CSCE 313-201
Introduction to Computer Systems
Fall 2020

Practice III
Dmitri Loguinov
Texas A&M University

November 5, 2020
• Write proper synchronization for a train tunnel

```c
Train
TryEnteringTunnel (int dir) {
    mutex[dir].Lock();
    if (trains[dir]++ == 0)
        occupied.Wait();
    mutex[dir].Unlock();

    semaMaxN.Wait();
    PassThruTunnel(x, dir);
    semaMaxN.Release();

    mutex[dir].Lock();
    if (--trains[dir] == 0)
        occupied.Release();
    mutex[dir].Unlock();
}
```
• Print spooler system
  - Main rule: combined size of Q1 and Q2 cannot exceed M
• Version #1: without the combined max, each queue has an independent size limit

```
P1
x = ObtainItem();
semaEmptyQ1.Wait();
m1.Lock();
Q1.push(x);
m1.Unlock();
semaFullQ1.Release();

P2
semaFullQ1.Wait();
m1.Lock();
y = Q1.pop();
m1.Unlock();
semaEmptyQ1.Release();
z = Process (y);
semaEmptyQ2.Wait();
m2.Lock();
Q2.push(z);
m2.Unlock();
semaFullQ2.Release();

P3
semaFullQ2.Wait();
m2.Lock();
w = Q2.pop();
m2.Unlock();
semaEmptyQ2.Release();
ProcessAndDiscard (w);
```
• Version #2: with the max, but deadlock-prone

Semaphore disk = \{M,M\};

\begin{itemize}
  \item P1
    \begin{itemize}
      \item x = ObtainItem();
      \item disk.Wait();
      \item m1.Lock();
      \item Q1.push(x);
      \item m1.Unlock();
      \item semaFullQ1.Release();
    \end{itemize}
  \item P2
    \begin{itemize}
      \item semaFullQ1.Wait();
      \item m1.Lock();
      \item y = Q1.pop();
      \item m1.Unlock();
      \item disk.Release();
      \item z = Process(y);
      \item disk.Wait();
      \item m2.Lock();
      \item Q2.push(z);
      \item m2.Unlock();
      \item semaFullQ2.Release();
    \end{itemize}
  \item P3
    \begin{itemize}
      \item semaFullQ2.Wait();
      \item m2.Lock();
      \item w = Q2.pop();
      \item m2.Unlock();
      \item disk.Release();
      \item ProcessAndDiscard(w);
    \end{itemize}
\end{itemize}

• When will this deadlock?
• Version #3: do not release disk semaphore in P2

Semaphore disk = {M,M};

\[ P1 \]
x = ObtainItem();
disk.Wait();
m1.Lock();
Q1.push(x);
m1.Unlock();
semaFullQ1.Release();

\[ P2 \]
semaFullQ1.Wait();
m1.Lock();
y = Q1.pop();
m1.Unlock();
// remove disk.Release();

z = Process (y);
// remove disk.Wait();
m2.Lock();
Q2.push(z);
m2.Unlock();
semaFullQ2.Release();

\[ P3 \]
semaFullQ2.Wait();
m2.Lock();
w = Q2.pop();
m2.Unlock();
disk.Release();
ProcessAndDiscard (w);

• What if P2 makes K items for each extracted from Q1?
Quiz 3

- Assume N processes sharing M resources
  - Process i eventually wants to hold $W_i$ resources
  - Resources are obtained non-atomically
  - After getting all of its resources, process releases them

- Maximum # of resources R that still lead to deadlock?
  - Suppose $W_1 = 6$, $W_2 = 3$, $W_3 = 14$
  - Then $M > R$ guarantees no deadlock and $M = R$ allows one

- Writing:
  \[
  R = \sum_{i=1}^{N} (W_i - 1) < M
  \]
  - we obtain:
  \[
  \sum_{i=1}^{N} W_i - N < M \quad \Rightarrow \quad \sum_{i=1}^{N} W_i < M + N
  \]
String Search

• How fast is homework #3 with 200K keywords?
  - Roughly 9.1 KB/s, 38 days to parse the big file
• Using all 8M unique words in large Wikipedia?
  - Speed 240 bytes/s, roughly 4 years to finish (using 12 cores)
• Focus of computer science has always been efficiency
  - Quicksort vs bubble sort, hashing vs sorting, binary vs linear search, min-heap vs linear min()
  - Substring search is another example
• Start with single-string search
  - Assume some text and a given keyword
  - Need to find all occurrences of keyword in text
  - Matches do not have to be complete words
**Single String**

- Naïve method #1: use strcmp or memcmp
- Naïve method #2: use strstr
  - Runs somewhat faster, but still far from optimal
- **Example of method #1:**
  - Worst-case complexity?
  - N = length of text, M = word size, then (N-M)*M

```c
while (off < bufSize - wordLen) {
    if (memcmp (buf + off, word, wordLen) == 0)
        found ++;
    off ++;
}
```

```c
char *match = buf;
buf [bufSize] = 0;
while (true) {
    match = strstr (match, word);
    if (match == NULL)
        break;
    found ++;
    match ++;
}
```

---

**A B C Q**

**word**

**text**

---

step 1
Single String

<table>
<thead>
<tr>
<th>text</th>
<th>A B C Q A B C D A B Z D ...</th>
</tr>
</thead>
<tbody>
<tr>
<td>miss</td>
<td>A B C D A B D Q</td>
</tr>
<tr>
<td>word</td>
<td>A B C D A B D Q</td>
</tr>
<tr>
<td>step 2</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>text</th>
<th>A B C Q A B C D A B Z D ...</th>
</tr>
</thead>
<tbody>
<tr>
<td>miss</td>
<td>A B C D A B D Q</td>
</tr>
<tr>
<td>word</td>
<td>A B C D A B D Q</td>
</tr>
<tr>
<td>step 3</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>text</th>
<th>A B C Q A B C D A B Z D ...</th>
</tr>
</thead>
<tbody>
<tr>
<td>miss</td>
<td>A B C D A B D Q</td>
</tr>
<tr>
<td>word</td>
<td>A B C D A B D Q</td>
</tr>
<tr>
<td>step 4</td>
<td></td>
</tr>
</tbody>
</table>
Single String

- Naïve takes 7 comparisons to move 4 bytes
  - Total complexity of getting past 12 bytes is 23 comparisons
- Knuth-Morris-Pratt (KMP), 1977:

  text: A B C Q A B C D A B Z D

  word: A B C D A B D Q

  step 1

  text: A B C Q A B C D A B Z D

  word: A B C D A B D Q

  step 2
Single String

text: A B C Q A B C D A B Z D...
word: A B C D A B D Q

Step 3: Miss

Step 4: A B C D A B D Q

Step 5: A B C D
**Single String**

- Total 6 steps, 15 comparisons to pass 12 bytes
- How does it work?
  - Each character needs two lookup tables (LUTs) – by how many bytes to move after a non-match in this position and where in the word to re-start on the next attempt

<table>
<thead>
<tr>
<th>word</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>A</th>
<th>B</th>
<th>D</th>
<th>Q</th>
</tr>
</thead>
<tbody>
<tr>
<td>move</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>5</td>
<td>5</td>
<td>4</td>
<td>7</td>
</tr>
<tr>
<td>re-start</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>0</td>
</tr>
</tbody>
</table>

{tables built offline, fit in L1 cache}
• **Boyer-Moore (BM), 1977:**
  - Uses not just distance, but also the mismatched character

• Matching goes right to left, until a mismatch
  - Off is examined position in text

• After a miss, two hash tables move the word forward:
  - Slide[dist]: based on the # of matched characters
  - Shift[char]: based on mismatched character text[off]
Single String

- In the example above
  - Mismatch distance is 0, so slide by 1 char
  - Mismatch char = C, so shift by 5

- After moving off by 5:

- In this case, mismatch occurs at \text{text}[\text{off}] = Z:
  - Mismatch distance = 2, slide word by 8
  - Mismatch char = Z, shift word by 6

when moving forward, take the larger of the two
For words that have rare letter combinations, we can be skipping by $M$ each time

- Best case complexity is sub-linear, i.e., $N/M$ comparisons

Typically faster than KMP for larger $M$
**Single String**

- Can we do better?
- Notice that BM gets stuck on popular characters, while ideally it should skip most examined locations
  - E.g., “zebra” incurs detailed inspection any time it hits an ‘e’
- **Idea**: set up a hash table with 2-byte combinations
  - E.g., “ze”, “eb”, “br”, “ra” which are much more rare
  - Then scan the text using an *unsigned short* (2-byte) pointer
- **Caveat**: don’t know alignment of the word, may hit something like “_z” and miss the word
  - Need to set up wildcard entries ?z and a? for all possible leading and trailing characters
  - If only full words are needed, ? will be a white space
Multiple Strings

• Why was homework #3 so inefficient?

  • **Idea**: do not compare current byte to all strings, only to those that can potentially be a match

  • **Rabin-Karp** (RK), 1987
    - Assume M is the smallest keyword length
    - Compute a hash H of the next M chars from current location
    - Hit a hash table, compare with words that tie for that hash
    - Speed is only based on the length of collision chains
Multiple Strings

- After hash table lookup, slide by one byte forward, recompute the hash of the next M chars

- Notice that M-1 chars are the same in both hashes
  - Main twist of the algorithm is to use a rolling hash, which obtains $H_{i+1}$ from $H_i$ in $O(1)$ time

- Treating hashes as base-$B$ integers, we have
  - $H_0 = str[0] \cdot B^{M-1} + str[1] \cdot B^{M-2} + \ldots + str[M-1]$
  - $H_{i+1} = (H_i \cdot B + str[i+M]) \mod B^M$
Wrap-up

- Larger M means fewer collisions and faster operation
- With M = 3 and 216K strings, RK runs at 20MB/s
  - 2000 times faster than the naïve method
- Indexing a file with unknown keywords is slightly different, but the idea is similar to RK
  - Homework #4 explores this in more detail
- Main goal is to design code that processes all 4.5B words in large Wikipedia in ~35 sec (135M wps)
  - 3.7M times faster than the method in homework #3
- Homework #4 has 3 checkpoints
  - The first two should be done early
  - Checkpoint #3 is more complex, uses virtual memory