Memory III
Dmitri Loguinov
Texas A&M University

April 24, 2024
Homework #3

- Tested Rabin-Karp performance on enwiki-all.txt
  - FILE_FLAG_NO_BUFFERING, B = 2 MB, 50 slots
  - 8-core Skylake-X server w/RAID @ 4 GB/s

<table>
<thead>
<tr>
<th></th>
<th>Time</th>
<th>Speed</th>
<th>Found</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>12.5</td>
<td>2.4 GB/s</td>
<td>318,798,734</td>
</tr>
<tr>
<td>B</td>
<td>22.1</td>
<td>1.4 GB/s</td>
<td>319,017,279</td>
</tr>
<tr>
<td>C</td>
<td>40.9</td>
<td>0.7 GB/s</td>
<td>319,017,279</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>Speed</th>
<th>Found</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ref</td>
<td>7.50</td>
<td>319,017,279</td>
</tr>
</tbody>
</table>

keywords-B

<table>
<thead>
<tr>
<th></th>
<th>Speed</th>
<th>Found</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ref</td>
<td>125 MB/s</td>
<td>3,374,677,735</td>
</tr>
</tbody>
</table>

keywords-D
Chapter 7: Roadmap

7.1 Requirements
7.2 Partitioning
7.3 Paging
7.4 Segmentation
7.5 Security
Paging

- Paging allows the OS to allocate non-contiguous chunks of space to application requests
  - Hardware finds the page in RAM by transparently mapping from logical to physical addresses
- Logical address consists of two parts
  - Page number
  - Offset within that page
- **Example:** 32 bit address, 4 KB pages

```c
char *ptr = 0x33567
0x33 = logical page number
0x567 = offset
0x33567 = physical address
```

RAM

```
P1
P1
P2
P2
```
Paging

- Conversion of page numbers is done using the TLB (Translation Lookaside Buffer):

\[
\text{char *ptr = 0x33 0x567}
\]

- Each process owns a page table controlled by OS
**Paging**

- **Example:** write 5000 bytes to array `ptr[]`

```c
char *ptr = 0x33567;
for (int i = 0; i < 5000; i++)
    ptr [i] = i;
```

- `Ptr + i = 0x33567-0x3FFF`
  - `i = 0-2712 (2713 iterations)`
  - Physical address range 0x453567-0x453FFF
- `Ptr + i = 0x34000-0x348EE`
  - `i = 2713-4999 (2287 iterations)`
  - Physical address range 0x621000-0x6218EE
Paging

• To avoid doubling RAM latency on random access, TLB is kept in **dedicated cache memory**
  – CPU performs a lookup before sending address to RAM
• Within a given page, no control of address validity
  – However, if a process goes far enough to hit next page, the TLB must have an entry for that page with correct permissions
  – If not, a page fault is thrown and the process is killed
• These concept allow allocation of pages beyond physical RAM, swapping to disk, loading to new addr
• **Example**: computer with 8 GB of RAM
  – Process requests 7 GB, but all other resident software and kernel occupy 2.5 GB
Paging

• Whatever pages aren’t being used are swapped to disk
  – Special pagefile provides space for this operation
  – Usually, pagefile.sys is twice the size of RAM
• Memory classification
  – Non-pageable memory: special types of pages that cannot be swapped to disk (e.g., parts of OS, locked pages, AWE segments, large-page allocations)
  – Commit set: all pageable memory of the process (i.e., allocated in the page file)
  – Working set: touched (accessed) pages in RAM
  – Private working set: a subset of the working set (e.g., heap-allocated) that is not shared with other processes
• The last three can be seen in Task Manager
Paging

• Access to page outside working set causes a **page fault**
• Types of page faults
  – **Hard**: requires the page to be read from disk
  – **Soft**: can be resolved with remapping (e.g., pages exists in working set of another process or first-time access)
  – **Violation**: access outside virtual space of this process or using incompatible permissions (e.g., writing to read-only page)
• Hard/soft faults are handled transparently by OS
• **Example**: allocate 1 GB of committed memory

```c
char *buf = (char *) VirtualAlloc (NULL, 1 << 30, MEM_COMMIT|MEM_RESERVE, PAGE_READWRITE);
```

• Commit size, working set size, and private set size?
Paging

• Examine Task Manager:

<table>
<thead>
<tr>
<th>Image Name</th>
<th>PID</th>
<th>User Name</th>
<th>CPU</th>
<th>Working Set (</th>
<th>Memory (Priv...</th>
<th>Commit Size</th>
<th>Paged Pool</th>
<th>Page Faults</th>
</tr>
</thead>
<tbody>
<tr>
<td>hw4.exe</td>
<td>5428</td>
<td>dmitri</td>
<td>00</td>
<td>3,440 K</td>
<td>1,500 K</td>
<td>1,052,260 K</td>
<td>75 K</td>
<td>874</td>
</tr>
</tbody>
</table>

• Commit size is 1 GB as expected, but none of that memory has been allocated in physical RAM yet
  - OS doesn’t know which pages we’ll need and in what order
  - Conserves physical RAM as much as possible

• Write something into each page:

```c
memset (buf, 0x55, 1 << 30);
```

both working sets change

260K soft page faults
• Suppose we intend to dynamically expand the region of allocated memory
  - But don’t want to copy data over to the new area each time
  - Similar to HeapReAlloc
• Would like to ask the kernel to map the continuation of the previous buffer to some additional physical pages:

```c
// allocation of initial 128 KB succeeds
int size = 1 << 17;
char *buf = (char *) VirtualAlloc (NULL, size, MEM_COMMIT|MEM_RESERVE, PAGE_READWRITE);
// attempt to add 16 MB to this buffer may fail
char *result = (char *) VirtualAlloc (buf + size, 1 << 24,
   MEM_COMMIT|MEM_RESERVE, PAGE_READWRITE);
```
The problem is that the virtual space beyond buf + size might have already been assigned
- Allocation in this case fails

**Idea:** reserve a huge amount of virtual space so that the heap can’t use it

Reserved memory is not mapped to pagefile until explicitly **committed**
- Reservation simply makes sure this address space is not used in other allocation requests
- In Server 2016, max reservation is 128 TB
Working with Buffers

• Can now commit memory in our reserved space

```c
// reserve 1 TB
char *bufMain = (char *) VirtualAlloc (NULL, (uint64) 1<<40,
    MEM_RESERVE, PAGE_READWRITE);
// allocate 128 KB
int size0 = 1 << 17;
char *buf0 = (char *) VirtualAlloc (bufMain, size0,
    MEM_COMMIT, PAGE_READWRITE);
// now add 16 MB to this buffer
int size1 = 1 << 24;
char *buf1 = (char *) VirtualAlloc (buf0 + size0, size1,
    MEM_COMMIT, PAGE_READWRITE);
// now add 1 GB
int size2 = 1 << 30;
char *buf2 = (char *) VirtualAlloc (buf1 + size1, size2,
    MEM_COMMIT, PAGE_READWRITE);
```

• Memory may be decommitted as needed

```c
// decommit 4KB from the middle of committed space
char *result = (char*) VirtualFree (buf1, 1 << 12, MEM_DECOMMIT);
```