CSCE 313-200
Introduction to Computer Systems
Spring 2019

Memory III
Dmitri Loguinov
Texas A&M University

April 23, 2019
Homework #3

- Tested Rabin-Karp performance on enwiki-all.txt
  - FILE_FLAG_NO_BUFFERING, B = 2 MB, 36 slots
  - First server w/RAID @ 2 GB/s + keywords-A; disk too slow
  - Second server w/RAID @ 3.9 GB/s + keywords-B

<table>
<thead>
<tr>
<th></th>
<th>CPU</th>
<th>Time</th>
<th>Speed</th>
<th>Found</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>100%</td>
<td>14.4</td>
<td>2.1 GB/s</td>
<td>319,017,279</td>
</tr>
<tr>
<td>B</td>
<td>100%</td>
<td>16.0</td>
<td>1.9 GB/s</td>
<td>319,017,279</td>
</tr>
<tr>
<td>C</td>
<td>100%</td>
<td>14.4</td>
<td>2.1 GB/s</td>
<td>319,026,630</td>
</tr>
<tr>
<td>D</td>
<td>100%</td>
<td>16.8</td>
<td>1.8 GB/s</td>
<td>318,798,734</td>
</tr>
<tr>
<td>Ref</td>
<td>64%</td>
<td>8.8</td>
<td>3.4 GB/s</td>
<td>319,017,279</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>Speed</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>22</td>
</tr>
<tr>
<td>B</td>
<td>24</td>
</tr>
<tr>
<td>C</td>
<td>crash</td>
</tr>
<tr>
<td>D</td>
<td>20</td>
</tr>
<tr>
<td>Ref</td>
<td>105</td>
</tr>
</tbody>
</table>
Why are lookup tables useful?
- Allows verification of set membership in 1 cache access

How to initialize?
- E.g., need to set up LUT to verify that character belongs to set {+, -, =, /, *}

When bool maps to 1 byte, can use it instead of char
- Keep in mind though that BOOL is 4 bytes

Make sure to test code on various input and buf size
- **Debugging**: elimination of crashes/incorrect output
- **Testing**: discovery of input configurations that expose previously unseen problems
Chapter 7: Roadmap

7.1 Requirements
7.2 Partitioning
7.3 Paging
7.4 Segmentation
7.5 Security
8.1 Hardware virtual memory
8.2 OS software
Memory Dumps

• Process crash is usually good news
  - Attach debugger, examine location of crash…

• Except when product has shipped to customers
  - Users do stuff with code that makes it crash
  - Developer is unable to replicate bug locally, what next?

• Idea: catch faults thrown by SEH (Structured Exception Handling)
  - Create a crash dump, send it to main server, then probably restart
Memory Dumps

• Instead of dumping entire RAM contents, Windows allows much smaller files called MiniDumps
  – Can be customized during exception handling to vary in size from a few KB to a few MB

• MiniDumps can be loaded into Visual Studio
  – Shows the exact location of crash, call stack, certain variables (even if crashed in release mode)

• Example:
  – Important application that must work 24/7, years in a row
  – When it crashes, saves internal data and dump, restarts
  – Debugging is done offline from a collection of minidumps

• See MiniDumpWriteDump on MSDN
Chapter 7: Roadmap

7.1 Requirements
7.2 Partitioning
7.3 Paging
7.4 Segmentation
7.5 Security

8.1 Hardware virtual memory
8.2 OS software
Buffer Overflow Attacks

• Example 1:

```c
int CheckPassword (HANDLE user) {
    char correctHash [16];
    char userHash [16];

    GetPassHash (user, correctHash);
    // remote desktop hashes password
    // and sends the hash to server
    Network.Read (userHash);
    if (!strcmp (correctHash, userHash))
        return MATCH;
    else
        return BOGUS;
}
```

• Example 2:

```c
void HandleServerRequest (void) {
    char request [256];
    Network.Read (request);
}
```

- Stack grows backwards
- Overflow of userHash rewrites correctHash
- Long overflow, contains virus code
- Execution continues from PC, virus runs
Buffer Overflow Attacks

- Modern OS usually puts a guard page between data, code, and stack.

Example 3:

```c
void HandleServerRequest (void) {
    char request [256];
    Network.Read (request);
    ...
}
```

Guard pages (reserved, but not committed) generate page faults.

- Modern OS marks data and stack pages as non-executable (DEP).

Firefox was closed
To help protect your computer, Data Execution Prevention has closed Firefox. Click to learn more.
Buffer Overflow Attacks

- **Example 4:**

```c
void HandleServerRequest (void) {
    char request [256];
    Network.Read (request);
    ...
}
```

- **Example 5:**

```c
void HandleServerRequest (void) {
    char request [256];
    Network.Read (request);
    ...
}
```

```
request
```

rewrites return address to jump to specific kernel function that gives elevated privileges

```
ret addr
```

ret NTdll.A

```
garbage
```

```
ptr
```

hijacked ptr

```
admin password in RAM
```

```
NTdll.A: admin user logged in
NTdll.B: change admin password
NTdll.C: wipe C:\
```

kernel space

more in CSCE 465
Heartbleed Bug

• OpenSSL is a library that encrypts/decrypts traffic
  – Commonly used in HTTPS, SSH, secure IMAP/SMTP
• Heartbeat extension introduced in 2011
  – OpenSSL periodically sends a request that is echoed back to verify the connection is alive
• Request message format:
  ```
  header  len  buffer
  ``
  ```
  len bytes
  ```
  • Response is supposed to echo the buffer
  – Implementation →
  ```
  size = Network.GetNextPacketSize();
  char *packet = new char [size];
  Network.Read (packet);
  len = ExtractLenField (packet);
  Network.Send (packet, len+sizeof(header)+sizeof(short));
  ```
Chapter 7: Roadmap

7.1 Requirements
7.2 Partitioning
7.3 Paging
7.4 Segmentation
7.5 Security

8.1 Hardware virtual memory
8.2 OS software
The OS has to make two main decisions when managing virtual memory and swapping:
- Which page to bring back to RAM (fetch policy)
- Which page to offload to disk (replacement policy)

Similar concepts may be useful in user-mode programs (e.g., object caching, browser prefetch)

Fetch policy
- Demand paging: bring page only on access (Windows)
- Prepaging: OS attempts to guess future demand, bring those pages in memory ahead of the request

Replacement policy
- FIFO: treats all pages as circular buffer, evicts the next one
Managing Virtual Memory

- **Replacement policy (cont’d)**
  - **LRU**: evicts the page that has not been used the longest
  - **Optimal**: evicts the page that won’t be used the longest
    (only used in simulations for comparison purposes)

- **How to implement LRU?**
  - Can’t tag each page with an access timestamp (updating timestamps incurs huge overhead)
  - Can’t organize all pages into a linked list either (moving items to the front of the list on access is expensive)

- **Idea**: replace LRU with an approximation algorithm
  - Assume a set of pages 0, ..., N-1 that the OS manages
  - Associate a bit B (e.g., in the TLB) with each page
  - CPU sets the bit to 1 upon each read/write access
Managing Virtual Memory

- Upon page fault:
  - OS scans from current position CP in [0, N-1] forward
  - If next page has B = 1, flag is reset to 0 and scan continues
  - If next page has B = 0, OS stops and evicts that page

- This policy is called **CLOCK**
  - Next page evicted?

- Quality of algorithm measured by number of hard page faults (PF)
  - FIFO 2x worse than optimal in PF
  - CLOCK better than FIFO, but not as good as LRU
Managing Virtual Memory

- Should pages that were read be replaced at the same rate as those that have been written to?
  - Probably more expensive to evict a modified page

- **Idea**: set up an extra bit $W$ for each page
  - CPU modifies them on access, CLOCK first evicts eligible pages with $W = 0$; if none left, then those with $W = 1$

- CLOCK is quicker than LRU even in user mode

- **Examples where CLOCK might be useful**:
  - Web crawler keeps a list of recently seen URLs
  - Search engine caches answers to popular queries
  - Homework #4: 50% of all hash table lookups refer to 1,270 words (20% to just 36 words), possible ways to speed up?