CSCE 313-200
Introduction to Computer Systems
Spring 2024

Memory IV
Dmitri Loguinov
Texas A&M University

April 26, 2024
Why are lookup tables useful?
- Allow 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 {+, -, =, /, *}

Cast buf[i] to uchar to prevent LUT underflows on ASCII > 127 codes

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

```c
char LUT[256];
memset(LUT, false, 256);
const char special[] = "+-=/*";
for (int i = 0; i < strlen(special); i++)
  LUT[special[i]] = true;
```
InterlockedMultiply

• Write code for an interlocked multiply that does not use mutexes (lock-free)
  – Multiple threads might be calling this function at the same time

• Idea: grab the target, multiply locally, then try to swap back into shared space
  – Main caveat is another thread might have changed the target between our read and write

// new function
LONG __cdecl InterlockedMultiply(
  __inout  LONG volatile *Target,
  __in     LONG Value
);

// standard functions in Windows
LONG __cdecl InterlockedExchange(
  __inout  LONG volatile *Target,
  __in     LONG Value
);
PVOID __cdecl InterlockedExchangePointer(
  __inout  PVOID volatile *Target,
  __in     PVOID Value
);
LONG __cdecl InterlockedCompareExchange(
  __inout  LONG volatile *Destination,
  __in     LONG Exchange,
  __in     LONG Comparand
);
PVOID __cdecl InterlockedCompareExchangePointer(
  __inout  PVOID volatile *Destination,
  __in     PVOID Exchange,
  __in     PVOID Comparand
);
Queue Example

- Design self-resizing Q that keeps data contiguous and never has to memcpy
  - Code below does not handle errors, nor does it compute how much to expand or shrink by

```c
Q::Q () {
    reserveSize = (uint64) 1<<40;
    char *bufMain = (char *) VirtualAlloc (NULL, reserveSize,
    MEM_RESERVE, PAGE_READWRITE);
    head = tail = (Item*) (next = last = bufMain);
}
void Q::push (Item x) {
    // overflow of current commit section?
    if (tail + sizeof(x) >= next) {
        // add some commit space in front of the tail
        VirtualAlloc (next, expandSize, MEM_COMMIT, PAGE_READWRITE);
        next += expandSize;
    }
    *tail++ = item;
}
```

```c
class Q {
    char *next, *last;
    char *bufMain;
    Item *head, *tail;
};
```
- **Shrink the committed region during pop**

```c
Item Q::pop (void) {
    if (head > last + shrinkSize) {
        // decommit old memory behind the head
        VirtualFree (last, shrinkSize, MEM_DECOMMIT);
        last += shrinkSize;
    }
    return *head++;
}
```

- **Problem #1**: cannot commit/decommit too fast
  - Keep expandSize and shrinkSize around 1 MB
- **Problem #2**: queue eventually overflows when reserveSize is exceeded
  - If 128 TB of virtual space is not enough, memcpy or linked lists of buffers cannot be avoided
Assume there exists some complex data processing library whose APIs only work with contiguous buffers.

- Can the library be hacked to work with shadow buffers?

If so, what if some records do not fit in shadow buffer?

- Recall that shadow buffers must be at least the size of the longest record (e.g., word) in the file.

Some files may have extremely long records.

- E.g., each record in a graph contains a node ID and a list of its neighbors; for 300M neighbors, 2.4 GB per record.

Worse yet, what if individual records do not fit in RAM?

- E.g., search engine index contains a keyword hash and a list of pages where the keyword appears; for a popular keyword found in 5B pages, this requires 40 GB.

Disk I/O Example

single-threaded application that reads a file larger than RAM
• Suppose the library is a streaming data processor
  - Operates on data only sequentially and going forward
  - Never returns by more than $X$ bytes, where $X$ is small
• **Goal**: use virtual memory to create an illusion of a continuous file in RAM for this library
• **Idea**: let the library run into page faults
  - Which we catch, commit the next chunk of virtual memory, read the next file block into it, and return control to the API
  - Blocks of memory that are 2 buffers behind are decommitted assuming buffer size is no smaller than $X$
• Performance (AMD Phenom II): page-fault rate is $\sim 900$K/sec
Disk I/O Example

• What’s a good reserve size?
  – Length of file

• This is how memory-mapped files work
  – Slightly more general as they allow random access
  – Read small buffer surrounding the page fault
  – Decommit old pages using LRU or some other technique
  – See CreateFileMapping andMapViewOfFile

• Problem: this method can only do single-buffering
  – Stalls processing while the next buffer is being read
  – Only solution is to read ahead into other RAM locations, then memcpy into buf_{i+2} during page faults
**Disk I/O Example**

- **Using AWE (Address Windowing Extensions)**
  - Six physical buffers allocated by disk thread, into which it reads the file, wrapping back to $B_0$ after $B_5$
  - Two green buffers are mapped to virtual addresses currently being processed by the library; $B_2$-$B_5$ are used for read-ahead
  - On page fault, the oldest buffer $B_0$ is unmapped, the next buffer $B_2$ is mapped where the page fault occurred

![Diagram of disk I/O example with buffers and page fault](image)
Disk I/O Example

- Writing-to-buffer benchmark
  - 1) No remapping or page-fault processing

```
char *buf = VirtualAlloc (NULL, 1e9, MEM_COMMIT|MEM_RESERVE, ...);
```

- 2) Reserve virtual memory, catch page faults, commit new chunks of size 1 MB, decommit old chunks

```
char *buf = VirtualAlloc (NULL, 1e9, MEM_RESERVE, ...);
__try {
    writeToPtr (buf, 1e9);
}
__except ( ...) {
}
```

- 3) Reserve physical memory (AWE), catch page faults, remap chunks of size 1MB, unmap old chunks
**Disk I/O Example**

- **Two versions of `writeToPtr()`**:

  ```c
  void writeToPtrA (char *buf, int size) {
    for (int i=0; i < size; i++)
      buf[i] = 55;
  }
  ```

  ```c
  void writeToPtrB (char *buf, int size) {
    memset (buf, 55, size);
  }
  ```

- **Benchmark results**:

<table>
<thead>
<tr>
<th>Mapping</th>
<th>writeToPtr</th>
<th>Working set</th>
<th>Page faults</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>1) None</td>
<td>loop</td>
<td>1 GB</td>
<td>245,493</td>
<td>3.4 sec</td>
</tr>
<tr>
<td></td>
<td>memset</td>
<td>same</td>
<td>245,493</td>
<td>343 ms</td>
</tr>
<tr>
<td>2) Commit</td>
<td>loop</td>
<td>5.3 MB</td>
<td>245,327</td>
<td>3.2 sec</td>
</tr>
<tr>
<td></td>
<td>memset</td>
<td>same</td>
<td>245,327</td>
<td>499 ms</td>
</tr>
<tr>
<td>3) Physical</td>
<td>loop</td>
<td>5.3 MB</td>
<td>1,361</td>
<td>3.1 sec</td>
</tr>
<tr>
<td></td>
<td>memset</td>
<td>same</td>
<td>1,361</td>
<td>156 ms</td>
</tr>
</tbody>
</table>