Swasti Mishra

Dr. Beck

COSC 360

21 November 2022

Midterm 2

1. **Explain in detail the steps in translating an 18-bit Virtual Address to a 32-bit Physical Address in a system with a two-level page table where the page directory is 32 entries and the size of each page table is 128 entries. Include the length of the bit fields that make up of the Virtual Address and the Physical address.**

1. List all your variables:

Virtual Address Size	= 18 bits
Page Directory Bits	= 5 bits (2^5 = 32)
Page Table Entry Bits	= 7 bits (2^7 = 128)
Page Offset	= 18 – 7 – 5 = 6 bits

2. Make a diagram of your virtual address format:

Page Directory: 5 bits	Page Table Entry: 7 bits	Page Offset: 6 bits

3. Make a diagram of your physical address format:

Frame Number: 28 bits	Frame Offset: 4 bits

4. Discuss the results:

In order to convert our 18-bit virtual address to a 32-bit physical address, we had to calculate the size of our page directory, page table, and page offset. Now that we have done that, we can convert an 18-bit virtual address by checking the first 5 bits for the appropriate page directory, which will contain the address for the page table.

We then go to that address and check the next nine bits, which will result in an entry that gives us the frame number. Once we add the frame number to the frame offset, we will have the frame address, which is our physical address.

2. **When an operating system switches between kernel-supported threads executing within the same process there is a reduction in performance which does not occur when there is a switch between threads supported at user level.**

    a) **Explain the difference between kernel-supported threads and user level threads which explains this difference.**

    User-level threads are, as the name suggests, begun by the user. The OS does not micromanage their performance in the same way it does for kernel-level threads. User-level threads run within processes and are managed within their respective process's time slices. Kernel threads, however, are scheduled and managed by the OS. This means that to swap between them, the OS must perform a context switch by updating registers, stack pointers, etc., which leads to the reduction in performance. User threads, on the other hand, do not require a system call for switching between them, which means they are much faster and do not lead to a reduction in performance.

    b) **Explain why threads supported at user level cannot support stack growth or protect against stack overflow.**

    While user-level threads can access the stack, it is usually through specific, thread-local storage that has been allocated for the task. This storage is protected, meaning it does not access the rest of the OS's stack and therefore does not interfere with other processes. Further, this storage is allocated at the beginning of the task and does not change size. For this reason, user-level threads cannot affect overall stack growth or overflow.

Occasionally, if necessary, user-level threads can access the stack, but only with a lot of protections. The OS will trigger an interrupt, allow the thread to access the stack, and then place the user thread back in the ready queue. This is another reason the user threads can't impact the stack in any meaningful way- any operations they "do" on the stack are temporary until the OS returns to the kernel threads.

3. **Explain in detail the implementation of fork() using copy-on-write, including changes in the page table entries and management of the Translation Lookaside Buffer.**

The goal of the fork system call is to create a duplicate process. The original process is called the parent process, and the new process is called the child process. In copy-on-write, the parent process and the child process originally share their pages in memory. These pages are marked with copy-on-write, meaning that if the parent or child modify one of the pages, then a copy of the page is created, and the changes will be applied to the newly created page. In this way, the new process gets pages with the modifications, and the old process is unaffected.

The problem with copy-on-write, however, is that it can require a large overhead in terms of memory. The table that manages pages requires two pieces of information- the frame number where the page is located, and the address within the frame. To manage this information, the Translation Lookaside Buffer was developed, which is a high-speed hardware address translation cache. The Translation Lookaside Buffer checks if a virtual to physical address is stored, and if it is, the processer gets a TLB hit, and the frame number and address are supplied. If the cache does not find the page and frame number, the processor gets a TLB miss, and the processor looks for the page number in the main memory, and then updates the Translation Lookaside Buffer using either first in first out, least recently used, or some other cache implementation.

4. **A single program which sequentially processes one-word elements in a large single loop array (reading, updating and then writing each element in turn).**

   a) **Explain why the program would experience a low rate in a Translation Look Aside Buffer that utilizes the Second Chance or Clock LRU approximation algorithm.**

   While LRU is a common eviction approach in Translation Lookaside Buffers, it would not be a suitable way to set up the cache in this instance. This is because the least recently used policy does not always work when the loop of the program is larger than the cache. This may seem counterintuitive- the whole point of the cache is that it is supposed to be smaller than the actual number of references being made to it. But if in the loop, the eviction policy continually the oldest pages even though they are supposed to be referenced later in the loop, the Translation Lookaside Buffer will always have a cache miss. This is why a random replacement policy would be better for this program. With a random replacement policy, the TLB would occasionally get a cache hit, while LRU will always fail.

   b) **Explain the data structure and operation of the Clock algorithm.**

   The clock algorithm considers whether a page in the cache has been used in the time since the last time the replacement policy was run. It does this by adding a "second chance" bit to the memory frames. This bit is originally set to zero by the hardware, and when a reference is made to a page inside the frame, the bit is swapped to one by the OS. This means that when the replacement policy is run, it sees the one, and swaps it to a zero, therefore giving the frame reference a "second chance." In this way, a page with a second chance bit of one will never be replaced while other pages have a zero. If all other pages also have a one, the Translation Lookaside Buffer will replace the page it started at.

5. **In Section 29.1 of the course textbook "Operating Systems in Three Easy Pieces" code which implements Approximate Counters is given in Figure 29.4.**

a) **Why does this implementation not work if the amt (amount) argument to the update() function can take negative values?**

When the amount argument takes negative values, the threshold S isn't reached. If that doesn't happen, the update function doesn't run, and the local values aren't updated to the global values. Further, even if the update function was changed to accommodate this, if the amount argument is negative, it would negate the global values and ruin the overall count.

b) **Extend this implementation with a function int getexact(counter *c) which always returns the globally exact value of the counter rather than an approximation. Your extension should not change the current implementation of get() or update().**

```
int getexact( counter *c ){
     pthread_mutex_lock(&c->glock) ;
     int val = c-> global ;
     pthread_mutex_unlock( &c->glock ) ;
     int i = 0 ;
     for ( i < NUMCPUS ; i++ ){
          pthread_mutex_lock( &c->llock[i] ) ;
          c->local[i] = val + c->local[i] ;
          pthread_mutex_unlock( &c->llock[i] ) ;
     }
     return val ;
}
```

c) **How would the mix of operations (get vs. getexact) affect the performance of your extended implementation.**

This function would require more overhead because of the for loop, but that is a tradeoff that happens because the value returned is not approximate. Also, using the mutex three times weighs on performance because it's a system call and will make this solution pretty slow.