Warm Up Problem

Please do your Teaching Evaluations! Go to `https://evaluate.uwaterloo.ca`

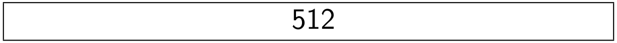
CS 241 Lecture 22

Heap Management and Loaders

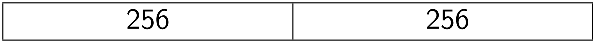
With thanks to Brad Lushman, Troy Vasiga and Kevin Lanctot

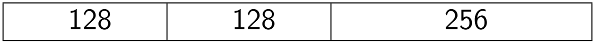
Binary Buddy System

Idea: Start off with 512 bytes of heap memory:

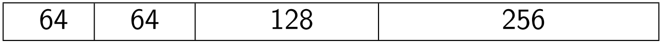


Suppose we try to allocate 19 bytes. We would need need an extra one for bookkeeping (so 20 total). This fits in a block of size $2^5 = 32$. We split our memory until we find such a block and reserve the entire block.

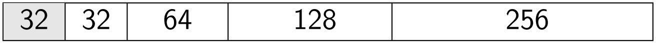




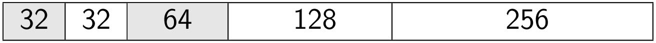
Continuing



and finally:

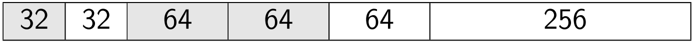


Now, we request 63 bytes (so we need 64 bytes of space) we have an exact place to put this so we allocate there:

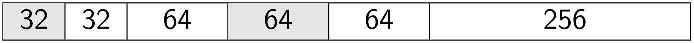


Continuing

Suppose now we need 40 bytes (so actually 41). We need a 64 byte block we don't have. We split the 128 block.



Now let's say we free the allocated 63 byte block:



Continuing

Further, let's say we also free the allocated 19 byte block:

32	32	64	64	64	256

Notice that 32 and its neighbouring buddy are both free so we collapse!

64	64	64	64	256

We can collapse again!

128	64	64	256

Continuing

Lastly, we free the allocated 40 byte block:

128	64	64	256

and we collapse:

128	128	256

and again:

256	256

and again:

512

Automatic Memory Management

Some languages are far too nice to you and they will take care of all of the garbage collecting for you so that you don't need to delete when you are done using the memory. Here are some examples from Java:

```
int f(){
  MyClass ob = new MyClass(); // Java
  ...
} //ob no longer accessible
// garbage collector reclaims.
```

Automatic Memory Management

Second example:

```
int f(){
  MyClass ob2 = null;
  if(x == y){
    MyClass ob1 = new MyClass();
    ob2 = ob1;
  }//ob1 goes out of scope;
  //BUT ob2 still holds the object
  //so not reclaimed
  ...
} //ob2 no longer accessible;
//no one else holds the obj so reclaimed
```

Technique 1: Reference Counting

- For each heap block, keep track of the number of pointers that point to it.
- Must watch every pointer and update reference counts each time a pointer is reassigned (decrement the old one and increment the new one).
- If a block's reference count reaches 0, then reclaim it.
- What issues are there to this?

Technique 1: Reference Counting

- For each heap block, keep track of the number of pointers that point to it.
- Must watch every pointer and update reference counts each time a pointer is reassigned (decrement the old one and increment the new one).
- If a block's reference count reaches 0, then reclaim it.
- What issues are there to this?
- If a block points to another block and vice versa (and nothing points to the cluster) then the cluster is unreachable and should be cleaned.

Technique 2: Mark and Sweep

- Scan the entire stack [and global variables] and search for pointers.
- Mark the heap blocks that have pointers to them. If these contain pointers, continue following etc.
- Then scan the heap: reclaim any blocks that aren't marked.
- Boils down to a graph traversal problem.

Technique 3: Copying the Collector

- Idea: Split the heap into two halves, say H_1 and H_2
- Allocate memory in H_1. When full (or when you cannot find enough memory), copy H_1 into H_2 (can use a mark and sweep here)
- After the copy, H_2 has all of the memory stored contiguously.
- Once finished copying, begin allocation to H_2 (that is, reverse the roles of H_1 and H_2
- We have no fragmentation in our memory; new and delete are very quick BUT we can only use half the heap at a time.
- This and common other variants (where you split your heap into 3 or 4 regions reserving only 1 region for the copy step) are widely used.

Loaders

Let's write an operating system! (Sort of...)

```
repeat:
  p <- next program to run
  copy P into memory at 0x0
  jalr $0
  beq $0, $0, repeat
```

Note that `mips.twoints` and `mips.array` do this.

Issues

- The operating system is a program that needs to be in memory - where should it be?

Issues

- The operating system is a program that needs to be in memory - where should it be?
- How can we fix this?
    - Could choose different addresses at assembly time - but how does the loader know where to put them?
    - Still have the issue of collisions.
    - Let's just let the loader decide...

Loader's New role:

Loader's job:

- Take a program P as input
- Find a location α in memory for P
- Copy P to memory, starting at α
- Return α to the OS.

OS 2.0:

```
repeat:
  p <- next program to run
  $3 <- loader(P)
  jalr $3
  beq $0, $0, repeat
```

Code for Loader From Previous Slide

Input: machine code in the form of words `w1` up to `wk`. Also, `n = k + stack_space` (pick how much stack space you want!).

```
for i from 1 to k:
  MEM[alpha + 4*i] = wi
$30 <- alpha + 4*n
return alpha
```

As always, there are issues! What goes wrong?

Problems

Labels may be resolved to incorrect addresses! Loader needs to fix this! What needs to be fixed for labels?

```
.word id <-- need to add alpha to id
.word constant <-- do not relocate!
branching commands etc. <-- do not relocate!
```

Seems easy but there's a problem...

Machine Code

Recall that we translate assembly code into machine code (zeroes and ones). Given:

```
0x00000018
```

Was this line of code a `.word constant` or a `.word id`? You can't know!

OS 3.0:

```
repeat:
  p <- next program to run
  $3 <- load_and_relocate(P)
  jalr $3
  beq $0, $0, repeat
```

(still need to determine how we relocate!)

Object Code

- Usually, the output of assemblers isn't pure machine code; it is *object code*!
- We've seen this - these are our MERL files (MIPS Executable Relocatable Linkable)
- In this file, we need *the code*, *location of .word id* and some auxiliary information.

MERL File in Assembly

```
beq $0, $0, 2
.word endfile ;file length
.word endcode ;code + header
;Insert MIPS Assembly here
.word 0x4 ;Constant No relocation
.word 0x8 ;Constant No relocation
.word A ;Needs relocation
B: jr$31
A : beq $0, $0, B ;No relocation
endcode:  ;MERL symbol table
.word 0x1 ;Format code 1 means relocate!
.word 0x14 ;Location of A.
endfile:
```

Loading

Requires two passes:

- Pass 1: Record the size of the file, start counting addresses at `0x0c` instead of `0x0` and record the location of `.word id` instructions
- Pass 2: Output the header, MIPS machine code and relocation table.

Caveat

Note: Even with this, it is possible to write code that only works at address 0:

```
lis $2
.word 12
jr $2
jr $31
```

You should never encode address as anything other than labels so that your loader can update the references! (That is, never hardcode addresses)

Loader Relocation Algorithm

```
read_line() //skip cookie!
endMod <-- read_line() //End of MERL file
codeLen <- read_line() - 12 //No header in codeLe
alpha <- findFreeRAM(codeLen)
for(int i = 0; i <codeLen; i+= 4)
  MEM[alpha + i] <- read_line()
i <- codeLen + 12 //start of reloc. table
while (i < endMod)
  format <- read_line()
  if (format == 1)
    rel <-- read_line()
    //go forward by alpha and back by header len
    //alpha + rel - 12 since we don't load header
    MEM[alpha + rel - 12] += alpha - 12
  else
    ERROR
  i += 8
```

Linkers

- How do we resolve situations where we have labels in different files?
- One option is to `cat` all such files together; but why should we have to reassemble these files more than once?
- Could we not assemble the files *first* and then cat?

Linkers

- How do we resolve situations where we have labels in different files?
- One option is to `cat` all such files together; but why should we have to reassemble these files more than once?
- Could we not assemble the files *first* and then cat?
- Sure but remember only one piece can be at `0x0` at a time - so these assembled files need to be MERL files and not just MIPS files.
- Worse still, recall that concatenating two MERL files does *not* give a valid MERL file!

But Wait! There's More!

- Still worse - we haven't really resolved the issue of labels in different files!
- We need to modify our assembler - when we encounter a `.word` where the label is not in the file, we need to print a place holder, `0x0` in our case, and indicate that we cannot run this program until the value of the `id` is given.
- For example, below, `a.asm` on the left and `b.asm` on the right (recall `asm` for assembly):

  `a.asm`

  ```
  lis $3
  .word label
  ```

  `b.asm`

  ```
  label: sw $4, -4($30)
  ```

  you cannot run `a.asm` without linking with `b.asm`.
- We will need to extend our MERL file so that it can notify us when we need to assemble with multiple files.

Error Checking

- Naively this works, but we make typos. Consider this:

  ```
  lis $3
  .word banana
  bananas:
  ```

  Did we make a mistake? Maybe we meant `.word bananas`? How could we recognize such errors? Without any other changes, our assembler will believe that a label `banana` exists somewhere and would load this with a placeholder (which might not be what we want!)
- How can we tell our assembler what is an error and what is intentional?
- Hint: You've already been doing this!

New (sort of...) Directive

- `.import id` is the command that asks for which `id` we should be linking in.
- This will not assemble to a word of MIPS.
- Errors occur if the label `id` is not in the current file *and* there is no `.import id` in the file.
- We need to add entries in the MERL symbol table
- Include format code `0x11` for External Symbol Reference (ESR).

ESR Entry

What needs to be in an ESR entry?

1. Where the symbol is being used
2. The name of said symbol.

Format:

```
0x11 ;Format code
;location used
;length of name of symbol (n)
;1st ASCII character of name of symbol
;2nd ASCII character of name of symbol
;...
;nth ASCII character of name of symbol
```

Still Problems

What about if labels are duplicated? Suppose we have a `c.asm` along with our other two files that has:

```
label: add $1, $0, $0
;more code here
beq $1, $0, label
```

Here, we want `label` to not be exported; rather it should be self-contained.

We include another assembler directive and another MERL entry type to handle this, namely the `.export` directive.

Exporting

- `.export label` will make `label` available for linking with other files. As with `.import`, it does not translate to a word in MIPS. It tells the assembler to make an entry in the MERL symbol table.
- The assembler makes an ESD, or an External Symbol Definition, for these types of words. It follows this format:

```
0x05 ;Format code
;address the symbol represents
;length of name of symbol (n)
;1st ASCII character of name of symbol
;2nd ASCII character of name of symbol
;...
;nth ASCII character of name of symbol
```

Now, our MERL file contains the code, the address that need relocating, as well as the addresses and names of each ESR and ESD. Our linker now has everything it needs to do its job

Linker Algorithm 1 of 3

Algorithm 1 Algorithm to link two merl files m_1 and m_2

```
1: $\alpha = m_1.codeLen - 12$
2: Relocate $m_2$ by $\alpha$.
3: Add $\alpha$ to each entry of $m_2$'s symbol table
4: **if** labels of the exports of $m_1$ and $m_2$ are non-empty **then**
5:     ERROR
6: **end if**
7: //(See next page...)
```

Linker Algorithm 2 of 3

Algorithm 2 Algorithm to link two merl files m_1 and m_2 con't

```
 1: for $\langle addr_1, label \rangle$ in $m_1$'s imports do
 2:     if there exists a $\langle addr_2, label \rangle$ in $m_2$'s exports then
 3:         $m_1.code[addr_1] = addr_2$
 4:         Remove $\langle addr_1, label \rangle$ from $m_1$'s imports
 5:         Add $addr_1$ to $m_1$'s relocates:
 6:     end if
 7: end for
 8: for $\langle addr_2, label \rangle$ in $m_2$'s imports do
 9:     if there exists a $\langle addr_1, label \rangle$ in $m_1$'s exports then
10:         $m_2.code[addr_2] = addr_1$
11:         Remove $\langle addr_2, label \rangle$ from $m_2$'s imports
12:         Add $addr_2$ to $m_2$'s relocates:
13:     end if
14: end for
15: // See last page
```

Linker Algorithm 3 of 3

Algorithm 3 Algorithm to link two merl files m_1 and m_2 con't

```
1: imports = union of $m_1$ and $m_2$'s imports
2: exports = union of $m_1$ and $m_2$'s exports
3: relocates = union of $m_1$ and $m_2$'s relocates
4: output 0x10000002 (MERL cookie)
5: output total codeLen + total (import, export, relocates) + 12
6: output total codeLen + 12
7: output $m_1$ code
8: output $m_2$ code
9: output Imports, exports, relocates
```