

# CSC 262

## Lab 6: Memory Management

Due Dec. 11

### 1 Overview

In this lab you'll be working on code to simulate a virtual memory manager inside an OS. Some of the questions will ask you to extend code contained in files available on the class website. Note that the lab is due on the eleventh, but the extra credit questions may be turned in later (but no later than the 20th). Do not kill yourself trying to get the extra credit, focus on the required questions first and do the extra credit only if you happen to have extra time on your hands!

### 2 Addresses

Virtual memory is all about addresses. The OS needs to multiplex multiple virtual address spaces onto real physical memory. This lab will be using 32-bit addresses and two-level page tables. Given a 32-bit address and 4K pages, a page number only consumes 20 bits of space. Of course, for reasons of efficiency secondary page tables consist of 4-byte integers. That means that there are 12 extra bits available in each entry (the low-order bits). There are two basic ways of getting at this structure in C. The first is to just treat each entry as an integer and construct functions (or macros) to extract particular bits. However, a more modern approach is to use a so-called bit-packed struct. This is a way of using `struct` semantics over primitive machine types. This has several advantages. First, it lets the programmer be lazier. The compiler gets to figure out all the tedious shifting and masking. Second, it introduces, at the language level, indirection. This indirection makes it easier to add, subtract, and move around bitfields without having to change any code that uses the struct. Consider the following definition for a secondary page table entry:

```
struct pte_int{
    unsigned int page_no    :20;
    unsigned int read_prot  :1;
    unsigned int write_prot:1;
    unsigned int exec_prot  :1;
    unsigned int resident   :1;
} __attribute__((__packed__));
typedef struct pte_int table_entry_t;
```

Ignore the `__attribute__` rigamarole at the end of the struct definition. That is a directive to GCC forcing it to pack the struct as tightly as possible (e.g. into a single int), rather than placing each field in a separate byte.

The colon-delimited numbers after each field declaration specify how many bits each take up. So, in this case, `page_no` is 20 bits long, and all the other fields are single bit flags. This is handy, given an instance of this struct, you can dereference the fields as with any other struct, like so:

```
table_entry_t* entry = get_entry(...);
if(entry->resident){
    //good, page is in memory
}else{
    //bad, need to swap page in
}
```

So, now we have defined a single entry in a secondary page table. Therefore, a secondary page table itself is just an array of `table_entry_t`'s. What goes in the main page table? On a real system, the main page table would contain entries that specified the physical memory location of the secondary page table. Of course, we don't have access to the physical addresses. So we'll just use pointers. Therefore the main page table is just an array of `table_entry_t*`. Each one of which points to a secondary table. In this lab, you will be writing code to manipulate and use these simulated page tables.

**Q1:** In GNU C, the type for a 64 bit integer is `long long`. Write out the bit-packed struct definition for an entry on a 64-bit machine with 8K pages. Assume that the flags needed are the same as on the 32-bit machine.

## 2.1 Bit Widths

Messing around with addresses requires manipulation of integers at the bit level. This is something that C programmers routinely do, either for reasons of efficiency or just simply to show off. There are 4 basic operations:

1. `&`, bitwise-AND. `&` returns the result of ANDing two values together bit-wise. AND is often used to mask off values (i.e. set irrelevant bits to 0).
2. `|`, bitwise-OR. `|` returns the result of ORing two values together bit-wise. OR is often used to combine several bit-fields together into one complete value.
3. `<<`, left shift. Shifting a value left means shifting all the bits toward the most-significant position. This is often used to move values around into the correct position before using an `|` to combine them.
4. `>>`, right shift. Shifting a value right means shifting the bits toward the least-significant position. This can be used to move values around, as well as moving masked-off high bits down to be interpreted as a value.

Most manipulations combine a logical operator and a shifting operator. For example, to extract the 2 high-order bits from an int:

```
int in = ...;
int hi_bits = in & 0xC0000000;
hi_bits = hi_bits >> 30;
```

The second line masks off the lower 30 bits of the value stored in `in`. This works because  $0 \wedge x = 0$  and  $1 \wedge x = x$ . Therefore the lower 30 bits are all set to 0 regardless of their value, and the upper 2 bits are preserved. At this point, we're only half done. Although we've preserved the values of the upper 2 bits, we need to shift them down, if we're going to interpret them as an integer value. That's what the last line does.

Similarly, these operations can be used to combine partial values into one complete value. The following code assembles a 20-bit value, a 10 bit value and a 2 bit value into a 32-bit value. This code assumes that the values of interest are all stored in the lower bits.

```
int low20 = ...;
int low10 = ...;
int low2 = ...;
int all32 = (low20 << 12) | (low10 << 2);
all32 = all32 | low2;
```

The fourth line combines `low20` and `low10` to create a 32-bit value. First, `low20` is shifted left so that it sits in the upper 20 bits of a 32-bit value. Second, `low10` is shifted left by 2 so that it sits in the middle 10 bits of the 32-bit value. These two values are ORed together to create a mostly-complete 32-bit value. Last, the low 2 bits are ORed in.

**Q2:** Extend the code contained in `simplePage.c` by implementing the methods: `get_main_idx`, `get_sec_idx`, `get_page_offs`. These methods each extract a different part of the virtual address. Remember that this code simulates a two-level page table with 4K pages, therefore both the main and secondary index are 10 bits large.

*The following questions all involve a 64-bit machine with 8K pages, using 3 level paging.*

- Q3:** Write code to extract the page offset from a `long long`.
- Q4:** If the main page table is to be one page large, and each entry is 8 bytes large, write code to extract the main page index from a `long long`.
- Q5:** If the secondary page table is 1MB large, write code to extract the secondary page table index from a `long long`.
- Q6:** How many entries are contained in the tertiary page table?

### 3 Using a page table

A page table is for translating virtual addresses into physical addresses. The MMU tears apart a virtual address and uses it to generate the physical address of the data in actual memory. The system

we're using simulates two-level paging with 4K pages. This means that both the main index and the secondary index are 10 bits large. The tables themselves are simply arrays of entries. In our system, the entries in the main page table are `table_entry_t*`. And each of these points to a secondary table that contains `table_entry_t`'s. The code defines the constant `INVALID_ENTRY` which is the value for an entry in the main table that doesn't map anywhere. Translation for a resident page is simply a matter of walking the page table until you hit an entry, and then use the entry to rewrite the address.

The code you will be extending constructs a page table for you to test your code on. When you invoke the program, it should be like so: `program num_physical_pages num_virtual_pages`.

**Q7:** Extend the code contained in `translateEntry.c` with your code from Q2. Then implement the `translate_addr` function. This function takes a virtual address (as a `void*`) and a pointer to a main page table, and returns the physical address indicated by the virtual address. For each successful translation, `translate_addr` should output the following:

```
"Virtual: 0x<virtual_addr> -> 0x<phys_addr>\n"
```

**Q8:** Extend the code contained in `printEntry.c` to implement the `print_entry` function. This function is very similar to the `translate_addr` function, except it prints out resident information. `print_entry` should output the same string as `translate_addr` as well as one line outputting the page number and another line for the resident information, like so:

```
"Virtual: 0x<virtual_addr> -> 0x<phys_addr>\n"  
"page_no = 0x<page_no>\n"  
"resident = <resident bit>\n"
```

## 4 Swapping Pages

It is often the case that all the address spaces of running programs are larger than main memory. When this occurs, the OS time shares main memory by writing out pages to disk (swapping out) to make room in main memory. Ideally, the OS should swap out those pages that are least likely to be used. Of course, the OS can't predict the future so virtual memory managers use various schemes to "guess" at future program behavior from past behavior.

### 4.1 FIFO

The FIFO algorithm is fairly simple. As pages are loaded into main memory, they are put into a queue. This queue ends up sorting pages by age, when a page must be swapped out, the FIFO algorithm chooses the oldest page (i.e. the one at the head of the queue). This queue is external to the page table, and is part of the metadata that the virtual memory manager keeps around. It would be possible to implement the queue in the standard fashion, by malloc-ing nodes as pages are added. However, this is grossly inefficient. Because the amount of physical memory available does

not change, the memory manager can actually pre-allocate the queue (one slot for each physical page). Rather than a linked list, the memory manager's queue can be implemented as an array (and using techniques similar to the bounded buffer to keep track of the head and tail).

For this next question, you will be extending code to implement FIFO page replacement. First, there is a new structure corresponding to FIFO elements defined thus:

```
typedef struct{
    table_entry_t* entry;
    unsigned int virtual_addr;
} fifo_ent;
```

This entry contains a reference to a page entry as well as a record of the virtual page it corresponds to (this makes producing output much easier). The fifo itself is just an array of these guys `fifo_ent* fifo`. Because the array is being managed like a queue, there are three integers that need to be maintained as well:

- `fifo_len` contains the length of the queue (the number of physical pages).
- `fifo_head` contains the index of the head of the queue.
- `fifo_tail` contains the index of the tail of the queue.

Conventionally, the head of the queue points to the oldest page, and the tail the newest. Therefore, when swapping the head should be swapped out and the newly swapped in page should be appended at the tail of the queue. Because this queue is actually a global array, appending is just overwriting the values at a particular index and advancing the tail "pointer". Additionally, both `fifo_head` and `fifo_tail` need to be advanced in a circular manner. That is, when they reach the end of the array `fifo_len - 1` they need to be wrapped around to the front 0.

There are three functions that the rest of the simulator uses to interact with the swapping system. `init_vm` initializes the data structures and should be called at start up (as soon as the number of physical and virtual pages are known). `add_page` is a function that appends a new page to the queue. It is called by the simulator when it is building the page table. Note that this function should only be called with **resident** pages! Last, `swap_in` is the function that is called when the user wants to translate a non-resident page. `swap_in` has to make room for the new page and then swap it in (because we're not actually managing a swap partition, swapping in is just printing to the screen). `swap_in` has two jobs: first, it must make room for the requested page by swapping out a resident page according to the FIFO algorithm; second, `swap_in` needs to swap in the requested page and update the FIFO and page table accordingly. Note, that you may have to create additional helper functions to implement these three functions (particularly `swap_in`). `swap_in` itself should be called from within `translate_addr` when a non-resident page is discovered. Because we're dealing with a fully occupied physical memory, all page faults will result in swapping.

**Q9:** Extend the code contained in `FifoSwap.c` to implement FIFO page replacement. Use your code from question 7 as necessary. When `swap_in` is called, it should output its replacement decision like so:

Swapping out virtual page <virtual page> to free physical page <phys pa

- Q10: *Extra Credit:*** Extend and modify the code from Q9 to implement LRU via move-to-front. With move-to-front, every time a page is touched, it is moved to the front of the queue (in this case, the page to be evicted sits at the tail, rather than the front).
- Q11: *Extra Credit:*** Extend and modify the code from Q9 to implement CLOCK (one-handed) replacement.