Runtime Environments

Ronghui Gu
Spring 2019
Columbia University

* Course website: https://www.cs.columbia.edu/rgu/courses/4115/spring2019
** These slides are borrowed from Prof. Edwards.
Storage Classes
Storage Classes and Memory Layout

**Stack**: objects created/destroyed in last-in, first-out order

**Heap**: objects created/destroyed in any order; automatic garbage collection optional

**Static**: objects allocated at compile time; persist throughout run
### Static Objects

```java
class Example {
    public static final int a = 3;
    public void hello() {
        System.out.println("Hello");
    }
}
```

#### Examples
- Static class variable
- String constant “Hello”
- Information about the Example class

#### Advantages
- Zero-cost memory management
- Often faster access (address a constant)
- No out-of-memory danger

#### Disadvantages
- Size and number must be known beforehand
- Wasteful
The Stack and Activation Records
Idea: some objects persist from when a procedure is called to when it returns.

Naturally implemented with a stack: linear array of memory that grows and shrinks at only one boundary.

Natural for supporting recursion.

Each invocation of a procedure gets its own frame (activation record) where it stores its own local variables and bookkeeping information.
int foo(int a, int b) {
    int c, d;
    bar(1, 2, 3);
}
Recursive Fibonacci

(Real C)

```c
int fib(int n) {
    if (n<2)
        return 1;
    else
        return fib(n-1) + fib(n-2);
}
```

(Assembly-like C)

```c
int fib(int n) {
    int tmp1, tmp2, tmp3;
    tmp1 = n < 2;
    if (!tmp1) goto L1;
    return 1;
L1: tmp1 = n - 1;
    tmp2 = fib(tmp1);
L2: tmp1 = n - 2;
    tmp3 = fib(tmp1);
L3: tmp1 = tmp2 + tmp3;
    return tmp1;
}
```

```
fib(3)
  `   `   
  fib(2) fib(1)
    `   `
  fib(1) fib(0)
```

Executing fib(3)
int fib(int n) {
    int tmp1, tmp2, tmp3;
    tmp1 = n < 2;
    if (!tmp1) goto L1;
    return 1;
L1: tmp1 = n - 1;
    tmp2 = fib(tmp1);
L2: tmp1 = n - 2;
    tmp3 = fib(tmp1);
L3: tmp1 = tmp2 + tmp3;
    return tmp1;
}

n = 3
int fib(int n) {
    int tmp1, tmp2, tmp3;
    tmp1 = n < 2;
    if (!tmp1) goto L1;
    return 1;
L1: tmp1 = n - 1;
    tmp2 = fib(tmp1);
L2: tmp1 = n - 2;
    tmp3 = fib(tmp1);
L3: tmp1 = tmp2 + tmp3;
    return tmp1;
}

n /equal.osf /three.osf
return address
last frame pointer
tmp/one.osf /equal.osf /two.osf
tmp/two.osf /equal.osf
tmp/three.osf /equal.osf
n /equal.osf /two.osf
FP
SP
/seven.osf
int fib(int n) {
    int tmp1, tmp2, tmp3;
    tmp1 = n < 2;
    if (!tmp1) goto L1;
    return 1;
L1: tmp1 = n - 1;
    tmp2 = fib(tmp1);
L2: tmp1 = n - 2;
    tmp3 = fib(tmp1);
L3: tmp1 = tmp2 + tmp3;
    return tmp1;
}
int fib(int n) {
    int tmp1, tmp2, tmp3;
    tmp1 = n < 2;
    if (!tmp1) goto L1;
    return 1;
L1: tmp1 = n - 1;
    tmp2 = fib(tmp1);
L2: tmp1 = n - 2;
    tmp3 = fib(tmp1);
L3: tmp1 = tmp2 + tmp3;
    return tmp1;
}
```c
int fib(int n) {
    int tmp1, tmp2, tmp3;
    tmp1 = n < 2;
    if (!tmp1) goto L1;
    return 1;
L1: tmp1 = n - 1;
    tmp2 = fib(tmp1);
L2: tmp1 = n - 2;
    tmp3 = fib(tmp1);
L3: tmp1 = tmp2 + tmp3;
    return tmp1;
}
```
int fib(int n) {
    int tmp1, tmp2, tmp3;
    tmp1 = n < 2;
    if (!tmp1) goto L1;
    return 1;
L1: tmp1 = n - 1;
    tmp2 = fib(tmp1);
L2: tmp1 = n - 2;
    tmp3 = fib(tmp1);
L3: tmp1 = tmp2 + tmp3;
    return tmp1;
}
int fib(int n) {
    int tmp1, tmp2, tmp3;
    tmp1 = n < 2;
    if (!tmp1) goto L1;
    return 1;
L1: tmp1 = n - 1;
    tmp2 = fib(tmp1);
L2: tmp1 = n - 2;
    tmp3 = fib(tmp1);
L3: tmp1 = tmp2 + tmp3;
    return tmp1;
}
int fib(int n) {
    int tmp1, tmp2, tmp3;
    tmp1 = n < 2;
    if (!tmp1) goto L1;
    return 1;
    L1: tmp1 = n - 1;
    tmp2 = fib(tmp1);
    L2: tmp1 = n - 2;
    tmp3 = fib(tmp1);
    L3: tmp1 = tmp2 + tmp3;
    return tmp1;
}
int fib(int n) {
    int tmp1, tmp2, tmp3;
    tmp1 = n < 2;
    if (!tmp1) goto L1;
    return 1;
L1: tmp1 = n - 1;
    tmp2 = fib(tmp1);
L2: tmp1 = n - 2;
    tmp3 = fib(tmp1);
L3: tmp1 = tmp2 + tmp3;
    return tmp1;
}
int fib(int n) {
    int tmp1, tmp2, tmp3;
    tmp1 = n < 2;
    if (!tmp1) goto L1;
    return 1;
L1: tmp1 = n - 1;
    tmp2 = fib(tmp1);
L2: tmp1 = n - 2;
    tmp3 = fib(tmp1);
L3: tmp1 = tmp2 + tmp3;
    return tmp1;
}
Local arrays with fixed size are easy to stack.

```c
void foo()
{
    int a;
    int b[10];
    int c;
}
```

<table>
<thead>
<tr>
<th>return address</th>
<th>← FP</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td></td>
</tr>
<tr>
<td>b[9]</td>
<td></td>
</tr>
<tr>
<td>:</td>
<td></td>
</tr>
<tr>
<td>b[0]</td>
<td></td>
</tr>
<tr>
<td>c</td>
<td>← FP − 48</td>
</tr>
</tbody>
</table>
Variable-sized local arrays aren’t as easy.

```c
void foo(int n)
{
    int a;
    int b[n];
    int c;
}
```

Doesn’t work: generated code expects a fixed offset for `c`. Even worse for multi-dimensional arrays.

<table>
<thead>
<tr>
<th>return address</th>
<th>← FP</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>← FP</td>
</tr>
<tr>
<td>b[n-1]</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
<tr>
<td>b[0]</td>
<td></td>
</tr>
<tr>
<td>c</td>
<td>← FP</td>
</tr>
</tbody>
</table>
Allocating Variable-Sized Arrays

As always:
add a level of indirection

```c
void foo(int n)
{
    int a;
    int b[n];
    int c;
}
```

Variables remain constant offset from frame pointer.
Implementing Nested Functions with Access Links

```
let a x s =
  let b y =
    let c z = z + s in
    let d w = c (w+1) in
    d (y+1) in (* b *)
  let e q = b (q+1) in
  e (x+1) (* a *)
```

What does “a 5 42” give?

(access link) •

a:
  x = 5
  s = 42
Implementing Nested Functions with Access Links

```
let a x s =
  let b y =
    let c z = z + s in
    let d w = c (w+1) in
    d (y+1) in (* b *)
  let e q = b (q+1) in
  e (x+1) (* a *)
```

What does “a 5 42” give?

```
(access link)

a: x = 5
  s = 42

e:
  (access link)

  q = 6
```

Implementing Nested Functions with Access Links

```plaintext
let a x s =
  let b y =
    let c z = z + s in
    let d w = c (w+1) in
    d (y+1) in (* b *)
  in (* b *)
  let e q = b (q+1) in
  e (x+1) (* a *)
```

What does “a 5 42” give?

```
a:
  x = 5
  s = 42

b:
  y = 7

e:
  q = 6
```

(access link)
Implementing Nested Functions with Access Links

```plaintext
let a x s =
  let b y =
    let c z = z + s in
    let d w = c (w+1) in
    d (y+1) (* b *)
  let e q = b (q+1) in
  e (x+1) (* a *)
```

What does “a 5 42” give?

```
| a: x = 5 |
| s = 42 |
| e: q = 6 |
| b: y = 7 |
| d: w = 8 |
```
Implementing Nested Functions with Access Links

```
let a x s =
  let b y =
    let c z = z + s in
    let d w = c (w+1) in
    d (y+1) in (* b *)
  let e q = b (q+1) in
  e (x+1) (* a *)
```

What does “a 5 42” give?

<table>
<thead>
<tr>
<th>a:</th>
<th>x = 5</th>
<th>s = 42</th>
</tr>
</thead>
<tbody>
<tr>
<td>b:</td>
<td>y = 7</td>
<td></td>
</tr>
<tr>
<td>c:</td>
<td>z = 9</td>
<td></td>
</tr>
<tr>
<td>d:</td>
<td>w = 8</td>
<td></td>
</tr>
<tr>
<td>e:</td>
<td>q = 6</td>
<td></td>
</tr>
</tbody>
</table>

(access link)
In-Memory Layout Issues
Modern processors have byte-addressable memory.

The IBM 360 (c. 1964) helped to popularize byte-addressable memory.

Many data types (integers, addresses, floating-point numbers) are wider than a byte.

16-bit integer: \[1\, 0\]

32-bit integer: \[3\, 2\, 1\, 0\]
Modern memory systems read data in 32-, 64-, or 128-bit chunks:

```
3 2 1 0
7 6 5 4
11 10 9 8
```

Reading an aligned 32-bit value is fast: a single operation.

```
3 2 1 0
7 6 5 4
11 10 9 8
```

It is harder to read an unaligned value: two reads plus shifting

```
3 2 1 0
7 6 5 4
11 10 9 8
```

SPARC and ARM prohibit unaligned accesses

MIPS has special unaligned load/store instructions

x86, 68k run more slowly with unaligned accesses
To avoid unaligned accesses, the C compiler pads the layout of unions and records. Rules:

- Each $n$-byte object must start on a multiple of $n$ bytes (no unaligned accesses).
- Any object containing an $n$-byte object must be of size $mn$ for some integer $m$ (aligned even when arrayed).

```c
struct padded {
    int x; /* 4 bytes */
    char z; /* 1 byte */
    short y; /* 2 bytes */
    char w; /* 1 byte */
};
```

```c
struct padded {
    char a; /* 1 byte */
    short b; /* 2 bytes */
    short c; /* 2 bytes */
};
```
To avoid unaligned accesses, the C compiler pads the layout of unions and records. Rules:

- Each $n$-byte object must start on a multiple of $n$ bytes (no unaligned accesses).
- Any object containing an $n$-byte object must be of size $mn$ for some integer $m$ (aligned even when arrayed).

```c
struct padded {
    int x; /* 4 bytes */
    char z; /* 1 byte */
    char w; /* 1 byte */
    short y; /* 2 bytes */
};
```

```c
struct padded {
    char a; /* 1 byte */
    short b; /* 2 bytes */
    short c; /* 2 bytes */
};
```
A C struct has a separate space for each field; a C union shares one space among all fields.

```c
union int char {
    int i; /* 4 bytes */
    char c; /* 1 byte */
};

union twostructs {
    struct {
        char c; /* 1 byte */
        int i; /* 4 bytes */
    } a;
    struct {
        short s1; /* 2 bytes */
        short s2; /* 2 bytes */
    } b;
};
```

or

```
s2 s2 s1 s1
```
Arrays

Basic policy in C: an array is just one object after another in memory.

```c
int a[10];
```

This is why you need padding at the end of structs.

```c
struct {
    int a;
    char c;
} b[2];
```
Arrays and Aggregate types

The largest primitive type dictates the alignment

```c
struct {
    short a;
    short b;
    char c;
} d[4];
```
Arrays of Arrays

char a[4];

char a[3][4];

```
char a[4];
```

```
char a[3][4];
```

```
```

```
a[0][3] a[0][2] a[0][1] a[0][0]
a[1][3] a[1][2] a[1][1] a[1][0]
```

```
a[0]
a[1]
a[2]
```

```
a[0]
a[1]
a[2]
```

```
a[0]
a[1]
a[2]
```

```
a[0]
a[1]
a[2]
```
The Heap
Static works when you know everything beforehand and always need it.

Stack enables, but also requires, recursive behavior.

A *heap* is a region of memory where blocks can be allocated and deallocated in any order.

(These heaps are different than those in, e.g., heapsort)
```c
struct point {
    int x, y;
};

int play_with_points(int n) {
    int i;
    struct point *points;

    points = malloc(n * sizeof(struct point));

    for (i = 0; i < n; i++) {
        points[i].x = random();
        points[i].y = random();
    }

    /* do something with the array */
    free(points);
}
```
Dynamic Storage Allocation
Dynamic Storage Allocation

↓ free([gray shaded block])
Dynamic Storage Allocation
Dynamic Storage Allocation

\[\downarrow \text{free}()\]

\[\downarrow \text{malloc}()\]
Dynamic Storage Allocation

Rules:

- Each allocated block contiguous (no holes)
- Blocks stay fixed once allocated

`malloc()`

- Find an area large enough for requested block
- Mark memory as allocated

`free()`

- Mark the block as unallocated
Simple Dynamic Storage Allocation

Maintaining information about free memory

Simplest: Linked list

The algorithm for locating a suitable block

Simplest: First-fit

The algorithm for freeing an allocated block

Simplest: Coalesce adjacent free blocks
Simple Dynamic Storage Allocation

malloc

free

/two.osf/five.osf
Simple Dynamic Storage Allocation

malloc( )
Simple Dynamic Storage Allocation

malloc()
Simple Dynamic Storage Allocation

malloc()
Simple Dynamic Storage Allocation

malloc( )

free( )
Dynamic Storage Allocation

Many, many other approaches.
Other “fit” algorithms
Segregation of objects by size
More clever data structures
Fragmentation

`malloc( )` seven times give

`free()` four times gives

`malloc( )`?

Need more memory; can’t use fragmented memory.

Zebra  Tapir
Standard CS solution: Add another layer of indirection. 

Always reference memory through “handles.”

The original Macintosh did this to save memory.
Fragmentation and Handles

Standard CS solution: Add another layer of indirection.
Always reference memory through “handles.”

The original Macintosh did this to save memory.
Automatic Garbage Collection
Automatic Garbage Collection

Entrust the runtime system with freeing heap objects

Now common: Java, C#, Javascript, Python, Ruby, OCaml and most functional languages

**Advantages**

- Much easier for the programmer
- Greatly improves reliability: no memory leaks, double-freeing, or other memory management errors

**Disadvantages**

- Slower, sometimes unpredictably so
- May consume more memory
Reference Counting

What and when to free?

• Maintain count of references to each object
• Free when count reaches zero

let a = (42, 17) in
let b = [a;a] in
let c = (1,2)::b in
b
Reference Counting

What and when to free?

- Maintain count of references to each object
- Free when count reaches zero

```plaintext
let a = (42, 17) in
let b = [a;a] in
let c = (1,2)::b in
b
```
Reference Counting

What and when to free?

• Maintain count of references to each object
• Free when count reaches zero

let a = (42, 17) in
let b = [a;a] in
let c = (1,2)::b in
b
Reference Counting

What and when to free?

- Maintain count of references to each object
- Free when count reaches zero

let a = (42, 17) in
let b = [a;a] in
let c = (1,2)::b in
b
Reference Counting

What and when to free?

- Maintain count of references to each object
- Free when count reaches zero

```haskell
let a = (42, 17) in
let b = [a;a] in
let c = (1,2)::b in
b
```
Reference Counting

What and when to free?

- Maintain count of references to each object
- Free when count reaches zero

let a = (42, 17) in
let b = [a;a] in
let c = (1,2)::b in
b
Reference Counting

What and when to free?

• Maintain count of references to each object
• Free when count reaches zero

let a = (42, 17) in
let b = [a;a] in
let c = (1,2)::b in
b
Reference Counting

What and when to free?

- Maintain count of references to each object
- Free when count reaches zero

```plaintext
let a = (42, 17) in
let b = [a;a] in
let c = (1,2)::b in
b
```
Reference Counting

What and when to free?

- Maintain count of references to each object
- Free when count reaches zero

```
let a = (42, 17) in
let b = [a;a] in
let c = (1,2)::b in
b
```
Reference Counting

What and when to free?

- Maintain count of references to each object
- Free when count reaches zero

```ocaml
let a = (42, 17) in
let b = [a;a] in
let c = (1,2)::b in
b
```
Issues with Reference Counting

Circular structures defy reference counting:

![Diagram of circular structure with nodes a and b]

Neither is reachable, yet both have non-zero reference counts.

High overhead (must update counts constantly), although incremental
Mark-and-Sweep

What and when to free?

- Stop-the-world algorithm invoked when memory full
- Breadth-first-search marks all reachable memory
- All unmarked items freed

```plaintext
let a = (42, 17) in
let b = [a;a] in
let c = (1,2)::b in
b
```
Mark-and-Sweep

What and when to free?

- Stop-the-world algorithm invoked when memory full
- Breadth-first-search marks all reachable memory
- All unmarked items freed

```plaintext
let a = (42, 17) in
let b = [a;a] in
let c = (1,2)::b in
b
```
Mark-and-Sweep

What and when to free?

- Stop-the-world algorithm invoked when memory full
- Breadth-first-search marks all reachable memory
- All unmarked items freed

let a = (42, 17) in
let b = [a;a] in
let c = (1,2)::b in
b
Mark-and-Sweep

What and when to free?

- Stop-the-world algorithm invoked when memory full
- Breadth-first-search marks all reachable memory
- All unmarked items freed

```
let a = (42, 17) in
let b = [a;a] in
let c = (1,2)::b in
b
```
Mark-and-Sweep

Mark-and-sweep is faster overall; may induce big pauses

Mark-and-compact variant also moves or copies reachable objects to eliminate fragmentation

Incremental garbage collectors try to avoid doing everything at once

Most objects die young; generational garbage collectors segregate heap objects by age

Parallel garbage collection tricky

Real-time garbage collection tricky