COSC441 Concurrent Programming

Stacks and Threads

Richard A. O'Keefe

July 24, 2017

Outline

- Procedure calls use stacks
- Expression evaluation stack
- Environment stack
- Control stack
- The Cactus Stack model
- The Multi-Stack model
- C11 Threads
- Posix Threads

Procedure calls use stacks

- Procedure calls are fundamental,
- especially in OOP.
- Even more basic than variables!
- This uses three stacks:
  - Expression evaluation stack
  - Environment stack
  - Control stack

Expression evaluation stack

- Holds values of subexpressions
- Could be numbers or pointers
- Forth, Postscript, Transputer, Burroughs B5500 to E-mode
- Still popular in VMs, Lua, Java, AWK, *etc.*

Expression evaluation stack 2

$$
\begin{aligned}
\mathcal{E}[\![c]\!] &= \mathsf{pushConst}(c) \\
\mathcal{E}[\![v]\!] &= \mathsf{pushVar}(v) \\
\mathcal{E}[\![e_1\,\theta\,e_2]\!] &= \mathcal{E}[\![e_1]\!]; \mathcal{E}[\![e_2]\!]; \mathsf{doOp}(\theta) \\
\mathcal{E}[\![f(e_1, \ldots, e_n)]\!] &= \mathcal{E}[\![e_1]\!]; \ldots \mathcal{E}[\![e_n]\!]; \mathcal{E}[\![f]\!]; \mathsf{call} \\
\mathcal{E}[\![e_a[e_i]]\!] &= ; \mathcal{E}[\![e_i]\!]; \mathcal{E}[\![e_a]\!]; \mathsf{index} \\
\mathcal{E}[\![e_c\,?\,e_t : e_f]\!] &= \mathcal{E}[\![e_c]\!]; \mathsf{jfalse}(L_1); \\
&\quad\ \mathcal{E}[\![e_t]\!]; \mathsf{jump}(L_2); L_1 : \mathcal{E}[\![e_f]\!]; L_2 :
\end{aligned}
$$

As always, *as if*

- Hardware (B6700, Transputer) or software can keep part of the stack in registers.
- Compilers try to avoid re-evaluating sub-expressions (as long as you can't tell).
- Register machines don't have *that* many expressions, so intermediate values are *spilled* to memory.

Environment stack

- type $Binding = Variable \mapsto Value$
- type $Frame = \text{Map}[Variable, Value]$
- type $Environment = \text{List}[Frame]$
- lookup v $[] =$ error "unbound variable"
- lookup v $(f : fs) =$
  $v \in \text{dom} f \; ? \; f(v) \; : \;$ lookup v fs
- This implements *lexical scoping*.

Independence of Environment Stack

- *Blocks* in Algol, C, Java, *etc* push new frames on entry and pop them on exit.
- That is, the environment stack can change without a procedure call.
- An *Object* is basically an environment. A Java class $\mathcal{O}$ may contain a nested class $\mathcal{I}$; an instance of $\mathcal{I}$ holds a pointer to the containing instance of $\mathcal{O}$ so that methods in $\mathcal{I}$ can refer to fields of $\mathcal{O}$.
- In a language with *Closures*, a frame may outlive the call that created it.

Nested class environment example

```
public class 𝒪 {
    private int x = 0;
    public class ℐ {
        public int inc() {
            return x++;
        }
    }
}
⋮
𝒪.ℐ foo = new 𝒪().new ℐ();
System.out.println(foo.inc());
```

Closure environment example

```
datatype Op = Inc | Dec | Get
fun make_counter initial =
    let val n = ref initial
        fun f Get = !n
          | f Inc = (n := !n + 1; 1)
          | f Dec = (n := !n - 1; ~1)
        in f
    end
⋮
val c = make_counter 10; (c Inc; c Inc; c Get);
⟹ 12
```

Control stack

- Handles procedure return
- Is a stack of *continuations*
- A continuation is “what to do next”
- Simplest case: just return addresses.
- `jsr` L = push(PC); PC $\leftarrow L$
  `ret` = PC $\leftarrow$ pop()

In Algol-like languages

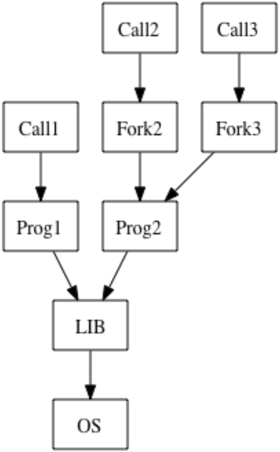
- All three stacks folded into one
- A *Stack Frame* contains
  - A return address
  - A dynamic link (where is caller's frame)
  - A static link (where is outer environment)
  - Bindings
  - Expression evaluation intermediate values
  - including arguments for next call

Confusions

- The return address is really the continuation address of the *caller's* frame, but it is pushed by the callee, so people think of it as part of the callee's frame.
- The arguments for the next call belong to the next callee, but this procedure pushes them, so people think of them as part of this frame.

The Cactus Stack model

Photo credit: Stevemarlett - Own work, CC BY-SA 3.0,
https://commons.wikimedia.org/w/index.php?curid=2838664



The Cactus Stack model, 2

- Used in Burroughs Extended Algol
- Used in Simula 67
- Used in some Algol 68 systems
- Used in Concurrent ML
- Hint: All support concurrency *well*.
- Snag: more complex memory management.

The Linear Stack model

- A stack is a single block of memory, frames are created on entry and removed on exit.
- Used in Algol 60, Pascal, BCPL, C.
- Classic UNIX memory model was

code	static data	heap $\rightarrow$	gap	$\leftarrow$ stack

- C compiler generates simple stack code.
- Cannot move or grow the stack.

Why can't C stacks move or grow?

- In order to move a chunk of memory, you have to adjust all the pointers (in)to it.
- The static and dynamic links are easy to find by "walking the stack" and could be expressed as offsets anyway.
- Languages like Lisp, Smalltalk, and Python store *tagged* pointers on the stack.
- Compilers for languages like ML and Java leave behind *stack maps* to find pointers.
- C and C++ do neither, and contain internal pointers (like &x).
- So moving C stacks breaks things.

What about scattered stacks?

- It is possible to have a linear stack broken into several segments.
- Doesn't move but does grow.
- That requires extra procedure entry/exit work.
- People wanted to add concurrency to C & Unix by adding a library and not changing the compiler.

Consequences of the linear stack

- The effect of a stack overflow in C[++] is *not defined*.
- It is not an exception you can catch.
- A stack must be pre-allocated big enough.
- If it isn't, that's *your* fault.
- You cannot find out how big it should be.

Creating a Thread in Erlang

spawn(**fun** () $\rightarrow$
    body of new thread
end)

Easy because there are nested functions, tagged pointers, and growable stacks.

Creating a Thread in C11

```
#include ⟨threads.h⟩
// C11 standard, section 7.26, not in El Capitan
thrd_t mythread;
void myfunc(void *ctxt) {
    code to run in new thread
    thrd_exit(result);
}
int e = thrd_create(&mythread, myfunc, &mydata);
// ⇒ thrd_success or thrd_nomem or thrd_error
e = thrd_join(mythread, &result);
```

C11 Thread Creation 2

- This is FORK-JOIN parallelism, just like fork() and wait() in classic UNIX.
- If you want the new thread to continue independently, you must do

  e = thrd_detach(mythread);

- thrd_current returns id of calling thread.
- thrd_equal compares two thrd_t-s.

Problems

- The machines I have access to don't support C11 yet.
- The C library picks a new stack size.
- Nothing you do affects that size.
- If the stack size is too small you are euchred.
- Parameters and locals are private to the thread, globals are available, anything else has to be accessed through a global or the **void*** argument.

Thread-local variables

- What if you want multiple functions to access a variable, but each thread should have its own copy?
- Declare such variables thread_local.
- Problem: each thread gets a copy whether it needs one or not.
- In Ada, a task is a kind of procedure. Variables declared in that are automatically thread local, and only relevant ones exist.
- C stinks as a concurrent language.

Creating threads in POSIX

```
#include <pthread.h>
pthread_t mythread;
pthread_attr_t mythreadattrs;
void myfunc(void *ctxt) {
    code to run in new thread
    pthread_exit(&myresult);
}
int e = pthread_create(&mythread, &myattr,
            myfunc, &mydata);
void *result;
e = pthread_join(mythread, &myresult);
```

POSIX thread creation 2

- This is FORK-JOIN parallelism.
- If you want the new thread to continue independently, you can do

  e = pthread_detach(mythread);

- or use PTHREAD_CREATE_DETACHED in myattrs.
- pthread_self returns id of calling thread.
- pthread_equal compares two thrd_t-s.

Problems

- OSX supports less of POSIX than Linux and Solaris.
- There is a default stack size which can be wrong.
- But you *can* set a stack size, even allocate it yourself, using attributes.
- If the stack size is too small you are euchred.
- Parameters and locals are private to the thread, globals are available, anything else has to be accessed through a global or the **void*** argument.

Thread-local variables

- GCC supports __thread as a storage class.
- Some other C compilers support it too.
- It doesn't work in OSX.
- The portable way is to create such variables *dynamically*, which hurts type checking.
- C stinks as a concurrent language.

A common mistake

- In Ada, a task will not exit until its child tasks have finished. (Yay!)
- In C, the program will exit as soon as main() returns or exit() is called, even if other threads are still running. (Boo!)
- If it is important that a thread should finish it is up to *you* to ensure this.

Next week

- Synchronisation between threads.
- Atomic variables in C and Java.
- How atomic variables work.
- The CAS and LL/SC instructions.
- Mutual exclusion locks (mutexes).
- Semaphores.