Secure Operating System Design and Implementation
Programming Language Issues

Jon A. Solworth

Dept. of Computer Science
University of Illinois at Chicago

February 4, 2015

Part I

Programming Language Background

Programming language

- What programming language to use for applications?
- What programming language to use in the kernel?
- Ideally these would be the same
- But there are different needs at different levels
- The kernel must be able to evade the type system (e.g., treat a pointer as an integer)
- The application may be written by programmers which are not security experts.

Pro and Con of C

Pro
- C is implemented on every platform ever invented
- C is very efficient
- C can embed assembly language, violate types
- C is "portable assembly language"
- C has a tiny runtime

Con
- C is not type safe
- C is not memory safe
- C is messy, you need to go through hoops to get it to do the right thing

Cross compiling

- NanoOS compilation done in Dom0 (cross compilation)
- Using the same tools to compile for Linux user space
- What is the difference between these two targets? In NanoOS
  - no standard library (printf, ...
  - include files usually used in conjunction with libraries, which don't exist
  - architecture may be different (e.g., 32-bit NanoOS, 64-bit Linux)
  - runtime not automatically added (crt0 providing main linkage, 64-bit support on 32-bit architectures)
  - linking (ld) has to ignore user space libraries, use kernel runtime

Part II

The tool chain

The compilation tool chain

- C compilation is designed to go through a number of phases
- C pre-processor (cpp) processes
  - #include
  - #define
  - #if ... #else ... #endif
- cpp produces one file from (potentially) many
- the c-compiler reads the output of cpp and produces assembly language
- optionally, the c-compiler runs the optimizer which attempts to improve performance
- the output of the compiler is assembly language
- the assembler (as) produces an object file
- object files are linked (ld) together to produce an executable
- the executable is loaded and dynamically linked (ldd) against libraries

gcc driver

- The standard compilation system is GCC, for the GNU Compiler Collection
- GCC supports multiple languages (e.g., FORTRAN, C, C++, Pascal, Java)
- the driver, gcc sequences the tools—cpp, c compiler, optimizer, as, ld
- Usually gcc is used if multiple phases are used
- Gcc also sets up environment, such as location of files
- Single tools use typically don't invoke gcc

C preprocessor

- The pre-processor handles the macro languages
- macros are confusing, they should only be used when there aren't other viable choices
- The preprocessor has a number of command line flags
  - Define a macro name (myname)

    ```
    −Dmyname
    ```
  - Define a macro name (myname) with a value

    ```
    −Dmyname=4
    ```
  - Prepend directory (dir) to Include path

    ```
    −I dir
    ```


C preprocessor input

- Include stdio.h from standard include path

  ```
  #include <stdio.h>
  ```
- Include file (myheader.h) from standard include path or from the current directory

  ```
  #include "myheader.h"
  ```
- Define a name (dog) which will be expanded to value ("stdio.h") when encountered in the future. If define spans multiple lines, each line but the last needs to end with "\".

  ```
  #define dog "stdio.h"
  ```


C-preprocessor input (cont'd)

- defines a macro (add) with parameters (a, b) which expands by replacing the value of parameter into the expression on the right.

  ```
  #define add(a,b) ((a) + (b))
  ```
- if the expression (expr) evaluates to true (it must be a preprocessor-time constant) than the code after expression to else appears in the output. The #else is optional, if no #else then the code after the expression to the #endif is included.

  ```
  #if expr
  ...
  #else
  ...
  #endif
  ```


C-preprocessor input (cont'd)

- if version which test whether a pre-processor name (add) is defined.

  ```
  #ifdef add
  ```
- if version which test whether a pre-processor name (add) is not defined.

  ```
  #ifndef add
  ```
- __FILE__ is the name of the file (useful in error messages)
- __LINE__ is the line number (useful in error messages)
- an error if this is output, used in partially defined conditional compile

  ```
  #error ''Unknown architecture''
  ```

Part III

Separate Compilation

Separate compilation

- C supports separate compilation
- each source file is independently compiled
- The below code compiles x.c and produces an object file x.o

```
gcc -c x.c
```

- multiple object files (`x.o` and `y.o`) are linked together and an executable (`p`) is produced

```
ld -o p x.o y.o
```

- normally, these multiple phases are sequenced by the overall gcc script.
- The linker resolves names which are used in an object file but not defined in that object file

Declarations and definitions

- C separates declarations (associate a type with a name)

```
float y;
int incr(int x);
```

- from definitions which in addition provides the value

```
float y = 3;
int incr(int x)
  {
    return ++x;
  }
```

- declarations can occur in any number of compilation units
- definitions can occur at most (and in some cases exactly) once

Extern and static

- Normally, variables at file scope have external linkage, meaning that the names can be linked together with other object files.
- To prevent a definition from being exported, `static` is used.
- Hence, two statically defined variables named `x` in different compilation units results in two, independent variables.
- Statically defined variables within a file are visible only within the file, but rather than existing on the stack, have a single global instance.
- Static variables in a process have reentrant issues.
- At file scope, `extern` is the opposite of static

Const and constant expressions

- const is a reserve word for a variable. It does not mean that the value is constant, only that the value is read-only through the variable name.
- This is very different from a constant expression which is a constant.
- (A const is often used to declare that a procedure which is passed a pointer does not change the underlying object).
- An array which is defined in global scope must have a size which is a constant expression
- Thus, for defining these global arrays, #define can be used.
- or an enum can be used.
- but a const variable cannot be used for this purpose

enum

- the elements of an enum are integer constant expressions
- if an explicit value is not given, they are numbered starting at 0

```
enum {
   a,
   b,
   c = 20};
```

- enums are not definitions but declarations and thus can be included in multiple compilation units without linkage implications.

Part IV

Libraries

Libraries

- code which is used in many applications is often grouped in libraries
- libraries define code which an application can “pull in” as needed
- the most well known of these libraries is the standard C library
- libraries are not so useful within the kernel, and in fact one of the issues which makes kernel programming more difficult is the lack of the standard C library.

Linking with libraries

- libx.a is a static library, it gets fully linked at link time
- libx.so is a dynamic library, it gets fully linked at load time (right before execution).
- the directory to be added to the path to search for libraries is defined with a library path command

```
-L dir
```

- to list a library file (libx.a) is can be listed with a flag (-lx) or by giving the name (libx.a)
- The -l version also uses the library path rather than the current directory

Linking with libraries

- Load lines are processed from left to right

```
ld x.o y.o -L /my/dir -lw -lz
```

  the executable output will include all the
  - definitions in x.o and y.o, and then
  - unresolved names with definitions in libw.a are pulled in, and then
  - unresolved names with definitions in libz.a are pulled in.
- if a name is defined in x.o and in libw.a, the x.o version will be used.
- if a name is defined in both x.o and y.o, that is a (fatal) conflict
- if a libz.a function requires a function in libw.a (that wasn't previously pulled in) then this will be an unresolved reference

Creating a static library

- To create a library (libz.a) from object files (x.o, y.o, q.o)

```
ar -r libz.a x.o y.o q.o
```

- Note that "-r" stand for replacement
- That means that duplicate names in x.o, y.o, q.o are silently replaced with the last to occur

Creating a dynamically library

- To create a dynamic library (libmystuff.so.1.0.1) from object files (a.c, b.c) first compile Position Independent Code (PIC)

```
gcc -fPIC -g -c -Wall a.c
gcc -fPIC -g -c -Wall b.c
```

- PIC code can be loaded anywhere
- Dynamic libraries are intended to be physically shared between multiple processes
- Dynamic libraries are logically private to each process
- To create the library, the following gcc command is used

```
gcc -shared -Wl,-soname,libmystuff.so.1 \
    -o libmystuff.so.1.0.1 a.o b.o -lc
```

- Note that 1.0.1 is a version number, multiple versions can exist.

Dynamic loading of libraries

- Dynamic loading occurs implicitly when executing
- Its all been checked a link time
- But the final linking is deferred to the execute
- This requires the dynamic libraries to be available at execute time
- We don't use dynamic libraries in Ethos
- (there is even later loading, after execution starts).

Ethos user space load script

- Ethos doesn't have a POSIX library
- And never will (POSIX is a UNIX thing)
- And so there is no standard library
- And thus we need to use a custom loader script

Part V

Declarations

Sizes of primitive types

- C does not guarantee sizes of primitive types
- the only thing which is guaranteed is that sizeof(short) $\leq$ sizeof(int) $\leq$ sizeof(long)
- hence the size depends on both the target architecture and compilation environment.
- common 32-bit an 64-bit architecture sizes

type	32-bit	64-bit
int	32	32
long	32	64
void *	32	64
long long	64	64

Casting

Consider a 64-bit architecture (sizeof(int) < sizeof(long))

```
int a;
long b;
uint au;
ulong bu;
char *p;

a = b;      // truncates high order bits
b = a;      // sign extends a and assigns
bu = au;    // zero extends a and assigns
p = (char *) bu; // copy of bits in bu to p
```


Constants

- Integer constants are of size long
- What happens when you do something like ~ 0 and assign it to an uint64 on a 32-bit arch?
- ~ 0 is 32-bit
- assignment to uint64 causes a 0 extends
- probably wanted ~(0ULL)
- which is 64-bit all ones.
- Similarly, LL used when creating large signed integers
- Without this high order bits silently truncated

Arrays and Pointer arithmetic

```
int a[9];
int *p;
char *cp;
char c;
```

- sizeof(c) is always equal to one
- p+i is equal to (int *) ((char *) p) + i * sizeof(*p))
- a is a constant pointer of type int *
- a[i] is the same as *(a+i)

Types in the OS kernel

- rather than using these types in the OS when size matters, type aliases are used such as int32, int64, uint32, ...
- uint (unsigned ints) are used for values which are always non-negative
- declaration are more sensitive in OS because an OS potentially uses all of the address space, so all of the bits of an unsigned value may be needed.
- other types are dependent on architecture
- For example, a pointer size will depend on whether the architecture is 32 or 64 bit

Printf and brethren

- Note that when using formatted print, it is important to use the correct '%' specifiers
- '%d' for signed
- '%u' or '%x' for unsigned
- '%llu' (or '%lld') for long long unsigned (or long long signed).
- Note that if you use a '%u' instead of a '%llu' not only will the value be partially printed, but some bytes will be left over on the stack for values after that in the format string.
- This will be very confusing

typedef

- typedef allows an alias type to be defined, e.g.

```
typedef int mytype;
mytype x;
```

- Of course, int can be used instead of mytype
- But mytype may be more succinct
- more modular (when referring to specific size)
- more portable across architectures

Part VI

Memory programming issues

Memory layout

an OS

- must deal with raw memory
- receives values from applications and
  - sends them over the network or
  - stores them on disk
- manages both pages and the kernel objects written on these pages

therefore, it is important to understand how values—and in particular, multibyte values—are mapped to memory,

Void*

- When referring to objects of unknown type, a `void *` pointer is used
- A `void *` pointer cannot be dereferenced
- Instead it needs to be cast, e.g., `(int *) v`, after which it can be assigned or deferenced
- A cast changes the way the object's bits are interpreted
- Note that `void *` is a pointer to an object of unknown type while
- `void` means none, i.e., `int p(void);` is a procedure without any parameters.

Endianess

Endianess has to do with byte ordering

Definition

A big endian (respectively, little endian) architecture stores the most (resp., least) significant byte at the lowest address.

For example, consider the number x04030201

address	0	1	2	3
big endian	04	03	02	01
little endian	01	02	03	04

- To ensure consistency between different architectures a common endianess is used
- For example, the Internet uses big endian port and IP address
- To ensure portability of storage, the storage should be written in some architecture-independent format

Alignment

Definition

A memory architecture is *n*-byte aligned if an *n* byte entity must begin at an address divisible by *n*.

Alignment issues may cause a `struct` to have

- unused space within the `struct` (before an aligned field)
- unused space at the end of a `struct` so that the size of a struct is the same whether the struct is an element or an array of elements.
- x86 allows alignment to be required and is little endian

Alignment example

```
struct A {
    char    a;
    uint    b;
}
```

- What is the sizeof(A)?
  - Unaligned, 32-bit: 5 bytes
  - Aligned, 32-bit: 8 bytes
  - x86 supports unaligned accesses, but since aligned accesses are faster, compilers use that.

Alignment example

```
struct B {
    uint   b;
    char   a;
}
```

- What is the sizeof(B), aligned?
  - 8 bytes
  - because sizeof must account for arrays of B, each element which needs to be 4-byte aligned.

Part VII

ANSI C

ANSI C

Two primary ANSI revisions of C, C89 and C99. C89 adds

- `void` and `void *`
- requires procedure declaration to fully specify types, eg.

```
int p(void); // OK, void means no parameter
q(int);      // BAD, procedure type not given
int q();     // BAD, void needed if no parameter
```

- `volatile` is used to describe variables which are not solely updated by the program. Examples of volatile variables include
  1. device registers
  2. synchronization variables (and thus updated by several different programs)

  Parts of the Xen ring buffer qualify as (2).

C99 features

- inline functions
- `long long int`: 64-bit or larger signed integers
- `unsigned long long int`: 64-bit or larger unsigned integers
- _Bool for boolean values
- mixing statements with declaration
- variable length arrays
- designated initiators
- compound literals
- variadic macros

C99

- // comments as in C++
- Initialization

```
 struct Pt {
     int x;
     int y;
} pt = {.x =1, .y=2};

// can construct a value
f((struct Pt) { .x=3, .y=4})
```


Other oddities

- A "struct hack" allows a variable sized array to be created on the heap

```
struct  S{
    int n;          // number of elements
    char array[]; // variable sized array
};
S s = (struct S) malloc(sizeof(struct S) + e);
s.n = e;
```

- A variable sized array can be created:

```
int p(int size) {
    char a[size];   // legal!
}
```


Part VIII

Linked lists

Linked lists

- Ethos uses Linux's doubly linked list structure
- Linked lists are declared within a structure
- using **struct** list_head listName;
- For example:

```
struct Point {
    int x,y;
    ListHead pointList
} point;
```

- initialized by

```
LIST_HEAD_INIT(&point.pointList);
```

Data structure

- List declaration (from `list.h`)

```
struct list_head {
  struct list_head *next;
  struct list_head *prev;
}

typedef struct list_head ListHead;
```


Linked list operations

- The following is a iterator of the list
  `list_for_each (pos, &myPoint.pointList)` where
  - `pos` is of type `ListHead *`
  - `&myPoint.pointList` is the pointer of a type `ListHead *`
- Use `list_for_each_safe (pos, q, &myPoint.pointList)` is used when deleting from a list (or moving elements between lists). (`q` is of type `ListHead*`)
- To extract the object at pos
  `t = list_entry(pos, struct Point, pointList)`
  - `pos` is of type `ListHead *`
  - `&myPoint.pointList` is of type `ListHead *`

Linked list operations (cont'd)

- To remove an element `list_del (pos)`
- To add an element `n`
  `list_add (&(n->list), &(myPoint.pointList))`
- To add an element `n`
  `list_add_tail (&(n->list), &(myPoint.pointList))`
- A variant is to use `list_for_each_entry (pos, head, member)`
  - `pos` is of type `struct Point *`
  - `head` is of type `ListHead *`
  - `member` is the field name `pointList`

Conclusions

- Building an OS in a high level language makes the job easier
- But also introduces a host of problems
- Which requires much detail work to get to fit together
- C is used to build almost all OSs today
- But it relies on the programmer to get it all right.
- It would be nice if there was a better way.