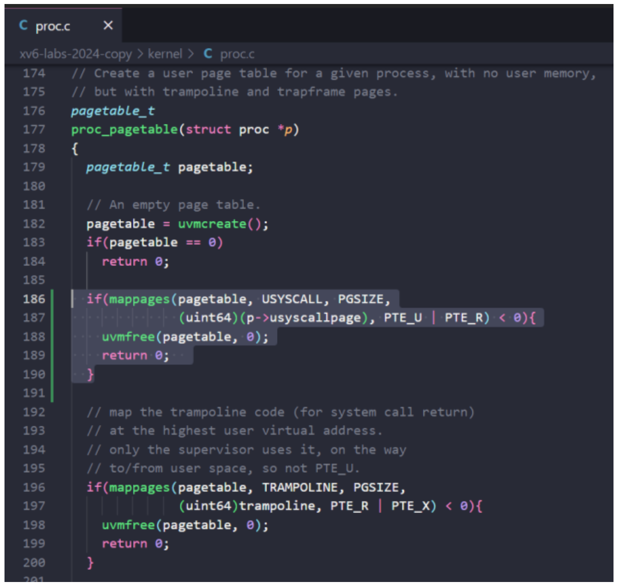
First, map `struct usyscall` to address `USYSCALL` (a virtual address defined in `memlayout.h`) when the process is created. In `struct proc` (defined in `kernel/proc.h`), declare variable of type `struct usyscall*`

proc.h

xv6-labs-2024-copy > kernel > proc.h

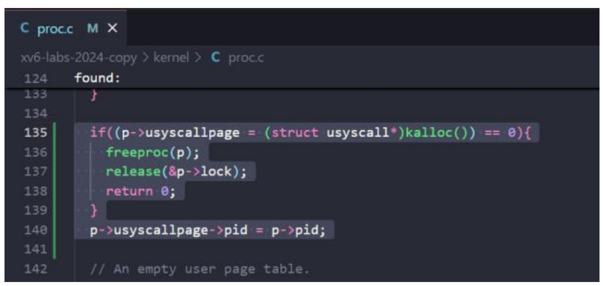
```c
struct proc {
  struct trapframe *trapframe; // data page for trampoline.S
  struct context context;      // swtch() here to run process
  struct file *ofile[NOFILE];  // Open files
  struct inode *cwd;           // Current directory
  char name[16];               // Process name (debugging)
  struct usyscall* usyscallpage;
};
```

Function `pagetable_t proc_pagetable(struct proc *p)` in `kernel/proc.c` will map page table for process. Function `mappages` will be used for mapping. Store `struct usyscall` with `PTE_R` and `PTE_U` flag.



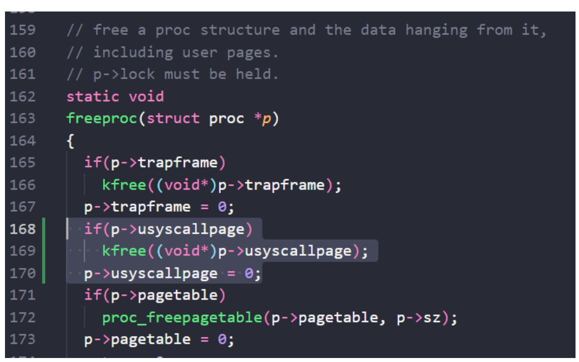
```
proc.c
xv6-labs-2024-copy > kernel > proc.c
// Create a user page table for a given process, with no user memory,
// but with trampoline and trapframe pages.
pagetable_t
proc_pagetable(struct proc *p)
{
  pagetable_t pagetable;

  // An empty page table.
  pagetable = uvmcreate();
  if(pagetable == 0)
    return 0;

  if(mappages(pagetable, USYSCALL, PGSIZE,
              (uint64)(p->usyscallpage), PTE_U | PTE_R) < 0){
    uvmfree(pagetable, 0);
    return 0;
  }

  // map the trampoline code (for system call return)
  // at the highest user virtual address.
  // only the supervisor uses it, on the way
  // to/from user space, so not PTE_U.
  if(mappages(pagetable, TRAMPOLINE, PGSIZE,
              (uint64)trampoline, PTE_R | PTE_X) < 0){
    uvmfree(pagetable, 0);
    return 0;
  }
```

Now store process ID when the process is created. `struct allocproc` allocates the process.



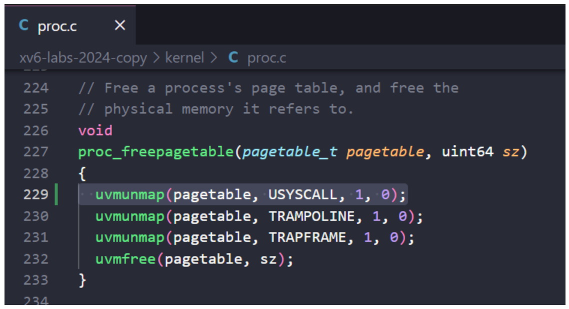
```
proc.c M
xv6-labs-2024-copy > kernel > proc.c
found:
  }

  if((p->usyscallpage = (struct usyscall*)kalloc()) == 0){
    freeproc(p);
    release(&p->lock);
    return 0;
  }
  p->usyscallpage->pid = p->pid;

  // An empty user page table.
```

We create a page when the process created, therefore we must delete the page when the process is deleted. The process is deleted in `freeproc` function.

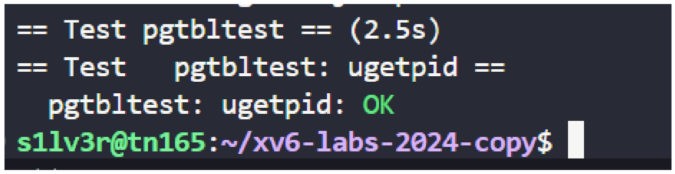


```
// free a proc structure and the data hanging from it,
// including user pages.
// p->lock must be held.
static void
freeproc(struct proc *p)
{
  if(p->trapframe)
    kfree((void*)p->trapframe);
  p->trapframe = 0;
  if(p->usyscallpage)
    kfree((void*)p->usyscallpage);
  p->usyscallpage = 0;
  if(p->pagetable)
    proc_freepagetable(p->pagetable, p->sz);
  p->pagetable = 0;
```

Remove mapping (created in `proc_pagetable` function)



```
proc.c
xv6-labs-2024-copy > kernel > proc.c
// Free a process's page table, and free the
// physical memory it refers to.
void
proc_freepagetable(pagetable_t pagetable, uint64 sz)
{
  uvmunmap(pagetable, USYSCALL, 1, 0);
  uvmunmap(pagetable, TRAMPOLINE, 1, 0);
  uvmunmap(pagetable, TRAPFRAME, 1, 0);
  uvmfree(pagetable, sz);
}
```



```
== Test pgtbltest == (2.5s)
== Test   pgtbltest: ugetpid ==
  pgtbltest: ugetpid: OK
s1lv3r@tn165:~/xv6-labs-2024-copy$
```

Which other xv6 system call(s) could be made faster using this shared page? Explain how.

The system call `copyout` will process faster, as it saves time by copying data directly from the buffer to the address in memory. Furthermore, read-only system calls will also be sped up, since a context switch to OS mode is no longer necessarily required. Data can be read directly in user mode using the `PTE_U` flag.

4. Print a page table (easy)

We added a system call `kpgtbl()`, which calls `vmprint()` in `vm.c`. It takes a `pagetable_t` argument, and your job is to print that pagetable in the format described below.

Function `print_pgtbl()` (the name is as the 2nd question) is defined to print the page table into depth recursion. Then function `vmprint()` (predeclared by the lab creator in `vm.c`) is defined.

```c
#ifdef LAB_PGTBL
void
print_pgtbl(pagetable_t pagetable, int depth)
{
  // there are 2^9 = 512 PTEs in a page table.
  for(int i=0; i < 512; i++){
    pte_t pte = pagetable[i];
    if (pte & PTE_V){
      printf("..");
      for(int j=0;j<depth;j++) {
        printf(" .."); // print the depth of the page table
      }
      // must casting to void*, else warning from compiler
      printf("%d: pte %p pa %p\n", i, (void*)pte, (void*)PTE2PA(pte));

    // check if the PTE is a page table
      if((pte & PTE_V) && (pte & (PTE_R | PTE_W | PTE_X)) == 0) {
        uint64 child = PTE2PA(pte); // get the physical address of the child page table
        print_pgtbl((pagetable_t)child, depth+1); // recursively print the child page table
      }
    }
  }
}

void
vmprint(pagetable_t pagetable) {
  printf("page table %p\n", pagetable);
  print_pgtbl(pagetable, 0);
}
#endif
```

The pointers must be cast to void*, else warning from compiler:

```
kernel/vm.c: In function ‘print_pgtbl’:
kernel/vm.c:502:24: error: format ‘%p’ expects argument of type ‘void *’, but argument 3 has type ‘pte_t’ {aka ‘long unsigned int’} [-Werror=format=]
  502 |       printf("%d: pte %p pa %p\n", i, pte, PTE2PA(pte)); // print the PTE and the physical address
      |                       ~^              ~~~
      |                        |              |
      |                        void *         pte_t {aka long unsigned int}
      |                       %ld
kernel/vm.c:502:30: error: format ‘%p’ expects argument of type ‘void *’, but argument 4 has type ‘pte_t’ {aka ‘long unsigned int’} [-Werror=format=]
  502 |       printf("%d: pte %p pa %p\n", i, pte, PTE2PA(pte)); // print the PTE and the physical address
      |                             ~^
      |                              |
      |                              void *
      |                             %ld
cc1: all warnings being treated as errors
make: *** [Makefile:136: kernel/vm.o] Error 1
```

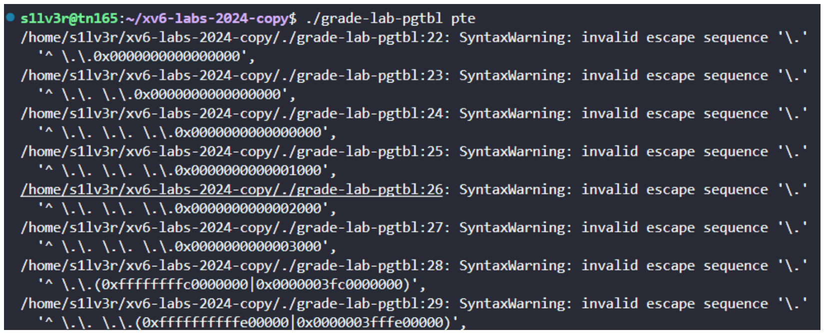
In `exec()` function in `kernel/exec.c`, call to print page table.

```c
{
  p->trapframe->epc = elf.entry;  // initial program counter = main
  p->trapframe->sp = sp; // initial stack pointer
  proc_freepagetable(oldpagetable, oldsz);

  if(p->pid == 1)
    vmprint(p->pagetable);

  return argc; // this ends up in a0, the first argument to main(argc, argv)
```

Run `make qemu` and the page table is printed

```
hart 2 starting
hart 1 starting
page table 0x0000000087f4d000
..0: pte 0x0000000021fd2401 pa 0x0000000087f49000
.. ..0: pte 0x0000000021fd2001 pa 0x0000000087f48000
.. .. ..0: pte 0x0000000021fd281b pa 0x0000000087f4a000
.. .. ..1: pte 0x0000000021fd1c17 pa 0x0000000087f47000
.. .. ..2: pte 0x0000000021fd1807 pa 0x0000000087f46000
.. .. ..3: pte 0x0000000021fd1417 pa 0x0000000087f45000
..255: pte 0x0000000021fd3001 pa 0x0000000087f4c000
.. ..511: pte 0x0000000021fd2c01 pa 0x0000000087f4b000
.. .. ..509: pte 0x0000000021fd5413 pa 0x0000000087f55000
.. .. ..510: pte 0x0000000021fd5807 pa 0x0000000087f56000
.. .. ..511: pte 0x000000002000180b pa 0x0000000080006000
init: starting sh
```

<u>Note:</u> We cannot run make grade. We have no idea it did not work.



5. Detect which pages have been accessed (hard)

> *Your job is to implement `pgaccess()`, a system call that reports which pages have been accessed. 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 (a datastructure that uses one bit per page and where the first page corresponds to the least significant bit). You will receive full credit for this part of the lab if the pgaccess test case passes when running pgtbltest.*

The objective of this assignment was to implement the `sys_pgaccess` system call in the xv6 operating system. This system call identifies which pages within a specified range have been accessed by a process by examining the `PTE_A` (Accessed) flag in each Page Table Entry (PTE). This functionality is essential for tasks such as memory profiling, security auditing, and optimizing page replacement algorithms.

1. System Call Implementation:

Integrated `sys_pgaccess` into the system call table to make it accessible to user programs.

2. Parameter Handling:

Utilized `argaddr` and `argint` to retrieve the three parameters passed by the user:

* **Start Page Address (addr)**: The virtual address marking the beginning of the page range.

* **Number of Pages (num)**: The total number of consecutive pages to inspect.

* **Bitmask Output (dest)**: A user-space pointer where the resultant bitmask will be stored.

3. Page Table Traversal:

Leveraged the existing `walk` function to navigate the three-level page table hierarchy of RISC-V Sv39, retrieving the PTE corresponding to each page within the specified range without allocating new pages (`alloc` set to 0).

4. Access Flag Processing:

For each page in the range:

- Checked the `PTE_A` flag to determine if the page had been accessed.
- If accessed, set the corresponding bit in an unsigned integer `abits` to 1.
- Cleared the `PTE_A` flag to reset the accessed status for future tracking.

5. Result Transmission

Employed the `copyout` function to safely transfer the constructed bitmask from kernel space to the user-provided address, ensuring accurate and secure data transfer.

Run code with `make qemu` and `pgtbltest`:

```
print_pgtbl: OK
ugetpid_test starting
ugetpid_test: OK
print_kpgtbl starting
page table 0x0000000087f22000
..0: pte 0x0000000021fc7801 pa 0x0000000087f1e000
.. ..0: pte 0x0000000021fc7401 pa 0x0000000087f1d000
.. .. ..0: pte 0x0000000021fc7c5b pa 0x0000000087f1f000
.. .. ..1: pte 0x0000000021fc701b pa 0x0000000087f1c000
.. .. ..2: pte 0x0000000021fc6cd7 pa 0x0000000087f1b000
.. .. ..3: pte 0x0000000021fc6807 pa 0x0000000087f1a000
.. .. ..4: pte 0x0000000021fc64d7 pa 0x0000000087f19000
..255: pte 0x0000000021fc8401 pa 0x0000000087f21000
.. ..511: pte 0x0000000021fc8001 pa 0x0000000087f20000
.. .. ..509: pte 0x0000000021fd4c13 pa 0x0000000087f53000
.. .. ..510: pte 0x0000000021fd00c7 pa 0x0000000087f40000
.. .. ..511: pte 0x000000002000184b pa 0x0000000080006000
print_kpgtbl: OK
superpg_test starting
pgtbltest: superpg_test failed: pte different, pid=3
```