[xv6] PROJECT 3 – Page Tables

University of Science - VNUHCM

- **Date**: 12/04/2025
- **Subject**: *Operating System*
- **Class**: *23CLC03*
- **Environment**: *Ubuntu-24.04*
- **Intructors:**
  - *Phan Quốc Kỳ*
  - *Lê Hà Minh*
  - *Lê Giang Thanh*

I. Group Information

No	Student's ID	Student's Full Name	Contribution
1	23127004	Lê Nhật Khôi	`33%` : Print a page table
2	23127113	Nguyễn Trần Phú Quý	`33%` : Detect which pages have been accessed
3	23127165	Nguyễn Hải Đăng	`33%` : Speed up system calls

II. Submission's Details

Here are all of the assignments stated in the lab.

1. Speed up system calls

1.1. Problem Overview

The goal was to optimize the `getpid()` system call in xv6 by eliminating the need for a kernel trap. This was achieved by mapping a read-only page containing the process ID

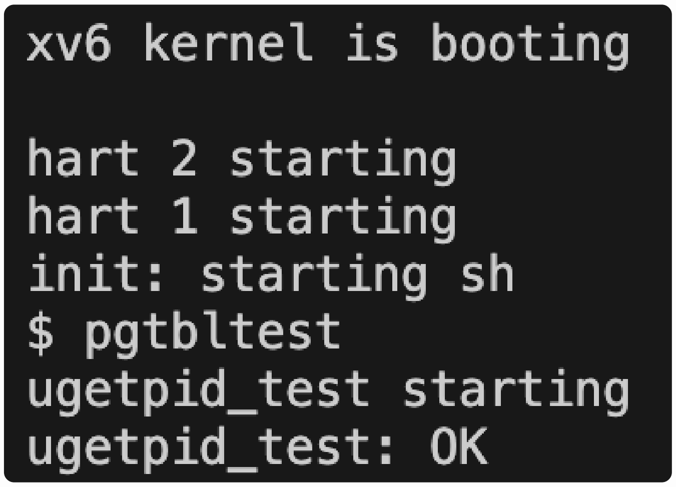
(PID) into the user's address space at a predefined virtual address (`USYSCALL`).

1.2. Implementation Details

- **Data Structures:**
  - Defined `struct usyscall` containing just an `int pid;` .
  - Added a `struct usyscall *usyscall` field to `struct proc` to store the kernel virtual address (KVA) of the allocated shared page.
- **Process Creation (`allocproc`):**
  - Allocated a new physical page using `kalloc()` and stored its KVA in `p->usyscall` .
  - Wrote the newly assigned process ID (`p->pid`) into the `pid` field of the structure on this page.
- **Page Table Management (`proc_pagetable` , `proc_freepagetable`):**
  - Modified `proc_pagetable` (used by `allocproc` and `exec`) to map the physical page corresponding to `p->usyscall` (obtained using `V2P`) to the virtual address `USYSCALL` .
  - The mapping permissions were set to `PTE_R | PTE_U` , allowing user read-only access.
  - Modified `proc_freepagetable` to unmap the `USYSCALL` page during page table cleanup.
- **Process Cleanup (`freeproc`):**
  - Added a call to `kfree(p->usyscall)` to deallocate the shared page when the process is freed.

1.3. Testing

The implementation was tested using the `pgtbltest` suite. The `ugetpid` test case, which specifically uses the `USYSCALL` mapping, passed successfully.



```
xv6 kernel is booting

hart 2 starting
hart 1 starting
init: starting sh
$ pgtbltest
ugetpid_test starting
ugetpid_test: OK
```

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

The same shared page mechanism could potentially speed up other system calls that return simple, relatively static process-specific information:

- **`getppid()`**: The parent PID could be added to `struct usyscall`. This would require updating the value in the child's shared page if the child is reparented (e.g., when its original parent exits and `init` adopts it). This update would need to happen within the kernel's `reparent` logic.
- **(Hypothetical) `getuid()` / `getgid()`**: If xv6 implemented user/group IDs, these could also be stored in the shared page, as they typically don't change frequently for a running process.

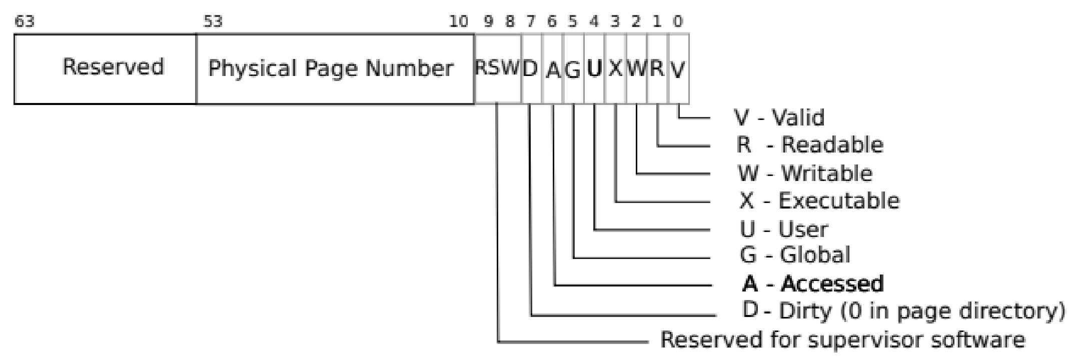
These optimizations follow the same principle: place read-only, frequently accessed, process-specific data in the shared page to avoid kernel crossings.

2. Print a page table

2.1. Brief introduction to page table

In the xv6 operating system, the **page table** is a data structure to manage virtual memory, translating virtual addresses into physical addresses. A page table is stored in physical memory as three-level tree: each of those pages contains 512 PTE(Page Table Entry), which contain the physical addresses for page-table pages in the next level of the tree.

The structure of a PTE includes two main components: the physical address of the mapped page and various flags that define the properties of the page. The lower 20 bits of the PTE store the physical address of the corresponding 4KB page in memory, while the upper 12 bits are used for flags that provide control over the page's behavior.



> Some details about a few PTE's flag:
>
> - PTE_V(Valid): Indicates that the page entry is valid and that the associated page is mapped to physical memory.
> - PTE_R(Readable): Indicate whether the page has been read or accessed.
> - PTE_W(Writable): Indicates whether the page is writable by the process.
> - PTE_X(Executable): Indicates whether the page is executable by the process.
> - PTE_U(User): Controls whether the page is accessible by user-level code.

2.2. Approach and Pseudocode

The goal is implementing `vmprint(pagetable_t pagetable)` in `kernel/vm.c` whose the definition is in `kernel/defs.h` so that we can call it from `exec.c`

Approach

To print page tables, recursive method will be used to print information of child page which required a help function for handling traversal.

```
1. The help function loops through all 512 entries
of the current page table

2. For each PTE, it first checks
if the entry is valid(PTE_V flag) or not

3. If the entry is valid, it extracts the physical address
using the PTE2PA macro, then it prints the index,
the PTE value, and the physical address.
Else we move to the next loop.

4. If PTE doesn't have read (PTE_R), write (PTE_W), or execute (PTE_X)
it indicates that the entry points to a lower-level
page table rather than a leaf data page.
In that case, the function recursively calls itself with the
new page table and increases the depth by one.
```

Pseudocode

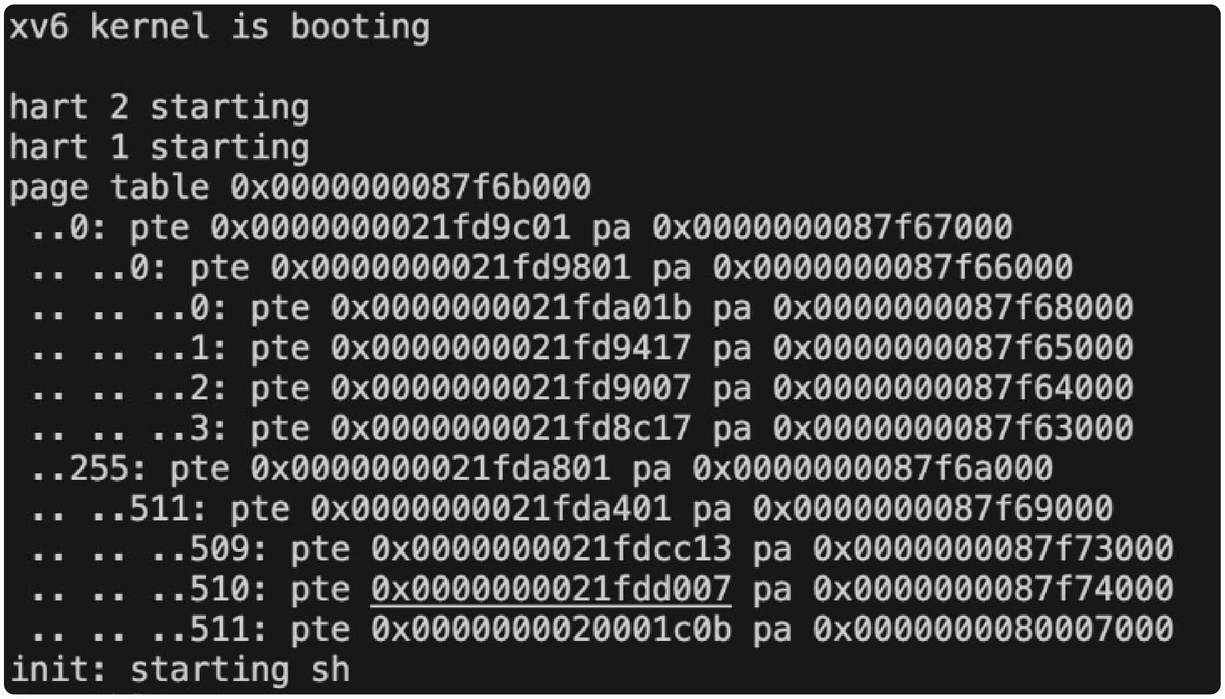
```
function recursion_vmprint(pagetable, id):
    for i from 0 to 511:
        pte = pagetable[i]

        if pte is valid (pte & PTE_V):
            print i, pte, and pa

            if pte does not have read/write/execute permissions
            ((pte & (PTE_R | PTE_W | PTE_X)) == 0):
                call recursion_vmprint(pa, id + 1)

function vmprint(pagetable):
    print "page table" and pagetable address
    call recursion_vmprint(pagetable, id = 1)
```

2.3. Final Result



```
xv6 kernel is booting

hart 2 starting
hart 1 starting
page table 0x0000000087f6b000
 ..0: pte 0x0000000021fd9c01 pa 0x0000000087f67000
 .. ..0: pte 0x0000000021fd9801 pa 0x0000000087f66000
 .. .. ..0: pte 0x0000000021fda01b pa 0x0000000087f68000
 .. .. ..1: pte 0x0000000021fd9417 pa 0x0000000087f65000
 .. .. ..2: pte 0x0000000021fd9007 pa 0x0000000087f64000
 .. .. ..3: pte 0x0000000021fd8c17 pa 0x0000000087f63000
 ..255: pte 0x0000000021fda801 pa 0x0000000087f6a000
 .. ..511: pte 0x0000000021fda401 pa 0x0000000087f69000
 .. .. ..509: pte 0x0000000021fdcc13 pa 0x0000000087f73000
 .. .. ..510: pte 0x0000000021fdd007 pa 0x0000000087f74000
 .. .. ..511: pte 0x0000000020001c0b pa 0x0000000080007000
init: starting sh
```

2.4. For every leaf page in the vmprint output, explain what it logically contains and what its permission bits are.

Explain every leaf page in the `vmprint` output:

Leaf Page	10 Permission Bits (from PTE)	Page Permissions (Flags)
0x0000000021fda01b	0000011011	V R X U
0x0000000021fd9417	0000010111	V R W U
0x0000000021fd9007	0000000111	V R W

0x0000000021fd8c17	0000010111	V R W U
0x0000000021fdcc13	0000010011	V R U
0x0000000021fdd007	0000000111	V R W
0x000000002001c0b	0000001011	V R X

3. Detect which pages have been accessed

3.1. Objective

Implement `pgaccess()` syscall that *detects which pages have been accessed* (read or write) and *reports this information to userspace*.

3.2. Approach

- We utilize the 6th bit (PTE_A) in the Page Table Entry (PTE) to track recent access. This bit is automatically set by the RISC-V architecture whenever a page is accessed. This is achieved by adding this line to `kernel/riscv.h`:

```
#define PTE_A (1L << 6) // recently-accessed bit
```

- We then implement `sys_pgaccess()` in `kernel/sysproc.c` to fulfill the following tasks:
    - *Retrieve arguments from the trapframe.*
    - *Iterate through the page range and check each PTE.*
    - *Check if the PTE_A bit is set.*
    - *Clear the PTE_A bit after checking.*
    - *Build a bitmask of accessed pages and copy it back to user space.*

3.3. Implementation

1. Get Inputs from Userspace:

- Read the virtual address (`addr`) from which to start checking.
- Read how many pages to check (`npages`).
- Read the address in userspace where we should store the result (`bitmask_addr`).
- Check for errors or invalid ranges.

2. Set Up a Bitmask to Record Results

- Use a 64-bit mask where each bit corresponds to a page.

3. Page Table Walking

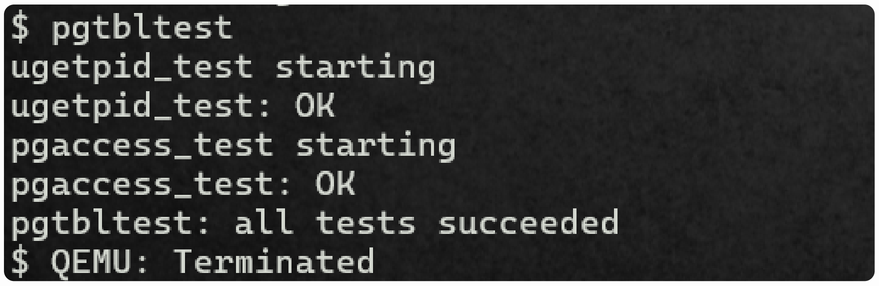
- Use `walk()` to locate the PTE for each page.
- If the `PTE_A` bit is set, mark the corresponding bit in the result bitmask.
- Clear the `PTE_A` bit so it can track future accesses.

4. Copy Result to User Space

- Uses `copyout()` to transfer the result bitmask back to the user-provided address.

3.4. Testing

Successfully passed the `pgaccess_test()` function in `user/pgtbltest.c`:



```
$ pgtbltest
ugetpid_test starting
ugetpid_test: OK
pgaccess_test starting
pgaccess_test: OK
pgtbltest: all tests succeeded
$ QEMU: Terminated
```

3.5. Conclusion

> To implement this syscall, we needed a deep understanding of virtual memory, page table traversal, and kernel-user data transfer. Gotta say, it was worth the 5 points.

III. References

- *Lab Answers from MIT*
- *GeeksforGeeks - System calls*
- *Solution from Western Steamed Buns*!
- *xv6 Lab answers from yixuaz*