

Homework 4: CS 361

Multilevel page tables

second revision

Jon A. Solworth

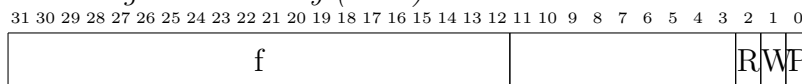
April 12, 2019

1 Overview

You are working as a Developer for LetsManage Inc. The development team has decided to implement a Proof of Concept for a new Memory Management Unit which needs to be developed for a new Operating System version. You must implement the 4 APIs described below. Your code will be tested against a series of calls to these APIs and the output. NOTE: insert(),remove() are invoked by the OS itself while write(),read() are user calls.

You are going to implement a 2-level page table for a 32-bit architecture.

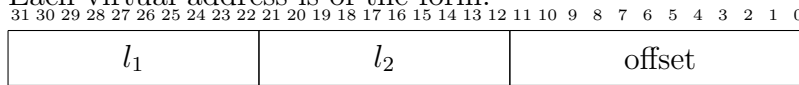
- Each page is 4096 bytes,
- Each *Page Table Entry (PTE)* is 32 bits



- The top 20 bits are the frame **f** and
- The bottom 12 bits are the permissions (only the low order 3 bits are used)
- The permissions are (by bit position):
 - 2:** R (read),
 - 1:** W (write), and
 - 0:** P (present)
- The contents of the PTE are a 32 bit unsigned int, but we shall write it as a pair ($f, perm$) where f is the frame and $perm$ is the permissions. The permissions will be written here using **RWP** notations.

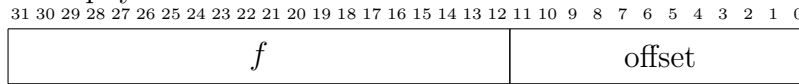
For example as **R-P** which means permissions R and P, but not W. Numerically, bit 2 and 0 are set, and thus the value is 5. Thus (17,5) has the numerical value $17 \cdot 4096 + 5$.

- Each virtual address is of the form:



- $l_1||l_2$ is the page number. Page numbers are 20-bit values.

- Each physical address is of the form:



- f is the frame. Frames are 20-bit values, although physical memory is smaller than this.

2 Interfaces and data structures

You should have the definitions as shown in Figure 1. Please read them carefully, they are critical for understanding the statements below.

- `pageTable` contains page table (enough to represent a single process). The `pageTable` is physically addressed.
- `userPages` contain the actual values which are read/written from user space. The `userPages` are physically addressed.
- `insert_pte` inserts the page table entry (mapping) in the page table from page `p` to frame `f`.
 - The 1st level page table entry will have all three permissions set, ie RWP.
 - The 2nd level page table entry set by the insert will have permissions `read_vaddr` and `write_vaddr` as given in the arguments and will also sent P.

It returns 1 if successful. For errors, see the below error list.

- `remove_pte` removes the page table entries for page `p`. The 2nd level page table entry is set to 0 i.e. (0, ---). It returns 1 if successful. For errors, see the below error list.
- `read_vaddr` reads a character at physical address `paddr` in `userPages`, after using the page table to map `vaddr` to `paddr`. If any page table entry used in the translation does not have both R and P bits set, the translation fails. It returns 1 if successful. For errors, see the below error list. On success, it returns the character read as `*c`.
- `write_vaddr` writes a character `c` n at physical address `paddr` in `userPages`, after using the page table to map `vaddr` to `paddr`. If any page table entry used in the translation does not have W and P bit sets, the translation fails. It returns 1 if successful. For errors, see the below error list.

3 Errors

The error codes are as follows:

- -1: missing read permission
- -2: missing write permission
- -3: not present
- -4: 2nd level page table entry already defined (when doing an insert)

```

#define PAGE_SIZE_IN_BYTES 4096
#define PAGE_SIZE_IN_UINTS 1024
#define PAGE_TABLE_PAGES 1025
#define USER_PAGE_PAGES 1000

typedef unsigned int uint32;

/* a single page of the page table holds
   1024 page table entries, each a uint
   */
typedef uint32 PtePage[1024];

/* the page table consists of 1025 pages:

   . pageTable[0] holds the level1 page table page

   . pageTable[1]...pageTable[1024]
     holds the 1024 level2 page table pages
   */
PtePage pageTable[PAGE_TABLE_PAGES];

/* holds the user pages, byte addressable, where
   userPages[0].. userPages[4095] holds the first userPage
   */
char userPages[PAGE_SIZE_IN_BYTES*USER_PAGE_PAGES];

int insert_pte(uint32 p, uint32 f, uint32 read, uint32 write);
int remove_pte(uint32 p);
int read_vaddr(uint32 vaddr, char *c);
int write_vaddr(uint32 vaddr, char c);

```

Figure 1: Declarations

4 Example

4.1 Page table walk

Consider a page table walk for the above virtual address for a **read**.

address	Page Table Entry	
	frame	permission check
pageTable[0][l_1]	f_1	R and P
pageTable[f_1][l_2]	f_2	R and P
userPage[$f_2 \ll 12 + \text{offset}$]		

4.2 Sequence of operations

Next, we consider a sequence of calls, including the most critical procedure `insert()`.

The key to this assignment is the `insert()` procedure call. Note that the page number `p` has the range 0x00000 to 0xFFFFF, and that both `pageTable` and `userPages` are accessed by physical addresses.

The `pageTable` is split into one 1st level page at `pageTable[0]` and 1024 2nd level pages, where 2nd level page i is at address `pageTable[i+1]`. Thus there are 1025 page frames for the `pageTable`.

The `userPages` is byte addressible and has a 1000 frames, indexed in the range 0 to 999; this corresponds to physical addresses 0 ... 1000*4096-1. Obviously you should not map to a physical address outside this range.

i	call	effect	return
1	<code>insert_pte(23, 5, 1, 0);</code>	<code>pageTable[0][0] = (1, RWP);</code> <code>pageTable[1][23] = (5, R-P)</code>	1
2	<code>insert_pte(1023, 999, 1, 1)</code>	<code>pageTable[0][0] = (1, RWP);</code> <code>pageTable[1][1023] = (999, RWP)</code>	1
3	<code>insert_pte(4097, 29, 0, 1)</code>	<code>pageTable[0][4] = (5, RWP);</code> <code>pageTable[5][1] = (29, -WP)</code>	1
4	<code>write_vaddr(1023*4096+131, 'x')</code>	<code>userPages[999*4096+131] = 'x'</code>	1
5	<code>read_vaddr(1023*4096+131, cP)</code>		1, *cP='x'
6	<code>write_vaddr(23*4096, cP)</code>		-2

1. This will map page 23 ($l_1 = 0, l_2 = 23$) to frame 5 with read permission
2. This will map page 1023 ($l_1 = 0, l_2 = 1023$) to frame 999 with read and write permissions
3. This will map page 4097 ($l_1 = 4, l_2 = 1$) to frame 29 with write permission
4. This writes an 'x' at physical address $999*4096 + 131$
5. This read an 'x' (*cPtr) at physical address $999*4096 + 131$
6. This will fail because of a lack of write permissions

5 Submission

Your implementation of the `insert_pte`, `remove_pte`, `write_vaddr`, and `read_vaddr` should be saved as `hw4.c` and submitted to the link provided on gradescope.

For your convenience we've provided the `hw4.h` and `makefile` that gradescope will use to compile and your implementation. We recommend coding your own unit tests or driver file to test your implementation.

The current grade scope output tests the cases shown in the "sequence of operations" section. Your implementation will be tested on a wider range of test cases after the due date.