C0

1. Introduction

C0is an extension of C programming language. It has its roots in the C family of languages and will be immediately familiar to C, C++, C# and Java programmers. C0 is aprocedure-oriented language, with low-level capabilities and language-level support of parallelism on L0.

1.1. Simple example

Here we have a parallel version of vector addition it in C0.

```
int a[10000], b[10000];

void main() {
    inti;

    // TODO: Initialize a[] and b[]

for (i = 0; i < 10000; i+=100) {
runner{ add_runner(i, a[i,,i+100], b[i,,i+100]); }
    }
commit;
}

voidadd_runner(int start, int p[], int q[])
{
    inti;

for (i = start; i< start + 100; i++) {
p[i] = p[i] + q[i];
    }
commit;
}
```

The above program adds two vectors of length 10000 with 100 runners, each runner adds up 100 elements. Runners is a separate execution of code which is similar to threads.

1.2. Program structure

The three key concepts in *C0* are **programs**, **types**, **vairables** and **functions**. A program is consist of one or more source file. Each source file defines some types or functions. The program must has a function named **main** with no parameter or return value. The **main** function is where the program starts.

1.3. Keywords

abort	default	goto	~~static~~
~~auto~~	continue	if	struct
bool	~~double~~	int	switch
break	else	long	true
case	~~enum~~	return	unsigned
char	~~extern~~	runner	void
commit	Commitd	false~~short~~	volatile
~~const~~	~~float~~	signed	watching
while	for	do~~sizeof~~	~~register~~

2. Types

There are kinds of types in C0: **simple types**, **structtypes**, **union types**, **function types**, **void type**, **pointer types**, **array types**, and **array segments**.

2.1. Simple types

Table 1 shows the simple types supported in C0

Table 1. Simple types in C0

category	bits	type	range/precision
Boolean	32	**bool**	**true** or **false**
signed integral	8	**char**	–128...127
	~~16~~	**-***	~~-32,768...32,767~~
	32	**int**	–2,147,483,648...2,147,483,647
	64	**long**	–9,223,372,036,854,775,808...9,223,372,036,854,775,807
Unsigned integral	8	**unsigned char**	0...255
	~~16~~	**-***	~~0...65,535~~
	32	**unsigned int**	0...4,294,967,295
	64	**unsigned long**	0...18,446,744,073,709,551,615
~~floating point~~	~~32~~	**-***	~~1.5 × 10−45 to 3.4 × 1038, 7-digit precision~~
	~~64~~	**-***	~~5.0 × 10−324 to 1.7 × 10308, 15-digit precision~~

*: Not supported in current version of C0, due to lack of instruction level support of L0

2.2. Struct/Union types

Structure types are user defined types which contains other types (including other structure types). The **struct** keyword is used to define a structure type. Each element of a structure is called field. Each **field** in a structure has its own storage space.

```
struct Foo {
int a;
int *b;
};

struct {
int (*func)(int, int);
    Foo foo;
} complex_var;
```

The union types are similar to structure types. But the field in union shares the common storage space, so at most one field contains a meaningful value at any given time.

2.3. Function types

In the program, you cannot directly define variables of function types. But you can define functions who has a function type, or define a function pointer to a specified function type.

A function type describes the function prototype, including the types of parameters and the type of return value.

2.4. Void type

Void type is a special type which means “no type”, it can only be used for the return type of function, which means the function does not return any value, or used for defining a pointer which can points to any kind of values.

2.5. Pointer types

A variable of pointer type stores the address of the underlying type. We can access the value stored in the memory location which the pointer points to. This operation is called **dereferencing** a pointer. However, a pointer whose underlying type is void type cannot be dereferenced.

2.6. Array types

An **array** is a data structure that contains a number of variables that are accessed through computed indices. The variables contained in an array, also called the **elements** of the array, are all of the same type, and this type is called the element type of the array. We use **array[index]** to access the elements of an array. The indices of the elements of an array range from 0 to Length - 1. Array segment

2.7. Array segment

An array segment is logically same as an array (or a pointer). However, it restricts the access of elements to a specified range. The array segment is represented as **array[start,,end]**, the start is inclusive and end is exclusive.

```
int a[10000];
int i;

for (i = 0; i < 10000; i++) {
       a[i] = i;
}
```

```
int seg[,,] = a[100,,200]; // From a[100] to a[199]
int b = seg[140]; // b has the value 140
int c = seg[20]; // Undefined behavior
```

3. Expressions and Statements

3.1. Expressions

Expressions are constructed from operands and operators. The operators of an expression indicate which operations to apply to the operands. Examples of operators include +, -, *, /. Examples of operands include literals, fields, local variables, and expressions.

When an expression contains multiple operators, the precedence of the operators controls the order in which the individual operators are evaluated. For example, the expression x + y * z is evaluated as x + (y * z) because the * operator has higher precedence than the + operator.

Table 2summarizes C0 operators, listing the operator categories in order of precedence from highest to lowest. Operators in the same category have equal precedence.

Table 2. Operators in C0

Category	Expression	Description
Primary	x.m	Field access
	x(...)	Method invocation
	x[...]	Array orarray segment access
	x++	Post-increment
	x--	Post-decrement
	x->y	Pointer
Unary	*x	Dereference
	&x	Referencing the address
	+x	Identity
	-x	Negation
	!x	Logical negation
	~x	Bitwise negation
	++x	Pre-increment
	--x	Pre-decrement
	(T)x	Explicitly convert x to type T
Multiplicative	x * y	Multiplication
	x / y	Division
	x % y	Remainder
Additive	x + y	Addition
	x – y	Subtraction
Shift	x << y	Shift left
	x >> y	Shift right
Relational	x < y	Less than
	x > y	Greater than
	x <= y	Less than or equal
	x >= y	Greater than or equal
Equality	x == y	Equal
	x != y	Not equal
Logical AND	x & y	Integer bitwise AND,
Logical XOR	x ^ y	Integer bitwise XOR,
Logical OR	x | y	Integer bitwise OR
Conditional AND	x && y	Boolean logical AND
Conditional OR	x || y	Boolean logical OR
Conditional	x ? y : z	Evaluates y if x is true, z if x is false
Assignment	x = y	Assignment
	x ***op***= y	Compound assignment; supported operators are ***= /= %= += -= <<= >>=&= ^= |=**

3.2. Statements

The actions of a program are expressed using **statements.**

A block permits multiple statements to be written in contexts where a single statement is allowed. A block consists of a list of statements written between the delimiters **{** and**}**.

Declaration statements are used to declare local variables and constants.

Expression statements are used to evaluate expressions. Expressions that can be used as statements include method invocations, assignments using = and the compound assignment operators, and increment and decrement operations using the ++ and -- operators.

Selection statements are used to select one of a number of possible statements for execution based on the value of some expression. In this group are the **if** and **switch** statements.

Iteration statements are used to repeatedly execute an embedded statement. In this group are the **while**, **do**, and **for** statements.

Jump statements are used to transfer control. In this group are the **break**, **continue**, **goto**, and**return** statements.

4. Runners and Watchers

4.1. Define a runner

Defining a runner in DISA is just the same as defining a function. Actually any function satisfying the necessary constraints (will be mentioned later) can be started as a runner. A same function can either be directly invoked or be started as a new runner.

Thefunction that can become a runner must have the prototype with the following constraints

- It has no return type (with return type **void**)
- The parameters can only be either 1) simple types, or 3) array segments, or 3) structure types whose fields meet the constraints of 1) or 3).

The above constraints ensure that the input parameters to a new runner will not reference external memory locations not in the range of the parameters. The use of array segments constraints the use of pointers so the runtime can create the snapshots efficiently.

4.2. Creating instance of runners

The syntax of creating a runner is the same as invoking a function, plus the keyword **runner**. Note that the runner will only start to execute after current runner exits.

Example (quick sort):

```
int a[100];

int partition(int *v, int legnth, int ipivot) {
// ...
}

void qsort(int v[], start, length) {
if (length < 2)
commit;

int ipivot = start + rand() * length;
    ipivot = partition(&v[start], length, ipivot);

    runner { qsort(v[start,,ipivot], start, ipivot - start)); }
    runner
    {qsort(v[ipivot+1,,legnth-ipivot-1],ipivot+1,legnth-ipivot-1);}
commit;
}

void main() {
for (int i = 0; i< 100; i++) {
        a[i] = rand() * 10000;
    }

runner{qsort(v[0,,100], 0, 100); }

commit;
}
```

4.3. Watchers

The watchers are runners with additional startup conditions. Specifically, it will start after the parent runner commits successfully[??] and the specified memory location has modified since the creation of the watcher.

Defining a watcher is exactly the same as defining a normal runner.

To create an instance of a watcher, we also use runner keywords, with additional parameters to specify the memory location to watch. The watcher will get executed if the content of the memory has changed. The parameter can either be the pointer to a simple type or structure type, or an array segment.

```
void foo() {
int a[100];
intflag;

runner(a[0,,10]) { watcher1(); }
    runner(&flag){ watcher2(a[0,,100]); }

    // ...

commit;
}
```

4.4. Creating runners in another space

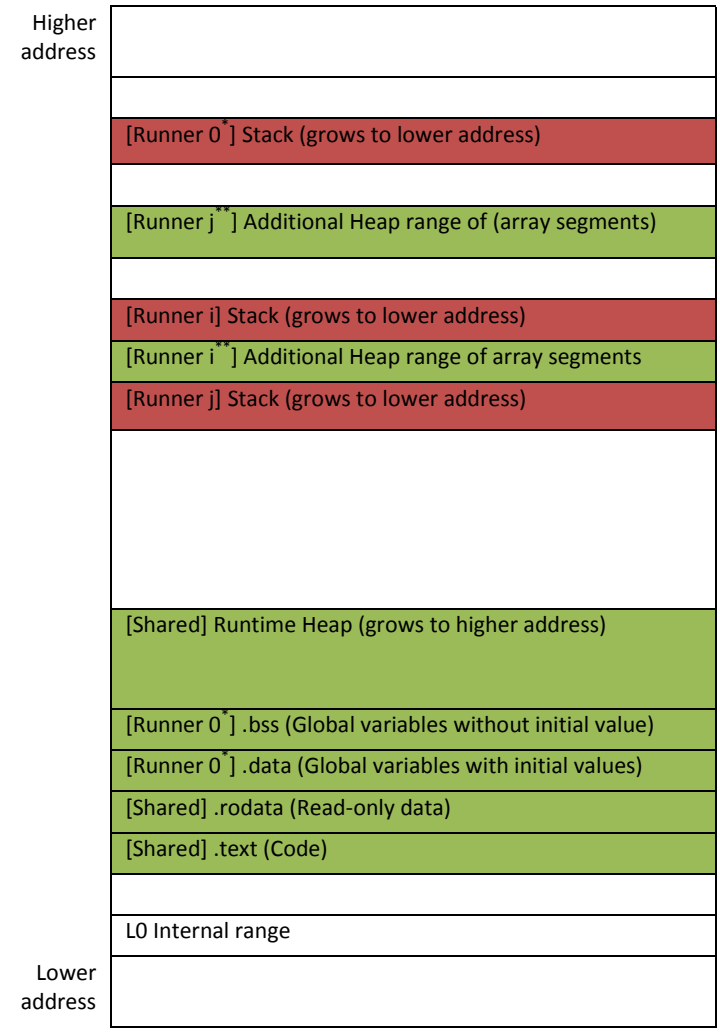
The memory space in i0 is separated into many spaces which is a continuous range. Each space hash a space specifier and the offset ranges for all spaces are the same. By default, the **runner** statement creates runners in the same space as the parent runner. The space can be specified by the **in** clause of the **runner** statement.

For example, to create a qsort runner in space SPACE1:

```
#include "libi0/stddef.h"

long space;

// To create a qsort runner in SPACE1

space = SPACE1;
runner qsort(0, 100)
    using v[0,,100]
    in space
;
```

5. Runtime Environment

5.1. Memory Layout



L0 memory type (indicated in colors):

Heap

Stack

* Suppose runner 0 is the first runner gets executed in the user application. It will has the access to all the global variables.

** The locations and sizes of additional heap ranges may be overlapped with other stack/heap ranges, depending on the location of array segments passed in the startup parameters.

5.2. Program loading

The DISA program will be compiled into DISA instructions as the ELF format.

At the start of the L0, the program loader will perform the following operations

- Load the ELF binary from the disk
- Parse the ELF headers
- For each section of ELF. (We only use the following sections: "**.text**", "**.data**", "**.rodata**", "**.bss**".)
    - Allocate the virtual memory range
    - Copy/map the data block into the memory; note that the length of data block might be less than the memory range. Fill the rest of the space with zeros.
- Create a snapshot, includes:
    - Heap: all the memory ranges of the ELF sections in memory
    - Initial dynamic heap with fixed size (e.g.1GB?, but we don't need to allocate memory pages now)
    - Fixed size (e.g. 64KB?) stack
- Start a new runner with the entry point and the created snapshot.

The memory layout is illustrated in Figure 1.[1]

[1] http://wiki.osdev.org/ELF

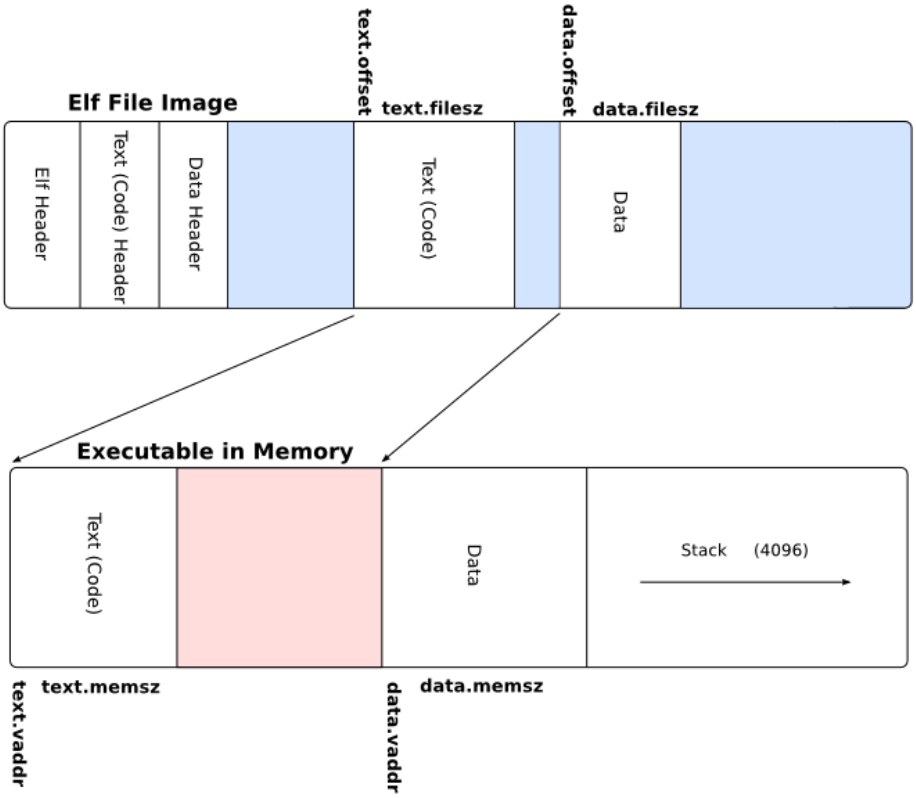


Figure 1. ELF file loading

Update History

May. 8, 2013. Add space for the runner statement. - zma