Section 8: Address Translation

March 9, 2018

Contents

1 Vocabulary

- **Virtual Memory** - Virtual Memory is a memory management technique in which every process operates in its own address space, under the assumption that it has the entire address space to itself. A virtual address requires translation into a physical address to actually access the system's memory.
- **Memory Management Unit** - The memory management unit (MMU) is responsible for translating a process' virtual addresses into the corresponding physical address for accessing physical memory. It does all the calculation associating with mapping virtual address to physical addresses, and then populates the address translation structures.
- **Address Translation Structures** - There are two kinds you learned about in lecture: segmentation and page tables. Segments are linearly addressed chunks of memory that typically contain logically-related information, such as program code, data, stack of a single process. They are of the form (s,i) where memory addresses must be within an offset of i from base segment s. A page table is the data structure used by a virtual memory system in a computer operating system to store the mapping between virtual addresses and physical addresses. Virtual addresses are used by the accessing process, while physical addresses are used by the hardware or more specifically to the RAM.
- **Inverted Page Table** - The inverted page table scheme uses a page table that contains an entry for each phiscial frame, not for each logical page. This ensures that the table occupies a fixed fraction of memory. The size is proportional to physical memory, not the virtual address space. The inverted page table is a global structure – there is only one in the entire system. It stores reverse mappings for all processes. Each entry in the inverted table contains has a tag containing the task id and the virtual address for each page. These mappings are usually stored in associative memory (remember fully associative caches from 61C?). Associatively addressed memory compares input search data (tag) against a table of stored data, and returns the address of matching data. They can also use actual hash maps.
- **translation lookaside buffer (TLB)** - A translation lookaside buffer (TLB) is a cache that memory management hardware uses to improve virtual address translation speed. It stores virtual address to physical address mappings, so that the MMU can store recently used address mappings instead of having to retrieve them mutliple times through page table accesses.

2 Problems

2.1 Conceptual Questions

If the physical memory size (in bytes) is doubled, how does the number of bits in each entry of the page table change?

increases by 1 bit. Assuming the page size remains the same, there are now twice as many physical pages, so the physical page number needs to expand by 1 bit.

If the physical memory size (in bytes) is doubled, how does the number of entries in the page map change?

no change. The number of entries in the page table is determined by the size of the virtual address and the size of a page – it's not affected by the size of physical memory.

If the virtual memory size (in bytes) is doubled, how does the number of bits in each entry of the page table change?

no change. The number of bits in a page table entry is determined by the number of control bits (usually 2: dirty and resident) and the number of physical pages – the size of each entry is not affected by the size of virtual memory.

If the virtual memory size (in bytes) is doubled, how does the number of entries in the page map change?

the number of entries doubles. Assuming the page size remains the same, there are now twice as many virtual pages and so there needs to be twice as many entries in the page map.

If the page size (in bytes) is doubled, how does the number of bits in each entry of the page table change?

each entry is one bit smaller. Doubling the page size while maintaining the size of physical memory means there are half as many physical pages as before. So the size of the physical page number field decreases by one bit.

If the page size (in bytes) is doubled, how does the number of entries in the page map change?

there are half as many entries. Doubling the page size while maintaining the size of virtual memory means there are half as many virtual pages as before. So the number of page table entries is also cut in half.

The following table shows the first 8 entries in the page map. Recall that the valid bit is 1 if the page is resident in physical memory and 0 if the page is on disk or hasn't been allocated.

Valid Bit	Physical Page
0	7
1	9
0	3
1	2
1	5
0	5
0	4
1	1

If there are 1024 bytes per page, what is the physical address corresponding to the hexadecimal virtual address 0xF74?

the virtual page number is 3 with a page offset of 0x374. Looking up page table entry for virtual page 3, we see that the page is resident in memory (valid bit = 1) and lives in physical page 2. So the corresponding physical address is `(2<<10)+0x374 = 0xB74`

2.2 Page Allocation

Suppose that you have a system with 8-bit virtual memory addresses, 8 pages of virtual memory, and 4 pages of physical memory.

How large is each page? Assume memory is byte addressed.

32 bytes

Suppose that a program has the following memory allocation and page table.

Memory Segment	Virtual Page Number	Physical Page Number
N/A	000	NULL
Code Segment	001	10
Heap	010	11
N/A	011	NULL
N/A	100	NULL
N/A	101	NULL
N/A	110	NULL
Stack	111	01

What will the page table look like if the program runs the following function? Page out the least recently used page of memory if a page needs to be allocated when physical memory is full. Assume that the stack will never exceed one page of memory.

What happens when the system runs out of physical memory? What if the program tries to access an address that isn't in physical memory? Describe what happens in the user program, the operating system, and the hardware in these situations.

```
#define PAGE_SIZE 1024; // replace with actual page size

void helper(void) {
    char *args[5];
    int i;
    for (i = 0; i < 5; i++) {
        // Assume malloc allocates an entire page every time
        args[i] = (char*) malloc(PAGE_SIZE);
    }
    printf("%s", args[0]);
}
```

Memory Segment	Virtual Page Number	Physical Page Number
N/A	000	NULL
Code Segment	001	10
Heap	010	11
Heap	011	00
N/A	100	NULL
N/A	101	NULL
N/A	110	NULL
Stack	111	01

Memory Segment	Virtual Page Number	Physical Page Number
N/A	000	NULL
Code Segment	001	10
Heap	010	PAGEOUT
Heap	011	00
Heap	100	11
N/A	101	NULL
N/A	110	NULL
Stack	111	01

Memory Segment	Virtual Page Number	Physical Page Number
N/A	000	NULL
Code Segment	001	10
Heap	010	PAGEOUT
Heap	011	PAGEOUT
Heap	100	11
Heap	101	00
N/A	110	NULL
Stack	111	01

2.3 Address Translation

Consider a machine with a physical memory of 8 GB, a page size of 8 KB, and a page table entry size of 4 bytes. How many levels of page tables would be required to map a 46-bit virtual address space if every page table fits into a single page?

Since each PTE is 4 bytes and each page contains 8KB, then a one-page page table would point to 2048 or 2^{11} pages, addressing a total of $2^{11} * 2^{13} = 2^{24}$ bytes.

Depth 1 = 2^{24} bytes
Depth 2 = 2^{35} bytes
Depth 3 = 2^{46} bytes
So in total, 3 levels of page tables are required.

List the fields of a Page Table Entry (PTE) in your scheme.

Each PTE will have a pointer to the proper page, PPN, plus several bits read, write, execute, and valid. This information can all fit into 4 bytes, since if physical memory is 2^{33} bytes, then 20 bits will be needed to point to the proper page, leaving ample space (12 bits) for the information bits.

Without a cache or TLB, how many memory operations are required to read or write a single 32-bit word?

Without extra hardware, performing a memory operation takes 4 actual memory operations: 3 page table lookups in addition to the actual memory operation.

With a TLB, how many memory operations can this be reduced to? Best-case scenario? Worst-case scenario?

Best-case scenario: 2 memory lookups. once in TLB, once for actual memory operation. Worst-case scenario: 5 memory lookups. once in TLB + 3 page table lookups in addition to the actual memory operation.

The pagemap is moved to main memory and accessed via a TLB. Each main memory access takes 50 ns and each TLB access takes 10 ns. Each virtual memory access involves:
- mapping VPN to PPN using TLB (10 ns)
- if TLB miss: mapping VPN to PPN using page map in main memory (50 ns)
- accessing main memory at appropriate physical address (50 ns)

Assuming no page faults (i.e. all virtual memory is resident) what TLB hit rate is required for an average virtual memory access time of 61ns.

```
(10+50)*x+(1-x)*(50+10+50) = 61

solve for x gives x = .98 = 98% hit rate
```

Assuming a TLB hit rate of .50, how does the average virtual memory access time of this scenario compare to no TLB?

2.4 Inverted Page Tables

Why IPTs? Consider the following case:
- 64-bit virtual address space
- 4 KB page size
- 512 MB physical memory

How much space (memory) needed for a single level page table? Hint: how many entries are there? 1 per virtual page. What is the size of a page table entry? access control bits + physical page #.

```
One entry per virtual page

 - 2^64 addressable bytes / 2^12 bytes per page = 2^52 page table entries

 Page table entry size

 - 512 MB physical memory = 2^29 bytes
- 2^29 bytes of memory/2^12 bytes per page = 2^17 physical pages
- 17 bits needed for physical page number
 Page table entry = ~4 bytes
```

```
- 17 bit physical page number = ~3 bytes
- Access control bits = ~1 byte

Page table size = page table entry size * # total entries

 2^52 page table entries * 2^2 bytes = 2^54 bytes (16 petabytes)

i.e. A WHOLE LOT OF MEMORY
```

How about multi level page tables? Do they serve us any better here?

What is the number of levels needed to ensure that any page table requires only a single page (4 KB)?

```
 Assume page table entry is 4 bytes
 4 KB page / 4 bytes per page table entry =
1024 entries
 10 bits of address space needed
 ceiling(52/10) = 6 levels needed

7 memory accesses to do something? SLOW!!!
```

Linear Inverted Page Table

What is the size of of the hashtable? What is the runtime of finding a particular entry?

Assume the following:

- 16 bits for process ID
- 52 bit virtual page number (same as calculated above)
- 12 bits of access information

```
 add up all bits = 80 bits = 10 bytes
- 10 bytes * # of physical pages = 10 * 2^17 = 2^3 * 2^17 = 1 MB

Iterate through all entries.
For each entry in the inverted page table,
compare process ID and virtual page
number in entry to the requested process
ID and virtual page number
Extremely slow. must iterate through 2^17 entries of the hash table
worst-case scenario.
```

Hashed Inverted Page Table

What is the size of of the hashtable? What is the runtime of finding a particular entry?

Assume the following:

- 16 bits for process ID
- 52 bit virtual page number (same as calculated above)
- 12 bits of access information

```
 add up all bits = 80 bits = 10 bytes
- 10 bytes * # of physical pages = 10 * 2^17 = 2^3 * 2^17 = 1 MB
```

```
Linear inverted page tables require too many
memory accesses.
- Keep another level before actual inverted page
table (hash anchor table)
 Contains a mapping of process ID and virtual page
number to page table entries
- Use separate chaining for collisions
- Lookup in hash anchor table for page table entry
 Compare process ID and virtual page number
- if match, then found
- if not match, check the next pointer for another page table
entry and check again

So, with a good hashing scheme and a hashmap proportional to
the size of physical memory, O(1) time. Very efficient!
```

2.5 Page Fault Handling for Pages Only On Disk

The page table maps vpn to ppn, but what if the page is not in main memory and only on disk? Think about structures/bits you might need to add to the page table/OS to account for this. Write pseudocode for a page fault handler to handle this.

```
Have a disk map structure that contains a disk address, and process id
for each ppn. Have each process be associated with a page table. Each of
these two tables describes the entire virtual memory address space, and
physical memory address space, respectively. The page table identifies
which ppn is associated with which vpn, and contains bits such as used,
modified, and presence to describe whether or not it is in physical
memory or only on disk. The disk map the corresponding disk address for
each ppn. The entire address space is on the disk, but only a subset of
it is resident in main memory.



page fault:

index: vpn, value: ppn

frame table:

index: ppn, value: process id, disk address



page table entry

p|u|m|f
```

```
p = presence flag
u = used flag
m = modified flag
f = page frame (ppn)

disk table entry

pid | disk address | bits/metadata for replacement algorithm

Page Fault Handler Pseudocode:

Using the replacement algorithm, iterate through the disk table and get
the number of a frame that will be used for the incoming page

Swap the page currently in that frame to its slot on the disk

Swap the requested page from its slot on disk into the above frame

Update the page table entry so that vpn -> ppn and the presence flag is
set to true (since it’s now in main memory)

return
```

2.6 Wait and Exit

This problem is designed to help you with implementing wait and exit in your project. Recall that wait suspends execution of the parent process until the child process specified by the parameter id exits, upon which it returns the exit code of the child process. In Pintos, there is a 1:1 mapping between processes and threads.

2.6.1 Thinking about what you need to do

"wait" requires communication between a process and its children, usually implemented through shared data. The shared data might be added to struct thread, but many solutions separate it into a separate structure. At least the following must be shared between a parent and each of its children: - Child's exit status, so that "wait" can return it. - Child's thread id, for "wait" to compare against its argument. - A way for the parent to block until the child dies (usually a semaphore). - A way for the parent and child to tell whether the other is already dead, in a race-free fashion (to ensure that their shared data can be freed).

2.6.2 Code

Data structures to add to thread.h for waiting logic:

```
Pseudocode:
process_wait (tid_t child_tid) {
    iterate through list of child processes
    if child process tid matches tid parameter, call sema_down
    on the semaphore associated with that child process (when
    child process exits, it will sema_up on that same semaphore,
    waking up the parent process)
    --- after waking ---
    set exit code to terminated child process's exit code
    decrease ref_cnt of child //why? need to free memory, "zombie processes"
    return exit_code
}
process_exit (void) {
    sema_up on semaphore that parent might be sleeping on
    remove all child processes from child process list
    decrement ref_cnt
}

Code:
struct wait_status
  {
    struct list_elem elem;      /* 'children' list element. */
    struct lock lock;           /* Protects ref_cnt. */
    int ref_cnt;                /* 2=child and parent both alive,
   1=either child or parent alive,
   0=child and parent both dead. */
    tid_t tid;                  /* Child thread id. */
    int exit_code;              /* Child exit code, if dead. */
    struct semaphore dead;      /* 1=child alive, 0=child dead. */
  };
```

```
struct wait_status *wait_status; /* This process's completion state. */
struct list children;            /* Completion status of children. */
```

Implement wait:

```
     - Find the child in the list of shared data structures.
       (If none is found, return -1.)
     - Wait for the child to die, by downing a semaphore in the
       shared data.
     - Obtain the child's exit code from the shared data.
     - Destroy the shared data structure and remove it from the
       list.
     - Return the exit code.
```

Implement exit:

```
     - Save the exit code in the shared data.
     - Up the semaphore in the data shared with our parent
       process (if any).  In some kind of race-free way (such
       as using a lock and a reference count or pair of boolean
       variables in the shared data area), mark the shared data
       as unused by us and free it if the parent is also dead.
     - Iterate the list of children and, as in the previous
       step, mark them as no longer used by us and free them if
       the child is also dead.
         - Terminate the thread.
```