CSCI 4100 – Assignment 7 – Virtual Memory

Learning Outcomes

Implement a simulated virtual address space.

Required Reading

PoCSD 5.3-5.4, 6.2

Instructions

For this assignment you will be writing a simulated virtual address space. The system you are simulating has the following attributes:

- Both virtual addresses and physical addresses are 16 bits or 2 bytes long.
- The page size is 256 (2^8) bytes, so a virtual address consists of an 8-bit page number and an 8-bit offset and a physical address consists of an 8-bit frame number and an 8-bit offset.
- Since a page number is 8 bits long, there must be 2^8 or 256 page table entries.
- Each page table entry is 16 bits long and contains several additional bits along with the frame number for the associated page.

Data Types

Working with page tables is largely a matter of manipulating bits. I have introduced several data types based on unsigned integers that make it a little clearer what all of these bits mean:

- `page_table` - An array of 256 page table entries. Use this data type to create your page table:

  ```
  page_table table;
  ```

- `pt_address` - A 16-bit virtual or physical address.
- `pt_index` - An 8-bit page or frame number.
- `pt_offset` - An 8-bit offset within a page.
- `pt_entry` - A 16-bit page table entry. Use this data type when you are looking up an entry in a page table:

  ```
  pt_entry entry = table[page_num];
  ```

- pt_bits - Eight bits that provide the following information:
  - PT_ALLOC - the page has been allocated in the virtual address space.
  - PT_DIRTY - the page has been written to recently.
  - PT_ACCESS - the page has been read from recently.
  - PT_PRESENT - the page is resident in physical memory.
  - PT_KERNEL - the page is only accessible in kernel mode.
  - PT_READ - the page may be read from.
  - PT_WRITE - the page may be written to.
  - PT_EXECUTE - the page contains executable instructions.

All of these data types are defined in the file va_space.h.

Functions

You must implement the following functions:

- pt_init - initializes all of the entries in a page table to zero, which among other things marks them as unallocated.
- pt_map - maps a page to a frame in a page table with the given permissions. The entry should have its allocated and present bits set, but not the accessed or dirty bits.
- pt_unmap - removes an entry from the page table by setting it to zero.
- pt_set_dirty, pt_set_accessed, pt_set_present - sets the corresponding bits.
- pt_clear_dirty, pt_clear_accessed, pt_clear_present - clears the corresponding bits.
- pt_allocated, pt_dirty, pt_accessed, pt_present - returns true if the corresponding bit is set, false otherwise.
- pt_not_permitted - returns true if the accesses requested are not permitted for the page in question, false otherwise.
- pt_translate - translates a virtual address to the corresponding physical address using the page table. May return one of the following errors:
  - ERR_PAGE_NOT_ALLOCATED - returned if the page in question has not been allocated.
  - ERR_PAGE_NOT_PRESENT - returned if the page is not currently resident in memory.
  - ERR_PERMISSION_DENIED - returned if any of the permissions requested are not allowed for that page.

I have provided the following functions in the file va_space.c to help you out with debugging:

- pt_display - displays the contents of the entire page table in hexadecimal notation.
- pt_display_entry - displays a single page table entry in hexadecimal notation.
- pt_display_address - displays a virtual or physical address in hexadecimal notation, prefaced by a string.

You must also write an executable program in a file called test_va_space.c that tests all of the functions you have implemented.

Hexadecimal Notation and Manipulating Bits

Hexadecimal notation is particularly helpful in our system because our virtual addresses, physical addresses, and page table entries are all 16 bits long and made up of two 8-bit pieces. What this means is that they can all be represented by 4-digit hexadecimal numbers where the first two digits represent the first 8 bits and the last two digits represent the last eight bits. For example, the virtual memory address 49421 would translate to the 16-bit binary number `1100000100001101`, which would be represented in hexadecimal as `0xc10d`. The page number is represented by the digits `c1`, and the offset is represented by the digits `0d`. These translate to 193 and 13 in decimal notation.

If you want to isolate individual bits, you can use a mask, which is another sequence of bits where the 1's represent the bits you want to keep and the 0's represent the bits you dont. If we want to isolate the offset in a virtual address we would use the mask `0x00ff`:

```
pt_address virtual_address = 0xc10d;
pt_offset offset = virtual_address & 0x00FF; // offset is 0x0d
```

Note the use of the bitwise and operator `&`, which uses the mask to zero out the first eight bits and leave the last eight bits alone.

To isolate higher order bits you can use a shift right operation move the bits in question to the right, discarding any bits that follow them. If we want to isolate the page number in a virtual address we would need to shift it eight bits to the right:

```
pt_address virtual_address = 0xc10d;
pt_index page_num = virtual_address >> 8; // page_num is 0xc1
```

The `>>` operator is used to shift bits to the right. You can use the `<<` operator to shift bits to the left, adding trailing zeroes as you go.

Manipulating individual bits involves the use of bitwise operators. If we want to set the allocated bit in a page table entry, we would need to use the bitwise or operator `|`, along with the appropriate bit.

```
entry = entry | PT_ALLOC; // set the allocated bit
```

To clear that bit we would need to use the `&` operator and the bitwise not operator `~` to create a mask from the bit:

```
entry = entry & ~PT_ALLOC; // clear the allocated bit
```

Note that this keeps every bit the same except the allocated bit.

A number of the functions I have asked you to implement return Boolean values. As there is no built-in Boolean type in C, these functions return values of type `int`, where zero is interpreted as false and any other number is interpreted as true. This is convenient for our purposes as we often want to return "true" when the result of an operation leaves at least one bit set and "false" when all bits are clear:

```
pt_entry entry = table[page_num];
if(entry & PT_ALLOC)
  puts("The allocated bit is set");
```

The bitwise or operator can also be used to add an entry to the page table with multiple permissions set:

```
pt_map(table, page_num, frame_num, PT_READ | PT_WRITE);
```

Testing Your Code

You may be tempted to write your implementation code first, then the test code second. I highly recommend that you work on both of these files at the same time. In fact, there's a few things you can do before implementing any of the functions in `va_space.c`. To make sure your code will compile, add the line `return 0` to every unimplemented function in `va_space.c` that has an `int` return value (you will change this later.) Then write the following code in your `test_va_space.c`:

```
/* <YOUR NAME HERE>
 * CSCI 4100
 * Assignment 7
 * Test code for virtual address space simulation
 */

#include <stdio.h>
#include "va_space.h"

int main() {
  /* Display the page table */
  page_table table;
  pt_display(table);
  puts("");

  /* Display a page table entry */
  pt_index page_num = 0xb1;
  pt_display_entry(table, page_num);
  puts("");

  /* Display an address */
  pt_address virtual_address = 0xb18f;
  pt_display_address("virtual address", virtual_address);
  puts("");

  return 0;
}
```

What this will do is demonstrate how to use the three functions I have already implemented for you. You will not want this code in your final version of the file, but you may use parts of it in the course of your testing.

The next thing I recommend is implementing and testing each of the functions one at a time. Here are some recommendations:

- Start with `pt_init`. Display the the page table after calling it.
- Work on the `pt_set` and `pt_clear` functions next. Display the page table entry you are modifying after each one to see how it changes.
- Work on `pt_dirty`, `pt_present`, and `pt_accessed` and use these along with your `pt_set` and `pt_clear` functions.
- Implement `pt_map` and `pt_allocated` and `pt_not_permitted` and test them with a variety of permissions.
- Implement `pt_translate`, and test all of the scenarios described earlier.

What to Hand In

You must hand in a zip file containing the source files `va_space.h`, `va_space.c`, `test_va_space.c`. Your source files should have comments at the top listing your name, CSCI 4100, Assignment 7, and a brief explanation of what the program does. Download the source files to your local machine, put them into a zip file then upload the zip file to D2L in the dropbox called Assignment 7. See Assignment 6 for instructions on how to do this.