StackVsHeap
Objectives

- Memory management
- Central shared resource in multiprocessing RTE
- Memory models that are used in Java and C++
- Services for Java/C++ programmer from RTE (JVM / OS).
Perspectives of Memory Services

- Hardware level:
  - READ / WRITE

- OS level:
  - sharing physical memory among processes
  - allocating memory to processors, etc.

- Programming Language:
  - how programming language relate to memory operations at runtime and at compile time.
Overview

- background on memory technologies,
- hardware level aspects
- programming language aspect
Memory Technologies

- **On-chip storage** - registers and cache storage
  - very fast storage
  - generally reside on the same chip as main processor
Memory Technologies

- Random Access Memory - family of technologies used to implement the computer main memory
Memory Technologies

- Persistent online storage - Hard disk/solid state disks using magnetic/electromagnetic technology
Flash memory

Single Bit Flash Memory—2 States
Programmed = 0
Erased = 1
Single Sense Amp

Intel StrataFlash® Memory—4 States
Programmed = 00
Programmed = 01
Programmed = 10
Erased = 11
2X Bits in 1X Space—Multiple Sense Amps

Two Technologies
One Core Memory Cell
Identical NOR Architecture
Memory Technologies

- Persistent offline storage - Tapes/DVD/CD are magnetic or optical offline storage technologies
Memory hierarchy

- faster memory available in smaller quantities
- slower memory available in larger quantities
  - Memory costs rise the faster the memory
Memory system characterization:

- Memory space: how much main memory the processor can address (maximum potential size)
- Word size: writes/reads from main memory are at least word-sized
  - words are the native units of data the CPU operates on - internal registers can store exactly one word
- **Width of address:**
  - Memory is viewed as an array of words.
  - Each word has an index (0, 1, …, up to the size of the memory space).
  - Index of a word in the main memory is called its address.
• Width of the data:
  • memory is read and written to/from registers in the processor
  • Each such operation is performed on a single word at a time. T
  • size of the word is the number of physical connections between processor and memory (BUS)
Hardware Perspective

- **Operation cycle:**
  - Load the data from memory into processor registers.
  - Apply operation on registers and put result in another register.
  - Write result from register into memory location.
Hardware Perspective

- primitive operations processor uses with relation to memory are:
  - Read (Address), (Register) : copy content of word stored at (Address) into (Register)
    - Registers are referenced by their number (0, 1, ... number of registers on the processor).
  - Write (Register), (Address): copy content of (Register) to memory cell at (Address).
• compiler generates abstract set of processor instructions
  • [var] - address in memory in which a variable is stored.

```plaintext
1. int i = 10;
2. int j = 20;
3. 
4. i = i + j;

;; Initialize the memory cells for i and j
WRITE $10, [i]  ;; 10 is a constant value
WRITE $20, [j]  ;; 20 is a constant value

;; Execute the sum
READ [i], R1  ;; copy the content of i in register 1 (R1)
READ [j], R2  ;; copy the content of j in register 2 (R2)
ADD R1, R2, R3 ;; execute the operation: R1 + R2 and store the result in R3
WRITE R3,[i]  ;; store the result in the memory location of i
```
Programming language perspective

- explicit/implicit memory-related services we already used in Java?
Programming language perspective

- **Explicit**
  - **Object allocation** - Object allocation using `new`
  - **Initialization** - Constructor invocation or variable initialization
  - **Copying** - Copying data from one variable to another
Programming language perspective

- **Implicit:**
  - **Object deletion** - done by the Garbage Collector
  - **Variable allocation** - Local (primitive) variables allocation
  - **Variable deallocation** - Local variable deallocation when leaving scope
Memory Access

- Usually one way to access memory: variable
- variable in Java is characterized by:
  - A name
  - A type
  - A scope: the program region in which the name is recognized and refers to the same variable
Memory Access

- Variables are **bound** to values at **runtime**.
- value is characterized by:
  - A type
  - A position in memory which is the value address
  - A size - how much memory does the value requires in memory
Memory Access

- Variable bound to value - type of value must be compatible with type of variable.

- Types are compatible when:
  - They are identical
  - type of the value is more specific in inheritance hierarchy than type of variable.
  - type of value implements type of variable (type of variable is an interface).
Procedures

- program is composed of several procedures.
- procedure is a small piece of code, performing a simple and defined operation.
- procedures may call other procedures or themselves.
- procedure may receive parameters, define local variables and return a value.
OO vs. procedures

- object oriented programs are procedural
- call to a member method of some object = call method with the object as one of the arguments.
- (in some languages, you are required to pass the object as the first argument!)
Stack model vs. Heap model

- two different memory models
Stack

- Procedural calls are viewed in a stack-like manner:
  - each procedure's activation requires a frame on a stack.
  - each time a procedure is called, a new activation frame is generated on the stack.
Each activation frame holds the following information:

- The parameters passed to the procedure
- Local primitive variables declared in the procedure
- The return value - Where to return to when the procedure ends
Each time a function is invoked, an activation frame will open on the stack

```java
public static void foo(int a, bool b) {
    int var_a;
    int var_b;
    
    
}
```

![Activation Frame of foo](image)
run-time/execution/call stack

- **Activation Frame Generation**: compiler generates the appropriate code, and inserts it as the preamble of each procedure.

- **Variable Bindings**: compiler hard-codes the location of each variable in the procedure's code, usually using a simple offset from the start of the activation frame.
void main(int argc, char *argv[])
{
    int x = 9;
    .......
    f1(x);
    .......
}

void f1(int i)
{
    int x;
    x = 7;
    ......
    f20;
    .......
}

void f20
{
    int x;
    .......
    x = 5;
    .......
}
When an activation frame is created, the binding between local variables and their location on the stack is made.

When a procedure terminates, its associated activation frame is popped from the stack, and (figuratively) discarded.

In Java, only variables located on top-most activation frame may be accessed = local variables/parameters. (not the case in C++)
Stack Implementation

- modern computer systems support *stack* data structure;
- For each execution element (process, thread)
  - memory region called *stack*.
- OS is responsible for allocating enough space for the stack of each process (or thread),
- process (or thread) use code generated by the compiler, manages the stack on its own.
Stack Implementation

- stack is defined by a single CPU register which points to top of the stack
  - Register directly modified by the process (or thread)
  - Indirectly modified, by pushing and popping words to/from the stack using machine code instructions.
Heap

- programs need to store information which is relevant across function calls
- too big to fit on the stack. Specify block memory size to store some information.
- *Unknown at compilation time*
- **Heap** - blocks of memory are heaped in a pile, and we can get to the block we need if we remember where we left it.

- **In Java:**
  - local variables are stored on the stack
  - objects are stored on the heap.
1. `int i = 5;`

2. `Object o = new Object();`
• i and o are stored on the stack.

• new Object allocated by new is stored on the heap.

• The section of the stack represented by the variable i contains the binary representation of 5 (00...00101).

• So, what is stored in o's section?
- [Object] - might be very big, with enough space to hold the entire state of Object
- [variable o] - holds the location of Object on the heap
  - section associated with o on stack holds reference to starting location of the block of memory new allocated to hold the instance of the new Object().
Objects vs. primitives

- In JAVA:
  - primitives always passed to functions by value
  - objects are always passed to functions by reference
Heap Implementation

- OS allocates a large block of memory to each process, to serve as its heap.
  - block is shared by all threads in same process
  - may be enlarged during the life time of the process by using special system calls (e.g., `brk(2)`).
  - Smaller blocks are (logically) allocated to hold specific data items (e.g., Objects).
Heap Implementation

- Heap responsibilities of the RTE:
  - Allocating smaller blocks from the heap
  - Keeping track of which parts of the heap are used at any given time
  - Requesting the OS to enlarge the heap

Managing the heap is done by the RTE
Managing the stack is done with code instructions inserted by the compiler
Man

- standard documentation system in unix (help files)
- arranged in sections, 1=utilities, 2=system calls, 3=library calls etc.
- `brk(2)` tells us that `brk` is a system call. 
  `[man 2 brk]`
Memory deallocation in Java

- blocks of memory are allocated in Java only when creating new Objects
- call to new operator= block of memory is allocated to hold the Object
Memory deallocation in Java

- What happens to an Object once we have no further use for it?
- e.g., we allocated an array inside some function, and then left the function without leaking the reference to the array outside of the scope of the function.
Memory deallocation in Java

- Should array be kept in memory indefinitely?
  - No. This would exhaust our memory rapidly.
  - We want to reuse the memory / tell the RTE that the memory can be recycled.
  - In Java, this is done automatically for us, by an active entity: garbage collector
Memory deallocation in Java

- Once in a while / user calls System.gc():
  - garbage collector kicks into action, and frees garbage Objects
  - **Object is considered garbage if it cannot be accessed from a variable on the stack**
algorithm of the garbage collector

1. Compose a set of all Objects referenced from the stack and from registers.
2. For each Object in the set, add all Objects it references.
3. Repeat step (2) until the set remains unchanged (compute transitive closure of the "a references b" relation).
4. Free all objects which are not in the set.
Stack and Heap in C++

- general abstractions of stack and heap are similar in C++ and Java
- fundamental differences in how they are managed.
- C++ allows the programmer more control over memory than Java. Price:
  - lines of code to write
  - requires the programmer completely understands the environment, avoid (the very large) pitfalls
Stack

- stack in C++ is managed in the same way the stack in Java is managed
  - one important difference: objects in C++ may be allocated on the stack
  - objects can be passed by value to functions!
Stack

- when an activation frame is deleted, objects are not just discarded as primitive types are,

- C++ RTE ensures that a special function of the object will be called - the destructor

- Destructor: function in charge of cleaning after the object:
  - Releasing object state and any resources the object might have acquired.
We later discuss three important methods of objects in C++:

- destructor
- copy constructor
- assignment operator
Objects on the stack and pass by value

```java
void doSomething(Foo foo2)
{
    foo2.changeMe(...);
}
Foo fool;
doSomthing(fool);
```
At runtime:

- A place to hold class Foo is reserved on the stack and is binded with the variable foo1.
  - The default constructor is called

- An activation frame for doSomething is created, with enough space to hold the return address and an object of class Foo.

```java
void doSomething(Foo foo2)
{
    foo2.changeMe(...);
}
Foo foo1;
doSomething(foo1);
```
A new object of class Foo is instantiated on the stack, by using the **copy constructor** of class Foo and passing foo1 as the argument.

The method `changeMe()` of the object foo2 is called.

The destructor of foo2 is called.

doSomething frame is popped from the stack and the program continues from the return address.

```cpp
void doSomething(Foo foo2)
{
    foo2.changeMe(...);
}
Foo foo1;
doSomthing(foo1);
```
changes made on foo2 are not reflected in foo1.

```java
donSomething(Foo foo2) {
    foo2.changeMe(...);
}
Foo foo1;
doSomething(foo1);
```
A place to hold class Foo is reserved on the stack and is binded with the variable foo1.

A place to hold class Foo is reserved on the stack and is binded with the variable foo3.

The default constructor is called for foo1.

An activation frame for doSomthing is created, with enough space to hold the return address and an object of class Foo.
● A new object of class Foo is instantiated on the stack, by using the copy constructor of class Foo and passing foo1 as the argument.

● The method changeMe() of the object foo2 is called.

● foo2 is copied by the copy constructor of class Foo to the location of foo3.

```cpp
Foo doSomething(Foo foo2) {
    foo2.changeMe(...);
    return foo2;
}
...
Foo foo1;
Foo foo3 = doSomething(foo1);
```
• The destructor of foo2 is called.
• The doSomething frame is popped from the stack and the program continues from the return address
• changes made on foo2 are not reflected in foo1
• doSomething implicitly employs three objects of type Foo, instantiating, copying and destructing them!
Heap

- As everything in C++ can be allocated on the stack, so can everything be allocated on the heap. Even primitive types...
1. `int *i = new int(8);`
2. `Foo *foo = new Foo(42);`
New Operator

- Allocating on the heap is achieved by calling the new operator.
- new is an operator which allocates space on the heap and initializes the space as we requested by calling a constructor.
- returned value of new a_type(vars) is always of type a_type * = a pointer to a_type
- memory is initialized using constructor (must exists, otherwise code will not compile)
Delete operator

- memory allocated on the heap in C++ is never garbage collected
  - responsibility of programmer to free memory.
- Each time we delete an object, its destructor is called.

1. int *i = new int(8);
2. Foo *foo = new Foo(42);
1. delete i;
2. delete foo;
Allocating and Deleting Arrays in C++

- Arrays in C++ are allocated by using a special operator, `new []`, which takes the size of the array as a parameter.

```
Foo *foo_array = new Foo[100];
```
Allocating and Deleting Arrays in C++

- type returned by the new [] operator is, type Foo *.
- allocated a block of memory on the heap, large enough to hold 100 Foo objects.
- Each object is initialized using the default constructor of class Foo
- how can we release the memory? by calling the delete[] operator!

```cpp
delete[] foo_array;
```

- calls the destructor of each object in the array, and then frees the memory
- new and new[] are not the same
- delete and delete[] are not the same
- call delete[] on memory allocated by new[] and delete for memory allocated by new.