Lab 5: String Interning

CMPUT 229

University of Alberta

Outline

Lab 5: String Interning

Motivation

- Suppose we have some text (possibly code) with many long, identical strings:

```
def long_and_useless_function(long_variable_name):
loop_increment_counter = 0;
while loop_increment_counter < long_variable_name:
loop_increment_counter = loop_increment_counter + 1
long_and_useless_function(long_variable_name  1)
print(‘‘This isn’t useful!")
```

- During a task such as compiling, we may need to compare these strings (such as variable and function names) many, many times.
- This is computationally expensive!
- $\Rightarrow$ The **solution**: string interning.

Motivation

- Instead of performing comparisons on long strings such as `long_and_useless_function` and `loop_increment_counter`, we will:
  - Create a 1 word (4 byte) unique identifier for these strings.
  - Compare these identifiers.
- For example:

`long_and_useless_function`	$\rightarrow$	`0x47fa018b`
`long_variable_name`	$\rightarrow$	`0xcc81b504`
`loop_increment_counter`	$\rightarrow$	`0x57cf47ab`

- Now the comparisons can be performed very quickly.

Hashing

- A hashing function takes some data as input and returns a fixed-length representation of that data.
- For example:

`"123password"`	`d2bc2f8d09990ebe87c809684fd78c66`
`"This is a sentence."`	`d15ba5f31fa7c797c09393132858164`
`"Hash me!"`	`e09f9e0c17051e3ad13f4176076cbb92`

- These are all examples of the MD5 hash algorithm.
- There are three things you should note about hash functions:
  1. The output is always the same length.
  2. Identical input will always produce identical output.
  3. If the input is unbounded (can be any length), multiple inputs will have the same output.

Hashing

- When two different inputs have the same output in a hash function, a collision occurs.
- For example:

```
"Both strings have the same value"    →  0x5f44a1b3
"for this particular hash function."  →  0x5f44a1b3
```

- One of the goals of a hash function is to reduce the number of collisions that occur.

Hashing

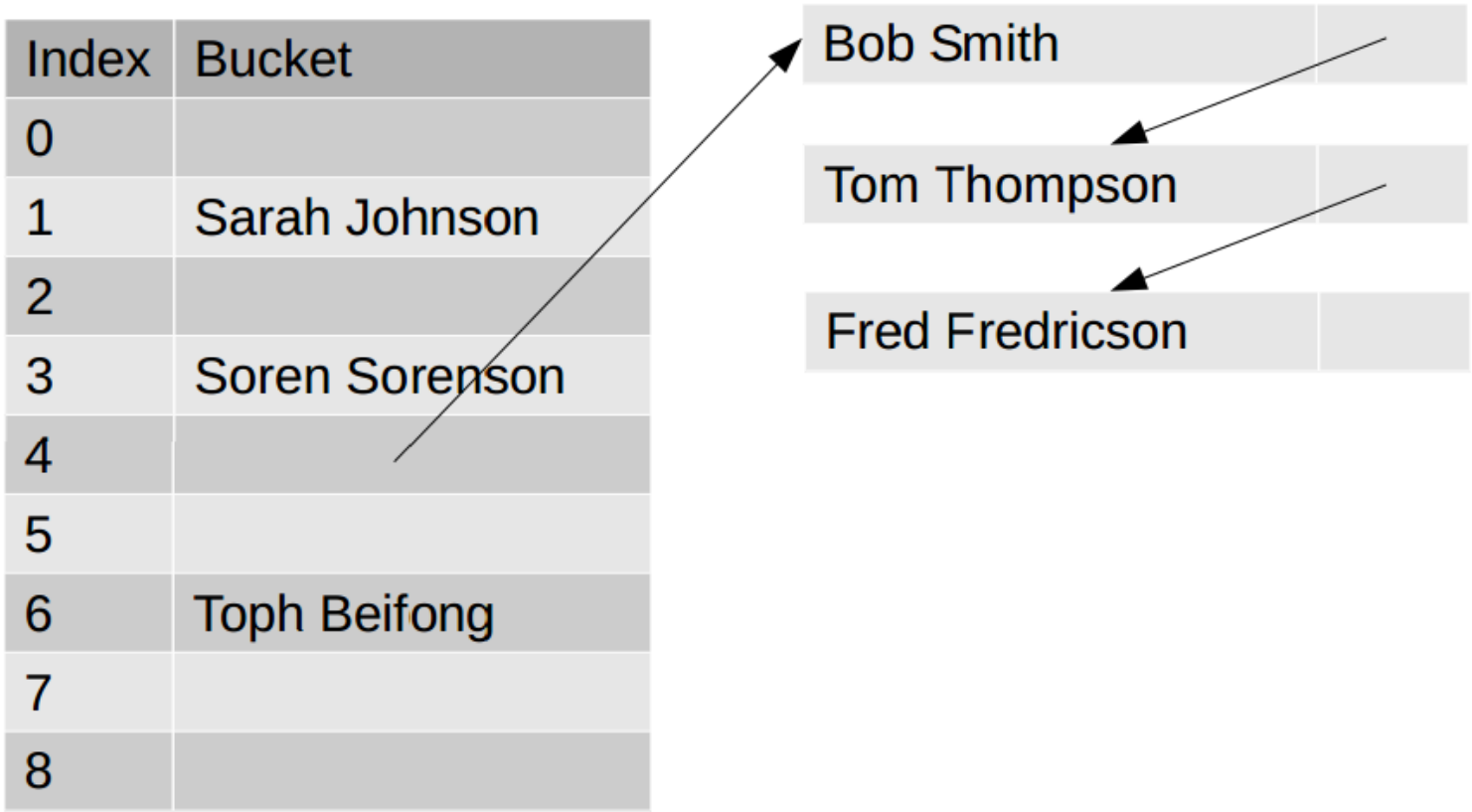
- In a hashtable, the output of a chosen hash function is used to specify the index at which to place data.

Index	Bucket
0	
1	Sarah Johnson
2	
3	Soren Sorenson
4	Bob Smith
5	
6	Toph Beifong
7	
8	

- The locations where data is stored in a hashtable are called **buckets**. In the example table above, each bucket stores one datum.

Hashing

- When a **collision** occurs in a hashtable, multiple items are stored in the same bucket. If the bucket is full, overflow has occurred.
- When it **overflows** it is replaced by a pointer to an unbounded data structure (e.g., linked list, stack, or even another hashtable).



Example

- We will step through a simple hashtable example. Our example will use the following checksum hash function (the same one you must use in your lab):

```
data []
        hash = 0
        for d in data
                do
                hash = (hash + d) mod n
        return hash
```

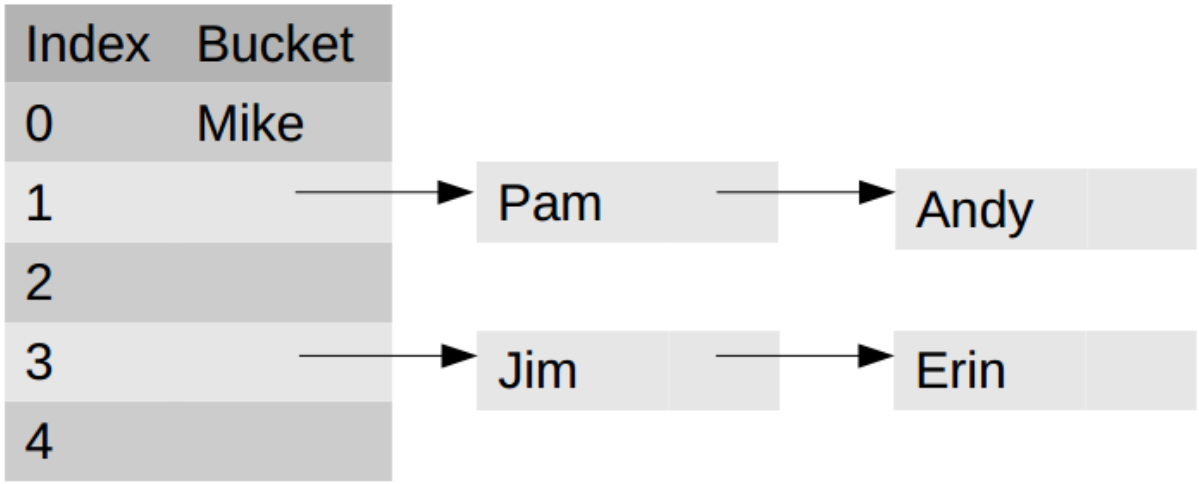
- Each bucket holds one item, and use a linked list for overflow.
- Finally, we set $n = 5$: n determines the range of possible outputs of our hash function, and therefore the size of our hashtable as well. When $n = 5$, our hash function will be in the range $f(x) \in [0, 5)$.

Example

- We will hash the following strings: `"Pam"`, `"Jim"`, `"Mike"`, `"Andy"`, `"Erin"`.
- Start by hashing `"Pam"`:
  1. Our hash starts at 0.
  2. The ASCII 'P' has a value of 80.
  3. $(0 + 80)\%5 = 0$
  4. The ASCII 'a' has a value of 97.
  5. $(0 + 97)\%5 = 2$
  6. The ASCII 'm' has a value of 109.
  7. $(2 + 109)\%5 = 1$
  8. `"Pam"` hashes to 1.

Example

- Using the same procedure, we can hash the other strings as well: `"Pam"` $\to 1$, `"Jim"` $\to 3$, `"Mike"` $\to 0$, `"Andy"` $\to 1$, `"Erin"` $\to 3$.



- Inserting `Jim` and then `Mike` into the hashtable is easy, because they fit into their respective buckets. However, when we insert `Andy` and then `Erin`, the buckets overflow, and we need to replace them with pointers to linked lists.

String Interning

- We will make use of a hashtable to perform **string interning**. String interning can be summarized with the following algorithm:
  1. Hash the string to find the hashtable index.
  2. Search the entry (either a bucket or unbounded data structure) for the string.
     1. If the string is found, return the unique identifier for the string.
     2. If the string is not found, store it (either in a bucket or unbounded data structure, whichever is appropriate) and then generate and return a unique identifier for it.
- Keep in mind that we are storing addresses to strings, not the strings themselves in our hashtable. Strings can be variable length, but addresses are always 1 word a 32-bit machine (e.g., spim).

String Interning: The Assignment

- From the assignment description, we can deduce that a bucket size of 1 should be used. When interning a string:
  1. **Empty bucket (`0x0`)**: store the string address, and return the unique identifier.
  2. **Bucket with linked list (bit 31st set)**: to obtain the pointer, switch bit 31 to 0. Search in the data structure: if there is a **match**, change nothing, and return the unique identifier. **Otherwise**, add the new string address to the data structure, and return the unique identifier.
  3. **Bucket with single entry**: if the entry **matches** to the new string, change nothing, and return the unique identifier. **Otherwise**, you must initialize an unbounded data structure, store both (old and new) string addresses in it, and then return the unique identifier for the new string.

String Interning: The Assignment

A few other important points about string interning:

- The string addresses given as arguments to subroutines are **mutable**: that is, the memory in that location may be changed/erased. Before saving a string address in the hashtable, make a copy of the string, and then save the address of this copy in the hashtable.
- Do not make a copy of a string if it is already stored in the hashtable.
- You must implement three subroutines: `internString`, `getInternedString` and `internFile`.

Subroutines: internString

internString:

Input: $a0 – address of a mutable string to be interned.
Return: $v0 – unique identifier for the string.

Consider:

- Identical strings must match exactly, and **do not** (necessarily) have the same address.
- You must make an immutable copy of the mutable strings given as input.
- The strings are null terminated.

Subroutines: getInternedString

getInternedString:

Input: $a0 – unique string identifier (interned string).
Return: $v0 – the immutable address (copy) of the string it if was interne

- Make sure your interned string identifier is unique and does not change even when your buckets may overflow.
- In other words, an identifier should always fetch the same string.

Subroutines: internFile

internFile:

Input: `$a0` – address of a mutable file to be interned.
Return: `$v0` – address of a list of unique identifiers for each string in the
`$v1` – the number of identifiers in the list.

Consider:

- Strings in the file are separated by either one or more space (0x20) or line feed (0x0A) characters (these separators are not part of strings: you must null-terminate them appropriately).
- The file ends with an end of transmission (0x04) character (this is also not part of any string).

Tips for the Lab

- The value that you chose for n in the hash function affects the size of your hashtable. Overall, you most have enough space to handle at most 128 unique strings.
- The strings passed to your functions are mutable and must be copied to immutable memory.
- Only string addresses are being stored in the hashtable.
- It is recommended (although not required) to create a string's unique identifier using a combination of the string's hash and index in the unbounded data structure.
- You can dynamically allocate memory in SPIM using system call 9: set `$a0` to the number of bytes desired, `$v0` to 9, and invoke syscall the address of the memory allocated will be in `$v0` after the call.
- As always, read the assignment carefully and follow all style and submission rules, including our **MIPS Callee/Caller convention**.

Questions?