Use of Heap Memory in Safety Critical Software

Kelvin Nil sen, Ludovic Gauthier, Atego Systems

Ada

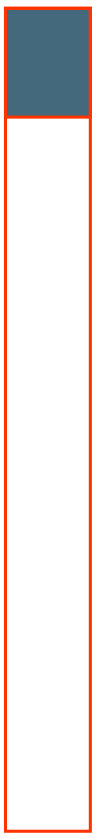
Why do developers use “heap memory” technologies?

- Can’t predict memory needs prior to execution, so can’t preallocate objects
  - But safety-critical software needs to budget memory conservatively
- Can’t predict lifetime of objects so can’t allocate on stack
  - But safety-critical software needs to know when memory can be reused
- Reuse off-the-shelf software components that assume heap memory
  - But off-the-shelf software rarely complies with safety-critical constraints
- Improve efficiency (use same memory for different purposes at different times)
  - Maybe relevant, if you can assure timing of repurpose events
- Improve type safety and encapsulation
  - “Modern” abstractions avoid dangling (type unsafe) pointers and assure proper object initialization (construction)

Important considerations for safety-critical heaps

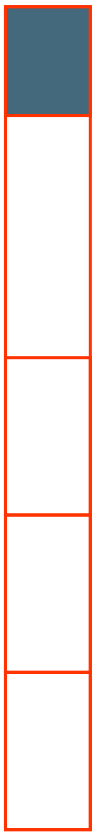
- The memory requirements of the application must be clearly understood and the “system” must assure requirements are satisfied
- Fragmentation of the heap must be “addressed”
- Static analysis must prove prior to execution that the program will run reliably without experiencing:
  - Out-of-memory or stack overflow errors
  - Dangling pointer errors
  - Illegal assignment errors (resulting from run-time prevention of dangling pointer errors)
- Modular composition of software components must support modular analysis of memory behavior
  - When adding a new software component to an existing software system, the impact must be constrained by the interface descriptions (without requiring complicated (undecideable?) reanalysis of entire system)

A stack-of-scopes execution model



Initially, the run-time stack (grows downward) for the main thread represents all non-immortal memory.

A stack-of-scopes execution model



The main thread may spawn additional threads, setting aside part of its own stack to represent the stack memory for the spawned threads.

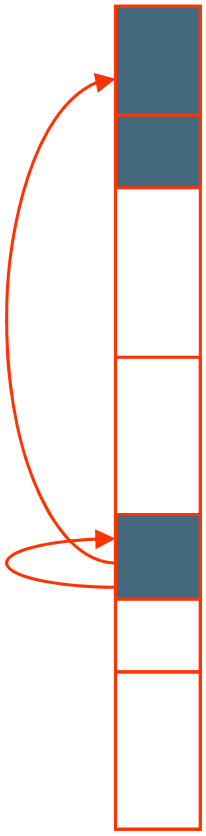
Any of the spawned threads may in turn carve up its stack in order to spawn “grandchildren” threads.

Memory for third spawned thread’s stack

Memory for second spawned thread’s stack

Memory for first spawned thread’s stack

A stack-of-scopes execution model



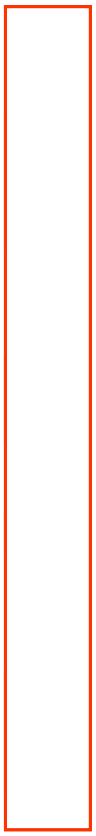
But objects residing in the parent thread’s stack below the point from which the child thread was spawned are not visible to the child thread. And outer-nested objects are not allowed to see objects residing in more inner-nested scopes.

And a child thread may see scoped objects that reside in the parent thread’s stack above the point at which the child thread was spawned.

Individual threads populate their run-time stacks as appropriate.

Each thread’s scoped objects can see scoped objects allocated in more outer-nested scopes of the same thread.

A stack-of-scopes execution model



The parent thread is required to join with its spawned threads before returning from the context from which it spawned those threads.

After the child threads have joined with the parent thread, their memory is fully reclaimed (and defragmented).

Perc Pico Approach to Safety-Critical Memory Management

- Enhance the Java type system to represent relevant memory management details
  - Type system rather than a “static analyzer”: runs faster, more deterministically
  - No false positives or negatives, just enforcement of “type system”
  - Straightforward modular composition of independently developed components enabled by enhanced interface descriptions
- Relevant memory management details
  - How much memory is required to serve the temporary allocation needs of this method?
  - What is this method allowed to do with its incoming reference (pointer) arguments?
  - What memory-relevant preconditions are imposed on arguments at each invocation of a particular service?

Motivation

- How much memory does this simple Java code allocate? Where?

```
String findExcerpt(String source) {
        int start_index = source.indexOf(':');
        int end_index = source.indexOf(':', start_index);

        return source.substring(start_index + 1, end_index);
}
```

Traditional Java mindset:

It's none of your business.
You don't need to know.
It's not part of the "contract".
In fact, this code may allocate different amounts of memory in different places at different times, and in different implementation.

Example Annotations

- In java.lang.String

```
@CallerAllocatedResult
@ExternallyConstructedScope
@ScopedThis
public String substring(int begin_index);
```

This method allocates a String in an outer-nested scope as specified by the caller.

If this method allocates additional objects which will be referenced from the result object, place these objects in the same external scope that holds the result object.

The implicit "this" argument must reside in a scope that encloses the scope that will hold the return object. The substring method is allowed to build a link from the return object to "parts of" the "this" object.

Discussion

- The annotations above identify where new objects may be allocated and which links (pointers) between objects might be created
- The annotations shown above do not enable developers to safely size scopes, scope backing store reservations, or run-time stacks
- A general solution to the memory sizing problem reduces to the halting problem. It is undecidable.
- Our approach is to allow programmers to "assert" bounds on loop iterations and recursion depths:
  - Memory budgeting trusts programmer-supplied assertions
  - Running boundary-case tests of the software with assertion checking enabled helps validate that assertions are correct
  - A "theorem proving" system (not provided by us) can verify that assertions are valid

Sample Resource Constraint Annotations

```
[1]    public class BubbleSort {
[2]        @StaticAnalyzable(parameters = "n")
       // n is length of a[]
[3]        public static void sort(@Scoped int a[]) {
[4]            int i, j, k, t;
[5]            int len = a.length;
[6]            boolean sorted = false;
[7]            for (i = 0, k = len; !sorted && (i < len); i++) {
[8]                assert StaticLimit.IterationBound("n");
[9]                k--;
[10]               sorted = true;    // assume array is sorted
[11]               for (j = 0; j < k; j++) {
[12]                   assert StaticLimit.IterationBound("((n-1)* n)/2)");
[13]                   if (a[i] < a[j]) {
[14]                       t = a[i]; a[i] = a[j]; a[j] = t; sorted = false;
[15]                   }
[16]               }
[17]           }
[18]       }
[19]   }
```

Overview of Development Process