Lecture 4 Virtual Memory

2025.11.26

中国科学院大学
University of Chinese Academy of Sciences

Schedule

- Project 4 assignment
- Project 3 due

Project 4 Virtual Memory

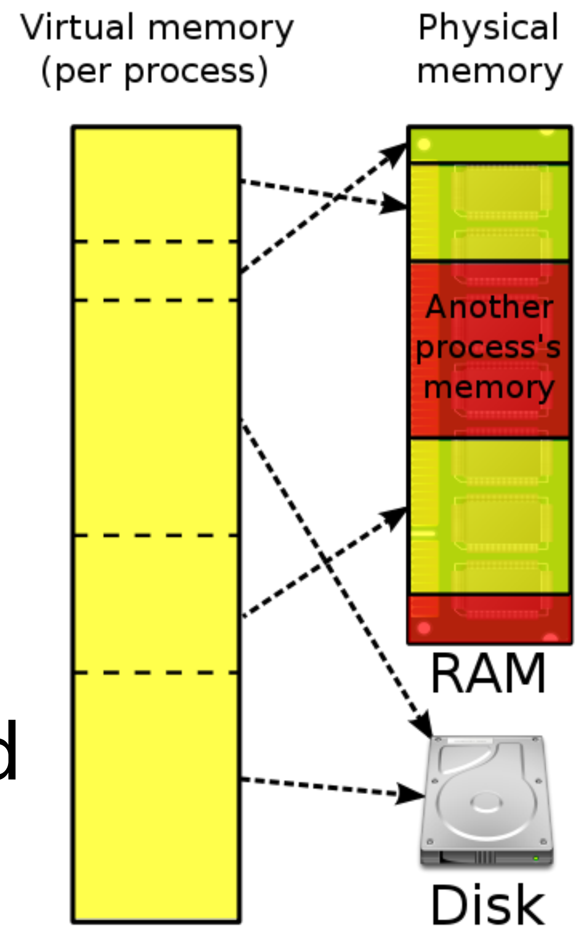
- Requirement
  - Implement virtual memory management
    - Enable VM for your OS
    - Setup page tables for user-level processes
    - Handle page fault to support on demand paging assuming physical memory is enough
    - Handle page swapping assuming physical memory is limited

Project 4 Virtual Memory

- Requirement
  - Implement virtual memory management
    - Support getting free memory
    - Zero-copy based pipe

Project 4 Virtual Memory

- Virtual memory
  - Each process sees a contiguous and linear address space, which is called virtual addresses
  - Virtual addresses are mapped into physical addresses by both HW and SW



[From Wikipedia]

Project 4 Virtual Memory

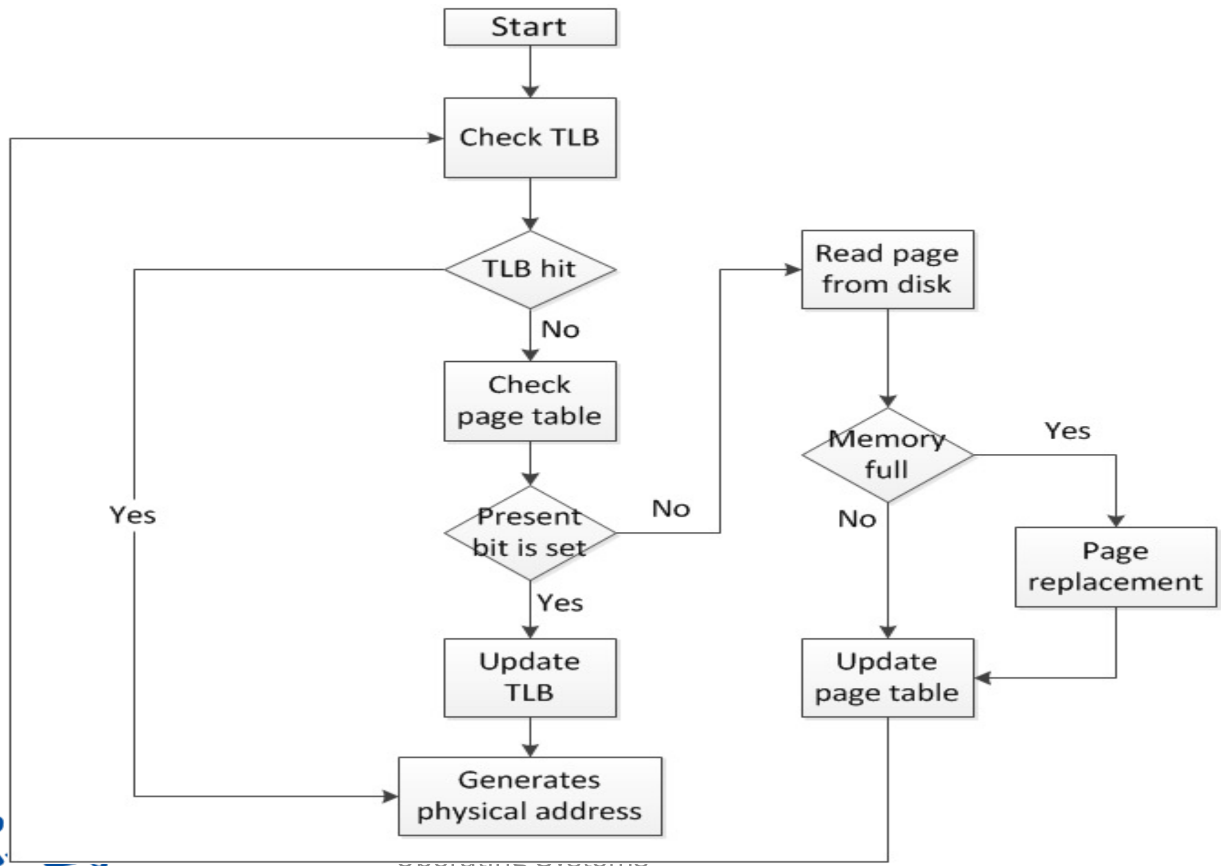
- Virtual memory
  - Virtual address space is divided into pages, which are blocks of contiguous virtual memory addresses
    - e.g. 4KB pages
  - Each page has a virtual address, and is mapped into a physical page frame (e.g. 4KB)

Project 4 Virtual Memory

- Virtual memory
  - Page tables
    - The data structure to store the mapping between virtual addresses and physical addresses
    - Each mapping is a page table entry
  - MMU and TLB
    - MMU stores a cache of recently used mappings from the page table, which is called translation lookaside buffer (TLB)
  - For RISC-V, TLB is handled by hardware
    - Hardware page table walker

Project 4 Virtual Memory

- Virtual memory
  - Address translation



Project 4 Virtual Memory

- Enabling virtual memory
  - You need to set ***satp*** register
    - PPN is the physical page frame of page directory
    - Fill 8 in MODE to enable sv39 mode (three-level page table)
    - ASID represents the ID of current process, which is used to distinguish different processes

63 … 60	59 … 44	43 … 0
MODE (**WARL**)	ASID (**WARL**)	PPN (**WARL**)
4	16	44

Project 4 Virtual Memory

- Enabling virtual memory
  - Example

```
// SATP_MODE_SV39 为 8, ASID 为 0, NOMAL_PAGE_SIZE 为 12, 代表 4KB 的偏移
// 这些宏定义定义在 arch/riscv/include/pgtable.h
set_satp(SATP_MODE_SV39, 0, 0x51000000 >> NORMAL_PAGE_SHIFT);
```

Project 4 Virtual Memory

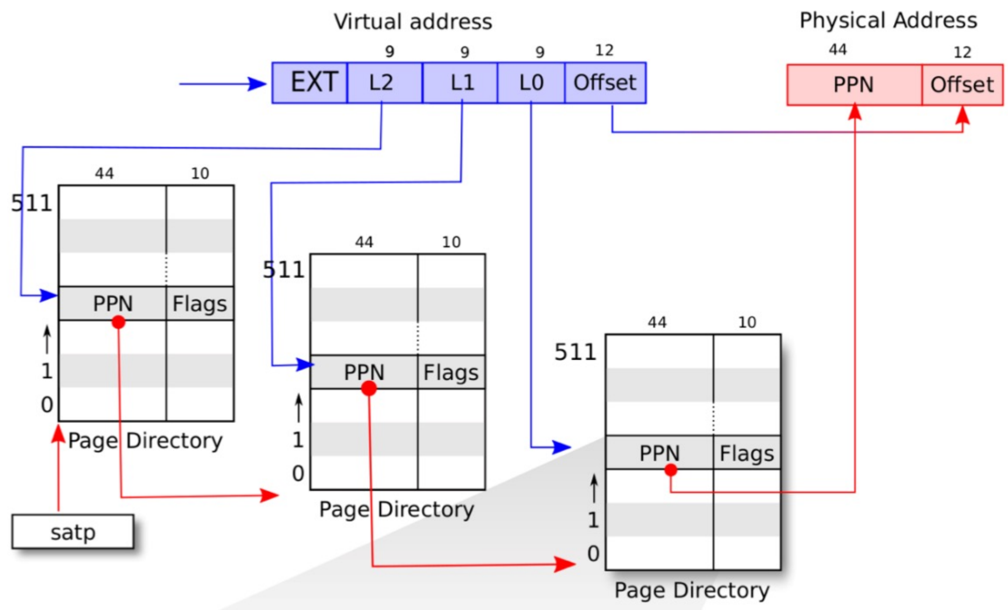
- Process switch
  - Modify ***satp*** register
  - Use *sfence.vma* to flush TLB manually
  - Pls. refer to *local_flush_tlb_all* 和 *local_flush_tlb_page* (arch/riscv/include/pgtable.h)
- Note that, when PTE is modified, you also need to flush TLB

Project 4 Virtual Memory

- Page table setup
  - After enabling virtual memory, both kernel and user-level processes use virtual addresses
  - Both kernel and user-level processes need to setup page tables
  - Pls. note that page table itself also needs page frames

Project 4 Virtual Memory

- Page table setup
  - Two-level page table is used for kernel, which is already setup in start code
  - Three-level page table is used for user-level process



University of Chinese Academy of Sciences

Project 4 Virtual Memory

- Page table setup
  - Design the structure of page table entries (PTE)
    - Physical address
    - X,W,R
    - U : Set for user-mode access
    - Valid
    - Access, Dirty
      - They are automatically set using QEMU, but need to be set by page fault handler when on board

63–54	53–28	27–19	18–10	9–8	7	6	5	4	3	2	1	0
Reserved	PPN[2]	PPN[1]	PPN[0]	RSW	D	A	G	U	X	W	R	V
10	26	9	9	2	1	1	1	1	1	1	1	1

Project 4 Virtual Memory

- Page table setup
  - Statically fill page table
    - Fill paired VA to PA mappings
  - On-demand paging
    - An empty page table
    - Fill the page table when page fault occurs

Project 4 Virtual Memory

- Kernel Address Space
  - The entry point of the kernel is 0xffffffc0_50202000, and its physical address is 0x50202000

地址范围	建议用途
0xffffffc0_50000000 - 0xffffffc0_50200000	BBL 代码及其运行所需的内存
0xffffffc0_50200000 - 0xffffffc0_51000000	Kernel 的数据段/代码段等
0xffffffc0_51000000 - 0xffffffc0_52000000	Kernel 页表以及跳转表等
0xffffffc0_52000000 - 0xffffffc0_60000000	供内核动态分配使用的内核虚拟地址空间

Project 4 Virtual Memory

- Process Address Space
  - Each Process has its private address space
    - Apps may start at same entry point and stack address
  - All processes share kernel address space
    - You need to copy the kernel page table into the page tables of all user-level processes

Project 4 Virtual Memory

- Load user program
  - First set up page table for user process
  - The kernel needs to load the program to the virtual address
  - The kernel needs to know the kernel virtual address mapping to the corresponding physical address for the above VA

```
// va 为需要映射的虚拟地址，pgdir 为页表目录，
// 返回值为为 va 映射的物理地址对应的内核虚地址
uintptr_t alloc_page_helper(uintptr_t va, uintptr_t pgdir);
```

Project 4 Virtual Memory

- Load user program

```
Elf file type is EXEC (Executable file)
Entry point 0x10000
There are 2 program headers, starting at offset 64

Program Headers:
  Type           Offset             VirtAddr           PhysAddr
                 FileSiz            MemSiz              Flags  Align
  LOAD           0x0000000000001000 0x0000000000010000 0x0000000000010000
                 0x0000000000000c35 0x0000000000001c40  RWE    0x1000
  GNU_STACK      0x0000000000000000 0x0000000000000000 0x0000000000000000
                 0x0000000000000000 0x0000000000000000  RW     0x10

 Section to Segment mapping:
  Segment Sections...
   00     .text .rodata .sdata .sbss .bss
   01
```

Virtual address range: 0x10000 ~ 0x11c40
Process image location: 0x10000 ~ 0x10c35
Fill zeros: 0x10c35 ~ 0x11c40

Project 4 Virtual Memory

- Load user program

```
Elf file type is EXEC (Executable file)
Entry point 0x10000
There are 2 program headers, starting at offset 64

Program Headers:
  Type           Offset             VirtAddr           PhysAddr
                 FileSiz            MemSiz              Flags  Align
  LOAD           0x0000000000001000 0x0000000000010000 0x0000000000010000
                 0x0000000000000c35 0x0000000000001c40  RWE    0x1000
  GNU_STACK      0x0000000000000000 0x0000000000000000 0x0000000000000000
                 0x0000000000000000 0x0000000000000000  RW     0x10

 Section to Segment mapping:
  Segment Sections...
   00     .text .rodata .sdata .sbss .bss
   01
```

Virtual address range: 0x10000 ~ 0x11c40
Assume mapped physical address range: 0x52000000 ~ 0x52000c35
Kernel load program image: 0xffffffc0 52000000 ~ 0xffffffc0 52000c35

Project 4 Virtual Memory

- Handling page fault
  - Page fault: not matching PTE in the page table
    - Allocate a physical page
    - Update the page table to setup the mapping between virtual page and physical page
    - Flush TLB entry
  - Use *scause* register to identify interrupt type
  - Develop page fault handler
  - The address of virtual page causing page fault is stored in *stval* register

Project 4 Virtual Memory

- Handling swapping
  - Assuming you have limit memory space, you need to swap in-memory pages out to disk when there are no available pages.
  - When these swapped-out pages are reused again, swap the pages in from disk
  - Use bios read/write functions we provided to read from and write to SD card

Project 4 Virtual Memory

- Step-by-step Task 1
  - Load kernel into memory
  - Finish APIs for manipulating kernel page table
  - Print "CPU #i has entered kernel with VM!"

```c
/*
 * Just start kernel with VM and print this string
 * in the first part of task 1 of project 4.
 * NOTE: if you use SMP, then every CPU core should call
 *  `kernel_brake()` to stop executing!
 */
printk("> [INIT] CPU #%u has entered kernel with VM!\n",
    (unsigned int)get_current_cpu_id());
// TODO: [p4-task1 cont.] remove the brake and continue to start user
   processes.
kernel_brake();
```

Project 4 Virtual Memory

- Step-by-step Task 1
  - Statically setup page table for user-processes *shell* and *fly*
    - Initialize page table entries and set all pages valid
    - Filling kernel page table entries into user-level process page table
    - Modify scheduler to allow switch page table among different processes
    - Reclaim memory space when executing *kill* and *exit*

Project 4 Virtual Memory

- Step-by-step: Task 2
  - Setup dynamic page table for user-level processes to support on-demand paging
  - Handling page fault for *rw*.
    - *rw* runs when you can input different virtual addresses
  - You need support in-kernel locking as you did in Project 2
    - The lock resides in the shared kernel space

Project 4 Virtual Memory

- Step-by-step: Task 3
  - Limit the available memory space, and design a test case to access a range of memory addresses, which is larger than the available memory space
  - Supporting swapping with your own designed replacement policy
    - We do not have any specific requirement for the replacement policy

Project 4 Virtual Memory

- Step-by-step: Task 4
  - Support user to use the shell command *free* to check the available memory space
  - Implement the corresponding syscall function sys_get_free_memory

Project 4 Virtual Memory

- Step-by-step: Task 5
  – Support zero copy based pipe
  – After sender sends data into the pipe, the kernel unmaps the mapping between the physical page frames and the virtual pages of the pipe
  – Then, the kernel maps these page frames holding the senders' data to the address space of the receiver
  – After that, the receiver can directly read the data

Project 4 Virtual Memory

- Requirements for design review
  - 请描述你对内核页表的理解，例如内核页表结构、PTE内容、页表大小、页表在内存中的存放位置等
  - 请描述用户态进程页表创建设置的过程。并简述内核加载一个用户进程时，如何将其运行在虚拟内存上？
  - 请描述page fault处理的流程，最好可以提供一些流程伪代码

Project 4 Virtual Memory

- Requirements for design review
  - Page swap时使用的替换策略是什么？
  - 任何想讨论的其他问题

Project 4 Virtual Memory

- Requirement for S/A/C-Core

Core type	Task requirements
S-Core	Task 1
A-Core	Tasks 1~3
C-Core	Tasks 1~5

Project 4 Virtual Memory

- P4 schedule
  - Design review: 3rd Dec.
  - Due
    - C core due : 10th Dec.
    - A core due: 17th Dec.
  - Please read the guidebook carefully, and contact us if you have any question