CS 241 Lecture 23

Graham Cooper

July 27th, 2015

1 Tail Recursion in WLP4

```
int f(...) {
-- if(...){
-- -- if (..){
---- } else {}
-- }
-- else {}
return x;
}
```

Is the same as:

```
int f(...) {
-- if(...){
-- -- if (..){
---- } else {}
-- return x;
-- }
-- else {
-- return x;
--}
}
```

is the same as:

```
int f(...) {
-- if(...){
-- -- if (..){
-- -- return x;
-- -- }
-- -- else {
-- -- return x;
-- }
-- return x;
```

```
-- }
-- else {
-- return x;
--}
}
```

When return x follows an assignment to x, merge:
x = f(...) $\rightarrow$ return f(...)
return x;

- may create some tail recursive calls

Generalization:

- tail call optimization
- when a function's last action is any function call (recursive or not) can reuse the stack frame

Overloading

What would happen if we wanted to compile:

```
int f(int a){...}
int f(int a, int *b){...}
```

Get duplicated labels for f.
How do we fix this?

Name Mangling

Encode the types of params as part of the label

Example naming convention:
F + typeinfo + _ + name

ie.
1. int f(){...}
2. int f(int a){...}
3. int f(int a, int *b){...}

1. F_f:
2. Fi_f:
3. Fip_f:

- C++ compilers wil ldo this because c++ has overloading
- there is no standard mangling convention
- all compilers are different
- makes it hard or impossible to link code from different compilers
- this is by design b/c compilers differ in other aspects as well

C doesn't have overloading so there is no mangling
- C and C++ code call each other routinely
- How is this done?
- Suppress mangling in c++
Call C from C++
- Extern "C" int f(int n); tells c++ f wot be mangled

Call c++ from C - tell c++ not to mangle the function

extern "C" int g(int x){...} // dont mangle g

and then obviously you cannot overload extern c functions

Memory Management and the Heap

WLP4, C, C++

- explicit memory management
- user must free own data using free/delete

Java, Scheme

- implicit memory management
- garbage collection

How do new/delete or malloc/free work?

There are a variety of algorithms

1)

List of free blocks:

- maintain a linked list of ptrs to blocks of free RAM
- Initially entire heap is free, list contains one entry
- Suppose heap is 1k
- suppose we allocate 16 bytes
  - actually allocate 20 bytes + 1 int (4 bytes)
  - return pointer to second word
  - store size just before the returned pointer
  - free list ctonains the rest of the loop

Note: Repeated allocation and deallocation creates "holes" in the heap
EG:

```
alloc 20 {xx 20 xx,... (140)... ...}
alloc 40 {xx 20 xx, xx 40 xx,...(100) ... ...}
alloc 20 {...(20)..., xx 40 xx, ... (100)...}
alloc 5 {xx 5 xx, ... (15). .., xx 40 xx, ... (100)...}
etc.
```

We get holes like the 15 block hole on the last line, this causes:
Code fragmentation - means even if n bytes are free, we may not be able to allocate n bytes

To reduce fragmentation:
- don't always pick the first block of RAM big enought to satisfy the request

2) Binary Buddy System

Assume: size of heap is a power of 2.
Example: heap is 4k (4096 byes) = 1024 words.

Suppose

- program requests 20 words
- we need 1 word for bookeeping

Memory allocated in blocks of size 2^k

- so allocate 23 words
- 1024 is too big so split into two heaps (buddies)
- allocate from one of the 512-word buddies
- still too big, split one of them again
- still to big, split the 256 word buddies
- continue to split until one of the blocks is the size you need for your 21 words
- we return a pointer to a block of 32 words, with the first one being for bookeeping

Now we request 63 words, so we allocate 64

- heap contains a 64-word block
- we allocate that 64-word block

Now request 50 words, llocate for 51

- split one of the 128 blocks into 64
- allocate one of the new 64 blocks

First 64 word block is released

- simply just free the memory

Free the first 32-word block

- we free the 32 block and see that it's buddy is also free
- since they are both free, merge them into a 64 word block
- now the 64 block and its buddy is free, so merge into 128 block

Finally free the last 64-word block

- merge the 64 block that was just freed and its buddy
- we can now merge the rest of the blocks since everything is free into the 1024 block again

Deallocation info

alloc.asm

- assignms each block a code
- entire heap has a code of 1
- the 512-word buddies have codes of 10, and 11

- 256 word buddies have codes of 100, 101, 110, 111
- these are all encoded like a trie
- to find your buddy's code : flip te last bit
- mergin buddies - drop the last big (>>1)

Implicity Memory Management: Garbage Collection

System reclaims memory once client can no longer access it

1) Mark and Sweep

- scan the entire stack looking for pointers
- for each pointer found mark the heap block its pointing to
- then if the heap block contains pointers, follow those as well and mark etc...
- scan the heap
- reclaim any blocks not marked
- clear all marks

2) Reference Counting

- for each heap block, keep track of the number of pointers that point to it (reference count)
- means you must eatch every pointer and update ref counts (decrement old, increment new) each time a pointer is reassigned
- if a block's ref count reaches 0, reclaim

There is a problem when we have circular references, each is a pointer on the heap, pointing at each other, their reference count is 0, but there is nothing on the stack pointing at them

3) Copying GC

Heap is in two halves, FROM and TO

- allocate only from FROM
- when FROM fills up, all reachable data is copied from FROM to TO
- Built in Compaction
- guaranteed that after the swap, the compactor will make it such that all reachable data occupies memory
- this causes there to be no fragmentation yay!
- BUT we can only use half of the heap at a time