Section 9: Calling Conventions, Cache and TLB, Demand Paging

CS162

October 18, 2016

Contents

1 Calling Conventions and Argument Passing 2
  1.1 Calling Conventions ................................................................. 2
  1.2 Argument Passing ................................................................. 2

2 Vocabulary 3

3 Problems 4
  3.1 Caching ................................................................. 4
  3.2 Demand Paging ................................................................. 5
  3.3 Virtual Memory ................................................................. 6
1 Calling Conventions and Argument Passing

1.1 Calling Conventions

Sketch the stack frame of helper before it returns.

```c
void helper(char* str, int len) {
    char word[len];
    strncpy(word, str, len);
    printf("%s", word);
    return;
}
```

```c
int main(int argc, char *argv[]) {
    char* str = "Hello World!";
    helper(str, 13);
}
```

1.2 Argument Passing

Fill out the following code to copy integer arguments onto the stack. You can change the ESP by modifying if_->esp.

```c
void populate_stack_args(int argc, uint32_t *argv, struct intr_frame *if_) {
    // The stack pointer is available in if_->esp (which has the type uint32_t*)
    int i;

    for (______________________________) {
        if_->esp = _________________;
        _________________ = argv[i];
    }
}
```
2 Vocabulary

- **Calling Conventions** - Describe the interface of called code. Define the order in which atomic parameters are allocated, how parameters are passed, which registers the callee must preserve for the caller, how the stack is prepared and restored, etc.

- **%esp, %ebp** - In a stack frame (on x86 architecture), the %esp register holds the address of the last thing on the stack. The %esp pointer gets moved as things get added or removed from the stack. The %ebp register stores the position of the beginning of the current stack frame.

- **Compulsory Miss** The miss that occurs on the first reference to a block. There’s essentially nothing that you can do about this type of miss, but over the course of time, compulsory misses become insignificant compared to all the other memory accesses that occur.

- **Capacity Miss** This miss occurs when the cache can’t contain all the blocks that the program accesses. Usually the solution to capacity misses is to increase the cache size.

- **Conflict Miss** Conflict misses occur when multiple memory locations are mapped to the same cache location. In order to prevent conflict misses, you should either increase the cache size or increase the associativity of the cache.

- **Coherence Miss** Coherence misses are caused by external processors or I/O devices that updates what’s in memory.

- **Working set** The subset of the address space that a process uses as it executes. Generally we can say that as the cache hit rate increases, more of the working set is being added to the cache.
3 Problems

3.1 Caching

An up-and-coming big data startup has just hired you to help design their new memory system for a byte addressable system. Suppose the virtual and physical memory address space is 32 bits with a 4KB page size.

First, you create 1) a direct mapped cache and 2) a fully associative cache of the same size that replaces the least recently used pages when the cache is full. You run a few tests and realize that the fully associative cache performs much worse than the direct mapped cache. What's a possible access pattern that could cause this to happen?

Instead, your boss tells you to build an 8KB 2-way set associative cache with 64 byte cache blocks. How would you split a given virtual address into its tag, index, and offset numbers?

You finish building the cache, and you want to show your boss that there was a significant improvement in average read time. Suppose your system uses a two level page table to translate virtual addresses and your system uses the cache for the translation tables and data. Each memory access takes 50ns, the cache lookup time is 5ns, and your cache hit rate is 90%. What is the average time to read a location from memory?
3.2 Demand Paging

Your boss has been so impressed with your work designing the caching that he has asked for your advice on designing a TLB to use for this system. Suppose you know that there will only be 4 process running at the same time, each with a maximum size of 512MB and a working set size of 256KB. Assuming the same system as the previous problem (32 bit virtual and physical address space, 4KB page size), what is the minimum amount of TLB entries that your system would need to support to be able to map the working set size for one process? What happens if you have more entries? What about less?

Suppose you run some benchmarks on the system and you see that the system is utilizing over 99% of its paging disk IO capacity, but only 10% of its CPU. What is a combination of the of disk space and memory size that can cause this to occur? Assume you have TLB entries equal to the answer from the previous part.

Out of increasing the size of the TLB, adding more disk space, and adding more memory, which one would lead to the largest performance increase and why?
### 3.3 Virtual Memory

`vmstat` is a Linux performance debugging tool that provides information about virtual memory on your system. When you run it, the output looks like this:

```
$ vmstat 1
procs -----------memory---------- ---swap-- -----io---- -system-- ------cpu-----
 r b swpd free buff cache si so bi bo in cs us sy id wa st
 1 0 0 174184 1007372 96316 49 642 3095 678 123 128 0 1 99 0 0
 0 0 0 174240 1007372 96316 0 0 0 0 48 88 0 0 100 0 0
 0 0 0 174240 1007372 96316 0 0 0 0 33 75 0 0 100 0 0
 0 0 0 174240 1007372 96316 0 0 0 0 32 73 0 0 100 0 0
```

The `1` means “recompute the stats every 1 second and print them out”. The first line contains the average values since boot time, while the second line contains the averages of the last second (current averages). Here’s a reference for what each one of the columns means.

**Procs**
- `r`: The number of runnable processes (running or waiting for run time).
- `b`: The number of processes in uninterruptible sleep.

**Memory**
- `swpd`: the amount of virtual memory used.
- `free`: the amount of idle memory.
- `buff`: the amount of memory used as buffers.
- `cache`: the amount of memory used as cache.
- `inact`: the amount of inactive memory. (-a option)
- `active`: the amount of active memory. (-a option)

**Swap**
- `si`: Amount of memory swapped in from disk (/s).
- `so`: Amount of memory swapped to disk (/s).

**IO**
- `bi`: Blocks received from a block device (blocks/s).
- `bo`: Blocks sent to a block device (blocks/s).

**System**
- `in`: The number of interrupts per second, including the clock.
- `cs`: The number of context switches per second.

**CPU**
- These are percentages of total CPU time.
  - `us`: Time spent running non-kernel code. (user time, including nice time)
  - `sy`: Time spent running kernel code. (system time)
  - `id`: Time spent idle. Prior to Linux 2.5.41, this includes IO-wait time.
  - `wa`: Time spent waiting for IO. Prior to Linux 2.5.41, included in idle.
  - `st`: Time stolen from a virtual machine. Prior to Linux 2.6.11, unknown.
Take a look at these 3 programs (A, B, C).

```c
char *buffer[4 * (1L << 20)];
int A(int in) {
    // "in" is a file descriptor for a file on disk
    while (read(in, buffer, sizeof(buffer)) > 0);
}

int B() {
    size_t size = 5 * (1L << 30);
    int *x = malloc(size);
    memset(x, 0xCC, size);
}

sem_t sema;
pthread_t thread;
void *foo() { while (1) sem_wait(&sema); }

int C() {
    pthread_create(&thread, NULL, foo, NULL);
    while (1) sem_post(&sema);
}
```

I ran these 3 programs one at a time, but in a random order. What order did I run them in? Can you tell where (in the vmstat output) one program stopped and another started? Explain.

I ran these 3 programs one at a time, but in a random order. What order did I run them in? Can you tell where (in the vmstat output) one program stopped and another started? Explain.

I ran these 3 programs one at a time, but in a random order. What order did I run them in? Can you tell where (in the vmstat output) one program stopped and another started? Explain.
<table>
<thead>
<tr>
<th>procs</th>
<th>memory</th>
<th>swap</th>
<th>io</th>
<th>system</th>
<th>cpu</th>
</tr>
</thead>
<tbody>
<tr>
<td>r</td>
<td>b</td>
<td>swpd</td>
<td>free</td>
<td>buff</td>
<td>cache</td>
</tr>
<tr>
<td>0 0</td>
<td>684688</td>
<td>25216</td>
<td>1822136</td>
<td>60860</td>
<td>75</td>
</tr>
<tr>
<td>1 0</td>
<td>684688</td>
<td>25268</td>
<td>1822136</td>
<td>60868</td>
<td>0 0</td>
</tr>
<tr>
<td>1 0</td>
<td>684688</td>
<td>25268</td>
<td>1822136</td>
<td>60868</td>
<td>0 0</td>
</tr>
<tr>
<td>1 0</td>
<td>684688</td>
<td>25268</td>
<td>1822136</td>
<td>60868</td>
<td>0 0</td>
</tr>
<tr>
<td>1 0</td>
<td>684688</td>
<td>25268</td>
<td>1822136</td>
<td>60868</td>
<td>0 0</td>
</tr>
<tr>
<td>0 0</td>
<td>684688</td>
<td>25612</td>
<td>1822136</td>
<td>60868</td>
<td>0 0</td>
</tr>
<tr>
<td>0 0</td>
<td>684688</td>
<td>25612</td>
<td>1822136</td>
<td>60868</td>
<td>0 0</td>
</tr>
<tr>
<td>3 0</td>
<td>694520</td>
<td>18544</td>
<td>3212</td>
<td>45040</td>
<td>64</td>
</tr>
<tr>
<td>1 1</td>
<td>1285828</td>
<td>20560</td>
<td>128</td>
<td>580</td>
<td>592440</td>
</tr>
<tr>
<td>3 0</td>
<td>1866176</td>
<td>21492</td>
<td>128</td>
<td>2132</td>
<td>0 578404</td>
</tr>
<tr>
<td>0 0</td>
<td>2350636</td>
<td>17820</td>
<td>136</td>
<td>2640</td>
<td>0 487732</td>
</tr>
<tr>
<td>1 1</td>
<td>2771016</td>
<td>22168</td>
<td>544</td>
<td>4360</td>
<td>2072</td>
</tr>
<tr>
<td>0 0</td>
<td>697036</td>
<td>1922160</td>
<td>560</td>
<td>9712</td>
<td>1516</td>
</tr>
<tr>
<td>0 0</td>
<td>697032</td>
<td>1921696</td>
<td>564</td>
<td>10096</td>
<td>28</td>
</tr>
<tr>
<td>1 0</td>
<td>696980</td>
<td>878128</td>
<td>1037720</td>
<td>11272</td>
<td>412</td>
</tr>
<tr>
<td>1 0</td>
<td>696980</td>
<td>21732</td>
<td>1895476</td>
<td>9348</td>
<td>0 0</td>
</tr>
<tr>
<td>0 2</td>
<td>696980</td>
<td>20992</td>
<td>1896496</td>
<td>9072</td>
<td>0 0</td>
</tr>
<tr>
<td>1 1</td>
<td>696980</td>
<td>20228</td>
<td>1897784</td>
<td>8648</td>
<td>0 0</td>
</tr>
<tr>
<td>1 1</td>
<td>696960</td>
<td>21048</td>
<td>1897404</td>
<td>8716</td>
<td>48</td>
</tr>
<tr>
<td>0 0</td>
<td>696952</td>
<td>23048</td>
<td>1899112</td>
<td>9004</td>
<td>8 0</td>
</tr>
<tr>
<td>0 0</td>
<td>696952</td>
<td>23048</td>
<td>1899112</td>
<td>9004</td>
<td>0 0</td>
</tr>
</tbody>
</table>
If you have extra available physical memory, Linux will use it to cache files on disk for performance benefits. This disk cache may also include parts of the swapfile. Why would caching the swapfile be better than paging-in those pages immediately?

If I remove the line “memset(x, 0xCC, size);” from program B, I notice that the `vmstat` output does not have a spike in swap (si and so) nor in io (bi and bo). My system doesn’t have enough physical memory for a 5GB array. Yet, the array is not swapped out to disk. Where does the array go? Why did the `memset` make a difference?

Program B has a 5GB array, but the whole thing just contains 0xCCCCCCCC. Based on this observation, can you think of a way to reduce program B’s memory footprint without changing any of program B’s code? (What can the kernel do to save memory?)