Runtime Environments

Ronghui Gu
Spring 2020
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

---

Diagram:
- Stack
- Heap
- Static
- Code

Stack Pointer
Program Break
High Memory
Low Memory
### 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.
An Activation Record: The State Before Calling `bar`

```c
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;
}
```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;
}
```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;
}
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;
}
```
Allocating Variable-Sized Arrays

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.
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

```latex
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 *)

(access link) ·

a: x = 5
s = 42
```

What does “a 5 42” give?
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 *)
  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
Implementing Nested Functions with Access Links

\[
\begin{align*}
\text{let } & \quad a \times s = \\
\text{let } & \quad b \ y = \\
\text{let } & \quad c \ z = z + s \text{ in} \\
\text{let } & \quad d \ w = c \ (w+1) \text{ in} \\
& \quad d \ (y+1) \text{ in } (* \ b \ *) \\
\text{let } & \quad e \ q = b \ (q+1) \text{ in} \\
& \quad e \ (x+1) \text{ (* a *)}
\end{align*}
\]

What does “a 5 42” give?
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>
</tr>
</thead>
<tbody>
<tr>
<td>s:</td>
<td>s = 42</td>
</tr>
<tr>
<td>e:</td>
<td>q = 6</td>
</tr>
<tr>
<td>b:</td>
<td>y = 7</td>
</tr>
<tr>
<td>d:</td>
<td>w = 8</td>
</tr>
</tbody>
</table>
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?

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

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

How about reading an unaligned value?
Padding

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).

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

```
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 */
};

struct padded {
    char a;    /* 1 byte */
    short b;   /* 2 bytes */
    short c;   /* 2 bytes */
};
```
Padding: (1) or (2)?

```c
struct padded {
    int a;    /* 4 bytes */
    char b;  /* 1 byte */
    char c;  /* 1 byte */
};
```

(1)

(2)
A C union shares one space among all fields

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

or

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

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

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

What if we remove rule 2 of padding?

```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];
```
The largest primitive type dictates the alignment

```
struct {
  short a;
  short b;
  char  c;
} d[4];
```
The Heap
Heap-Allocated Storage

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

\[\text{free}()\]

\[\text{malloc}()\]

\[\text{free}()\]
Dynamic Storage Allocation

\[
\text{free(} \quad \text{malloc(} \quad /two.osf/four.osf
\]
Dynamic Storage Allocation

↓ free()

↓ malloc()
Dynamic Storage Allocation

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

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

Rules:

Each allocated block contiguous
Blocks stay fixed once allocated

malloc()

free()
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(SNSNSN)
frees(SNSNSN)
Simple Dynamic Storage Allocation

```
malloc()
```

```
free()
```
Simple Dynamic Storage Allocation

malloc(

)
Simple Dynamic Storage Allocation

malloc()  

free()
Fragmentation

```c
malloc( ( ) seven times give
```

```c
free() four times gives
```

```c
malloc( ( ) ?
```

Need more memory; can’t use fragmented memory.

Zebra  Tapir
Fragmentation and Handles

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

The original Macintosh did this to save memory.
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? | Disadvantages? |
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
```
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

```latex
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

```
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?

\[ \text{a} \rightarrow \text{b} \rightarrow \text{a} \]
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

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
define a = (42, 17)
define b = [a; a]
define c = (1, 2)::b
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