Control Abstraction

Hwansoo Han
Review of Static Allocation

- Static allocation strategies
  - Code
  - Global variables
  - Own variables (live within an encapsulation - static in C)
  - Explicit constants (including strings, sets, other aggregates)
    - Small scalars may be stored in the instructions themselves
Review Of Stack Layout
Review of Allocation Strategies

- Stack allocation
  - Parameters
  - Local variables
  - Temporaries
  - Bookkeeping

- Heap allocation
  - Dynamic allocation
Stack Frame

- Contents of a stack frame
  - Bookkeeping
    - Return PC (dynamic link)
    - Saved registers
    - Static link
  - Arguments and returns
  - Local variables
  - Temporaries
Stack Frame Maintenance

- Maintenance of stack is the responsibility of:
  - Calling sequence
  - Subroutine prolog and epilog

- Space is saved
  - By putting as much in the prolog and epilog as possible

- Time *may* be saved
  - By putting stuff in the caller instead, where more information may be known
Register Save & Local Variables

- Common strategy is to divide registers into caller-saves and callee-saves sets
  - Caller uses the callee-saves registers first
    - callee-saves registers are for local variables which are more likely live across function calls
  - Caller-saves registers if necessary
    - Caller-saves registers are mostly for temporary values (less likely live across function calls)

- Local variables and arguments
  - Assigned at fixed offsets from the frame (or stack) pointer at compile time
Calling Sequences

- Before the call, Caller
  - Saves into the temporaries any caller-saves registers whose values will be needed after the call (i.e. live registers)
  - Puts small arguments into registers (or in the stack)
    - It depends on the types of the parameters and the order in which they appear in the argument list
  - Puts the rest of the arguments into the argument build area at the top of the stack frame
  - Use a special call instruction to jump to the subroutine and store return address on the stack or in a register
Calling Sequences

- In the prolog, Callee
  - Subtracts the frame size from \( sp \) (frame grows towards lower address)
  - Saves callee-saves registers if used anywhere inside callee

- In the epilog, Callee
  - Puts return value into registers (memory if large)
  - Restores callee-saves registers
  - Adds to \( sp \) to deallocate the frame (restore \( sp \))
  - Restore \( fp \) to old \( fp \)
  - Return to caller
Calling Sequences

- After call, Caller
  - Moves return value from register to wherever it's needed (if appropriate)
  - Restores caller-saves registers if needed (lazily over time, as their values are needed)
Arguments in Registers

- All arguments have space in the stack, whether passed in registers or not
  - The subroutine may begin with some of the arguments already cached in registers, and *stale* values in memory

- This is a normal state of affairs
  - Optimizing compilers keep things in registers whenever possible,
  - Flushing to memory only when they run out of registers, or when code may attempt to access the data through a pointer or from an inner scope
Optimization for Calling Sequence

- Many parts of the calling sequence, prologue, and/or epilogue can be omitted in common cases.

- Particularly, *leaf* routines don't call other routines.
  - If another routine is called, ra (which contains return address) should be saved, but leaf routines don't save it.
  - Simple leaf routines can be exceptionally fast.
    - No local variables and nothing to save and restore in the stack.
    - Don't even use memory, just compute with *(caller-saves)* registers.
Parameter Passing

- Three basic implementations for parameter passing
  - Call-by-value
  - Call-by-reference
  - Call-by-closure (subroutine + environment)

- Variations
  - Call-by-result — copying back when return
  - Call-by-value/result — copying and copying back
  - Call-by-sharing — copying in reference model
  - Call-by-name — substitute as macros
  - Read-only — prevent modification
  - Named parameter — specify corresponding formal parameters
Parameter Passing in C/C++

- C/C++: functions
  - Parameters passed by value (C)
  - Parameters passed by reference can be simulated with pointers (C)

```
void proc(int* x, int y) { *x = *x+y; }
proc(&a,b);
```

- Or directly passed by reference (C++)

```
void proc(int& x, int y) { x = x + y; }
proc(a,b);
```
Parameter Passing in Ada

- Ada goes for semantics: who can do what
  - in: callee reads only
  - out: callee writes and can then read (formal not initialized); actual modified
  - in out: callee reads and writes; actual modified

- Ada in/out is always implemented as
  - Value/result for scalars, and either
  - Value/result or reference for structured objects
Parameter Passing in Others

- Language with a reference model of variables (Lisp, Clu)
  - Pass by reference (*sharing*) is the obvious approach

- Fortran always uses call-by-reference (only option)
  - If you pass a constant or expression, the compiler creates a temporary location to hold the value and pass its reference
  - If you modify the temporary, who cares?

- Call-by-name is an old Algol technique
  - Think of it as call by textual substitution
    (procedure with all name parameters works like macro)
  - A way to mimic macros for assembly language programmers
## Parameter Passing Modes

<table>
<thead>
<tr>
<th>parameter mode</th>
<th>representative languages</th>
<th>implementation mechanism</th>
<th>permissible operations</th>
<th>change to actual?</th>
<th>alias?</th>
</tr>
</thead>
<tbody>
<tr>
<td>value</td>
<td>C/C++, Pascal, Java/C# (value types)</td>
<td>value</td>
<td>read, write</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>in, const</td>
<td>Ada, C/C++, Modula-3</td>
<td>value or reference</td>
<td>read only</td>
<td>no</td>
<td>maybe</td>
</tr>
<tr>
<td>out</td>
<td>Ada</td>
<td>value or reference</td>
<td>write only</td>
<td>yes</td>
<td>maybe</td>
</tr>
<tr>
<td>value/result</td>
<td>Algol W</td>
<td>value</td>
<td>read, write</td>
<td>yes</td>
<td>no</td>
</tr>
<tr>
<td>var, ref</td>
<td>Fortran, Pascal, C++</td>
<td>reference</td>
<td>read, write</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>sharing</td>
<td>Lisp/Scheme, ML, Java/C# (reference types)</td>
<td>value or reference</td>
<td>read, write</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>in out</td>
<td>Ada</td>
<td>value or reference</td>
<td>read, write</td>
<td>yes</td>
<td>maybe</td>
</tr>
<tr>
<td>name</td>
<td>Algol 60, Simula</td>
<td>closure (thunk)</td>
<td>read, write</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>need</td>
<td>Haskell, R</td>
<td>closure (thunk) with memoization</td>
<td>read, write*</td>
<td>yes*</td>
<td>yes*</td>
</tr>
</tbody>
</table>
Exceptions

- What is an exception?
  - A hardware-detected run-time error or
  - Unusual condition detected by software

- Examples
  - Arithmetic overflow
  - End-of-file on input
  - Wrong type for input data
  - User-defined conditions, not necessarily errors
Exception Handling

- What is an exception handler?
  - Code executed when exception occurs
  - May need a different handler for each type of exception

- Why design in exception handling facilities?
  - Allow users to explicitly handle errors in a uniform manner
  - Allow users to handle errors without having to check these conditions before
  - Explicitly in the program everywhere they might occur
Events

- What is an event?
  - Running programs need to respond to the events
  - Events occurs outside the programs at unpredictable times

- Examples
  - Inputs from users (keyboard, mouse motion/click)
  - I/O from network, disk

- Event handlers
  - A special callback function
  - A dedicated thread to handle
Event Handlers

- Sequential handlers
  - Works in an asynchronous way
    - Register *handler* and return
  - Use a hardware interrupt mechanism
  - OS calls handler routines
  - Return to the program

- Thread-based handlers
  - Separate thread control thread
  - Works in a synchronous way
    - A dedicated thread is called
    - Makes a system call for the next event
    - Waits for it to occur

Signal delivery for sequential handler
Coroutines

- Concurrently calling one another
  - Coroutines are execution contexts that exist concurrently, but that execute one at a time, and that transfer control to each other explicitly, by name

- Coroutines can be used to implement
  - Iterators
  - Threads

- Separate stack should be maintained
  - Because they are concurrent (i.e., simultaneously started but not completed), coroutines cannot share a single stack
Multiple Stacks for Coroutines

B calls subroutine S and created coroutine D

Cactus Stack

static link
Summary

- **Function**
  - Stack frame
  - Passing parameters

- **Exception**
  - Unexpected/unusual HW/SW condition caused by current program
  - Try-catch (Java, C++)

- **Event**
  - HW/SW condition required to respond by current program
  - Event handling mechanism

- **Coroutine**
  - Concurrent execution of multiple sequences, but execute one at a time
  - Transfer:
    - *non-local goto* to jump to other coroutine’s last transfer (continuation)
  - Cactus stack: Multiple concurrent stacks