Melon Developer Guide

A software platform for simplifying-development on

UNIX

Version 0.8.1

Niklaus F. Schen Ran Xuxin

Melon-c@hotmail.com ran83816@gmail.com

Content

Preface

Melon is a software platform for simplifying-development on UNIX. Why is a platform? Because it allows other programs running on it and Melon will keep them alive automatically (of course, you can set not to restart processes after they are dead). In addition, it provides many interfaces to develop application programs.

Melon supports both multi-process and multithreads models. And it provides a method which is the same as process development to develop thread modules.

The goal of Melon is trying to simplify the module development (no matter a process or thread) and guarantee the performance.

People will see some customized data types, such as *mln_size_t*, their definitions can be found in *include/mln_types.h*, as shown in Figure 1.

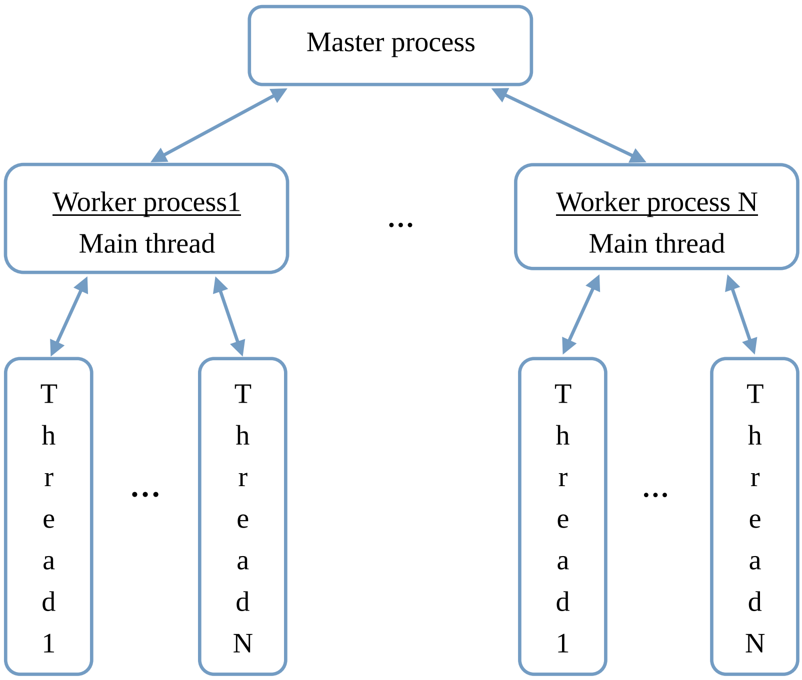

Figure 1. Melon's architecture.

Installation

We can install Melon following these steps:

a) Download and decompress Melon.
b) Change directory into the Melon.
c) Execute shell command
   sh configure
   to create makefile and some source and header files.
   There are only two options for *configure:*
   --help can show us the configuration information.
   --prefix can be used to indicate the Melon installation path.
d) Execute *make* to compile all source code.
e) Execute *make install* to install Melon to the specified path.
   Default installation path is */usr/local/*.

Interface

1. String

a) **extern mln_string_t *mln_new_string(const char *s);**
This interface returns a *mln_string_t* structure which is allocated by *malloc()*. And its content is varied by its argument. If the memory is not enough, NULL would be returned.

b) **extern mln_string_t *mln_dup_string(mln_string_t *str) __NONNULL1(1);**
It duplicates the argument and returns a new structure. The new structure is allocated by *malloc()*.

c) **extern mln_string_t *mln_ndup_string(mln_string_t *str, mln_s32_t size) __NONNULL1(1);**
It is the same as *mln_dup_string()*, but there is another argument to specify the length of the duplication.

d) **extern mln_string_t *mln_refer_string(mln_string_t *str) __NONNULL1(1);**
This interface creates a new pointer which points to the same address pointed by the argument. We don't free the string buffer when we are calling *mln_free_string()* to free *mln_string_t* because the string buffer is referenced from the other place.

e) **extern mln_string_t *mln_refer_const_string(char *s);**
It is the same as *mln_refer_string*, but the argument type.

f) **extern void mln_free_string(mln_string_t *str);**
It is used to free *mln_string_t* and its string buffer (if *str* is

not initialized by *mln_refer_string()* or *mln_refer_const_string()*).

g) **extern int mln_strcmp(mln_string_t *s1, mln_string_t *s2) __NONNULL2(1,2);**
It is the same as *strcmp()*, but the arguments' type is *mln_string_t*.

h) **extern int mln_const_strcmp(mln_string_t *s1, const char *s2) __NONNULL1(1);**
This interface is the same as *mln_strcmp()*, but it supports the type of *const char **.

i) **extern int mln_strncmp(mln_string_t *s1, mln_string_t *s2, mln_u32_t n) __NONNULL2(1,2);**
It is the same as *strncmp()*.

j) **extern int mln_const_strncmp(mln_string_t *s1, const char *s2, mln_u32_t n) __NONNULL1(1);**
It is the same as mln_strncmp(), but it supports the type of *const char **.

k) **extern int mln_strcasecmp(mln_string_t *s1, mln_string_t *s2) __NONNULL2(1,2);**
It is the same as *strcasecmp()*.

l) **extern int mln_const_strcasecmp(mln_string_t *s1, const char *s2) __NONNULL1(1);**
It is the same as *mln_strcasecmp()*, but it supports the type of *const char **.

m) **extern int mln_const_strncasecmp(mln_string_t *s1, const char *s2, mln_u32_t n) __NONNULL1(1);**
It is the same as *strncasecmp()*.

n) **extern int mln_strncasecmp(mln_string_t *s1, mln_string_t *s2, mln_u32_t n) __NONNULL2(1,2);**

It is the same as *mln_strncasecmp()*, but it supports the type of *const char **.

o) **extern char *mln_strstr(mln_string_t *text, mln_string_t *pattern) __NONNULL2(1,2);**
It is the same as *strstr()*.

p) **extern char *mln_const_strstr(mln_string_t *text, const char *pattern) __NONNULL2(1,2);**
It is the same as *mln_strstr()*, but it supports the type of *const char **.

q) **extern mln_string_t *mln_str_strstr(mln_string_t *text, mln_string_t *pattern) __NONNULL2(1,2);**
It is the same as *mln_strstr()*, but the type of the return value is *mln_string_t*.

r) **extern mln_string_t *mln_str_const_strstr(mln_string_t *text, const char *pattern) __NONNULL2(1,2);**
The same as *mln_const_strstr()*, but the type of return value is *mln_string_t*.

s) **extern char *mln_kmp_strstr(mln_string_t *text, mln_string_t *pattern) __NONNULL2(1,2);**
t) **extern char *mln_const_kmp_strstr(mln_string_t *text, const char *pattern) __NONNULL1(1);**
u) **extern mln_string_t *mln_str_kmp_strstr(mln_string_t *text, mln_string_t *pattern) __NONNULL2(1,2);**
v) **extern mln_string_t *mln_str_const_kmp_strstr(mln_string_t *text, const char *pattern) __NONNULL2(1,2);**
These four interfaces are the same as *strstr()*, but implemented by KMP algorithm.

w) **extern mln_string_t *mln_slice(mln_string_t *s, const char *sep_array/*ended by \0*/) __NONNULL2(1,2);**
Slice string into several pieces. Slicing characters can be

given by the second argument. This interface has a side-effect that the string buffer will be destroyed. So you can call *mln_dup_string()* or *mln_ndup_string()* at first. The return value is a vector, and its last element is NULL.

x) **extern void mln_slice_free(mln_string_t *array) __NONNULL1(1);**
This interface frees the vector that *mln_slice()* created.

2. Hash

```
struct mln_hash_attr {
        hash_calc_handler       hash;
        hash_cmp_handler        cmp;
        hash_free_handler       free_key;
        hash_free_handler       free_val;
        mln_u32_t               len_base;
        mln_u32_t               expandable;
};
```

This structure is used to be the argument of *mln_hash_init()* to initialize a hash table.

hash is a function pointer defined as

*typedef int (*hash_calc_handler)(mln_hash_t *, void *);*

to calculate and return a hash value to be the hash table index.

cmp is also a function pointer, it is defined as

*typedef int (*hash_cmp_handler) (mln_hash_t *, void *, void *);*

This function is used to compare with two hash elements' value. Its return value: !0 -- equal, 0 -- not equal.

free_key and *free_val* are two function pointers defined as

*typedef void (*hash_free_handler)(void *);*

They are used to free hash element's key and value. And these two pointers can be NULL.

len_base is an ideal value as a seed to be passed to a prime generator. And the generator returns a prime number which is equal to or greater than the seed.

expandable is a flag to indicate whether operations expand hash table dynamically. 0 means no, otherwise yes.

a) **extern mln_hash_t *mln_hash_init(struct mln_hash_attr *attr) __NONNULL1(1);**

   Initialize a hash table. The argument is a structure pointer discussed before.

b) **extern void mln_hash_destroy(mln_hash_t *h, enum mln_hash_flag flg) __NONNULL1(1);**
Destroy the hash table. If people set *free_key* and (or) *free_value*, this function will free hash element's key and (or) value at the same time.

c) **extern void *mln_hash_search(mln_hash_t *h, void *key) __NONNULL2(1,2);**
This interface searches a value which is specified by the *key*.

d) **extern int mln_hash_insert(mln_hash_t *h, void *key, void *val) __NONNULL3(1,2,3);**
It inserts a key-value into the hash table. If *malloc()* cannot allocate memory, -1 would be returned.

e) **extern void mln_hash_remove(mln_hash_t *h, void *key, enum mln_hash_flag flg) __NONNULL2(1,2);**
It removes a key-value from the hash table. If the *free_key* and (or) *free_value* are set, the element's key and (or) value would be freed at the same time.

3. Fibonacci Heap

```
struct mln_fheap_attr {
    fheap_cmp          cmp;
    fheap_copy         copy;
    fheap_key_free     key_free;
    void               *min_val;
    mln_size_t         min_val_size;
};
```

cmp is a function pointer defined as

*typedef int (*fheap_cmp)(const void *, const void *);*

The two arguments are customized structure pointers. And the return value is: 0 - ptr1 < ptr2, !0 - ptr1 >= ptr2.

copy is also a function pointer defined as

*typedef void (*fheap_copy)(void *dest, void *src);*

This function is used to copy data from *src* to *dest*.

key_free is a function pointer to free key's value in a heap node. It is defined as

*typedef void (*fheap_key_free)(void *);*

This pointer can be NULL.

min_val is a key's value actually. This is the minimum value in the heap.

min_val_size is the size of *min_val* structure.

a) **extern mln_fheap_node_t* mln_fheap_node_init(mln_fheap_t *fh, void *key) __NONNULL2(1,2);**

It initializes a fibonacci heap node. The first argument is a pointer points to a fibonacci heap. The second one is a user data. If memory is not enough, NULL would be returned.

b) **extern void mln_fheap_node_destroy(mln_fheap_t *fh, mln_fheap_node_t *fn) __NONNULL2(1,2);**

It destroys a heap node and frees its system resources. The second argument is the pointer that *mln_fheap_node_init()*

returned. If *key_free* isn't NULL, the key's value in the heap node would be freed.

c) **extern mln_fheap_t *mln_fheap_init(struct mln_fheap_attr *attr) __NONNULL1(1);**
It initializes and returns a fibonacci heap.

d) **extern void mln_fheap_destroy(mln_fheap_t *fh);**
It destroys a fibonacci heap. If *key_free* is not NULL, the key's value in every heap node would be freed by *key_free* at the same time.

e) extern void mln_fheap_insert(mln_fheap_t *fh, mln_fheap_node_t *fn) __NONNULL2(1,2);
This interface inserts a heap node.

f) extern void mln_fheap_delete(mln_fheap_t *fh, mln_fheap_node_t *node) __NONNULL2(1,2);
This interface removes a heap.

g) extern mln_fheap_node_t* mln_fheap_minimum(mln_fheap_t *fh) __NONNULL1(1);
It retrieves a minimum heap node. If the heap is empty, NULL would be returned.

h) extern mln_fheap_node_t* mln_fheap_extract_min(mln_fheap_t *fh) __NONNULL1(1);
Extract the node whose key value is minimum in the heap. If the heap is empty, NULL would be returned.

i) extern int mln_fheap_decrease_key(mln_fheap_t *fh, mln_fheap_node_t *node, void *key) __NONNULL3(1,2,3);
It decreases the key's value of the specified *node* to the *key*. The type of *key* must be identical to the type of keys in heap nodes. The return value is: -1 - key error, 0 - succeed.

4. Red-Black Tree

struct mln_rbtree_attr {
rbtree_cmp cmp;
rbtree_free_data data_free;
};

cmp is a function pointer to compare with two RB-tree data. It is defined as
*typedef int (*rbtree_cmp)(const void *, const void *);*
The type of two arguments is customized. And the return value is: >0 -- the first one is greater than the second; ==0 -- equal; <0 -- less.

data_free is a function pointer to free data in a tree node. This pointer can be NULL. It is defined as
*typedef void (*rbtree_free_data)(void *);*

a) **extern mln_rbtree_node_t *mln_rbtree_new_node(mln_rbtree_t *t, void *data) __NONNULL2(1,2);**
It creates a new RB-tree node. The first argument is the pointer points to a RB-tree which is the one that the new node is going to be inserted. The second one is a user data pointer. If memory is not enough, NULL would be returned.

b) **extern void mln_rbtree_free_node(mln_rbtree_t *t, mln_rbtree_node_t *n) __NONNULL2(1,2);**
This function destroys RB-tree node and frees its resources. If RB-tree's *data_free* is not NULL, the user data in the node would be freed by calling function *data_free()*.

c) **extern mln_rbtree_t *mln_rbtree_init(struct mln_rbtree_attr *attr) __NONNULL1(1);**
It initializes an RB-tree. If memory is not enough, NULL would be returned.

d) **extern void mln_rbtree_destroy(mln_rbtree_t *t);**
It destroys an RB-tree. If *data_free* is not NULL, all user data in the whole tree would be freed.

e) **extern void mln_rbtree_insert(mln_rbtree_t *t, mln_rbtree_node_t *n) __NONNULL2(1,2);**
It inserts an RB-tree node into a tree.

f) **extern void mln_rbtree_delete(mln_rbtree_t *t, mln_rbtree_node_t *n) __NONNULL2(1,2);**
It removes an RB-tree node from a tree.

g) **extern mln_rbtree_node_t* mln_rbtree_successor(mln_rbtree_t *t, mln_rbtree_node_t *n) __NONNULL2(1,2);**
It finds *n*'s successor. If nothing can be found, *&(t->nil)* would be returned. This return value is the address of tree's variable *nil*.

h) **extern mln_rbtree_node_t* mln_rbtree_search(mln_rbtree_t *t, mln_rbtree_node_t *root, const void *key) __NONNULL3(1,2,3);**
It searches an RB-tree node whose key is equal to the *key*. The routine will start from *root*. If nothing can be found, *&(t->nil)* would be returned.

i) **extern mln_rbtree_node_t* mln_rbtree_min(mln_rbtree_t *t) __NONNULL1(1);**
It returns the node whose user data is minimum in a tree. If the tree is empty, *&(t->nil)* would be returned.

5. Path

a) **extern char *mln_get_path(void);**
Depending on configuration, the installation path can be specified. So we have to get the absolute path because some components need it.

6. Prime Generator

a) **extern mln_u32_t mln_calc_prime(mln_u32_t n);**
It returns a prime number which is equal to or greater than *n*.

7. Lock

The lock that we discuss here is the spin lock.

a) MLN_LOCK_INIT(lock_ptr)
It initializes a spin lock. The argument is a lock pointer. The return value is an integer, 0 means OK, otherwise the return value is equivalent to the error number.

b) MLN_LOCK_DESTROY(lock_ptr)
It destroys a spin lock. The return value is an integer, 0 means OK, otherwise the return value is equivalent to the error number.

c) MLN_LOCK(lock_ptr)
This interface triggers a lock.

d) MLN_TRYLOCK(lock_ptr)
This interface tries to lock. When it fails, !0 will be returned. Otherwise 0.

e) MLN_UNLOCK(lock_ptr)

This interface releases the lock.

8. Log

There are some interfaces about log files, but not all of them are useful for developers. So we just discuss some useful interfaces.

a) **mln_log(err_lv,msg,...);**
This interface is widely used in Melon. We use it to record log message. The first argument is log level. There are five levels: *none, report*, *debug*, *error* and *nolog*. The initial log level can be set in a configuration file. The second argument is a string. It likes the first argument *fmt* in *printf()*. And it supports some format control characters, such as %s, %l, %d, %c, %f, %x, %X (=%lx in *printf()*), %u and %U (=%lu in *printf()*). Depending on *msg*'s format control characters, the rest arguments could be given or not.

b) **extern char *mln_get_log_dir_path(void);**
It returns a path of log files' directory.

c) **extern char *mln_get_log_file_path(void);**
It returns a path of a log file.

d) **extern char *mln_get_pid_file_path(void);**
It returns the path of a pid file. The pid of the parent process is recorded in that file.

9. Lexer

In Melon, Lexer is very easy to use and customize. Even though, there are some open source lexer generators, such as flex, but they are not so much simple as Melon's.

If you would like to write a lexer, you only need four steps.

a) *#include "mln_lex.h"*

b) Define structures and enumerations and declare functions.

   We just need to write a macro *MLN_DEFINE_TOKEN_TYPE_AND_STRUCT()* at the beginning of the file behind *#include "mln_lex.h"*.

   For example, if we are going to define a lexer, its functions' scope is "*extern*". *T*he prefix of every function, structure and enumeration name is "*mln_test_lex*". Its token prefix is "*TEST*" and it has three keywords "*for*", "*while*" and "*switch*", as shown below.

   MLN_DEFINE_TOKEN_TYPE_AND_STRUCT(extern, mln_test_lex, TEST, TEST_TK_FOR, TEST_TK_WHILE, TEST_TK_SWITCH);

   The *TEST_TK_FOR*, *TEST_TK_WHILE* and *TEST_TK_SWITCH* are token types of keywords "*for*", "*while*" and "*switch*".

   We should pay attention to the enumeration sequence that the customized special character declarations should be written before keywords'. And the sequence of keywords' enumerations should be identical to the sequence written in a keywords array.

c) Then we need to define functions and related arrays, e.g.,

   MLN_DEFINE_TOKEN(mln_test_lex, TEST);

   Now we have already defined 39 functions and a structure array. There are 33 functions are associated with processing special characters. One function is for setting customized functions to process special characters. One function is for creating token structure. One is for destroying token structure. One is for processing keywords. One is for processing all special characters' hooks. The last one function is for getting a token from a file or string buffer. The array maintains all special characters' handlers. You can see *include/mln_lex.h* and *src/mln_conf.c*. It is essential for the configuration.

d) Set lexer's attributions and call *MLN_LEX_INIT_WITH_HOOKS()* to initialize a lexer object, e.g.,

```
struct mln_lex_attr attr;
... //Set some special characters' processing hooks.
mln_lex_t *lex = NULL;
MLN_LEX_INIT_WITH_HOOKS(mln_test_lex, lex,
&attr);
```

Now a lexer has been made. The *attr* is defined as

```
struct mln_lex_attr {
        enum {
                mln_lex_file,
                mln_lex_buf
        } input_type;
        union {
                mln_s8ptr_t      filename;
                /*
                  * file_buf must be ended by '\0'.
                  */
                mln_s8ptr_t      file_buf;
        } input;
        /*
          * keywords must be ended by NULL
          */
        char                     **keywords;
```


*mln_lex_hooks_t *hooks;*

};

input_type is used to indicate the type of input, file or string buffer.

input maintains the relevant information of the input, filename or string buffer.

keywords is a vector which records all keywords that we need and the last element in vector is NULL.

Even though, there are 39 functions will be built in step c), but only three of them should be concerned.

A) **mln_test_lex_struct_t *mln_test_lex_new(mln_lex_t *lex, enum mln_test_lex_enum type);**

It initializes a new token. If memory is not enough, NULL would be returned.

B) **void mln_test_lex_free(mln_test_lex_struct_t *ptr);**

It frees a token structure which is given by the argument.

C) **mln_test_lex_struct_t *mln_test_lex_token(mln_lex_t *lex);**

It returns a token structure. If people call it again, the next token would be returned. If the lexer encounters the EOF, a token structure with the type *TK_xxx_TK_EOF* would be returned. If lexer encounters an error, NULL would be returned and the error number would be set in a lexer object.

Besides these three interfaces, there are some others we also need to know.

A) **extern void mln_lex_destroy(mln_lex_t *lex);**

It destroys the lexer object. But this function does not destroy all token structures created by a lexer object.

B) **extern char *mln_lex_strerror(mln_lex_t *lex) __NONNULL1(1);**

As *strerror()*, this function will return an error message.

C) **extern char mln_geta_char(mln_lex_t *lex) __NONNULL1(1);**
Get the next character from a file or a string buffer. If lexer encounters EOF, *MLN_EOF* would be returned. If it encounters an error, *MLN_ERR* would be returned and error number would be set.

D) **extern int mln_puta_char(mln_lex_t *lex, char c) __NONNULL1(1);**
It puts a character into the result buffer which is a string that *xxx_token()* returned. If it encounters an error, *MLN_ERR* would be returned and the error number would be set.

E) **extern void mln_step_back(mln_lex_t *lex) __NONNULL1(1);**
It steps back a character. If this interface is called between calling *mln_geta_char()* twice, the returned characters via calling *mln_geta_char()* twice are the same.

F) **extern int mln_isletter(char c);**
This interface is equivalent to *if (c == '_' || isalpha(c))*.

G) **extern int mln_isoctal(char c);**
This interface is equivalent to *if (c >= '0' && c < '8')*.

H) **extern int mln_ishex(char c);**
This interface is equivalent to *if (isdigit(c) || (c >= 'a' && c <= 'f') || (c >= 'A' && c <= 'F'))*.

10. Configuration

As shown below, this is the architecture of configuration.

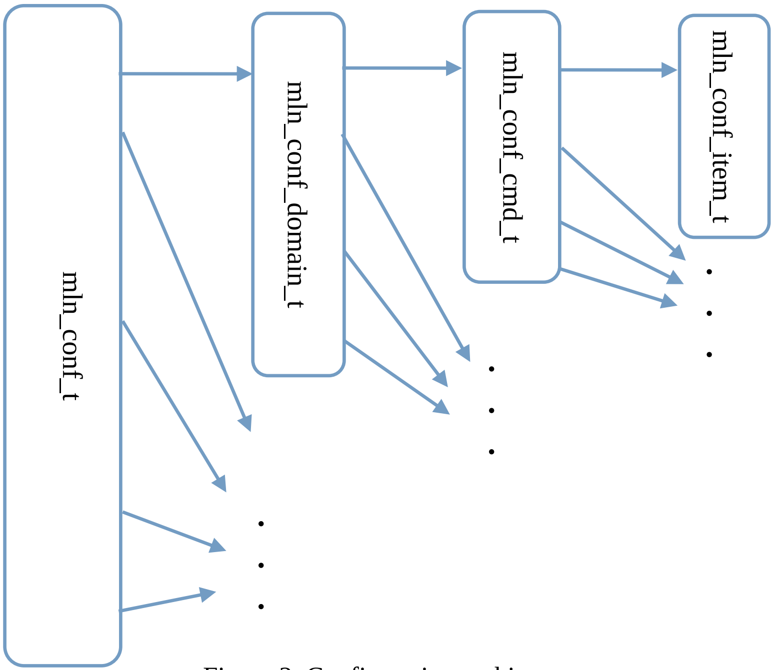

Figure 2. Configuration architecture.

Except *mln_conf_item_t*, other structures have a function pointer named *search*. In *mln_conf_t*, it is defined as

*typedef mln_conf_domain_t *(*search_domain)(mln_conf_t *, char *);*

This function is used to search a conf-domain. The first argument is a configuration object returned by *mln_get_conf()*. The second argument is a string indicating the domain name. If domain is existed, the domain object (*mln_conf_domain_t)* would be returned. Otherwise, the return value is NULL.

In *mln_conf_domain_t*, the function *search* is defined as

typedef mln_conf_cmd_t (*search_cmd) (mln_conf_domain_t *, char *);*

This function is used to search a conf-command. The first argument is a domain object. The second argument is the command name that we are looking for. If command is not existed, NULL would be returned.

In *mln_conf_cmd_t*, the function *search* is defined as

*typedef mln_conf_item_t *(*search_item) (mln_conf_cmd_t *, mln_u32_t);*

The first argument is a command object. The second one is an index indicating the position of the item in command. This index is start from 1. If index value is invalid, NULL would be returned.

The *mln_conf_item_t* is the smallest unit in configuration. Its structure is

```
struct mln_conf_item_s {
    enum {
        CONF_NONE = 0,
        CONF_STR,
        CONF_CHAR,
        CONF_BOOL,
        CONF_INT,
        CONF_FLOAT
    } type;
    union {
        mln_string_t *s;
        mln_s8_t c;
        mln_u8_t b;
        mln_sauto_t i;
        float f;
    } val;
};
```

We can operate this structure directly to get the value that we need.

About the usage of configuration file, there are something we need to know.

1. Configuration file has a concept named domain. The number of domain is unlimited. But one domain cannot be nested in the other one, which means the level of domain is

always 1.

2. We can define any domains or commands in a configuration file (of course, their tokens should be valid to the lexer) and Melon won't report any warnings or errors while there is an unused command or domain written in file.

3. The specification of configuration lexical analyzer follows these rules below:

   I. The name in the configuration file should be start with a letter or '_'. And the rest characters can be composed by digit, letter and '_'.

   II. Domain can be defined by a closure which is composed by a domain name, a left brace and a right brace at least.

   III. Except domain, the others are all called command. The first token in a command is the command name. And the rest tokens are called item. The number of item in a command can be 0.

   IV. The type of an item can be the one of following types: string, character, boolean, integer and float. Their written form is compatible with C language.

   V. Every command should be ended by a ';'.

   VI. The way of writing comments is the same as C language.

4. There are two keywords that we support, *on* and *off*. Their value is 1 and 0 recorded in union *val*'s *b* in an item. The type of them is *CONF_BOOL*.

There are some other interfaces we should know.

a) **extern mln_conf_t *mln_get_conf(void);**

   It gets a *mln_conf_t* object. Then we can use its *search* to find out the domain that we need. The return value won't be NULL until Melon crashed.

b) **extern mln_u32_t mln_get_cmd_num(mln_conf_t *cf, char *domain) __NONNULL2(1,2);**

   It returns the number of commands in the *domain*. If *domain* is not existed, NULL would be returned.

c) **extern void mln_get_all_cmds(mln_conf_t *cf, char *domain, mln_conf_cmd_t **vector) __NONNULL3(1,2,3);**
It puts all commands in the *domain* into the third argument. v*ector* should be allocated before calling.

d) **extern mln_u32_t mln_get_cmd_args_num(mln_conf_cmd_t *cc) __NONNULL1(1);**
It returns the number of items which belong to *cc* (command object).

11. Event

As libevent, Melon's event module integrates epoll, kqueue and select. And it supports three kinds of event: timer event, signal event and file descriptor event.

a) **extern mln_event_t *mln_event_init(mln_u32_t is_main);**
It initializes an event object. *is_main* indicates whether the event object is initialized in a main thread.

b) **extern void mln_event_destroy(mln_event_t *ev) __NONNULL1(1);**
It destroys an event object.

c) **extern int mln_event_set_fd(mln_event_t *event, int fd, mln_u32_t flag, void *data, void (*fd_handler)(mln_event_t *, int, void *)) __NONNULL1(1);**
This function sets a file descriptor event. The first argument is an event object. The second one is the file descriptor that we are interested. The third one is a flag indicating the type of this event. Some flags are listed below:
I. M_EV_RECV
This flag indicates the event is a read event.
II. M_EV_SEND
Indicates the event is a write event.
III. M_EV_ERROR
Indicates the event is an error event.
IV. M_EV_ONESHOT
Indicates the event only triggered once.
V. M_EV_NONBLOCK
Set file descriptor to be non-blocking mode.
VI. M_EV_BLOCK
Set file descriptor to be blocking mode.

VII. M_EV_APPEND

Set this flag will retain the old event type and the new type.

VIII. M_EV_CLR

Remove this file descriptor event.

I, II, III, IV, V (or VI), VII can be used in the same one event combined with '|'.

The fourth is a user data that will be passed to a function which is given by the last argument. The arguments of the *fh_handler* are an event object, a file descriptor and a user data.

If an event is failed to set, -1 would be returned. Otherwise, the returned value is 0.

d) **extern int mln_event_set_timer(mln_event_t *event, mln_u32_t msec, void *data, void (*tm_handler)(mln_event_t *, void *)) __NONNULL1(1);**

It sets a timer event. The first argument is an event object. The second one is a millisecond timer. Its unit is 10ms. Third argument is a user data. The last one is an event handler. The arguments of *tm_handler* are an event object and a user data.

If an event is failed to set, -1 would be returned. Otherwise, the returned value is 0.

e) **extern int mln_event_set_signal(mln_event_t *event, mln_u32_t flag, int signo, void *data, void (*sg_handler)(mln_event_t *, int, void *)) __NONNULL1(1);**

It sets a signal event. The first argument is an event object. The second one is a flag indicating the type of event to install or uninstall the event handler. Its value can be *M_EV_SET* or *M_EV_UNSET*. The third one is the signal number that we catch. The fourth one is a user data. Last one is an event handler. The arguments of *sg_handler* are an event object, a signal number and a user data.

If an event is failed to set, -1 would be returned. Otherwise, 0 would be returned.

It is different between this interface and the one in libevent. We allow to set signal event handlers (no matter they are different or not) for the same one signal into one or many event objects. When a signal raised, the handlers that are whether in the same event object or not, will be called in their threads.

f) **extern void mln_dispatch(mln_event_t *event) __NONNULL1(1);**

It dispatches every event. If a routine jumps into this function, it wouldn't be out until it encounters two situations: calling *mln_event_set_break()* to break out or a child process is forked.

g) **extern void mln_event_set_break(mln_event_t *ev) __NONNULL1(1);**

It lets the routine break out from *mln_dispatch()*.

12. Connection I/O

Connection I/O, for now, only supports TCP (actually, it also supports file I/O).

a) **extern void mln_tcp_connection_init(mln_tcp_connection_t *c, int fd) __NONNULL1(1);**
Initialize a connection object which is provided by the first argument.

b) **extern void mln_tcp_connection_destroy(mln_tcp_connection_t *c) __NONNULL1(1);**
Destroy a connection object. Free *c*'s buffers.

c) **extern int mln_tcp_connection_set_buf(mln_tcp_connection_t *c, void *buf, mln_u32_t len, int type) __NONNULL1(1);**
Set a connection's buffer. The connection object has two buffers. One is for sending, the other for receiving. We can identify the buffer type from the last argument. It has two value: *M_C_SEND* and *M_C_RECV*. We will copy *len* bytes from *buf* to the connection's buffer that we set.

d) **extern void mln_tcp_connection_clr_buf(mln_tcp_connection_t *c, int type) __NONNULL1(1);**
Clear and free connection's buffer. The buffer type is indicated by *type*. *Type* value can be *M_C_SEND* or *M_C_RECV*.

e) **extern void *mln_tcp_connection_get_buf(mln_tcp_connection_t *c, int type) __NONNULL1(1);**
Get a connection's buffer. The buffer type is indicated by

the second argument, its value can be *M_C_SEND* or *M_C_RECV*.

f) **extern int mln_tcp_connection_send(mln_tcp_connection_t *c) __NONNULL1(1);**
Send data via TCP. There are four kinds of return value.
  I. M_C_FINISH
  Data transfer is completed.
  II. M_C_NOTYET
  Data transfer is uncompleted.
  III. M_C_ERROR
  Transfer error.
  IV. M_C_CLOSED
  Connection closed by peer.
If a return value is M_C_FINISH, we could get buffer via *mln_tcp_connection_get_buf()*. And we should clear connection's buffer after we get the concerned buffer via *mln_tcp_connection_clr_buf()*.

g) **extern int mln_tcp_connection_recv(mln_tcp_connection_t *c) __NONNULL1(1);**
Receive data via TCP. The return value is the same as *mln_tcp_connection_send()*.

h) **M_C_SND_EMPTY(c_ptr);**
Test whether the sent buffer is empty or not.

i) **M_C_RCV_EMPTY(c_ptr);**
Test whether the receive buffer is empty or not.

j) **M_C_SND_NULL(c_ptr);**
Test whether the sent buffer pointer is NULL or not.

k) **M_C_RCV_NULL(c_ptr);**
Test whether the received buffer pointer is NULL or not.

Process Module Development

This chapter we do not discuss the interfaces in *include/mln_ipc.h* and *include/mln_fork.h*. Those interfaces are used in Melon's fundamental components.

Actually, it is not much difference between application program development and Melon process module development. We can write a normal application program and configure its path and parameters in Melon's configuration file.

Now, let's follow these steps to develop our process module. Of course, we assume Melon has already installed.

1. Write a program.

EXAMPLE:

```
#include <stdio.h>
int main(int argc, char *argv[])
{
    printf("This is a test. %s\n", argv[1]);
    return 0;
}
```

2. Compile this source file.

```
cc -o a a.c
```

3. Modify Melon's configuration file.

   vim xxx/melon/conf/melon.conf

   We assume the new program's path is */home/John/a*.

   There is a domain named *exec_proc*. We can add a command in it.

   keepalive/default "/home/John/a" "argument1";

   Command name indicates the process type, there are two types: *keepalive* and *default*. *Keepalive* will restart process when the process is killed by an unexpected error or system command *kill*. *Default* will do nothing after the process is dead.

4. Modify *configuration* file to build our new program automatically. This step is optional.

5. Start up Melon platform.

Now, out program *a* is running.

In the above example, the last argument in program *a* is not *argument1* but a string of a file descriptor, even though there is no argument written in the command. This file descriptor is connecting with Melon master process. If command name is *keepalive* and program *a* is killed, this connection would be closed, and the master process of Melon would receive this event and restart program *a*. If command name is *default* and the file descriptor is closed, Melon would never restart *a* and ignore this event.

Now we can supervise our child process. The next essential is IPC. How to implement IPC? We provide an easy way. We don't need to modify the original source file. There is a special directory existing for this, named *ipc_handlers*. All files in this directory are used to define the IPC message types and their handlers.

File name can be separated into two parts. The first one is the prefix of handlers' name. There are two handlers. One is for master process named *xxx_master* and the other is for worker process named *xxx_worker*. The other part is the message type. This type will be defined in *include/mln_ipc.h*, and we don't need to modify this file. Executing shell script *configure*, this file will be re-created automatically, then all message types are re-defined.

The declaration of IPC handler is:

void ***prefix****_master/worker(mln_event_t *ev, void *f_ptr, void *buf, mln_u32_t len, void **udata_ptr);*

The first argument is an event object that is related with this connection file descriptor. The second one is a *mln_fork_t* pointer, we need this pointer to provide us some essentials about child process, such as TCP connection object. The third one is the data that master or worker received. *Len* indicates the length of *buf*. The last one is a customized pointer. We can allocate a block of memory to restore other useful data. And this memory block should be freed by the handler. Melon won't free it.

Thread Module Development

In Melon, thread module is not only like process module development, but also like IPC development.

There is a directory named *threads* that is provided for maintaining thread module files. Every file is a thread module. The file name is the thread's alias.

Every thread has an entrance named ***alias**_main*, it's defined as

*int **alias**_main(int argc, char **argv);*

Thread will start from this function. This function is the same as *main()* in application program except its name.

Not all of these thread modules in directory *threads* can be started up, but only the commands written in the domain *thread_exec* in configuration file.

Now, let's see an example.

A) Create a thread module file.

```
touch threads/hello
```

Then the thread's alias is *hello*, and its main function should be named *hello_main*.

B) Edit the thread file.

```
#include ... //include some essential header files.
int hello_main(int argc, char *argv[])
{
        //this is a thread module.
        mln_log(debug, "hello thread!\n");
        return 0;
}
```

This thread is going to do one thing that output *hello thread!* to the log. The last available element in the second argument is a string indicating a file descriptor that is combined with a TCP connection for communicating with the main thread. And the first element in the second argument is the thread alias. The rest elements are all parameters.

C) Modify configuration file.

thread_exec {

// format: restart|default "alias" ["parameter", ...];

//... Other threads

restart "hello";

}

In this example, *restart* is the command name that indicates Melon to restart this thread when it exited. There is another command name *default*, it indicates Melon just to clean up the thread's resources when the thread exited.

The type of alias and parameters must be the string.

D) Start up Melon

There is a command in the configuration file which is *worker_process*. Its item notices Melon how many worker processes will be started up. What's different between worker process and the process module? The worker process is only used to run thread modules. If there is no thread in domain *thread_exec*, the worker process would still be started up but do nothing. However, the process module is a new process that it's not for running thread modules (Of course, you can implement multithread in your process).

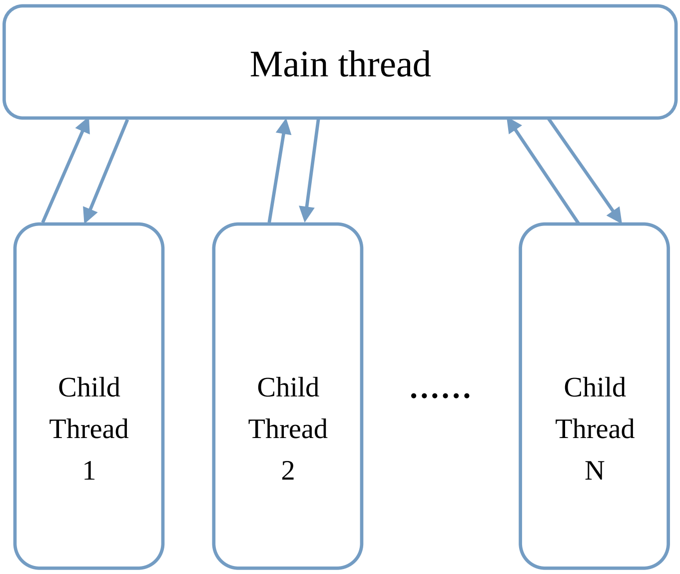

Figure 3. Inter-thread communication.

Now, we discuss inter-thread communication.

As shown in Figure 3, it is the inter-thread communication architecture.

We can see the message is only transferred between the main thread and a child thread. If *child_thread1* wants to send a message to *child_thread2*, the message should be delivered to the main thread at first, and then it would be transferred to the destination thread. This mechanism effectively reduces the coupling degree. It is very like the micro kernel model.

Now, let's see the format of inter-thread communication message.

```
typedef struct {
        mln_string_t                *dest;
        mln_string_t                *src;
        double                       f;
        void                        *pfunc;
        void                        *pdata;
        mln_sauto_t                  sauto;
        mln_uauto_t                  uauto;
        mln_s8_t                     c;
        mln_s8_t                     padding[7];
        enum {
                ITC_REQUEST,
                ITC_RESPONSE
        }                            type;
        int                          need_clear;
} mln_thread_msg_t;
```

dest is an alias indicating the destination thread. If this alias is not existed, message would be dropped by main thread, and main thread wouldn't send any error message to the source thread.

src is an alias to indicate the source thread. This variable is set by main thread.

type indicates the type of message is a request or response.

need_clear indicates whether the *pdata* should be freed.

The rest variables can be used to pass some arguments and (or) a function pointer.

There are three interfaces we should know:

A) ***extern void mln_thread_clear_msg(mln_thread_msg_t *msg);***
Free *msg*’s memory.

B) ***extern void mln_thread_exit(int sockfd, int exit_code);***
Exit the thread.

C) ***extern void mln_set_cleanup(void (*tcleanup)(void *), void *data);***
Set a cleanup function that will be called after your thread module main function exits.