Warm Up Problem

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

CS 241 Lecture 21

Heap Management

With thanks to Brad Lushman, Troy Vasiga and Kevin Lanctot

Strength Reduction

In the real world, addition is much faster than multiplication. If you can avoid multiplying this is best. Consider `code(n*2;)`:

```
;load n into $3
sw $3, 0($30)
sub $30, $30, $4
lis $3
.word 2
lw $5, 0($30)
add $30, $30, $4
mult $3, $5
mflo $3
```

This is more work than:

```
add $3, $3, $3
```

Inlining Procedures

Consider the following:

```
int f(int a, int b){
  return a + b;
}

int wain(int a, int b){
   return f(a,b)
}
```

It would be far faster to use the equivalent code:

```
int wain(int a, int b){
   return a + b;
}
```

as we wouldn't need to have all of the overhead associated with the function call!

Inlining Procedures

Does the previous slide always give us shorter code?

Inlining Procedures

Does the previous slide always give us shorter code?

- If we call `f` a lot, then we might get many copies of `f`.
- This can be a win if the body of `f` is shorter than the code to call `f`.
- Sometimes this is harder to tell. Can also cause problems with recursive calls.
- If all calls to `f` are inline, then we never need the code for the procedure `f`.

Tail Recursion

Consider the following:

```
int fact(int n, int acc){
  if (n == 0){return acc;}
  else {return fact(n-1, acc*n);}
}
```

- Notice that the last operation made in each branch is just returning a value (that is, after the recursive call, there is no more work to do in this stack frame).
- We could reuse the current stack frame to save some operations.
- This is called **tail recursion**.
- Can we do this in our WLP4 grammar?

Tail Recursion

Consider the following:

```
int fact(int n, int acc){
  if (n == 0){return acc;}
  else {return fact(n-1, acc*n);}
}
```

- Notice that the last operation made in each branch is just returning a value (that is, after the recursive call, there is no more work to do in this stack frame).
- We could reuse the current stack frame to save some operations.
- This is called **tail recursion**.
- Can we do this in our WLP4 grammar? No! return statements are not allowed in if statements and we cannot refactor the code to get this behaviour!

Hypothetically

Let's suppose that we did allow for multiple return statements like this however. What would tail recursive code look like? Let's look at

```
return fact(n-1, acc*n);
```

which is of the form `factor` $\rightarrow$ `ID(expr1,...,exprn)`. Then this would be (see next slide)...

Code for Factor

factor → ID(expr1,...,exprn)

```
code(factor) = code(expr1) + push ($3)
             + ...
             + code(exprn) + push($3)
             + pop($5)
             + sw $5, 4n($29)
             + pop($5)
             + sw $5, 4(n-4)($29)
             + ...
             + pop($5)
             + sw$5, 4($29)
             + lis $5
             +.word FID
             + add $30, $29, $4 ;reset stack ptr
             + jr $5 ;NOT jalr don't save
                     ;$31, $29 or pop args
```

Code for Procedure

```
code(procedure) = FID: sub $29, $30, $4
                + push(registers)
                + code(statements)
                + code(expr)
                + pop(registers)
                + add $29, $30, $4
                + jr $31
```

Making This Work

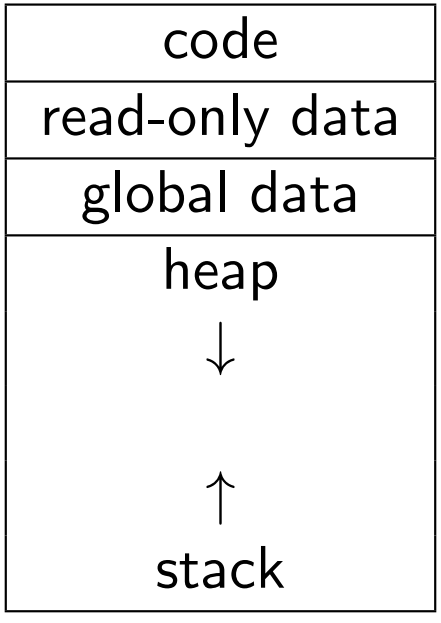
- To make the above work, what is often done is to write equivalent but simpler code *inside the current language (WLP4)*.
- This is sometimes called *intermediate code*.
- Once you have this code, run this through your compiler and then use the MIPS output.
- This concludes our brief discussion on optimization. Best of luck with A10!

Heap Management

- In our course, we gave you a library to deal with all of memory management features which included `init`, `new` and `delete`.
- This allows for data to exist in memory that is out of scope, that is, out of the boundaries of your stack frame.
- How do we manage this memory?
- The memory not on the stack is said to be *on the heap*. The command `init` initializes a heap for us to use.
- Much more problematic than a stack to take care of. Stacks are nice and ordered - calls can be made to heaps using delete or new in arbitrary orders making it harder to manage.

Heap Management

Our World:



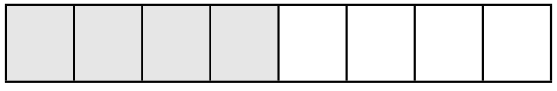
Note that our stack contains a pointer to where the heap is.

Heaps

- But how exactly does a heap work?
- We have a variety of implementations that we will discuss ranging in practicality and utility.
- Our first example makes use of the fact that heap management is easy if you never allow for `delete`

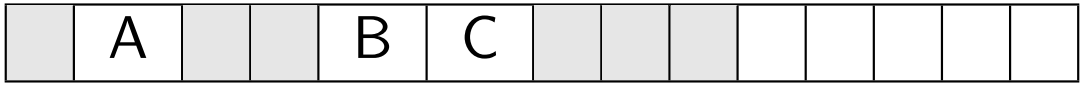
Example 1: No Reclamation of Memory and Fixed Blocks

- After `init`, you get two pointers, one to the start of memory on the heap and one at the end.
- Initialization is $O(1)$.
- Allocation is also $O(1)$.
- Never delete so this is fine.
- Clearly not our best choice; we run out of memory quickly if we don't reuse reclaimed memory. (Think: actual garbage waste)

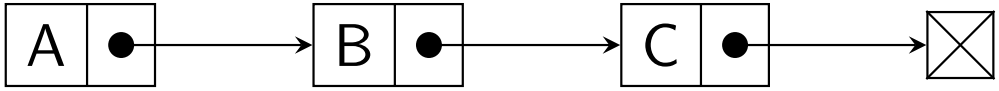


Example 2: Explicit Reclamation and Linked List of Fixed Sized Blocks

- We can keep the fixed size idea but this time, we keep track of a free list (a linked list of free memory blocks) and we can allocate from this linked list.



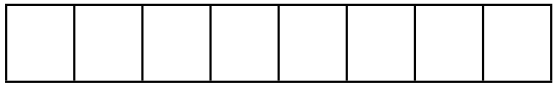
where A, B, and C are the starting memory addresses of the free blocks and then the associated linked list:



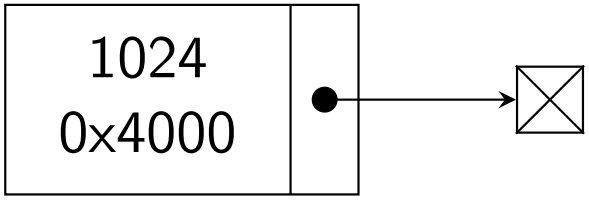
- Works only because you allow a single block of memory each time. You keep track of the memory in the free list and allocate from the free list first if memory is being requested.

Example 3: Variable Sized Blocks

- Idea: We once again used a linked list but here our linked list will store a number of bytes, where those bytes can be found and the next node.
- Init: Start with the entire heap being free. As an example, suppose we have 1024 bytes:
  Memory (first address is at 0x4000):

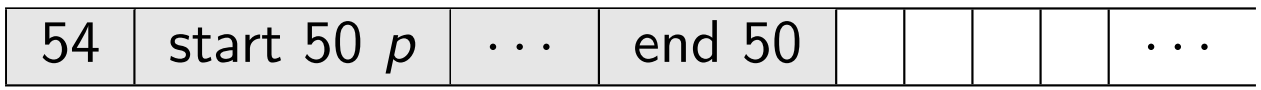


  Free Linked List:

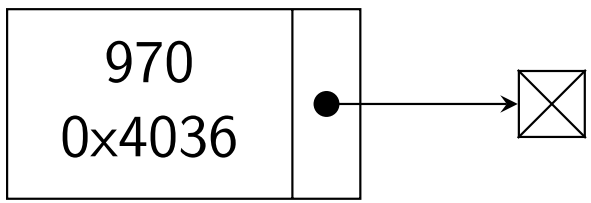


Next

We want to allocate 50 bytes. What we will do is allocate 54 bytes - the first 4 bytes are the size of the block (an integer) and the rest is the number of bytes. We return a pointer to the start of the 50 bytes.
Memory:

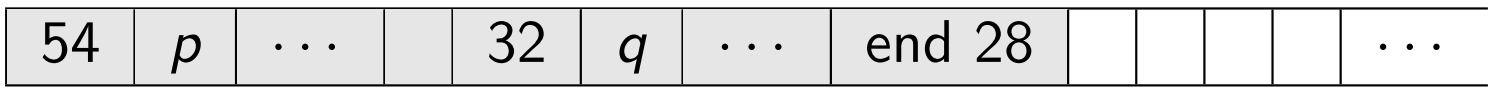


Free Linked List:

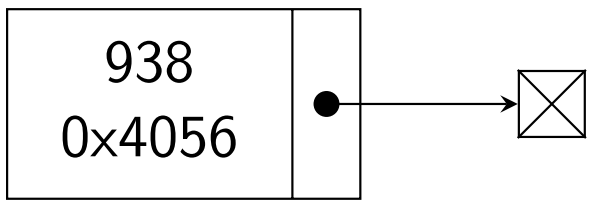


Next

Next, we want to allocate 28 bytes. What we will do is allocate 32 bytes - the first 4 bytes are the size of the block (an integer) and the rest is the number of bytes. We return a pointer to the start of the 28 bytes.
Memory:

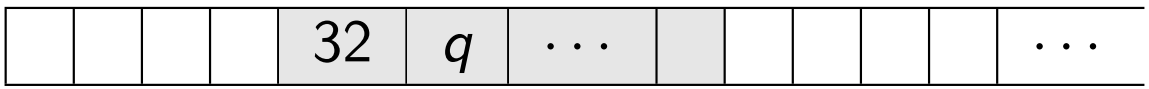


Free Linked List:

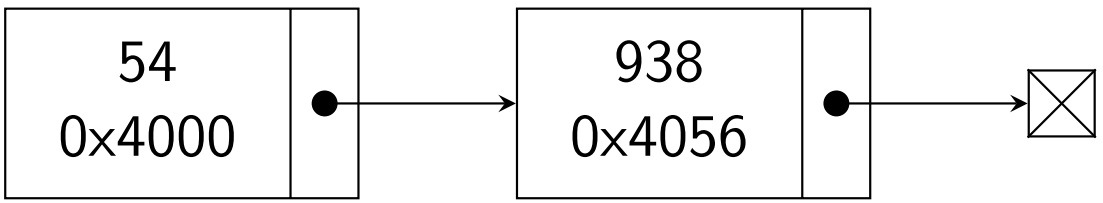


Free!

Next, we free the 50 bytes!
Memory:

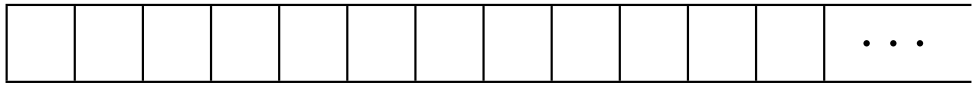


Free Linked List:

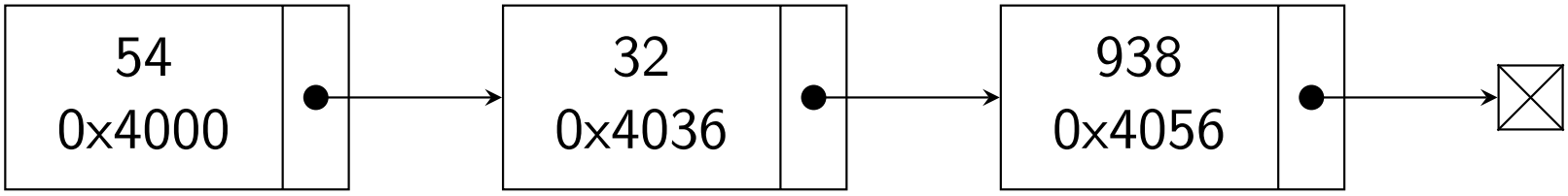


Free!

Lastly, we free the other 32 bytes
Memory:

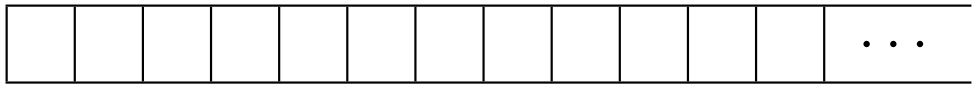


Free Linked List:

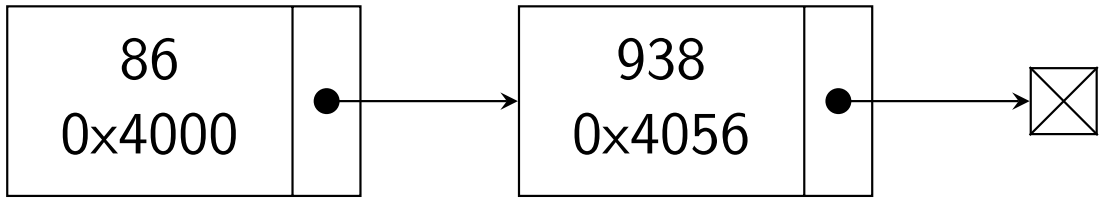


Consolidate

Now, we can do some consolidation. Notice that $54 + 0\text{x}4000 = 0\text{x}4036$ and so the first two nodes in our linked list can collapse to a single node:
Memory:

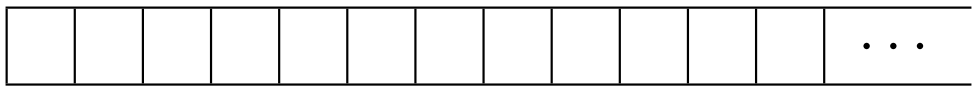


Free Linked List:

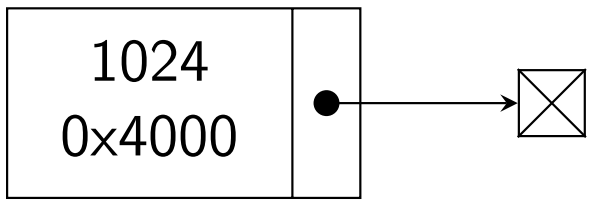


Consolidate

Again we can consolidate. Notice that $86 + \texttt{0x4000} = \texttt{0x4056}$ and so the first two nodes in our linked list can collapse to a single node:
Memory:



Free Linked List:



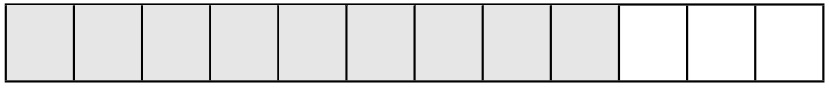
Issues

The biggest issue with this approach is fragmentation. Suppose we have 48 bytes and we make the following calls:

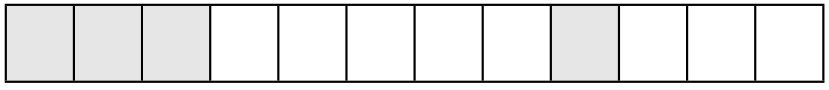
Allocate 12

Allocate 20

Allocate 4



Free 20



Allocate 8

Cannot allocate 16, despite having 24 bytes free!

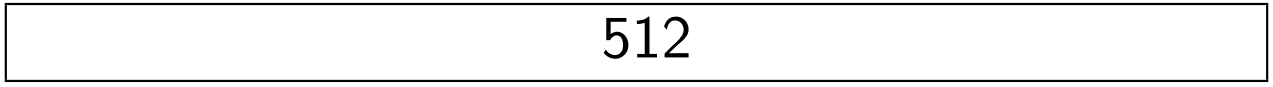
Dealing With Fragmentation

Heuristics:

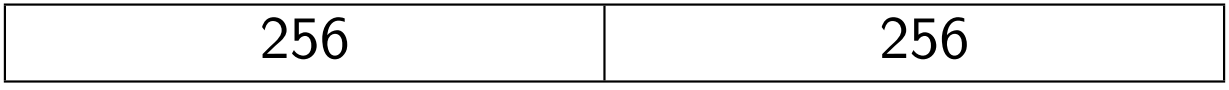
- First fit: Put memory in the first available spot
- Best fit: Put the block in an exact match (or as close to it so there is less waste)
- Problem: Both would require a large amount of time investment (potentially) and we want fast memory access.
- Other ideas include *dmalloc* and `binary buddy system` or *buddy memory allocation*. We discuss the latter.

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

64	64	128	256

and finally:

32	32	64	128	256

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

32	32	64	128	256

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.

32	32	64	64	64	256

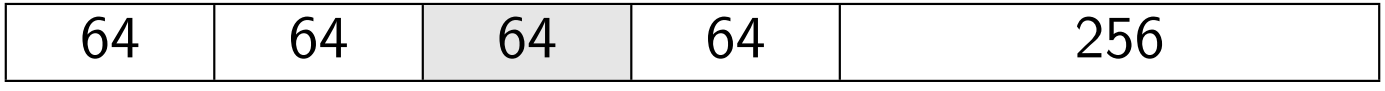
Now let's say we free the allocated 63 byte block:

32	32	64	64	64	256

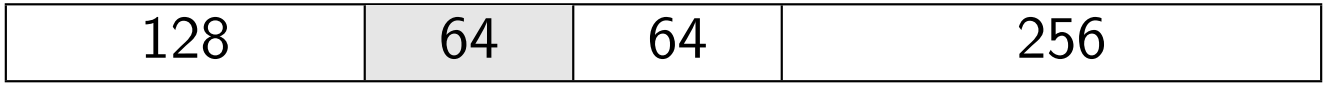
Continuing

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

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

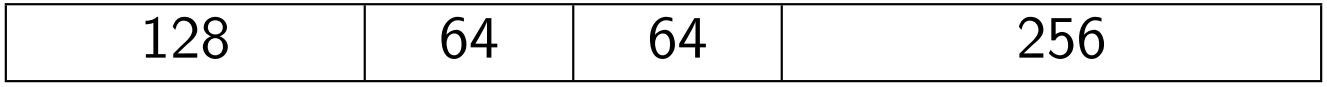


We can collapse again!

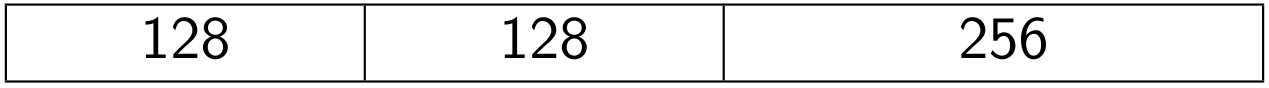


Continuing

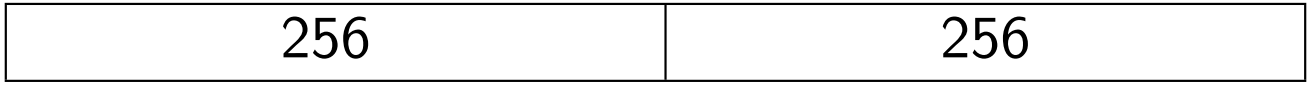
Lastly, we free the allocated 40 byte block:



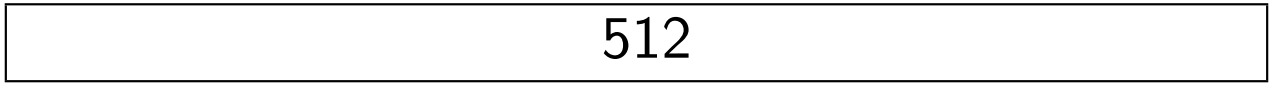
and we collapse:



and again:



and again:



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 ob1 = 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.