Honors Compilers

Run-Time Support for Compilers
Mar 21st 2002
Major Issues

What is a compiler run-time?

Major components of run-time
- Computational Support
- I/O interfacing
- Storage Management
- Tasking interfaces
- Exception Handling
- Other operating system issues
The Compiler Run-Time

Simple operations generate direct machine language (e.g. addition)
But modern languages have many high level constructs
Compiler generates calls to hidden routines called the run-time.
Run-time is included silently in linked program (standard libraries used)
The Compiler Run-Time

Part of the implementation
Not available directly to program
- Interface may not be documented
- Special calling sequences can be used
- Tailored asm may be appropriate
- Special operating system interface

Delivered as part of the compiler
- Licensing considerations apply
Computational Support

Some simple operations may not be supported by instruction set

- 64-bit integer arithmetic
- Conversion of fpt to integer
- Overflow checking
- Floating-point (emulation needed)
- Long shifts
- Square root
Handling Computational Support

Need very small high efficiency routines, might possibly be inlinable.

Special calling sequences may be useful (e.g. args in registers).

Assembly language may be reasonable in this case.

Implementation is target dependent.
Language Considerations

Some languages have no high level constructs, so need minimal run time

- For example, C, everything is done with explicit procedure calls.

But other languages, e.g.

- C++ (stream operations)
- Ada (tasking)
- Java (automatic storage mgmt)
The Run-Time in C

Very small

- Usually only some computational routines.

Standard has many standard functions as part of language

- These are explicit library routines
- Often part of the underlying OS
  - E.g. fopen, delete, etc.

Command line interface needed
The Run-Time in Ada

Much more extensive

- Tasking
- Timing
- Storage Management
- Conversions (e.g. string to fpt)

Requires extensive library

- Only partly target dependent
Other Languages

C++ nearer to C, but has
- Streams
- Exception Handling

Java, more extensive, adds
- Storage mgmt (garbage collection)
- Standardized arithmetic (IEEE fpt)
- Interface to JBC/JVM
Other Languages (cont)

COBOL, much more extensive

- Detailed complex I/O model
  - Includes full indexed files
- SORT subsystem
- INSPECT REPLACING
- PICTURE editing
- Display and packed numeric formats
Other Languages (cont)

Very high level languages
- Setl, Python, GUILE, Visual Basic
- Set operations
- GUI operations
- High level operating systems interfaces
  - E.g. COM/DCOM interfacing
  - Web interfacing
I/O interface

Language may have explicit I/O statements, or rely on proc interface. In either case, have to map the language notion of I/O to op sys:

- End of line (record vs stream)
- Notion of file system (directories?)
- Exception handling
- Character sets (esp wide chars)
Implementing the I/O Interface

Two parts

- Target independent code
- Target dependent code

Target Independent code

- Simply implements the language semantics
- Using some abstractions
Target Dependent I/O

On top of operating system

- Map language semantics to OS semantics, deal with differences as well as possible.
- On a bare board, have to actually implement basic I/O
  - Perhaps at the port level
  - Basically compiler includes an O/S
Storage Management

Stack Allocation
Heap Allocation
Controlled Allocation
Automatic Garbage Collection
Stack Management

Stack must be allocated for each task in the program

- Usually handled by operating system
- Have to worry about size of stack
- How/whether to extend it as needed
- How/whether to check stack overflow
  - May or may not be language required
Heap Management

Basic semantics is Allocate/Free

- **Parameters for allocation**
  - Size in bytes
  - Alignment (often just use max alignment as required for example in C by malloc)

- **Parameters for free**
  - Probably just address (but perhaps size and alignment as well, as in Ada)
Heap Algorithms

Many algorithms available
- Free list
- Multiple free lists
- Binary allocator

Performance considerations
- Average case time
- Worst case time (real-time critical)
- Fragmentation considerations
OS Heap Allocation

Typical O/S provides malloc/free
Malloc takes size in bytes
☞ Returns max aligned address
Free takes address
Optimized for average performance
Often algorithm is not documented
More on malloc/free

Can map language requirements
- E.g. new in C++ or Ada maps to malloc
- Unchecked_Deallocation to free

May want to allocate large blocks with malloc, subdivide for use
- Built in for PL/1 areas
- Built in for Ada collections with storage size clause given.
Commit Strategies

Physical Allocation

Virtual Allocation
- Storage committed at allocate time
- Storage committed at use time

Overflow checking
- Indication of out of memory
- Difficulties with commit on use
Specialized Strategies

Trade off fragmentation with time performance.

Fragmentation

- Internal, block allocated is larger than needed by the allocation request
- External, storage available, but not contiguous.
Example, Binary Storage Allocator (Buddy System)

N free lists for $2^{**1}$, $2^{**2}$, ... $2^{**N}$ units

Block of length $2^{**K}$ is on $2^{**K}$ boundary (last K address bits zero)

To allocate block of length M

- Use next higher power of 2 ($2^{**K}$)
- If available on free list grab
- If not, allocate block of $2^{**(K+1)}$, split

Free recombines if possible
BinaryAllocator Performance Issues

Internal Fragmentation
- Need 40 bytes, get 64 bytes, 24 wasted

External Fragmentation
- May have 2 non-contiguous 128 byte blocks, but cannot allocate 256 bytes

Performance
- Bounded (at most K allocation steps)
Controlled Allocation

In C++, constructors automatically allocate storage, destructors free storage.
In Ada, controlled type Initialize allocates, Finalize frees storage.
Can be used for other than storage issues, but 99% of time usage is for allocate/free of memory
Implementing Controlled Storage

Compiler inserts automatic calls to constructors and destructors.

In GNAT, you can use -gnatG to see this happening in the expanded code.

Constructors/destructors contain appropriate storage mgmt calls.
Automatic Garbage Collection

Allocate storage as needed
Free storage automatically when no longer needed.
- Concept of reachability
- Can garbage collect when non-reachable.
Possibly compact storage
- Considerations of adjusting pointers
Determining Blocks in Use

Assume that we can tell what blocks are currently allocated.

We have certainly starting points:
- Global and local variables
- Register contents
- These are called “roots”
Tracing Blocks in Use

Need to find all pointers in blocks in use, and recursively trace other blocks in use, following all pointers.

Two approaches
  - Conservative
  - Type-Accurate
Conservative Tracing

Conservative Garbage Collection
- If it looks like a pointer, assume it is
- Will never release used storage
- May hold onto garbage

Type-Accurate Garbage Collection
- Know where pointers are
- Trace only pointers, knowing types
Further Steps in GC

Once all blocks in use are traced
Free all remaining blocks
Possibly compact blocks in use
Adjust pointers if compaction

- Requires type accurate tracing
- Since only pointers must be adjusted
- Eliminates external fragmentation
Concerns with GC

Stop the world and GC
- Not good for a rocket launch!
- Or even for an ATM/Web use if too slow

Parallel garbage collection
- GC as you go along
- Have a separate processor doing GC
  - Requires delicate synchronization
Reference Counts

Each block has a count of number of pointers to the block.

Copying a pointer increments count

Destroying a pointer decrements count

If count goes to zero, free block

But cycles can be complete garbage and never freed.
Tasking

Might be done with explicit library

- E.g. use of POSIX threads in C
- No special compiler considerations
- Except for synchronization issues
  - E.g. when stuff can be held in registers
  - VOLATILE/ATOMIC considerations

POSIX/Threads is almost standard

- Not perfect, some variations
- Not always completely implemented yet
Language Tasking Constructs

Threads in Java
Tasks in Ada
Built in defined semantics
May be more or less precisely defined with respect to
- Priority handling
- Guaranteed performance
- Dispatching issues (e.g. time slicing?)
Implementing Tasking

Map tasks onto OS threads
- 1/1 (each task is a thread)
- Many/1 (multiple tasks to a thread)
- All/1 (don’t need OS threads)

Cannot map tasks to processes
- Because of address space issues

May be difficult to get exact semantic equivalence.
Difficulties in Implementing Tasking, an Example

Ceiling priority protocol
- Raise priority to ceiling
- Grab resource, do processing
- Lower priority to ceiling

But does last step lose control
- In Ada, never to equal prio task
- In POSIX, may well add to end of queue, thus losing control.
Bare Board Implementations of Tasking

If no operating system, then no threads to map to. Basically must write an executive that provides this capability. The compiler system ends up including a small tasking executive, a mini-operating system.
Implementations of Exception Handling

C++, Java and Ada share same basic model of exceptions.

- Replacement semantics
- Raise/Throw an exception
- Abandons current execution
- Strips stack frames
- Till exception is handled/caught
The "setjmp/longjmp" method for exceptions.

When a handler is encountered, save enough machine state to restore as required (setjmp)

When a throw occurs, restore most recently saved state (longjmp)

Requires a stack of saved states

Fast throw, but handlers cost
“Zero-cost” Exception Handling

A handler generates only static tables (PC range and handler ID)

Throw unwinds stack frames
- Restoring all registers
- Find return points

Check return point against PC table
- Jump to handler on a match
More on “Zero-cost” approach

Requires tight integration with compiler

Stack traceback

Requires traceback code to understand generated code

- Either by looking at it
- Or by reading external tables
- For example, DWARF-2 unwind tables
Other Operating System Issues

Calendar/Time interface
Distribution issues (RPC/Streams)
Interface to other Languages
Interface to linker
Certification Issues
Calendar/Time Issues

Match of semantics
  - Time changes, daylight saving
Formats of dates etc.
Use for delays, alarms
Resolution
  - Special target dependent capabilities
  - E.g. high resolution timer on NT
Distribution Issues

CORBA interfaces

Annex E in Ada

- Requires RPC support
- Stream handling
  - Target dependence
  - Use of
- Build/synchronization/network issues
  - Use of sockets as binding layer
Interface To Other Languages

Compatibility of Calling Sequences
Compatibility of Data

In Ada

Pragma Convention on procedure sets appropriate calling sequence
Pragma Convention on data sets appropriate representation, e.g. columnwise arrays for Fortran...
Interface to Linker

Language may require special linker capabilities for

- Elaboration checking (Ada)
- Elaboration code (Ada, C++)
- Other consistency checks

May need special linker, or add-on

- In GNAT, gnatbind does extra steps
- In C++, collec2 does extra steps
Certification Issues

Safety-Critical certification
- Requires complete testing
- And documentation of process

For a run-time
- Must provide testing materials
- Certify the process
- One approach: no run-time at all!

Applies to OS as well
- WRS new certifiable kernel