z:\game\ff_server\skynet\skynet-src\atomic.h

```
1  #ifndef SKYNET_ATOMIC_H
2  #define SKYNET_ATOMIC_H
3
4  #define ATOM_CAS(ptr, oval, nval)
   __sync_bool_compare_and_swap(ptr, oval, nval)
5  #define ATOM_CAS_POINTER(ptr, oval, nval)
   __sync_bool_compare_and_swap(ptr, oval, nval)
6  #define ATOM_INC(ptr) __sync_add_and_fetch(ptr, 1)
7  #define ATOM_FINC(ptr) __sync_fetch_and_add(ptr, 1)
8  #define ATOM_DEC(ptr) __sync_sub_and_fetch(ptr, 1)
9  #define ATOM_FDEC(ptr) __sync_fetch_and_sub(ptr, 1)
10 #define ATOM_ADD(ptr,n) __sync_add_and_fetch(ptr, n)
11 #define ATOM_SUB(ptr,n) __sync_sub_and_fetch(ptr, n)
12 #define ATOM_AND(ptr,n) __sync_and_and_fetch(ptr, n)
13
14 #endif
```

z:\game\ff_server\skynet\skynet-src\luashrtbl.h

```
1  #ifndef LUA_SHORT_STRING_TABLE_H
2  #define LUA_SHORT_STRING_TABLE_H
3
4  #include "lstring.h"
5
6  // If you use modified lua, this macro would be defined in
   lstring.h
7  #ifndef ENABLE_SHORT_STRING_TABLE
8
9  static inline int luaS_shrinfo(lua_State *L) { return 0; }
10 static inline void luaS_initshr() {}
11 static inline void luaS_exitshr() {}
12 static inline void luaS_expandshr(int n) {}
13
14 #endif
15
16 #endif
```

z:\game\ff_server\skynet\skynet-src\malloc_hook.c

```c
#include <stdio.h>
#include <string.h>
#include <assert.h>
#include <stdlib.h>
#include <lua.h>
#include <stdio.h>

#include "malloc_hook.h"
#include "skynet.h"
#include "atomic.h"

static size_t _used_memory = 0;
static size_t _memory_block = 0;
typedef struct _mem_data {
    uint32_t handle;
    ssize_t allocated;
} mem_data;

#define SLOT_SIZE 0x10000
#define PREFIX_SIZE sizeof(uint32_t)

static mem_data mem_stats[SLOT_SIZE];


#ifndef NOUSE_JEMALLOC

#include "jemalloc.h"

// for skynet_lalloc use
#define raw_realloc je_realloc
#define raw_free je_free

static ssize_t*
get_allocated_field(uint32_t handle) {
    int h = (int)(handle & (SLOT_SIZE - 1));
    mem_data *data = &mem_stats[h];
    uint32_t old_handle = data->handle;
    ssize_t old_alloc = data->allocated;
    if(old_handle == 0 || old_alloc <= 0) {
```

```
40          // data->allocated may less than zero, because it may
    not count at start.
41          if(!ATOM_CAS(&data->handle, old_handle, handle)) {
42              return 0;
43          }
44          if (old_alloc < 0) {
45              ATOM_CAS(&data->allocated, old_alloc, 0);
46          }
47      }
48      if(data->handle != handle) {
49          return 0;
50      }
51      return &data->allocated;
52  }
53
54  inline static void
55  update_xmalloc_stat_alloc(uint32_t handle, size_t __n) {
56      ATOM_ADD(&_used_memory, __n);
57      ATOM_INC(&_memory_block);
58      ssize_t* allocated = get_allocated_field(handle);
59      if(allocated) {
60          ATOM_ADD(allocated, __n);
61      }
62  }
63
64  inline static void
65  update_xmalloc_stat_free(uint32_t handle, size_t __n) {
66      ATOM_SUB(&_used_memory, __n);
67      ATOM_DEC(&_memory_block);
68      ssize_t* allocated = get_allocated_field(handle);
69      if(allocated) {
70          ATOM_SUB(allocated, __n);
71      }
72  }
73
74  inline static void*
75  fill_prefix(char* ptr) {
76      uint32_t handle = skynet_current_handle();
77      size_t size = je_malloc_usable_size(ptr);
78      uint32_t *p = (uint32_t *)(ptr + size -
    sizeof(uint32_t));
79      memcpy(p, &handle, sizeof(handle));
```

```
80
81      update_xmalloc_stat_alloc(handle, size);
82      return ptr;
83  }
84
85  inline static void*
86  clean_prefix(char* ptr) {
87      size_t size = je_malloc_usable_size(ptr);
88      uint32_t *p = (uint32_t *)(ptr + size -
    sizeof(uint32_t));
89      uint32_t handle;
90      memcpy(&handle, p, sizeof(handle));
91      update_xmalloc_stat_free(handle, size);
92      return ptr;
93  }
94
95  static void malloc_oom(size_t size) {
96      fprintf(stderr, "xmalloc: Out of memory trying to allocate
    %zu bytes\n",
97          size);
98      fflush(stderr);
99      abort();
100 }
101
102 void
103 memory_info_dump(void) {
104     je_malloc_stats_print(0,0,0);
105 }
106
107 size_t
108 mallctl_int64(const char* name, size_t* newval) {
109     size_t v = 0;
110     size_t len = sizeof(v);
111     if(newval) {
112         je_mallctl(name, &v, &len, newval, sizeof(size_t));
113     } else {
114         je_mallctl(name, &v, &len, NULL, 0);
115     }
116     // skynet_error(NULL, "name: %s, value: %zd\n", name, v);
117     return v;
118 }
119
```

```
120 int
121 mallctl_opt(const char* name, int* newval) {
122     int v = 0;
123     size_t len = sizeof(v);
124     if(newval) {
125         int ret = je_mallctl(name, &v, &len, newval,
    sizeof(int));
126         if(ret == 0) {
127             skynet_error(NULL, "set new value(%d) for (%s)
    succeed\n", *newval, name);
128         } else {
129             skynet_error(NULL, "set new value(%d) for (%s)
    failed: error -> %d\n", *newval, name, ret);
130         }
131     } else {
132         je_mallctl(name, &v, &len, NULL, 0);
133     }
134 
135     return v;
136 }
137 
138 // hook : malloc, realloc, free, calloc
139 
140 void *
141 skynet_malloc(size_t size) {
142     void* ptr = je_malloc(size + PREFIX_SIZE);
143     if(!ptr) malloc_oom(size);
144     return fill_prefix(ptr);
145 }
146 
147 void *
148 skynet_realloc(void *ptr, size_t size) {
149     if (ptr == NULL) return skynet_malloc(size);
150 
151     void* rawptr = clean_prefix(ptr);
152     void *newptr = je_realloc(rawptr, size+PREFIX_SIZE);
153     if(!newptr) malloc_oom(size);
154     return fill_prefix(newptr);
155 }
156 
157 void
158 skynet_free(void *ptr) {
```

```
159     if (ptr == NULL) return;
160     void* rawptr = clean_prefix(ptr);
161     je_free(rawptr);
162 }
163 
164 void *
165 skynet_calloc(size_t nmemb,size_t size) {
166     void* ptr = je_calloc(nmemb + ((PREFIX_SIZE+size-1)/size),
    size );
167     if(!ptr) malloc_oom(size);
168     return fill_prefix(ptr);
169 }
170 
171 #else
172 
173 // for skynet_lalloc use
174 #define raw_realloc realloc
175 #define raw_free free
176 
177 void
178 memory_info_dump(void) {
179     skynet_error(NULL, "No jemalloc");
180 }
181 
182 size_t
183 mallctl_int64(const char* name, size_t* newval) {
184     skynet_error(NULL, "No jemalloc : mallctl_int64 %s.",
    name);
185     return 0;
186 }
187 
188 int
189 mallctl_opt(const char* name, int* newval) {
190     skynet_error(NULL, "No jemalloc : mallctl_opt %s.", name);
191     return 0;
192 }
193 
194 #endif
195 
196 size_t
197 malloc_used_memory(void) {
198     return _used_memory;
```

```
199 }
200
201 size_t
202 malloc_memory_block(void) {
203     return _memory_block;
204 }
205
206 void
207 dump_c_mem() {
208     int i;
209     size_t total = 0;
210     skynet_error(NULL, "dump all service mem:");
211     for(i=0; i<SLOT_SIZE; i++) {
212         mem_data* data = &mem_stats[i];
213         if(data->handle != 0 && data->allocated != 0) {
214             total += data->allocated;
215             skynet_error(NULL, "0x%x -> %zdkb", data->handle,
    data->allocated >> 10);
216         }
217     }
218     skynet_error(NULL, "+total: %zdkb",total >> 10);
219 }
220
221 char *
222 skynet_strdup(const char *str) {
223     size_t sz = strlen(str);
224     char * ret = skynet_malloc(sz+1);
225     memcpy(ret, str, sz+1);
226     return ret;
227 }
228
229 void *
230 skynet_lalloc(void *ptr, size_t osize, size_t nsize) {
231     if (nsize == 0) {
232         raw_free(ptr);
233         return NULL;
234     } else {
235         return raw_realloc(ptr, nsize);
236     }
237 }
238
239 int
```

```
240 dump_mem_lua(lua_State *L) {
241     int i;
242     lua_newtable(L);
243     for(i=0; i<SLOT_SIZE; i++) {
244         mem_data* data = &mem_stats[i];
245         if(data->handle != 0 && data->allocated != 0) {
246             lua_pushinteger(L, data->allocated);
247             lua_rawseti(L, -2, (lua_Integer)data->handle);
248         }
249     }
250     return 1;
251 }
252 
253 size_t
254 malloc_current_memory(void) {
255     uint32_t handle = skynet_current_handle();
256     int i;
257     for(i=0; i<SLOT_SIZE; i++) {
258         mem_data* data = &mem_stats[i];
259         if(data->handle == (uint32_t)handle && data-
>allocated != 0) {
260             return (size_t) data->allocated;
261         }
262     }
263     return 0;
264 }
265 
266 void
267 skynet_debug_memory(const char *info) {
268     // for debug use
269     uint32_t handle = skynet_current_handle();
270     size_t mem = malloc_current_memory();
271     fprintf(stderr, "[:%08x] %s %p\n", handle, info, (void
*)mem);
272 }
```

z:\game\ff_server\skynet\skynet-src\malloc_hook.h

```h
#ifndef SKYNET_MALLOC_HOOK_H
#define SKYNET_MALLOC_HOOK_H

#include <stdlib.h>
#include <lua.h>

extern size_t malloc_used_memory(void);
extern size_t malloc_memory_block(void);
extern void   memory_info_dump(void);
extern size_t mallctl_int64(const char* name, size_t* newval);
extern int    mallctl_opt(const char* name, int* newval);
extern void   dump_c_mem(void);
extern int    dump_mem_lua(lua_State *L);
extern size_t malloc_current_memory(void);

#endif /* SKYNET_MALLOC_HOOK_H */
```

z:\game\ff_server\skynet\skynet-src\rwlock.h

```
1  #ifndef SKYNET_RWLOCK_H
2  #define SKYNET_RWLOCK_H
3
4  #ifndef USE_PTHREAD_LOCK
5
6  struct rwlock {
7      int write;
8      int read;
9  };
10
11 static inline void
12 rwlock_init(struct rwlock *lock) {
13     lock->write = 0;
14     lock->read = 0;
15 }
16
17 static inline void
18 rwlock_rlock(struct rwlock *lock) {
19     for (;;) {
20         while(lock->write) {
21             __sync_synchronize();
22         }
23         __sync_add_and_fetch(&lock->read,1);
24         if (lock->write) {
25             __sync_sub_and_fetch(&lock->read,1);
26         } else {
27             break;
28         }
29     }
30 }
31
32 static inline void
33 rwlock_wlock(struct rwlock *lock) {
34     while (__sync_lock_test_and_set(&lock->write,1)) {}
35     while(lock->read) {
36         __sync_synchronize();
37     }
38 }
39
40 static inline void
```

```
41 rwlock_wunlock(struct rwlock *lock) {
42     __sync_lock_release(&lock->write);
43 }
44 
45 static inline void
46 rwlock_runlock(struct rwlock *lock) {
47     __sync_sub_and_fetch(&lock->read,1);
48 }
49 
50 #else
51 
52 #include <pthread.h>
53 
54 // only for some platform doesn't have __sync_*
55 // todo: check the result of pthread api
56 
57 struct rwlock {
58     pthread_rwlock_t lock;
59 };
60 
61 static inline void
62 rwlock_init(struct rwlock *lock) {
63     pthread_rwlock_init(&lock->lock, NULL);
64 }
65 
66 static inline void
67 rwlock_rlock(struct rwlock *lock) {
68      pthread_rwlock_rdlock(&lock->lock);
69 }
70 
71 static inline void
72 rwlock_wlock(struct rwlock *lock) {
73      pthread_rwlock_wrlock(&lock->lock);
74 }
75 
76 static inline void
77 rwlock_wunlock(struct rwlock *lock) {
78     pthread_rwlock_unlock(&lock->lock);
79 }
80 
81 static inline void
82 rwlock_runlock(struct rwlock *lock) {
```

```
83      pthread_rwlock_unlock(&lock->lock);
84  }
85
86  #endif
87
88  #endif
```

z:\game\ff_server\skynet\skynet-src\skynet_daemon.c

```c
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/file.h>
#include <signal.h>
#include <errno.h>
#include <stdlib.h>

#include "skynet_daemon.h"

static int
check_pid(const char *pidfile) {
    int pid = 0;
    FILE *f = fopen(pidfile,"r");
    if (f == NULL)
        return 0;
    int n = fscanf(f,"%d", &pid);
    fclose(f);

    if (n !=1 || pid == 0 || pid == getpid()) {
        return 0;
    }

    if (kill(pid, 0) && errno == ESRCH)
        return 0;

    return pid;
}

static int
write_pid(const char *pidfile) {
    FILE *f;
    int pid = 0;
    int fd = open(pidfile, O_RDWR|O_CREAT, 0644);
    if (fd == -1) {
        fprintf(stderr, "Can't create %s.\n", pidfile);
        return 0;
    }
    f = fdopen(fd, "r+");
```

```
40      if (f == NULL) {
41          fprintf(stderr, "Can't open %s.\n", pidfile);
42          return 0;
43      }
44
45      if (flock(fd, LOCK_EX|LOCK_NB) == -1) {
46          int n = fscanf(f, "%d", &pid);
47          fclose(f);
48          if (n != 1) {
49              fprintf(stderr, "Can't lock and read pidfile.\n");
50          } else {
51              fprintf(stderr, "Can't lock pidfile, lock is held by
    pid %d.\n", pid);
52          }
53          return 0;
54      }
55
56      pid = getpid();
57      if (!fprintf(f,"%d\n", pid)) {
58          fprintf(stderr, "Can't write pid.\n");
59          close(fd);
60          return 0;
61      }
62      fflush(f);
63
64      return pid;
65  }
66
67  int
68  daemon_init(const char *pidfile) {
69      int pid = check_pid(pidfile);
70
71      if (pid) {
72          fprintf(stderr, "Skynet is already running, pid =
    %d.\n", pid);
73          return 1;
74      }
75
76  #ifdef __APPLE__
77      fprintf(stderr, "'daemon' is deprecated: first deprecated in
    OS X 10.5 , use launchd instead.\n");
78  #else
```

```
    if (daemon(1,0)) {
        fprintf(stderr, "Can't daemonize.\n");
        return 1;
    }
#endif

    pid = write_pid(pidfile);
    if (pid == 0) {
        return 1;
    }

    return 0;
}

int
daemon_exit(const char *pidfile) {
    return unlink(pidfile);
}
```

z:\game\ff_server\skynet\skynet-src\skynet_daemon.h

```
#ifndef skynet_daemon_h
#define skynet_daemon_h

int daemon_init(const char *pidfile);
int daemon_exit(const char *pidfile);

#endif
```

z:\game\ff_server\skynet\skynet-src\skynet_env.c

```c
#include "skynet.h"
#include "skynet_env.h"
#include "spinlock.h"

#include <lua.h>
#include <lauxlib.h>

#include <stdlib.h>
#include <assert.h>

struct skynet_env {
    struct spinlock lock;
    lua_State *L;
};

static struct skynet_env *E = NULL;

const char *
skynet_getenv(const char *key) {
    SPIN_LOCK(E)

    lua_State *L = E->L;

    lua_getglobal(L, key);
    const char * result = lua_tostring(L, -1);
    lua_pop(L, 1);

    SPIN_UNLOCK(E)

    return result;
}

void
skynet_setenv(const char *key, const char *value) {
    SPIN_LOCK(E)

    lua_State *L = E->L;
    lua_getglobal(L, key);
    assert(lua_isnil(L, -1));
```

```
40      lua_pop(L,1);
41      lua_pushstring(L,value);
42      lua_setglobal(L,key);
43
44      SPIN_UNLOCK(E)
45  }
46
47  void
48  skynet_env_init() {
49      E = skynet_malloc(sizeof(*E));
50      SPIN_INIT(E)
51      E->L = luaL_newstate();
52  }
```

z:\game\ff_server\skynet\skynet-src\skynet_env.h

```
1  #ifndef SKYNET_ENV_H
2  #define SKYNET_ENV_H
3
4  const char * skynet_getenv(const char *key);
5  void skynet_setenv(const char *key, const char *value);
6
7  void skynet_env_init();
8
9  #endif
```

z:\game\ff_server\skynet\skynet-src\skynet_error.c

```c
#include "skynet.h"
#include "skynet_handle.h"
#include "skynet_mq.h"
#include "skynet_server.h"

#include <stdarg.h>
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

#define LOG_MESSAGE_SIZE 256

void
skynet_error(struct skynet_context * context, const char *msg, ...) {
    static uint32_t logger = 0;
    if (logger == 0) {
        logger = skynet_handle_findname("logger");
    }
    if (logger == 0) {
        return;
    }

    char tmp[LOG_MESSAGE_SIZE];
    char *data = NULL;

    va_list ap;

    va_start(ap,msg);
    int len = vsnprintf(tmp, LOG_MESSAGE_SIZE, msg, ap);
    va_end(ap);
    if (len >=0 && len < LOG_MESSAGE_SIZE) {
        data = skynet_strdup(tmp);
    } else {
        int max_size = LOG_MESSAGE_SIZE;
        for (;;) {
            max_size *= 2;
            data = skynet_malloc(max_size);
            va_start(ap,msg);
```

```
39              len = vsnprintf(data, max_size, msg, ap);
40              va_end(ap);
41              if (len < max_size) {
42                  break;
43              }
44              skynet_free(data);
45          }
46      }
47      if (len < 0) {
48          skynet_free(data);
49          perror("vsnprintf error :");
50          return;
51      }
52
53
54      struct skynet_message smsg;
55      if (context == NULL) {
56          smsg.source = 0;
57      } else {
58          smsg.source = skynet_context_handle(context);
59      }
60      smsg.session = 0;
61      smsg.data = data;
62      smsg.sz = len | ((size_t)PTYPE_TEXT << MESSAGE_TYPE_SHIFT);
63      skynet_context_push(logger, &smsg);
64  }
```

z:\game\ff_server\skynet\skynet-src\skynet_handle.c

```c
#include "skynet.h"

#include "skynet_handle.h"
#include "skynet_server.h"
#include "rwlock.h"

#include <stdlib.h>
#include <assert.h>
#include <string.h>

#define DEFAULT_SLOT_SIZE 4
#define MAX_SLOT_SIZE 0x40000000

struct handle_name {
    char * name;
    uint32_t handle;
};

struct handle_storage {
    struct rwlock lock;

    uint32_t harbor;
    uint32_t handle_index;
    int slot_size;
    struct skynet_context ** slot;

    int name_cap;
    int name_count;
    struct handle_name *name;
};

static struct handle_storage *H = NULL;

uint32_t
skynet_handle_register(struct skynet_context *ctx) {
    struct handle_storage *s = H;

    rwlock_wlock(&s->lock);

```

```
40      for (;;) {
41          int i;
42          for (i=0;i<s->slot_size;i++) {
43              uint32_t handle = (i+s->handle_index) &
HANDLE_MASK;
44              int hash = handle & (s->slot_size-1);
45              if (s->slot[hash] == NULL) {
46                  s->slot[hash] = ctx;
47                  s->handle_index = handle + 1;
48
49                  rwlock_wunlock(&s->lock);
50
51                  handle |= s->harbor;
52                  return handle;
53              }
54          }
55          assert((s->slot_size*2 - 1) <= HANDLE_MASK);
56          struct skynet_context ** new_slot =
skynet_malloc(s->slot_size * 2 * sizeof(struct skynet_context
*));
57          memset(new_slot, 0, s->slot_size * 2 * sizeof(struct
skynet_context *));
58          for (i=0;i<s->slot_size;i++) {
59              int hash = skynet_context_handle(s->slot[i]) & (s-
>slot_size * 2 - 1);
60              assert(new_slot[hash] == NULL);
61              new_slot[hash] = s->slot[i];
62          }
63          skynet_free(s->slot);
64          s->slot = new_slot;
65          s->slot_size *= 2;
66      }
67  }
68
69  int
70  skynet_handle_retire(uint32_t handle) {
71      int ret = 0;
72      struct handle_storage *s = H;
73
74      rwlock_wlock(&s->lock);
75
76      uint32_t hash = handle & (s->slot_size-1);
77      struct skynet_context * ctx = s->slot[hash];
```

```
78
79      if (ctx != NULL && skynet_context_handle(ctx) == handle) {
80          s->slot[hash] = NULL;
81          ret = 1;
82          int i;
83          int j=0, n=s->name_count;
84          for (i=0; i<n; ++i) {
85              if (s->name[i].handle == handle) {
86                  skynet_free(s->name[i].name);
87                  continue;
88              } else if (i!=j) {
89                  s->name[j] = s->name[i];
90              }
91              ++j;
92          }
93          s->name_count = j;
94      } else {
95          ctx = NULL;
96      }
97
98      rwlock_wunlock(&s->lock);
99
100     if (ctx) {
101         // release ctx may call skynet_handle_* , so wunlock
    first.
102         skynet_context_release(ctx);
103     }
104
105     return ret;
106 }
107
108 void
109 skynet_handle_retireall() {
110     struct handle_storage *s = H;
111     for (;;) {
112         int n=0;
113         int i;
114         for (i=0;i<s->slot_size;i++) {
115             rwlock_rlock(&s->lock);
116             struct skynet_context * ctx = s->slot[i];
117             uint32_t handle = 0;
118             if (ctx)
```

```
119                 handle = skynet_context_handle(ctx);
120             rwlock_runlock(&s->lock);
121             if (handle != 0) {
122                 if (skynet_handle_retire(handle)) {
123                     ++n;
124                 }
125             }
126         }
127         if (n==0)
128             return;
129     }
130 }
131 
132 struct skynet_context *
133 skynet_handle_grab(uint32_t handle) {
134     struct handle_storage *s = H;
135     struct skynet_context * result = NULL;
136 
137     rwlock_rlock(&s->lock);
138 
139     uint32_t hash = handle & (s->slot_size-1);
140     struct skynet_context * ctx = s->slot[hash];
141     if (ctx && skynet_context_handle(ctx) == handle) {
142         result = ctx;
143         skynet_context_grab(result);
144     }
145 
146     rwlock_runlock(&s->lock);
147 
148     return result;
149 }
150 
151 uint32_t
152 skynet_handle_findname(const char * name) {
153     struct handle_storage *s = H;
154 
155     rwlock_rlock(&s->lock);
156 
157     uint32_t handle = 0;
158 
159     int begin = 0;
160     int end = s->name_count - 1;
```

```
161     while (begin<=end) {
162         int mid = (begin+end)/2;
163         struct handle_name *n = &s->name[mid];
164         int c = strcmp(n->name, name);
165         if (c==0) {
166             handle = n->handle;
167             break;
168         }
169         if (c<0) {
170             begin = mid + 1;
171         } else {
172             end = mid - 1;
173         }
174     }
175 
176     rwlock_runlock(&s->lock);
177 
178     return handle;
179 }
180 
181 static void
182 _insert_name_before(struct handle_storage *s, char *name,
    uint32_t handle, int before) {
183     if (s->name_count >= s->name_cap) {
184         s->name_cap *= 2;
185         assert(s->name_cap <= MAX_SLOT_SIZE);
186         struct handle_name * n = skynet_malloc(s->name_cap *
    sizeof(struct handle_name));
187         int i;
188         for (i=0;i<before;i++) {
189             n[i] = s->name[i];
190         }
191         for (i=before;i<s->name_count;i++) {
192             n[i+1] = s->name[i];
193         }
194         skynet_free(s->name);
195         s->name = n;
196     } else {
197         int i;
198         for (i=s->name_count;i>before;i--) {
199             s->name[i] = s->name[i-1];
200         }
```

```
201     }
202     s->name[before].name = name;
203     s->name[before].handle = handle;
204     s->name_count ++;
205 }
206
207 static const char *
208 _insert_name(struct handle_storage *s, const char * name,
    uint32_t handle) {
209     int begin = 0;
210     int end = s->name_count - 1;
211     while (begin<=end) {
212         int mid = (begin+end)/2;
213         struct handle_name *n = &s->name[mid];
214         int c = strcmp(n->name, name);
215         if (c==0) {
216             return NULL;
217         }
218         if (c<0) {
219             begin = mid + 1;
220         } else {
221             end = mid - 1;
222         }
223     }
224     char * result = skynet_strdup(name);
225
226     _insert_name_before(s, result, handle, begin);
227
228     return result;
229 }
230
231 const char *
232 skynet_handle_namehandle(uint32_t handle, const char
    *name) {
233     rwlock_wlock(&H->lock);
234
235     const char * ret = _insert_name(H, name, handle);
236
237     rwlock_wunlock(&H->lock);
238
239     return ret;
240 }
```

```
241
242 void
243 skynet_handle_init(int harbor) {
244     assert(H==NULL);
245     struct handle_storage * s = skynet_malloc(sizeof(*H));
246     s->slot_size = DEFAULT_SLOT_SIZE;
247     s->slot = skynet_malloc(s->slot_size * sizeof(struct
    skynet_context *));
248     memset(s->slot, 0, s->slot_size * sizeof(struct
    skynet_context *));
249
250     rwlock_init(&s->lock);
251     // reserve 0 for system
252     s->harbor = (uint32_t) (harbor & 0xff) <<
    HANDLE_REMOTE_SHIFT;
253     s->handle_index = 1;
254     s->name_cap = 2;
255     s->name_count = 0;
256     s->name = skynet_malloc(s->name_cap * sizeof(struct
    handle_name));
257
258     H = s;
259
260     // Don't need to free H
261 }
```

z:\game\ff_server\skynet\skynet-src\skynet_handle.h

```
#ifndef SKYNET_CONTEXT_HANDLE_H
#define SKYNET_CONTEXT_HANDLE_H

#include <stdint.h>

// reserve high 8 bits for remote id
#define HANDLE_MASK 0xffffff
#define HANDLE_REMOTE_SHIFT 24

struct skynet_context;

uint32_t skynet_handle_register(struct skynet_context *);
int skynet_handle_retire(uint32_t handle);
struct skynet_context * skynet_handle_grab(uint32_t handle);
void skynet_handle_retireall();

uint32_t skynet_handle_findname(const char * name);
const char * skynet_handle_namehandle(uint32_t handle,
const char *name);

void skynet_handle_init(int harbor);

#endif
```

z:\game\ff_server\skynet\skynet-src\skynet_harbor.c

```c
#include "skynet.h"
#include "skynet_harbor.h"
#include "skynet_server.h"
#include "skynet_mq.h"
#include "skynet_handle.h"

#include <string.h>
#include <stdio.h>
#include <assert.h>

static struct skynet_context * REMOTE = 0;
static unsigned int HARBOR = ~0;

void
skynet_harbor_send(struct remote_message *rmsg, uint32_t source, int session) {
    int type = rmsg->sz >> MESSAGE_TYPE_SHIFT;
    rmsg->sz &= MESSAGE_TYPE_MASK;
    assert(type != PTYPE_SYSTEM && type != PTYPE_HARBOR && REMOTE);
    skynet_context_send(REMOTE, rmsg, sizeof(*rmsg) , source, type , session);
}

int
skynet_harbor_message_isremote(uint32_t handle) {
    assert(HARBOR != ~0);
    int h = (handle & ~HANDLE_MASK);
    return h != HARBOR && h !=0;
}

void
skynet_harbor_init(int harbor) {
    HARBOR = (unsigned int)harbor << HANDLE_REMOTE_SHIFT;
}

void
skynet_harbor_start(void *ctx) {
    // the HARBOR must be reserved to ensure the pointer is
```

```
valid.
    // It will be released at last by calling skynet_harbor_exit
    skynet_context_reserve(ctx);
    REMOTE = ctx;
}

void
skynet_harbor_exit() {
    struct skynet_context * ctx = REMOTE;
    REMOTE= NULL;
    if (ctx) {
        skynet_context_release(ctx);
    }
}
```

z:\game\ff_server\skynet\skynet-src\skynet_harbor.h

```h
#ifndef SKYNET_HARBOR_H
#define SKYNET_HARBOR_H

#include <stdint.h>
#include <stdlib.h>

#define GLOBALNAME_LENGTH 16
#define REMOTE_MAX 256

struct remote_name {
    char name[GLOBALNAME_LENGTH];
    uint32_t handle;
};

struct remote_message {
    struct remote_name destination;
    const void * message;
    size_t sz;
};

void skynet_harbor_send(struct remote_message *rmsg, uint32_t source, int session);
int skynet_harbor_message_isremote(uint32_t handle);
void skynet_harbor_init(int harbor);
void skynet_harbor_start(void * ctx);
void skynet_harbor_exit();

#endif
```

z:\game\ff_server\skynet\skynet-src\skynet_imp.h

```
#ifndef SKYNET_IMP_H
#define SKYNET_IMP_H

struct skynet_config {
    int thread;
    int harbor;
    const char * daemon;
    const char * module_path;
    const char * bootstrap;
    const char * logger;
    const char * logservice;
};

#define THREAD_WORKER 0
#define THREAD_MAIN 1
#define THREAD_SOCKET 2
#define THREAD_TIMER 3
#define THREAD_MONITOR 4

void skynet_start(struct skynet_config * config);

#endif
```

z:\game\ff_server\skynet\skynet-src\skynet_log.c

```c
#include "skynet_log.h"
#include "skynet_timer.h"
#include "skynet.h"
#include "skynet_socket.h"
#include <string.h>
#include <time.h>

FILE *
skynet_log_open(struct skynet_context * ctx, uint32_t handle) {
    const char * logpath = skynet_getenv("logpath");
    if (logpath == NULL)
        return NULL;
    size_t sz = strlen(logpath);
    char tmp[sz + 16];
    sprintf(tmp, "%s/%08x.log", logpath, handle);
    FILE *f = fopen(tmp, "ab");
    if (f) {
        uint32_t starttime = skynet_starttime();
        uint64_t currenttime = skynet_now();
        time_t ti = starttime + currenttime/100;
        skynet_error(ctx, "Open log file %s", tmp);
        fprintf(f, "open time: %u %s", (uint32_t)currenttime, ctime(&ti));
        fflush(f);
    } else {
        skynet_error(ctx, "Open log file %s fail", tmp);
    }
    return f;
}

void
skynet_log_close(struct skynet_context * ctx, FILE *f, uint32_t handle) {
    skynet_error(ctx, "Close log file :%08x", handle);
    fprintf(f, "close time: %u\n", (uint32_t)skynet_now());
    fclose(f);
}

```

```
37 static void
38 log_blob(FILE *f, void * buffer, size_t sz) {
39     size_t i;
40     uint8_t * buf = buffer;
41     for (i=0;i!=sz;i++) {
42         fprintf(f, "%02x", buf[i]);
43     }
44 }
45
46 static void
47 log_socket(FILE * f, struct skynet_socket_message * message,
   size_t sz) {
48     fprintf(f, "[socket] %d %d %d ", message->type, message->id,
   message->ud);
49
50     if (message->buffer == NULL) {
51         const char *buffer = (const char *)(message + 1);
52         sz -= sizeof(*message);
53         const char * eol = memchr(buffer, '\0', sz);
54         if (eol) {
55             sz = eol - buffer;
56         }
57         fprintf(f, "[%*s]", (int)sz, (const char *)buffer);
58     } else {
59         sz = message->ud;
60         log_blob(f, message->buffer, sz);
61     }
62     fprintf(f, "\n");
63     fflush(f);
64 }
65
66 void
67 skynet_log_output(FILE *f, uint32_t source, int type, int
   session, void * buffer, size_t sz) {
68     if (type == PTYPE_SOCKET) {
69         log_socket(f, buffer, sz);
70     } else {
71         uint32_t ti = (uint32_t)skynet_now();
72         fprintf(f, ":%08x %d %d %u ", source, type, session,
   ti);
73         log_blob(f, buffer, sz);
74         fprintf(f,"\n");
75         fflush(f);
```

```
76     }
77 }
```

z:\game\ff_server\skynet\skynet-src\skynet_log.h

```
1  #ifndef skynet_log_h
2  #define skynet_log_h
3
4  #include "skynet_env.h"
5  #include "skynet.h"
6
7  #include <stdio.h>
8  #include <stdint.h>
9
10 FILE * skynet_log_open(struct skynet_context * ctx, uint32_t handle);
11 void skynet_log_close(struct skynet_context * ctx, FILE *f, uint32_t handle);
12 void skynet_log_output(FILE *f, uint32_t source, int type, int session, void * buffer, size_t sz);
13
14 #endif
```

z:\game\ff_server\skynet\skynet-src\skynet_main.c

```c
#include "skynet.h"

#include "skynet_imp.h"
#include "skynet_env.h"
#include "skynet_server.h"
#include "luashrtbl.h"

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <lua.h>
#include <lualib.h>
#include <lauxlib.h>
#include <signal.h>
#include <assert.h>

static int
optint(const char *key, int opt) {
    const char * str = skynet_getenv(key);
    if (str == NULL) {
        char tmp[20];
        sprintf(tmp,"%d",opt);
        skynet_setenv(key, tmp);
        return opt;
    }
    return strtol(str, NULL, 10);
}

/*
static int
optboolean(const char *key, int opt) {
    const char * str = skynet_getenv(key);
    if (str == NULL) {
        skynet_setenv(key, opt ? "true" : "false");
        return opt;
    }
    return strcmp(str,"true")==0;
}
*/
```

```
40
41 static const char *
42 optstring(const char *key,const char * opt) {
43     const char * str = skynet_getenv(key);
44     if (str == NULL) {
45         if (opt) {
46             skynet_setenv(key, opt);
47             opt = skynet_getenv(key);
48         }
49         return opt;
50     }
51     return str;
52 }
53
54 static void
55 _init_env(lua_State *L) {
56     lua_pushnil(L);  /* first key */
57     while (lua_next(L, -2) != 0) {
58         int keyt = lua_type(L, -2);
59         if (keyt != LUA_TSTRING) {
60             fprintf(stderr, "Invalid config table\n");
61             exit(1);
62         }
63         const char * key = lua_tostring(L,-2);
64         if (lua_type(L,-1) == LUA_TBOOLEAN) {
65             int b = lua_toboolean(L,-1);
66             skynet_setenv(key,b ? "true" : "false" );
67         } else {
68             const char * value = lua_tostring(L,-1);
69             if (value == NULL) {
70                 fprintf(stderr, "Invalid config table key =
   %s\n", key);
71                 exit(1);
72             }
73             skynet_setenv(key,value);
74         }
75         lua_pop(L,1);
76     }
77     lua_pop(L,1);
78 }
79
80 int sigign() {
```

```
81      struct sigaction sa;
82      sa.sa_handler = SIG_IGN;
83      sigaction(SIGPIPE, &sa, 0);
84      return 0;
85  }
86
87  static const char * load_config = "\
88      local config_name = ...\
89      local f = assert(io.open(config_name))\
90      local code = assert(f:read \'*a\')\
91      local function getenv(name) return assert(os.getenv(name),
    \'os.getenv() failed: \' .. name) end\
92      code = string.gsub(code, \'%$([%w_%d]+)\', getenv)\
93      f:close()\
94      local result = {}\
95      assert(load(code,\'=(load)\',\'t\',result))()\
96      return result\
97  ";
98
99  int
100 main(int argc, char *argv[]) {
101     const char * config_file = NULL ;
102     if (argc > 1) {
103         config_file = argv[1];
104     } else {
105         fprintf(stderr, "Need a config file. Please read skynet
    wiki : https://github.com/cloudwu/skynet/wiki/Config\n"
106             "usage: skynet configfilename\n");
107         return 1;
108     }
109
110     luaS_initshr();
111     skynet_globalinit();
112     skynet_env_init();
113
114     sigign();
115
116     struct skynet_config config;
117
118     struct lua_State *L = luaL_newstate();
119     luaL_openlibs(L);   // link lua lib
120
```

```
121     int err = luaL_loadstring(L, load_config);
122     assert(err == LUA_OK);
123     lua_pushstring(L, config_file);
124 
125     err = lua_pcall(L, 1, 1, 0);
126     if (err) {
127         fprintf(stderr,"%s\n",lua_tostring(L,-1));
128         lua_close(L);
129         return 1;
130     }
131     _init_env(L);
132 
133     config.thread =  optint("thread",8);
134     config.module_path = optstring("cpath","./cservice/?.so");
135     config.harbor = optint("harbor", 1);
136     config.bootstrap = optstring("bootstrap","snlua
    bootstrap");
137     config.daemon = optstring("daemon", NULL);
138     config.logger = optstring("logger", NULL);
139     config.logservice = optstring("logservice", "logger");
140 
141     lua_close(L);
142 
143     skynet_start(&config);
144     skynet_globalexit();
145     luaS_exitshr();
146 
147     return 0;
148 }
```

z:\game\ff_server\skynet\skynet-src\skynet_malloc.h

```h
#ifndef skynet_malloc_h
#define skynet_malloc_h

#include <stddef.h>

#define skynet_malloc malloc
#define skynet_calloc calloc
#define skynet_realloc realloc
#define skynet_free free

void * skynet_malloc(size_t sz);
void * skynet_calloc(size_t nmemb,size_t size);
void * skynet_realloc(void *ptr, size_t size);
void skynet_free(void *ptr);
char * skynet_strdup(const char *str);
void * skynet_lalloc(void *ptr, size_t osize, size_t nsize); // use for lua

#endif
```

z:\game\ff_server\skynet\skynet-src\skynet_module.c

```c
#include "skynet.h"

#include "skynet_module.h"
#include "spinlock.h"

#include <assert.h>
#include <string.h>
#include <dlfcn.h>
#include <stdlib.h>
#include <stdint.h>
#include <stdio.h>

#define MAX_MODULE_TYPE 32

struct modules {
    int count;
    struct spinlock lock;
    const char * path;
    struct skynet_module m[MAX_MODULE_TYPE];
};

static struct modules * M = NULL;

static void *
_try_open(struct modules *m, const char * name) {
    const char *l;
    const char * path = m->path;
    size_t path_size = strlen(path);
    size_t name_size = strlen(name);

    int sz = path_size + name_size;
    //search path
    void * dl = NULL;
    char tmp[sz];
    do
    {
        memset(tmp,0,sz);
        while (*path == ';') path++;
        if (*path == '\0') break;
```

```
40          l = strchr(path, ';');
41          if (l == NULL) l = path + strlen(path);
42          int len = l - path;
43          int i;
44          for (i=0;path[i]!='?' && i < len ;i++) {
45              tmp[i] = path[i];
46          }
47          memcpy(tmp+i,name,name_size);
48          if (path[i] == '?') {
49              strncpy(tmp+i+name_size,path+i+1,len - i - 1);
50          } else {
51              fprintf(stderr,"Invalid C service path\n");
52              exit(1);
53          }
54          dl = dlopen(tmp, RTLD_NOW | RTLD_GLOBAL);
55          path = l;
56      }while(dl == NULL);
57
58      if (dl == NULL) {
59          fprintf(stderr, "try open %s failed :
    %s\n",name,dlerror());
60      }
61
62      return dl;
63  }
64
65  static struct skynet_module *
66  _query(const char * name) {
67      int i;
68      for (i=0;i<M->count;i++) {
69          if (strcmp(M->m[i].name,name)==0) {
70              return &M->m[i];
71          }
72      }
73      return NULL;
74  }
75
76  static int
77  _open_sym(struct skynet_module *mod) {
78      size_t name_size = strlen(mod->name);
79      char tmp[name_size + 9]; // create/init/release/signal ,
    longest name is release (7)
```

```
80      memcpy(tmp, mod->name, name_size);
81      strcpy(tmp+name_size, "_create");
82      mod->create = dlsym(mod->module, tmp);
83      strcpy(tmp+name_size, "_init");
84      mod->init = dlsym(mod->module, tmp);
85      strcpy(tmp+name_size, "_release");
86      mod->release = dlsym(mod->module, tmp);
87      strcpy(tmp+name_size, "_signal");
88      mod->signal = dlsym(mod->module, tmp);
89
90      return mod->init == NULL;
91  }
92
93  struct skynet_module *
94  skynet_module_query(const char * name) {
95      struct skynet_module * result = _query(name);
96      if (result)
97          return result;
98
99      SPIN_LOCK(M)
100
101     result = _query(name); // double check
102
103     if (result == NULL && M->count < MAX_MODULE_TYPE) {
104         int index = M->count;
105         void * dl = _try_open(M,name);
106         if (dl) {
107             M->m[index].name = name;
108             M->m[index].module = dl;
109
110             if (_open_sym(&M->m[index]) == 0) {
111                 M->m[index].name = skynet_strdup(name);
112                 M->count ++;
113                 result = &M->m[index];
114             }
115         }
116     }
117
118     SPIN_UNLOCK(M)
119
120     return result;
121 }
```

```
122
123 void
124 skynet_module_insert(struct skynet_module *mod) {
125     SPIN_LOCK(M)
126
127     struct skynet_module * m = _query(mod->name);
128     assert(m == NULL && M->count < MAX_MODULE_TYPE);
129     int index = M->count;
130     M->m[index] = *mod;
131     ++M->count;
132
133     SPIN_UNLOCK(M)
134 }
135
136 void *
137 skynet_module_instance_create(struct skynet_module *m) {
138     if (m->create) {
139         return m->create();
140     } else {
141         return (void *)(intptr_t)(~0);
142     }
143 }
144
145 int
146 skynet_module_instance_init(struct skynet_module *m, void
    * inst, struct skynet_context *ctx, const char * parm) {
147     return m->init(inst, ctx, parm);
148 }
149
150 void
151 skynet_module_instance_release(struct skynet_module *m,
    void *inst) {
152     if (m->release) {
153         m->release(inst);
154     }
155 }
156
157 void
158 skynet_module_instance_signal(struct skynet_module *m,
    void *inst, int signal) {
159     if (m->signal) {
160         m->signal(inst, signal);
```

```
161     }
162 }
163
164 void
165 skynet_module_init(const char *path) {
166     struct modules *m = skynet_malloc(sizeof(*m));
167     m->count = 0;
168     m->path = skynet_strdup(path);
169
170     SPIN_INIT(m)
171
172     M = m;
173 }
```

z:\game\ff_server\skynet\skynet-src\skynet_module.h

```h
#ifndef SKYNET_MODULE_H
#define SKYNET_MODULE_H

struct skynet_context;

typedef void * (*skynet_dl_create)(void);
typedef int (*skynet_dl_init)(void * inst, struct skynet_context *, const char * parm);
typedef void (*skynet_dl_release)(void * inst);
typedef void (*skynet_dl_signal)(void * inst, int signal);

struct skynet_module {
    const char * name;
    void * module;
    skynet_dl_create create;
    skynet_dl_init init;
    skynet_dl_release release;
    skynet_dl_signal signal;
};

void skynet_module_insert(struct skynet_module *mod);
struct skynet_module * skynet_module_query(const char * name);
void * skynet_module_instance_create(struct skynet_module *);
int skynet_module_instance_init(struct skynet_module *, void * inst, struct skynet_context *ctx, const char * parm);
void skynet_module_instance_release(struct skynet_module *, void *inst);
void skynet_module_instance_signal(struct skynet_module *, void *inst, int signal);

void skynet_module_init(const char *path);

#endif
```

z:\game\ff_server\skynet\skynet-src\skynet_monitor.c

```c
#include "skynet.h"

#include "skynet_monitor.h"
#include "skynet_server.h"
#include "skynet.h"
#include "atomic.h"

#include <stdlib.h>
#include <string.h>

struct skynet_monitor {
    int version;
    int check_version;
    uint32_t source;
    uint32_t destination;
};

struct skynet_monitor *
skynet_monitor_new() {
    struct skynet_monitor * ret = skynet_malloc(sizeof(*ret));
    memset(ret, 0, sizeof(*ret));
    return ret;
}

void
skynet_monitor_delete(struct skynet_monitor *sm) {
    skynet_free(sm);
}

void
skynet_monitor_trigger(struct skynet_monitor *sm, uint32_t source, uint32_t destination) {
    sm->source = source;
    sm->destination = destination;
    ATOM_INC(&sm->version);
}

void
```

```
38 skynet_monitor_check(struct skynet_monitor *sm) {
39     if (sm->version == sm->check_version) {
40         if (sm->destination) {
41             skynet_context_endless(sm->destination);
42             skynet_error(NULL, "A message from [ :%08x ] to [ :%08x ] maybe in an endless loop (version = %d)", sm->source , sm->destination, sm->version);
43         }
44     } else {
45         sm->check_version = sm->version;
46     }
47 }
```

z:\game\ff_server\skynet\skynet-src\skynet_monitor.h

```
1  #ifndef SKYNET_MONITOR_H
2  #define SKYNET_MONITOR_H
3
4  #include <stdint.h>
5
6  struct skynet_monitor;
7
8  struct skynet_monitor * skynet_monitor_new();
9  void skynet_monitor_delete(struct skynet_monitor *);
10 void skynet_monitor_trigger(struct skynet_monitor *, uint32_t source, uint32_t destination);
11 void skynet_monitor_check(struct skynet_monitor *);
12
13 #endif
```

z:\game\ff_server\skynet\skynet-src\skynet_mq.c

```c
#include "skynet.h"
#include "skynet_mq.h"
#include "skynet_handle.h"
#include "spinlock.h"

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <assert.h>
#include <stdbool.h>

#define DEFAULT_QUEUE_SIZE 64
#define MAX_GLOBAL_MQ 0x10000

// 0 means mq is not in global mq.
// 1 means mq is in global mq , or the message is dispatching.

#define MQ_IN_GLOBAL 1
#define MQ_OVERLOAD 1024

struct message_queue {
    struct spinlock lock;
    uint32_t handle;
    int cap;
    int head;
    int tail;
    int release;
    int in_global;
    int overload;
    int overload_threshold;
    struct skynet_message *queue;
    struct message_queue *next;
};

struct global_queue {
    struct message_queue *head;
    struct message_queue *tail;
    struct spinlock lock;
};

```

```
41 static struct global_queue *Q = NULL;
42
43 void
44 skynet_globalmq_push(struct message_queue * queue) {
45     struct global_queue *q= Q;
46
47     SPIN_LOCK(q)
48     assert(queue->next == NULL);
49     if(q->tail) {
50         q->tail->next = queue;
51         q->tail = queue;
52     } else {
53         q->head = q->tail = queue;
54     }
55     SPIN_UNLOCK(q)
56 }
57
58 struct message_queue *
59 skynet_globalmq_pop() {
60     struct global_queue *q = Q;
61
62     SPIN_LOCK(q)
63     struct message_queue *mq = q->head;
64     if(mq) {
65         q->head = mq->next;
66         if(q->head == NULL) {
67             assert(mq == q->tail);
68             q->tail = NULL;
69         }
70         mq->next = NULL;
71     }
72     SPIN_UNLOCK(q)
73
74     return mq;
75 }
76
77 struct message_queue *
78 skynet_mq_create(uint32_t handle) {
79     struct message_queue *q = skynet_malloc(sizeof(*q));
80     q->handle = handle;
81     q->cap = DEFAULT_QUEUE_SIZE;
82     q->head = 0;
```

```
83      q->tail = 0;
84      SPIN_INIT(q)
85      // When the queue is create (always between service create
    and service init) ,
86      // set in_global flag to avoid push it to global queue .
87      // If the service init success, skynet_context_new will
    call skynet_mq_push to push it to global queue.
88      q->in_global = MQ_IN_GLOBAL;
89      q->release = 0;
90      q->overload = 0;
91      q->overload_threshold = MQ_OVERLOAD;
92      q->queue = skynet_malloc(sizeof(struct skynet_message) *
    q->cap);
93      q->next = NULL;
94
95      return q;
96  }
97
98  static void
99  _release(struct message_queue *q) {
100     assert(q->next == NULL);
101     SPIN_DESTROY(q)
102     skynet_free(q->queue);
103     skynet_free(q);
104 }
105
106 uint32_t
107 skynet_mq_handle(struct message_queue *q) {
108     return q->handle;
109 }
110
111 int
112 skynet_mq_length(struct message_queue *q) {
113     int head, tail,cap;
114
115     SPIN_LOCK(q)
116     head = q->head;
117     tail = q->tail;
118     cap = q->cap;
119     SPIN_UNLOCK(q)
120
121     if (head <= tail) {
```

```
122         return tail - head;
123     }
124     return tail + cap - head;
125 }
126 
127 int
128 skynet_mq_overload(struct message_queue *q) {
129     if (q->overload) {
130         int overload = q->overload;
131         q->overload = 0;
132         return overload;
133     }
134     return 0;
135 }
136 
137 int
138 skynet_mq_pop(struct message_queue *q, struct skynet_message
    *message) {
139     int ret = 1;
140     SPIN_LOCK(q)
141 
142     if (q->head != q->tail) {
143         *message = q->queue[q->head++];
144         ret = 0;
145         int head = q->head;
146         int tail = q->tail;
147         int cap = q->cap;
148 
149         if (head >= cap) {
150             q->head = head = 0;
151         }
152         int length = tail - head;
153         if (length < 0) {
154             length += cap;
155         }
156         while (length > q->overload_threshold) {
157             q->overload = length;
158             q->overload_threshold *= 2;
159         }
160     } else {
161         // reset overload_threshold when queue is empty
162         q->overload_threshold = MQ_OVERLOAD;
```

```
163     }
164
165     if (ret) {
166         q->in_global = 0;
167     }
168
169     SPIN_UNLOCK(q)
170
171     return ret;
172 }
173
174 static void
175 expand_queue(struct message_queue *q) {
176     struct skynet_message *new_queue =
    skynet_malloc(sizeof(struct skynet_message) * q->cap * 2);
177     int i;
178     for (i=0;i<q->cap;i++) {
179         new_queue[i] = q->queue[(q->head + i) % q->cap];
180     }
181     q->head = 0;
182     q->tail = q->cap;
183     q->cap *= 2;
184
185     skynet_free(q->queue);
186     q->queue = new_queue;
187 }
188
189 void
190 skynet_mq_push(struct message_queue *q, struct skynet_message
    *message) {
191     assert(message);
192     SPIN_LOCK(q)
193
194     q->queue[q->tail] = *message;
195     if (++ q->tail >= q->cap) {
196         q->tail = 0;
197     }
198
199     if (q->head == q->tail) {
200         expand_queue(q);
201     }
202
```

```
203     if (q->in_global == 0) {
204         q->in_global = MQ_IN_GLOBAL;
205         skynet_globalmq_push(q);
206     }
207 
208     SPIN_UNLOCK(q)
209 }
210 
211 void
212 skynet_mq_init() {
213     struct global_queue *q = skynet_malloc(sizeof(*q));
214     memset(q,0,sizeof(*q));
215     SPIN_INIT(q);
216     Q=q;
217 }
218 
219 void
220 skynet_mq_mark_release(struct message_queue *q) {
221     SPIN_LOCK(q)
222     assert(q->release == 0);
223     q->release = 1;
224     if (q->in_global != MQ_IN_GLOBAL) {
225         skynet_globalmq_push(q);
226     }
227     SPIN_UNLOCK(q)
228 }
229 
230 static void
231 _drop_queue(struct message_queue *q, message_drop drop_func,
    void *ud) {
232     struct skynet_message msg;
233     while(!skynet_mq_pop(q, &msg)) {
234         drop_func(&msg, ud);
235     }
236     _release(q);
237 }
238 
239 void
240 skynet_mq_release(struct message_queue *q, message_drop
    drop_func, void *ud) {
241     SPIN_LOCK(q)
242 
```

```
243     if (q->release) {
244         SPIN_UNLOCK(q)
245         _drop_queue(q, drop_func, ud);
246     } else {
247         skynet_globalmq_push(q);
248         SPIN_UNLOCK(q)
249     }
250 }
```

z:\game\ff_server\skynet\skynet-src\skynet_mq.h

```
1  #ifndef SKYNET_MESSAGE_QUEUE_H
2  #define SKYNET_MESSAGE_QUEUE_H
3
4  #include <stdlib.h>
5  #include <stdint.h>
6
7  struct skynet_message {
8      uint32_t source;
9      int session;
10     void * data;
11     size_t sz;
12 };
13
14 // type is encoding in skynet_message.sz high 8bit
15 #define MESSAGE_TYPE_MASK (SIZE_MAX >> 8)
16 #define MESSAGE_TYPE_SHIFT ((sizeof(size_t)-1) * 8)
17
18 struct message_queue;
19
20 void skynet_globalmq_push(struct message_queue * queue);
21 struct message_queue * skynet_globalmq_pop(void);
22
23 struct message_queue * skynet_mq_create(uint32_t handle);
24 void skynet_mq_mark_release(struct message_queue *q);
25
26 typedef void (*message_drop)(struct skynet_message *, void
   *);
27
28 void skynet_mq_release(struct message_queue *q, message_drop
   drop_func, void *ud);
29 uint32_t skynet_mq_handle(struct message_queue *);
30
31 // 0 for success
32 int skynet_mq_pop(struct message_queue *q, struct
   skynet_message *message);
33 void skynet_mq_push(struct message_queue *q, struct
   skynet_message *message);
34
35 // return the length of message queue, for debug
36 int skynet_mq_length(struct message_queue *q);
37 int skynet_mq_overload(struct message_queue *q);
```

```
38
39 void skynet_mq_init();
40
41 #endif
```

z:\game\ff_server\skynet\skynet-src\skynet_server.c

```c
#include "skynet.h"

#include "skynet_server.h"
#include "skynet_module.h"
#include "skynet_handle.h"
#include "skynet_mq.h"
#include "skynet_timer.h"
#include "skynet_harbor.h"
#include "skynet_env.h"
#include "skynet_monitor.h"
#include "skynet_imp.h"
#include "skynet_log.h"
#include "spinlock.h"
#include "atomic.h"

#include <pthread.h>

#include <string.h>
#include <assert.h>
#include <stdint.h>
#include <stdio.h>
#include <stdbool.h>

#ifdef CALLING_CHECK

#define CHECKCALLING_BEGIN(ctx) if (!(spinlock_trylock(&ctx->calling))) { assert(0); }
#define CHECKCALLING_END(ctx) spinlock_unlock(&ctx->calling);
#define CHECKCALLING_INIT(ctx) spinlock_init(&ctx->calling);
#define CHECKCALLING_DESTROY(ctx) spinlock_destroy(&ctx->calling);
#define CHECKCALLING_DECL struct spinlock calling;

#else

#define CHECKCALLING_BEGIN(ctx)
#define CHECKCALLING_END(ctx)
#define CHECKCALLING_INIT(ctx)
#define CHECKCALLING_DESTROY(ctx)
```

```
38 #define CHECKCALLING_DECL
39
40 #endif
41
42 struct skynet_context {
43     void * instance;
44     struct skynet_module * mod;
45     void * cb_ud;
46     skynet_cb cb;
47     struct message_queue *queue;
48     FILE * logfile;
49     char result[32];
50     uint32_t handle;
51     int session_id;
52     int ref;
53     bool init;
54     bool endless;
55
56     CHECKCALLING_DECL
57 };
58
59 struct skynet_node {
60     int total;
61     int init;
62     uint32_t monitor_exit;
63     pthread_key_t handle_key;
64 };
65
66 static struct skynet_node G_NODE;
67
68 int
69 skynet_context_total() {
70     return G_NODE.total;
71 }
72
73 static void
74 context_inc() {
75     ATOM_INC(&G_NODE.total);
76 }
77
78 static void
79 context_dec() {
```

```
80      ATOM_DEC(&G_NODE.total);
81  }
82
83  uint32_t
84  skynet_current_handle(void) {
85      if (G_NODE.init) {
86          void * handle =
    pthread_getspecific(G_NODE.handle_key);
87          return (uint32_t)(uintptr_t)handle;
88      } else {
89          uint32_t v = (uint32_t)(-THREAD_MAIN);
90          return v;
91      }
92  }
93
94  static void
95  id_to_hex(char * str, uint32_t id) {
96      int i;
97      static char hex[16] = {
    '0','1','2','3','4','5','6','7','8','9','A','B','C','D','E','F'
    };
98      str[0] = ':';
99      for (i=0;i<8;i++) {
100         str[i+1] = hex[(id >> ((7-i) * 4))&0xf];
101     }
102     str[9] = '\0';
103 }
104
105 struct drop_t {
106     uint32_t handle;
107 };
108
109 static void
110 drop_message(struct skynet_message *msg, void *ud) {
111     struct drop_t *d = ud;
112     skynet_free(msg->data);
113     uint32_t source = d->handle;
114     assert(source);
115     // report error to the message source
116     skynet_send(NULL, source, msg->source, PTYPE_ERROR, 0,
    NULL, 0);
117 }
```

```
118
119 struct skynet_context *
120 skynet_context_new(const char * name, const char *param)
    {
121     struct skynet_module * mod = skynet_module_query(name);
122
123     if (mod == NULL)
124         return NULL;
125
126     void *inst = skynet_module_instance_create(mod);
127     if (inst == NULL)
128         return NULL;
129     struct skynet_context * ctx =
    skynet_malloc(sizeof(*ctx));
130     CHECKCALLING_INIT(ctx)
131
132     ctx->mod = mod;
133     ctx->instance = inst;
134     ctx->ref = 2;
135     ctx->cb = NULL;
136     ctx->cb_ud = NULL;
137     ctx->session_id = 0;
138     ctx->logfile = NULL;
139
140     ctx->init = false;
141     ctx->endless = false;
142     // Should set to 0 first to avoid skynet_handle_retireall
    get an uninitialized handle
143     ctx->handle = 0;
144     ctx->handle = skynet_handle_register(ctx);
145     struct message_queue * queue = ctx->queue =
    skynet_mq_create(ctx->handle);
146     // init function maybe use ctx->handle, so it must init at
    last
147     context_inc();
148
149     CHECKCALLING_BEGIN(ctx)
150     int r = skynet_module_instance_init(mod, inst, ctx,
    param);
151     CHECKCALLING_END(ctx)
152     if (r == 0) {
153         struct skynet_context * ret =
    skynet_context_release(ctx);
```

```
154         if (ret) {
155             ctx->init = true;
156         }
157         skynet_globalmq_push(queue);
158         if (ret) {
159             skynet_error(ret, "LAUNCH %s %s", name, param ?
param : "");
160         }
161         return ret;
162     } else {
163         skynet_error(ctx, "FAILED launch %s", name);
164         uint32_t handle = ctx->handle;
165         skynet_context_release(ctx);
166         skynet_handle_retire(handle);
167         struct drop_t d = { handle };
168         skynet_mq_release(queue, drop_message, &d);
169         return NULL;
170     }
171 }
172 
173 int
174 skynet_context_newsession(struct skynet_context *ctx) {
175     // session always be a positive number
176     int session = ++ctx->session_id;
177     if (session <= 0) {
178         ctx->session_id = 1;
179         return 1;
180     }
181     return session;
182 }
183 
184 void
185 skynet_context_grab(struct skynet_context *ctx) {
186     ATOM_INC(&ctx->ref);
187 }
188 
189 void
190 skynet_context_reserve(struct skynet_context *ctx) {
191     skynet_context_grab(ctx);
192     // don't count the context reserved, because skynet abort
(the worker threads terminate) only when the total context is 0
.
```

```
193     // the reserved context will be release at last.
194     context_dec();
195 }
196 
197 static void
198 delete_context(struct skynet_context *ctx) {
199     if (ctx->logfile) {
200         fclose(ctx->logfile);
201     }
202     skynet_module_instance_release(ctx->mod, ctx->instance);
203     skynet_mq_mark_release(ctx->queue);
204     CHECKCALLING_DESTROY(ctx)
205     skynet_free(ctx);
206     context_dec();
207 }
208 
209 struct skynet_context *
210 skynet_context_release(struct skynet_context *ctx) {
211     if (ATOM_DEC(&ctx->ref) == 0) {
212         delete_context(ctx);
213         return NULL;
214     }
215     return ctx;
216 }
217 
218 int
219 skynet_context_push(uint32_t handle, struct skynet_message
    *message) {
220     struct skynet_context * ctx =
    skynet_handle_grab(handle);
221     if (ctx == NULL) {
222         return -1;
223     }
224     skynet_mq_push(ctx->queue, message);
225     skynet_context_release(ctx);
226 
227     return 0;
228 }
229 
230 void
231 skynet_context_endless(uint32_t handle) {
232     struct skynet_context * ctx =
```

```
    skynet_handle_grab(handle);
233     if (ctx == NULL) {
234         return;
235     }
236     ctx->endless = true;
237     skynet_context_release(ctx);
238 }
239 
240 int
241 skynet_isremote(struct skynet_context * ctx, uint32_t
    handle, int * harbor) {
242     int ret = skynet_harbor_message_isremote(handle);
243     if (harbor) {
244         *harbor = (int)(handle >> HANDLE_REMOTE_SHIFT);
245     }
246     return ret;
247 }
248 
249 static void
250 dispatch_message(struct skynet_context *ctx, struct
    skynet_message *msg) {
251     assert(ctx->init);
252     CHECKCALLING_BEGIN(ctx)
253     pthread_setspecific(G_NODE.handle_key, (void *)
    (uintptr_t)(ctx->handle));
254     int type = msg->sz >> MESSAGE_TYPE_SHIFT;
255     size_t sz = msg->sz & MESSAGE_TYPE_MASK;
256     if (ctx->logfile) {
257         skynet_log_output(ctx->logfile, msg->source, type, msg-
    >session, msg->data, sz);
258     }
259     if (!ctx->cb(ctx, ctx->cb_ud, type, msg->session, msg-
    >source, msg->data, sz)) {
260         skynet_free(msg->data);
261     }
262     CHECKCALLING_END(ctx)
263 }
264 
265 void
266 skynet_context_dispatchall(struct skynet_context * ctx) {
267     // for skynet_error
268     struct skynet_message msg;
269     struct message_queue *q = ctx->queue;
```

```
270     while (!skynet_mq_pop(q,&msg)) {
271         dispatch_message(ctx, &msg);
272     }
273 }
274 
275 struct message_queue *
276 skynet_context_message_dispatch(struct skynet_monitor *sm,
    struct message_queue *q, int weight) {
277     if (q == NULL) {
278         q = skynet_globalmq_pop();
279         if (q==NULL)
280             return NULL;
281     }
282 
283     uint32_t handle = skynet_mq_handle(q);
284 
285     struct skynet_context * ctx =
    skynet_handle_grab(handle);
286     if (ctx == NULL) {
287         struct drop_t d = { handle };
288         skynet_mq_release(q, drop_message, &d);
289         return skynet_globalmq_pop();
290     }
291 
292     int i,n=1;
293     struct skynet_message msg;
294 
295     for (i=0;i<n;i++) {
296         if (skynet_mq_pop(q,&msg)) {
297             skynet_context_release(ctx);
298             return skynet_globalmq_pop();
299         } else if (i==0 && weight >= 0) {
300             n = skynet_mq_length(q);
301             n >>= weight;
302         }
303         int overload = skynet_mq_overload(q);
304         if (overload) {
305             skynet_error(ctx, "May overload, message queue
    length = %d", overload);
306         }
307 
308         skynet_monitor_trigger(sm, msg.source , handle);
```

```
309
310         if (ctx->cb == NULL) {
311             skynet_free(msg.data);
312         } else {
313             dispatch_message(ctx, &msg);
314         }
315
316         skynet_monitor_trigger(sm, 0,0);
317     }
318
319     assert(q == ctx->queue);
320     struct message_queue *nq = skynet_globalmq_pop();
321     if (nq) {
322         // If global mq is not empty , push q back, and return
    next queue (nq)
323         // Else (global mq is empty or block, don't push q
    back, and return q again (for next dispatch)
324         skynet_globalmq_push(q);
325         q = nq;
326     }
327     skynet_context_release(ctx);
328
329     return q;
330 }
331
332 static void
333 copy_name(char name[GLOBALNAME_LENGTH], const char * addr)
    {
334     int i;
335     for (i=0;i<GLOBALNAME_LENGTH && addr[i];i++) {
336         name[i] = addr[i];
337     }
338     for (;i<GLOBALNAME_LENGTH;i++) {
339         name[i] = '\0';
340     }
341 }
342
343 uint32_t
344 skynet_queryname(struct skynet_context * context, const
    char * name) {
345     switch(name[0]) {
346     case ':':
347         return strtoul(name+1,NULL,16);
```

```
348     case '.':
349         return skynet_handle_findname(name + 1);
350     }
351     skynet_error(context, "Don't support query global name
    %s",name);
352     return 0;
353 }
354 
355 static void
356 handle_exit(struct skynet_context * context, uint32_t
    handle) {
357     if (handle == 0) {
358         handle = context->handle;
359         skynet_error(context, "KILL self");
360     } else {
361         skynet_error(context, "KILL :%0x", handle);
362     }
363     if (G_NODE.monitor_exit) {
364         skynet_send(context,  handle, G_NODE.monitor_exit,
    PTYPE_CLIENT, 0, NULL, 0);
365     }
366     skynet_handle_retire(handle);
367 }
368 
369 // skynet command
370 
371 struct command_func {
372     const char *name;
373     const char * (*func)(struct skynet_context * context,
    const char * param);
374 };
375 
376 static const char *
377 cmd_timeout(struct skynet_context * context, const char *
    param) {
378     char * session_ptr = NULL;
379     int ti = strtol(param, &session_ptr, 10);
380     int session = skynet_context_newsession(context);
381     skynet_timeout(context->handle, ti, session);
382     sprintf(context->result, "%d", session);
383     return context->result;
384 }
385 
```

```
386 static const char *
387 cmd_reg(struct skynet_context * context, const char * param)
    {
388     if (param == NULL || param[0] == '\0') {
389         sprintf(context->result, ":%x", context->handle);
390         return context->result;
391     } else if (param[0] == '.') {
392         return skynet_handle_namehandle(context->handle, param
    + 1);
393     } else {
394         skynet_error(context, "Can't register global name %s in
    C", param);
395         return NULL;
396     }
397 }
398
399 static const char *
400 cmd_query(struct skynet_context * context, const char *
    param) {
401     if (param[0] == '.') {
402         uint32_t handle = skynet_handle_findname(param+1);
403         if (handle) {
404             sprintf(context->result, ":%x", handle);
405             return context->result;
406         }
407     }
408     return NULL;
409 }
410
411 static const char *
412 cmd_name(struct skynet_context * context, const char *
    param) {
413     int size = strlen(param);
414     char name[size+1];
415     char handle[size+1];
416     sscanf(param,"%s %s",name,handle);
417     if (handle[0] != ':') {
418         return NULL;
419     }
420     uint32_t handle_id = strtoul(handle+1, NULL, 16);
421     if (handle_id == 0) {
422         return NULL;
423     }
```

```
424     if (name[0] == '.') {
425         return skynet_handle_namehandle(handle_id, name + 1);
426     } else {
427         skynet_error(context, "Can't set global name %s in C",
    name);
428     }
429     return NULL;
430 }
431 
432 static const char *
433 cmd_exit(struct skynet_context * context, const char *
    param) {
434     handle_exit(context, 0);
435     return NULL;
436 }
437 
438 static uint32_t
439 tohandle(struct skynet_context * context, const char *
    param) {
440     uint32_t handle = 0;
441     if (param[0] == ':') {
442         handle = strtoul(param+1, NULL, 16);
443     } else if (param[0] == '.') {
444         handle = skynet_handle_findname(param+1);
445     } else {
446         skynet_error(context, "Can't convert %s to
    handle",param);
447     }
448 
449     return handle;
450 }
451 
452 static const char *
453 cmd_kill(struct skynet_context * context, const char *
    param) {
454     uint32_t handle = tohandle(context, param);
455     if (handle) {
456         handle_exit(context, handle);
457     }
458     return NULL;
459 }
460 
461 static const char *
```

```
462 cmd_launch(struct skynet_context * context, const char *
    param) {
463     size_t sz = strlen(param);
464     char tmp[sz+1];
465     strcpy(tmp,param);
466     char * args = tmp;
467     char * mod = strsep(&args, " \t\r\n");
468     args = strsep(&args, "\r\n");
469     struct skynet_context * inst =
    skynet_context_new(mod,args);
470     if (inst == NULL) {
471         return NULL;
472     } else {
473         id_to_hex(context->result, inst->handle);
474         return context->result;
475     }
476 }
477
478 static const char *
479 cmd_getenv(struct skynet_context * context, const char *
    param) {
480     return skynet_getenv(param);
481 }
482
483 static const char *
484 cmd_setenv(struct skynet_context * context, const char *
    param) {
485     size_t sz = strlen(param);
486     char key[sz+1];
487     int i;
488     for (i=0;param[i] != ' ' && param[i];i++) {
489         key[i] = param[i];
490     }
491     if (param[i] == '\0')
492         return NULL;
493
494     key[i] = '\0';
495     param += i+1;
496
497     skynet_setenv(key,param);
498     return NULL;
499 }
500
```

```
501 static const char *
502 cmd_starttime(struct skynet_context * context, const char *
    param) {
503     uint32_t sec = skynet_starttime();
504     sprintf(context->result,"%u",sec);
505     return context->result;
506 }
507
508 static const char *
509 cmd_endless(struct skynet_context * context, const char *
    param) {
510     if (context->endless) {
511         strcpy(context->result, "1");
512         context->endless = false;
513         return context->result;
514     }
515     return NULL;
516 }
517
518 static const char *
519 cmd_abort(struct skynet_context * context, const char *
    param) {
520     skynet_handle_retireall();
521     return NULL;
522 }
523
524 static const char *
525 cmd_monitor(struct skynet_context * context, const char *
    param) {
526     uint32_t handle=0;
527     if (param == NULL || param[0] == '\0') {
528         if (G_NODE.monitor_exit) {
529             // return current monitor serivce
530             sprintf(context->result, ":%x",
    G_NODE.monitor_exit);
531             return context->result;
532         }
533         return NULL;
534     } else {
535         handle = tohandle(context, param);
536     }
537     G_NODE.monitor_exit = handle;
538     return NULL;
```

```
539 }
540
541 static const char *
542 cmd_mqlen(struct skynet_context * context, const char *
    param) {
543     int len = skynet_mq_length(context->queue);
544     sprintf(context->result, "%d", len);
545     return context->result;
546 }
547
548 static const char *
549 cmd_logon(struct skynet_context * context, const char *
    param) {
550     uint32_t handle = tohandle(context, param);
551     if (handle == 0)
552         return NULL;
553     struct skynet_context * ctx =
    skynet_handle_grab(handle);
554     if (ctx == NULL)
555         return NULL;
556     FILE *f = NULL;
557     FILE * lastf = ctx->logfile;
558     if (lastf == NULL) {
559         f = skynet_log_open(context, handle);
560         if (f) {
561             if (!ATOM_CAS_POINTER(&ctx->logfile, NULL, f)) {
562                 // logfile opens in other thread, close this
    one.
563                 fclose(f);
564             }
565         }
566     }
567     skynet_context_release(ctx);
568     return NULL;
569 }
570
571 static const char *
572 cmd_logoff(struct skynet_context * context, const char *
    param) {
573     uint32_t handle = tohandle(context, param);
574     if (handle == 0)
575         return NULL;
576     struct skynet_context * ctx =
```

```
    skynet_handle_grab(handle);
577     if (ctx == NULL)
578         return NULL;
579     FILE * f = ctx->logfile;
580     if (f) {
581         // logfile may close in other thread
582         if (ATOM_CAS_POINTER(&ctx->logfile, f, NULL)) {
583             skynet_log_close(context, f, handle);
584         }
585     }
586     skynet_context_release(ctx);
587     return NULL;
588 }
589
590 static const char *
591 cmd_signal(struct skynet_context * context, const char *
    param) {
592     uint32_t handle = tohandle(context, param);
593     if (handle == 0)
594         return NULL;
595     struct skynet_context * ctx =
    skynet_handle_grab(handle);
596     if (ctx == NULL)
597         return NULL;
598     param = strchr(param, ' ');
599     int sig = 0;
600     if (param) {
601         sig = strtol(param, NULL, 0);
602     }
603     // NOTICE: the signal function should be thread safe.
604     skynet_module_instance_signal(ctx->mod, ctx->instance,
    sig);
605
606     skynet_context_release(ctx);
607     return NULL;
608 }
609
610 static struct command_func cmd_funcs[] = {
611     { "TIMEOUT", cmd_timeout },
612     { "REG", cmd_reg },
613     { "QUERY", cmd_query },
614     { "NAME", cmd_name },
```

```
615     { "EXIT", cmd_exit },
616     { "KILL", cmd_kill },
617     { "LAUNCH", cmd_launch },
618     { "GETENV", cmd_getenv },
619     { "SETENV", cmd_setenv },
620     { "STARTTIME", cmd_starttime },
621     { "ENDLESS", cmd_endless },
622     { "ABORT", cmd_abort },
623     { "MONITOR", cmd_monitor },
624     { "MQLEN", cmd_mqlen },
625     { "LOGON", cmd_logon },
626     { "LOGOFF", cmd_logoff },
627     { "SIGNAL", cmd_signal },
628     { NULL, NULL },
629 };
630
631 const char *
632 skynet_command(struct skynet_context * context, const char
    * cmd , const char * param) {
633     struct command_func * method = &cmd_funcs[0];
634     while(method->name) {
635         if (strcmp(cmd, method->name) == 0) {
636             return method->func(context, param);
637         }
638         ++method;
639     }
640
641     return NULL;
642 }
643
644 static void
645 _filter_args(struct skynet_context * context, int type, int
    *session, void ** data, size_t * sz) {
646     int needcopy = !(type & PTYPE_TAG_DONTCOPY);
647     int allocsession = type & PTYPE_TAG_ALLOCSESSION;
648     type &= 0xff;
649
650     if (allocsession) {
651         assert(*session == 0);
652         *session = skynet_context_newsession(context);
653     }
654
```

```
655     if (needcopy && *data) {
656         char * msg = skynet_malloc(*sz+1);
657         memcpy(msg, *data, *sz);
658         msg[*sz] = '\0';
659         *data = msg;
660     }
661 
662     *sz |= (size_t)type << MESSAGE_TYPE_SHIFT;
663 }
664 
665 int
666 skynet_send(struct skynet_context * context, uint32_t
    source, uint32_t destination , int type, int session, void *
    data, size_t sz) {
667     if ((sz & MESSAGE_TYPE_MASK) != sz) {
668         skynet_error(context, "The message to %x is too large",
    destination);
669         if (type & PTYPE_TAG_DONTCOPY) {
670             skynet_free(data);
671         }
672         return -1;
673     }
674     _filter_args(context, type, &session, (void **)&data,
    &sz);
675 
676     if (source == 0) {
677         source = context->handle;
678     }
679 
680     if (destination == 0) {
681         return session;
682     }
683     if (skynet_harbor_message_isremote(destination)) {
684         struct remote_message * rmsg =
    skynet_malloc(sizeof(*rmsg));
685         rmsg->destination.handle = destination;
686         rmsg->message = data;
687         rmsg->sz = sz;
688         skynet_harbor_send(rmsg, source, session);
689     } else {
690         struct skynet_message smsg;
691         smsg.source = source;
692         smsg.session = session;
```

```
693         smsg.data = data;
694         smsg.sz = sz;
695 
696         if (skynet_context_push(destination, &smsg)) {
697             skynet_free(data);
698             return -1;
699         }
700     }
701     return session;
702 }
703 
704 int
705 skynet_sendname(struct skynet_context * context, uint32_t
    source, const char * addr , int type, int session, void *
    data, size_t sz) {
706     if (source == 0) {
707         source = context->handle;
708     }
709     uint32_t des = 0;
710     if (addr[0] == ':') {
711         des = strtoul(addr+1, NULL, 16);
712     } else if (addr[0] == '.') {
713         des = skynet_handle_findname(addr + 1);
714         if (des == 0) {
715             if (type & PTYPE_TAG_DONTCOPY) {
716                 skynet_free(data);
717             }
718             return -1;
719         }
720     } else {
721         _filter_args(context, type, &session, (void **)&data,
    &sz);
722 
723         struct remote_message * rmsg =
    skynet_malloc(sizeof(*rmsg));
724         copy_name(rmsg->destination.name, addr);
725         rmsg->destination.handle = 0;
726         rmsg->message = data;
727         rmsg->sz = sz;
728 
729         skynet_harbor_send(rmsg, source, session);
730         return session;
```

```
731     }
732
733     return skynet_send(context, source, des, type, session,
    data, sz);
734 }
735
736 uint32_t
737 skynet_context_handle(struct skynet_context *ctx) {
738     return ctx->handle;
739 }
740
741 void
742 skynet_callback(struct skynet_context * context, void *ud,
    skynet_cb cb) {
743     context->cb = cb;
744     context->cb_ud = ud;
745 }
746
747 void
748 skynet_context_send(struct skynet_context * ctx, void *
    msg, size_t sz, uint32_t source, int type, int session) {
749     struct skynet_message smsg;
750     smsg.source = source;
751     smsg.session = session;
752     smsg.data = msg;
753     smsg.sz = sz | (size_t)type << MESSAGE_TYPE_SHIFT;
754
755     skynet_mq_push(ctx->queue, &smsg);
756 }
757
758 void
759 skynet_globalinit(void) {
760     G_NODE.total = 0;
761     G_NODE.monitor_exit = 0;
762     G_NODE.init = 1;
763     if (pthread_key_create(&G_NODE.handle_key, NULL)) {
764         fprintf(stderr, "pthread_key_create failed");
765         exit(1);
766     }
767     // set mainthread's key
768     skynet_initthread(THREAD_MAIN);
769 }
```

```
770
771 void
772 skynet_globalexit(void) {
773     pthread_key_delete(G_NODE.handle_key);
774 }
775
776 void
777 skynet_initthread(int m) {
778     uintptr_t v = (uint32_t)(-m);
779     pthread_setspecific(G_NODE.handle_key, (void *)v);
780 }
```

z:\game\ff_server\skynet\skynet-src\skynet_server.h

```h
#ifndef SKYNET_SERVER_H
#define SKYNET_SERVER_H

#include <stdint.h>
#include <stdlib.h>

struct skynet_context;
struct skynet_message;
struct skynet_monitor;

struct skynet_context * skynet_context_new(const char * name, const char * parm);
void skynet_context_grab(struct skynet_context *);
void skynet_context_reserve(struct skynet_context *ctx);
struct skynet_context * skynet_context_release(struct skynet_context *);
uint32_t skynet_context_handle(struct skynet_context *);
int skynet_context_push(uint32_t handle, struct skynet_message *message);
void skynet_context_send(struct skynet_context * context, void * msg, size_t sz, uint32_t source, int type, int session);
int skynet_context_newsession(struct skynet_context *);
struct message_queue * skynet_context_message_dispatch(struct skynet_monitor *, struct message_queue *, int weight);    // return next queue
int skynet_context_total();
void skynet_context_dispatchall(struct skynet_context * context);   // for skynet_error output before exit

void skynet_context_endless(uint32_t handle);  // for monitor

void skynet_globalinit(void);
void skynet_globalexit(void);
void skynet_initthread(int m);

#endif
```

z:\game\ff_server\skynet\skynet-src\skynet_socket.c

```c
#include "skynet.h"

#include "skynet_socket.h"
#include "socket_server.h"
#include "skynet_server.h"
#include "skynet_mq.h"
#include "skynet_harbor.h"

#include <assert.h>
#include <stdlib.h>
#include <string.h>
#include <stdbool.h>

static struct socket_server * SOCKET_SERVER = NULL;

void
skynet_socket_init() {
    SOCKET_SERVER = socket_server_create();
}

void
skynet_socket_exit() {
    socket_server_exit(SOCKET_SERVER);
}

void
skynet_socket_free() {
    socket_server_release(SOCKET_SERVER);
    SOCKET_SERVER = NULL;
}

// mainloop thread
static void
forward_message(int type, bool padding, struct socket_message * result) {
    struct skynet_socket_message *sm;
    size_t sz = sizeof(*sm);
    if (padding) {
        if (result->data) {
```

```
39              size_t msg_sz = strlen(result->data);
40              if (msg_sz > 128) {
41                  msg_sz = 128;
42              }
43              sz += msg_sz;
44          } else {
45              result->data = "";
46          }
47      }
48      sm = (struct skynet_socket_message *)skynet_malloc(sz);
49      sm->type = type;
50      sm->id = result->id;
51      sm->ud = result->ud;
52      if (padding) {
53          sm->buffer = NULL;
54          memcpy(sm+1, result->data, sz - sizeof(*sm));
55      } else {
56          sm->buffer = result->data;
57      }
58
59      struct skynet_message message;
60      message.source = 0;
61      message.session = 0;
62      message.data = sm;
63      message.sz = sz | ((size_t)PTYPE_SOCKET <<
MESSAGE_TYPE_SHIFT);
64
65      if (skynet_context_push((uint32_t)result->opaque,
&message)) {
66          // todo: report somewhere to close socket
67          // don't call skynet_socket_close here (It will block
mainloop)
68          skynet_free(sm->buffer);
69          skynet_free(sm);
70      }
71  }
72
73  int
74  skynet_socket_poll() {
75      struct socket_server *ss = SOCKET_SERVER;
76      assert(ss);
77      struct socket_message result;
```

```
78      int more = 1;
79      int type = socket_server_poll(ss, &result, &more);
80      switch (type) {
81      case SOCKET_EXIT:
82          return 0;
83      case SOCKET_DATA:
84          forward_message(SKYNET_SOCKET_TYPE_DATA, false,
    &result);
85          break;
86      case SOCKET_CLOSE:
87          forward_message(SKYNET_SOCKET_TYPE_CLOSE, false,
    &result);
88          break;
89      case SOCKET_OPEN:
90          forward_message(SKYNET_SOCKET_TYPE_CONNECT, true,
    &result);
91          break;
92      case SOCKET_ERROR:
93          forward_message(SKYNET_SOCKET_TYPE_ERROR, true,
    &result);
94          break;
95      case SOCKET_ACCEPT:
96          forward_message(SKYNET_SOCKET_TYPE_ACCEPT, true,
    &result);
97          break;
98      case SOCKET_UDP:
99          forward_message(SKYNET_SOCKET_TYPE_UDP, false,
    &result);
100         break;
101     default:
102         skynet_error(NULL, "Unknown socket message type
    %d.",type);
103         return -1;
104     }
105     if (more) {
106         return -1;
107     }
108     return 1;
109 }
110
111 static int
112 check_wsz(struct skynet_context *ctx, int id, void *buffer,
    int64_t wsz) {
```

```
113     if (wsz < 0) {
114         return -1;
115     } else if (wsz > 1024 * 1024) {
116         struct skynet_socket_message tmp;
117         tmp.type = SKYNET_SOCKET_TYPE_WARNING;
118         tmp.id = id;
119         tmp.ud = (int)(wsz / 1024);
120         tmp.buffer = NULL;
121         skynet_send(ctx, 0, skynet_context_handle(ctx),
PTYPE_SOCKET, 0 , &tmp, sizeof(tmp));
122 //      skynet_error(ctx, "%d Mb bytes on socket %d need to
send out", (int)(wsz / (1024 * 1024)), id);
123     }
124     return 0;
125 }
126
127 int
128 skynet_socket_send(struct skynet_context *ctx, int id, void
*buffer, int sz) {
129     int64_t wsz = socket_server_send(SOCKET_SERVER, id,
buffer, sz);
130     return check_wsz(ctx, id, buffer, wsz);
131 }
132
133 void
134 skynet_socket_send_lowpriority(struct skynet_context *ctx,
int id, void *buffer, int sz) {
135     socket_server_send_lowpriority(SOCKET_SERVER, id, buffer,
sz);
136 }
137
138 int
139 skynet_socket_listen(struct skynet_context *ctx, const
char *host, int port, int backlog) {
140     uint32_t source = skynet_context_handle(ctx);
141     return socket_server_listen(SOCKET_SERVER, source, host,
port, backlog);
142 }
143
144 int
145 skynet_socket_connect(struct skynet_context *ctx, const
char *host, int port) {
146     uint32_t source = skynet_context_handle(ctx);
```

```
147     return socket_server_connect(SOCKET_SERVER, source, host,
    port);
148 }
149 
150 int
151 skynet_socket_bind(struct skynet_context *ctx, int fd) {
152     uint32_t source = skynet_context_handle(ctx);
153     return socket_server_bind(SOCKET_SERVER, source, fd);
154 }
155 
156 void
157 skynet_socket_close(struct skynet_context *ctx, int id) {
158     uint32_t source = skynet_context_handle(ctx);
159     socket_server_close(SOCKET_SERVER, source, id);
160 }
161 
162 void
163 skynet_socket_shutdown(struct skynet_context *ctx, int id)
    {
164     uint32_t source = skynet_context_handle(ctx);
165     socket_server_shutdown(SOCKET_SERVER, source, id);
166 }
167 
168 void
169 skynet_socket_start(struct skynet_context *ctx, int id) {
170     uint32_t source = skynet_context_handle(ctx);
171     socket_server_start(SOCKET_SERVER, source, id);
172 }
173 
174 void
175 skynet_socket_nodelay(struct skynet_context *ctx, int id) {
176     socket_server_nodelay(SOCKET_SERVER, id);
177 }
178 
179 int
180 skynet_socket_udp(struct skynet_context *ctx, const char *
    addr, int port) {
181     uint32_t source = skynet_context_handle(ctx);
182     return socket_server_udp(SOCKET_SERVER, source, addr,
    port);
183 }
184 
185 int
```

```
186 skynet_socket_udp_connect(struct skynet_context *ctx, int
    id, const char * addr, int port) {
187     return socket_server_udp_connect(SOCKET_SERVER, id, addr,
    port);
188 }
189
190 int
191 skynet_socket_udp_send(struct skynet_context *ctx, int id,
    const char * address, const void *buffer, int sz) {
192     int64_t wsz = socket_server_udp_send(SOCKET_SERVER, id,
    (const struct socket_udp_address *)address, buffer, sz);
193     return check_wsz(ctx, id, (void *)buffer, wsz);
194 }
195
196 const char *
197 skynet_socket_udp_address(struct skynet_socket_message
    *msg, int *addrsz) {
198     if (msg->type != SKYNET_SOCKET_TYPE_UDP) {
199         return NULL;
200     }
201     struct socket_message sm;
202     sm.id = msg->id;
203     sm.opaque = 0;
204     sm.ud = msg->ud;
205     sm.data = msg->buffer;
206     return (const char
    *)socket_server_udp_address(SOCKET_SERVER, &sm, addrsz);
207 }
```

z:\game\ff_server\skynet\skynet-src\skynet_socket.h

```
1  #ifndef skynet_socket_h
2  #define skynet_socket_h
3
4  struct skynet_context;
5
6  #define SKYNET_SOCKET_TYPE_DATA 1
7  #define SKYNET_SOCKET_TYPE_CONNECT 2
8  #define SKYNET_SOCKET_TYPE_CLOSE 3
9  #define SKYNET_SOCKET_TYPE_ACCEPT 4
10 #define SKYNET_SOCKET_TYPE_ERROR 5
11 #define SKYNET_SOCKET_TYPE_UDP 6
12 #define SKYNET_SOCKET_TYPE_WARNING 7
13
14 struct skynet_socket_message {
15     int type;
16     int id;
17     int ud;
18     char * buffer;
19 };
20
21 void skynet_socket_init();
22 void skynet_socket_exit();
23 void skynet_socket_free();
24 int skynet_socket_poll();
25
26 int skynet_socket_send(struct skynet_context *ctx, int id,
   void *buffer, int sz);
27 void skynet_socket_send_lowpriority(struct skynet_context
   *ctx, int id, void *buffer, int sz);
28 int skynet_socket_listen(struct skynet_context *ctx, const
   char *host, int port, int backlog);
29 int skynet_socket_connect(struct skynet_context *ctx, const
   char *host, int port);
30 int skynet_socket_bind(struct skynet_context *ctx, int fd);
31 void skynet_socket_close(struct skynet_context *ctx, int
   id);
32 void skynet_socket_shutdown(struct skynet_context *ctx, int
   id);
33 void skynet_socket_start(struct skynet_context *ctx, int
   id);
```

```
34 void skynet_socket_nodelay(struct skynet_context *ctx, int
   id);
35
36 int skynet_socket_udp(struct skynet_context *ctx, const
   char * addr, int port);
37 int skynet_socket_udp_connect(struct skynet_context *ctx,
   int id, const char * addr, int port);
38 int skynet_socket_udp_send(struct skynet_context *ctx, int
   id, const char * address, const void *buffer, int sz);
39 const char * skynet_socket_udp_address(struct
   skynet_socket_message *, int *addrsz);
40
41 #endif
```

z:\game\ff_server\skynet\skynet-src\skynet_start.c

```c
#include "skynet.h"
#include "skynet_server.h"
#include "skynet_imp.h"
#include "skynet_mq.h"
#include "skynet_handle.h"
#include "skynet_module.h"
#include "skynet_timer.h"
#include "skynet_monitor.h"
#include "skynet_socket.h"
#include "skynet_daemon.h"
#include "skynet_harbor.h"

#include <pthread.h>
#include <unistd.h>
#include <assert.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <signal.h>

struct monitor {
    int count;
    struct skynet_monitor ** m;
    pthread_cond_t cond;
    pthread_mutex_t mutex;
    int sleep;
    int quit;
};

struct worker_parm {
    struct monitor *m;
    int id;
    int weight;
};

static int SIG = 0;

static void
handle_hup(int signal) {
```

```
40      if (signal == SIGHUP) {
41          SIG = 1;
42      }
43  }
44
45  #define CHECK_ABORT if (skynet_context_total()==0) break;
46
47  static void
48  create_thread(pthread_t *thread, void *(*start_routine)
    (void *), void *arg) {
49      if (pthread_create(thread,NULL, start_routine, arg)) {
50          fprintf(stderr, "Create thread failed");
51          exit(1);
52      }
53  }
54
55  static void
56  wakeup(struct monitor *m, int busy) {
57      if (m->sleep >= m->count - busy) {
58          // signal sleep worker, "spurious wakeup" is harmless
59          pthread_cond_signal(&m->cond);
60      }
61  }
62
63  static void *
64  thread_socket(void *p) {
65      struct monitor * m = p;
66      skynet_initthread(THREAD_SOCKET);
67      for (;;) {
68          int r = skynet_socket_poll();
69          if (r==0)
70              break;
71          if (r<0) {
72              CHECK_ABORT
73              continue;
74          }
75          wakeup(m,0);
76      }
77      return NULL;
78  }
79
80  static void
```

```
81 free_monitor(struct monitor *m) {
82     int i;
83     int n = m->count;
84     for (i=0;i<n;i++) {
85         skynet_monitor_delete(m->m[i]);
86     }
87     pthread_mutex_destroy(&m->mutex);
88     pthread_cond_destroy(&m->cond);
89     skynet_free(m->m);
90     skynet_free(m);
91 }
92
93 static void *
94 thread_monitor(void *p) {
95     struct monitor * m = p;
96     int i;
97     int n = m->count;
98     skynet_initthread(THREAD_MONITOR);
99     for (;;) {
100        CHECK_ABORT
101        for (i=0;i<n;i++) {
102            skynet_monitor_check(m->m[i]);
103        }
104        for (i=0;i<5;i++) {
105            CHECK_ABORT
106            sleep(1);
107        }
108    }
109
110    return NULL;
111 }
112
113 static void
114 signal_hup() {
115    // make log file reopen
116
117    struct skynet_message smsg;
118    smsg.source = 0;
119    smsg.session = 0;
120    smsg.data = NULL;
121    smsg.sz = (size_t)PTYPE_SYSTEM << MESSAGE_TYPE_SHIFT;
122    uint32_t logger = skynet_handle_findname("logger");
```

```
123     if (logger) {
124         skynet_context_push(logger, &smsg);
125     }
126 }
127 
128 static void *
129 thread_timer(void *p) {
130     struct monitor * m = p;
131     skynet_initthread(THREAD_TIMER);
132     for (;;) {
133         skynet_updatetime();
134         CHECK_ABORT
135         wakeup(m,m->count-1);
136         usleep(2500);
137         if (SIG) {
138             signal_hup();
139             SIG = 0;
140         }
141     }
142     // wakeup socket thread
143     skynet_socket_exit();
144     // wakeup all worker thread
145     pthread_mutex_lock(&m->mutex);
146     m->quit = 1;
147     pthread_cond_broadcast(&m->cond);
148     pthread_mutex_unlock(&m->mutex);
149     return NULL;
150 }
151 
152 static void *
153 thread_worker(void *p) {
154     struct worker_parm *wp = p;
155     int id = wp->id;
156     int weight = wp->weight;
157     struct monitor *m = wp->m;
158     struct skynet_monitor *sm = m->m[id];
159     skynet_initthread(THREAD_WORKER);
160     struct message_queue * q = NULL;
161     while (!m->quit) {
162         q = skynet_context_message_dispatch(sm, q, weight);
163         if (q == NULL) {
164             if (pthread_mutex_lock(&m->mutex) == 0) {
```

```
165                 ++ m->sleep;
166                 // "spurious wakeup" is harmless,
167                 // because skynet_context_message_dispatch()
    can be call at any time.
168                 if (!m->quit)
169                     pthread_cond_wait(&m->cond, &m->mutex);
170                 -- m->sleep;
171                 if (pthread_mutex_unlock(&m->mutex)) {
172                     fprintf(stderr, "unlock mutex error");
173                     exit(1);
174                 }
175             }
176         }
177     }
178     return NULL;
179 }
180 
181 static void
182 start(int thread) {
183     pthread_t pid[thread+3];
184 
185     struct monitor *m = skynet_malloc(sizeof(*m));
186     memset(m, 0, sizeof(*m));
187     m->count = thread;
188     m->sleep = 0;
189 
190     m->m = skynet_malloc(thread * sizeof(struct skynet_monitor
    *));
191     int i;
192     for (i=0;i<thread;i++) {
193         m->m[i] = skynet_monitor_new();
194     }
195     if (pthread_mutex_init(&m->mutex, NULL)) {
196         fprintf(stderr, "Init mutex error");
197         exit(1);
198     }
199     if (pthread_cond_init(&m->cond, NULL)) {
200         fprintf(stderr, "Init cond error");
201         exit(1);
202     }
203 
204     create_thread(&pid[0], thread_monitor, m);
```

```
205     create_thread(&pid[1], thread_timer, m);
206     create_thread(&pid[2], thread_socket, m);
207 
208     static int weight[] = {
209         -1, -1, -1, -1, 0, 0, 0, 0,
210         1, 1, 1, 1, 1, 1, 1, 1,
211         2, 2, 2, 2, 2, 2, 2, 2,
212         3, 3, 3, 3, 3, 3, 3, 3, };
213     struct worker_parm wp[thread];
214     for (i=0;i<thread;i++) {
215         wp[i].m = m;
216         wp[i].id = i;
217         if (i < sizeof(weight)/sizeof(weight[0])) {
218             wp[i].weight= weight[i];
219         } else {
220             wp[i].weight = 0;
221         }
222         create_thread(&pid[i+3], thread_worker, &wp[i]);
223     }
224 
225     for (i=0;i<thread+3;i++) {
226         pthread_join(pid[i], NULL);
227     }
228 
229     free_monitor(m);
230 }
231 
232 static void
233 bootstrap(struct skynet_context * logger, const char *
    cmdline) {
234     int sz = strlen(cmdline);
235     char name[sz+1];
236     char args[sz+1];
237     sscanf(cmdline, "%s %s", name, args);
238     struct skynet_context *ctx = skynet_context_new(name,
    args);
239     if (ctx == NULL) {
240         skynet_error(NULL, "Bootstrap error : %s\n", cmdline);
241         skynet_context_dispatchall(logger);
242         exit(1);
243     }
244 }
```

```
245
246 void
247 skynet_start(struct skynet_config * config) {
248     // register SIGHUP for log file reopen
249     struct sigaction sa;
250     sa.sa_handler = &handle_hup;
251     sa.sa_flags = SA_RESTART;
252     sigfillset(&sa.sa_mask);
253     sigaction(SIGHUP, &sa, NULL);
254
255     if (config->daemon) {
256         if (daemon_init(config->daemon)) {
257             exit(1);
258         }
259     }
260     skynet_harbor_init(config->harbor);
261     skynet_handle_init(config->harbor);
262     skynet_mq_init();
263     skynet_module_init(config->module_path);
264     skynet_timer_init();
265     skynet_socket_init();
266
267     struct skynet_context *ctx = skynet_context_new(config-
>logservice, config->logger);
268     if (ctx == NULL) {
269         fprintf(stderr, "Can't launch %s service\n", config-
>logservice);
270         exit(1);
271     }
272
273     bootstrap(ctx, config->bootstrap);
274
275     start(config->thread);
276
277     // harbor_exit may call socket send, so it should exit
before socket_free
278     skynet_harbor_exit();
279     skynet_socket_free();
280     if (config->daemon) {
281         daemon_exit(config->daemon);
282     }
283 }
```

z:\game\ff_server\skynet\skynet-src\skynet_timer.c

```c
#include "skynet.h"

#include "skynet_timer.h"
#include "skynet_mq.h"
#include "skynet_server.h"
#include "skynet_handle.h"
#include "spinlock.h"

#include <time.h>
#include <assert.h>
#include <string.h>
#include <stdlib.h>
#include <stdint.h>

#if defined(__APPLE__)
#include <sys/time.h>
#endif

typedef void (*timer_execute_func)(void *ud,void *arg);

#define TIME_NEAR_SHIFT 8
#define TIME_NEAR (1 << TIME_NEAR_SHIFT)
#define TIME_LEVEL_SHIFT 6
#define TIME_LEVEL (1 << TIME_LEVEL_SHIFT)
#define TIME_NEAR_MASK (TIME_NEAR-1)
#define TIME_LEVEL_MASK (TIME_LEVEL-1)

struct timer_event {
    uint32_t handle;
    int session;
};

struct timer_node {
    struct timer_node *next;
    uint32_t expire;
};

struct link_list {
    struct timer_node head;
```

```
40      struct timer_node *tail;
41  };
42
43  struct timer {
44      struct link_list near[TIME_NEAR];
45      struct link_list t[4][TIME_LEVEL];
46      struct spinlock lock;
47      uint32_t time;
48      uint32_t starttime;
49      uint64_t current;
50      uint64_t current_point;
51  };
52
53  static struct timer * TI = NULL;
54
55  static inline struct timer_node *
56  link_clear(struct link_list *list) {
57      struct timer_node * ret = list->head.next;
58      list->head.next = 0;
59      list->tail = &(list->head);
60
61      return ret;
62  }
63
64  static inline void
65  link(struct link_list *list,struct timer_node *node) {
66      list->tail->next = node;
67      list->tail = node;
68      node->next=0;
69  }
70
71  static void
72  add_node(struct timer *T,struct timer_node *node) {
73      uint32_t time=node->expire;
74      uint32_t current_time=T->time;
75
76      if ((time|TIME_NEAR_MASK)==(current_time|TIME_NEAR_MASK))
    {
77          link(&T->near[time&TIME_NEAR_MASK],node);
78      } else {
79          int i;
80          uint32_t mask=TIME_NEAR << TIME_LEVEL_SHIFT;
```

```
81          for (i=0;i<3;i++) {
82              if ((time|(mask-1))==(current_time|(mask-1))) {
83                  break;
84              }
85              mask <<= TIME_LEVEL_SHIFT;
86          }
87
88          link(&T->t[i][((time>>(TIME_NEAR_SHIFT +
    i*TIME_LEVEL_SHIFT)) & TIME_LEVEL_MASK)],node);
89      }
90  }
91
92  static void
93  timer_add(struct timer *T,void *arg,size_t sz,int time) {
94      struct timer_node *node = (struct timer_node
    *)skynet_malloc(sizeof(*node)+sz);
95      memcpy(node+1,arg,sz);
96
97      SPIN_LOCK(T);
98
99          node->expire=time+T->time;
100         add_node(T,node);
101
102     SPIN_UNLOCK(T);
103 }
104
105 static void
106 move_list(struct timer *T, int level, int idx) {
107     struct timer_node *current = link_clear(&T->t[level]
    [idx]);
108     while (current) {
109         struct timer_node *temp=current->next;
110         add_node(T,current);
111         current=temp;
112     }
113 }
114
115 static void
116 timer_shift(struct timer *T) {
117     int mask = TIME_NEAR;
118     uint32_t ct = ++T->time;
119     if (ct == 0) {
```

```
120         move_list(T, 3, 0);
121     } else {
122         uint32_t time = ct >> TIME_NEAR_SHIFT;
123         int i=0;
124 
125         while ((ct & (mask-1))==0) {
126             int idx=time & TIME_LEVEL_MASK;
127             if (idx!=0) {
128                 move_list(T, i, idx);
129                 break;
130             }
131             mask <<= TIME_LEVEL_SHIFT;
132             time >>= TIME_LEVEL_SHIFT;
133             ++i;
134         }
135     }
136 }
137 
138 static inline void
139 dispatch_list(struct timer_node *current) {
140     do {
141         struct timer_event * event = (struct timer_event *)
    (current+1);
142         struct skynet_message message;
143         message.source = 0;
144         message.session = event->session;
145         message.data = NULL;
146         message.sz = (size_t)PTYPE_RESPONSE <<
    MESSAGE_TYPE_SHIFT;
147 
148         skynet_context_push(event->handle, &message);
149 
150         struct timer_node * temp = current;
151         current=current->next;
152         skynet_free(temp);
153     } while (current);
154 }
155 
156 static inline void
157 timer_execute(struct timer *T) {
158     int idx = T->time & TIME_NEAR_MASK;
159 
```

```
160     while (T->near[idx].head.next) {
161         struct timer_node *current = link_clear(&T-
>near[idx]);
162         SPIN_UNLOCK(T);
163         // dispatch_list don't need lock T
164         dispatch_list(current);
165         SPIN_LOCK(T);
166     }
167 }
168 
169 static void
170 timer_update(struct timer *T) {
171     SPIN_LOCK(T);
172 
173     // try to dispatch timeout 0 (rare condition)
174     timer_execute(T);
175 
176     // shift time first, and then dispatch timer message
177     timer_shift(T);
178 
179     timer_execute(T);
180 
181     SPIN_UNLOCK(T);
182 }
183 
184 static struct timer *
185 timer_create_timer() {
186     struct timer *r=(struct timer
*)skynet_malloc(sizeof(struct timer));
187     memset(r,0,sizeof(*r));
188 
189     int i,j;
190 
191     for (i=0;i<TIME_NEAR;i++) {
192         link_clear(&r->near[i]);
193     }
194 
195     for (i=0;i<4;i++) {
196         for (j=0;j<TIME_LEVEL;j++) {
197             link_clear(&r->t[i][j]);
198         }
199     }
```

```
200
201     SPIN_INIT(r)
202
203     r->current = 0;
204
205     return r;
206 }
207
208 int
209 skynet_timeout(uint32_t handle, int time, int session) {
210     if (time <= 0) {
211         struct skynet_message message;
212         message.source = 0;
213         message.session = session;
214         message.data = NULL;
215         message.sz = (size_t)PTYPE_RESPONSE <<
MESSAGE_TYPE_SHIFT;
216
217         if (skynet_context_push(handle, &message)) {
218             return -1;
219         }
220     } else {
221         struct timer_event event;
222         event.handle = handle;
223         event.session = session;
224         timer_add(TI, &event, sizeof(event), time);
225     }
226
227     return session;
228 }
229
230 // centisecond: 1/100 second
231 static void
232 systime(uint32_t *sec, uint32_t *cs) {
233 #if !defined(__APPLE__)
234     struct timespec ti;
235     clock_gettime(CLOCK_REALTIME, &ti);
236     *sec = (uint32_t)ti.tv_sec;
237     *cs = (uint32_t)(ti.tv_nsec / 10000000);
238 #else
239     struct timeval tv;
240     gettimeofday(&tv, NULL);
```

```
241     *sec = tv.tv_sec;
242     *cs = tv.tv_usec / 10000;
243 #endif
244 }
245 
246 static uint64_t
247 gettime() {
248     uint64_t t;
249 #if !defined(__APPLE__)
250     struct timespec ti;
251     clock_gettime(CLOCK_MONOTONIC, &ti);
252     t = (uint64_t)ti.tv_sec * 100;
253     t += ti.tv_nsec / 10000000;
254 #else
255     struct timeval tv;
256     gettimeofday(&tv, NULL);
257     t = (uint64_t)tv.tv_sec * 100;
258     t += tv.tv_usec / 10000;
259 #endif
260     return t;
261 }
262 
263 void
264 skynet_updatetime(void) {
265     uint64_t cp = gettime();
266     if(cp < TI->current_point) {
267         skynet_error(NULL, "time diff error: change from %lld
    to %lld", cp, TI->current_point);
268         TI->current_point = cp;
269     } else if (cp != TI->current_point) {
270         uint32_t diff = (uint32_t)(cp - TI->current_point);
271         TI->current_point = cp;
272         TI->current += diff;
273         int i;
274         for (i=0;i<diff;i++) {
275             timer_update(TI);
276         }
277     }
278 }
279 
280 uint32_t
281 skynet_starttime(void) {
```

```
282     return TI->starttime;
283 }
284 
285 uint64_t
286 skynet_now(void) {
287     return TI->current;
288 }
289 
290 void
291 skynet_timer_init(void) {
292     TI = timer_create_timer();
293     uint32_t current = 0;
294     systime(&TI->starttime, &current);
295     TI->current = current;
296     TI->current_point = gettime();
297 }
```

z:\game\ff_server\skynet\skynet-src\skynet_timer.h

```
1  #ifndef SKYNET_TIMER_H
2  #define SKYNET_TIMER_H
3  
4  #include <stdint.h>
5  
6  int skynet_timeout(uint32_t handle, int time, int session);
7  void skynet_updatetime(void);
8  uint32_t skynet_starttime(void);
9  
10 void skynet_timer_init(void);
11 
12 #endif
```

z:\game\ff_server\skynet\skynet-src\skynet.h

```
1  #ifndef SKYNET_H
2  #define SKYNET_H
3
4  #include "skynet_malloc.h"
5
6  #include <stddef.h>
7  #include <stdint.h>
8
9  #define PTYPE_TEXT 0
10 #define PTYPE_RESPONSE 1
11 #define PTYPE_MULTICAST 2
12 #define PTYPE_CLIENT 3
13 #define PTYPE_SYSTEM 4
14 #define PTYPE_HARBOR 5
15 #define PTYPE_SOCKET 6
16 // read lualib/skynet.lua examples/simplemonitor.lua
17 #define PTYPE_ERROR 7
18 // read lualib/skynet.lua lualib/mqueue.lua lualib/snax.lua
19 #define PTYPE_RESERVED_QUEUE 8
20 #define PTYPE_RESERVED_DEBUG 9
21 #define PTYPE_RESERVED_LUA 10
22 #define PTYPE_RESERVED_SNAX 11
23
24 #define PTYPE_TAG_DONTCOPY 0x10000
25 #define PTYPE_TAG_ALLOCSESSION 0x20000
26
27 struct skynet_context;
28
29 void skynet_error(struct skynet_context * context, const
   char *msg, ...);
30 const char * skynet_command(struct skynet_context * context,
   const char * cmd , const char * parm);
31 uint32_t skynet_queryname(struct skynet_context * context,
   const char * name);
32 int skynet_send(struct skynet_context * context, uint32_t
   source, uint32_t destination , int type, int session, void *
   msg, size_t sz);
33 int skynet_sendname(struct skynet_context * context,
   uint32_t source, const char * destination , int type, int
   session, void * msg, size_t sz);
34
```

```
35 int skynet_isremote(struct skynet_context *, uint32_t
   handle, int * harbor);
36
37 typedef int (*skynet_cb)(struct skynet_context * context, void
   *ud, int type, int session, uint32_t source , const void *
   msg, size_t sz);
38 void skynet_callback(struct skynet_context * context, void
   *ud, skynet_cb cb);
39
40 uint32_t skynet_current_handle(void);
41 uint64_t skynet_now(void);
42 void skynet_debug_memory(const char *info);     // for debug
   use, output current service memory to stderr
43
44 #endif
```

z:\game\ff_server\skynet\skynet-src\socket_epoll.h

```
1  #ifndef poll_socket_epoll_h
2  #define poll_socket_epoll_h
3
4  #include <netdb.h>
5  #include <unistd.h>
6  #include <sys/epoll.h>
7  #include <sys/types.h>
8  #include <sys/socket.h>
9  #include <netinet/in.h>
10 #include <arpa/inet.h>
11 #include <fcntl.h>
12
13 static bool
14 sp_invalid(int efd) {
15     return efd == -1;
16 }
17
18 static int
19 sp_create() {
20     return epoll_create(1024);
21 }
22
23 static void
24 sp_release(int efd) {
25     close(efd);
26 }
27
28 static int
29 sp_add(int efd, int sock, void *ud) {
30     struct epoll_event ev;
31     ev.events = EPOLLIN;
32     ev.data.ptr = ud;
33     if (epoll_ctl(efd, EPOLL_CTL_ADD, sock, &ev) == -1) {
34         return 1;
35     }
36     return 0;
37 }
38
39 static void
```

```
40 sp_del(int efd, int sock) {
41     epoll_ctl(efd, EPOLL_CTL_DEL, sock , NULL);
42 }
43 
44 static void
45 sp_write(int efd, int sock, void *ud, bool enable) {
46     struct epoll_event ev;
47     ev.events = EPOLLIN | (enable ? EPOLLOUT : 0);
48     ev.data.ptr = ud;
49     epoll_ctl(efd, EPOLL_CTL_MOD, sock, &ev);
50 }
51 
52 static int
53 sp_wait(int efd, struct event *e, int max) {
54     struct epoll_event ev[max];
55     int n = epoll_wait(efd , ev, max, -1);
56     int i;
57     for (i=0;i<n;i++) {
58         e[i].s = ev[i].data.ptr;
59         unsigned flag = ev[i].events;
60         e[i].write = (flag & EPOLLOUT) != 0;
61         e[i].read = (flag & EPOLLIN) != 0;
62     }
63 
64     return n;
65 }
66 
67 static void
68 sp_nonblocking(int fd) {
69     int flag = fcntl(fd, F_GETFL, 0);
70     if ( -1 == flag ) {
71         return;
72     }
73 
74     fcntl(fd, F_SETFL, flag | O_NONBLOCK);
75 }
76 
77 #endif
```

z:\game\ff_server\skynet\skynet-src\socket_kqueue.h

```
1  #ifndef poll_socket_kqueue_h
2  #define poll_socket_kqueue_h
3
4  #include <netdb.h>
5  #include <unistd.h>
6  #include <fcntl.h>
7  #include <sys/event.h>
8  #include <sys/types.h>
9  #include <sys/socket.h>
10 #include <netinet/in.h>
11 #include <arpa/inet.h>
12
13 static bool
14 sp_invalid(int kfd) {
15     return kfd == -1;
16 }
17
18 static int
19 sp_create() {
20     return kqueue();
21 }
22
23 static void
24 sp_release(int kfd) {
25     close(kfd);
26 }
27
28 static void
29 sp_del(int kfd, int sock) {
30     struct kevent ke;
31     EV_SET(&ke, sock, EVFILT_READ, EV_DELETE, 0, 0, NULL);
32     kevent(kfd, &ke, 1, NULL, 0, NULL);
33     EV_SET(&ke, sock, EVFILT_WRITE, EV_DELETE, 0, 0, NULL);
34     kevent(kfd, &ke, 1, NULL, 0, NULL);
35 }
36
37 static int
38 sp_add(int kfd, int sock, void *ud) {
39     struct kevent ke;
```

```
40      EV_SET(&ke, sock, EVFILT_READ, EV_ADD, 0, 0, ud);
41      if (kevent(kfd, &ke, 1, NULL, 0, NULL) == -1) {
42          return 1;
43      }
44      EV_SET(&ke, sock, EVFILT_WRITE, EV_ADD, 0, 0, ud);
45      if (kevent(kfd, &ke, 1, NULL, 0, NULL) == -1) {
46          EV_SET(&ke, sock, EVFILT_READ, EV_DELETE, 0, 0, NULL);
47          kevent(kfd, &ke, 1, NULL, 0, NULL);
48          return 1;
49      }
50      EV_SET(&ke, sock, EVFILT_WRITE, EV_DISABLE, 0, 0, ud);
51      if (kevent(kfd, &ke, 1, NULL, 0, NULL) == -1) {
52          sp_del(kfd, sock);
53          return 1;
54      }
55      return 0;
56  }
57
58  static void
59  sp_write(int kfd, int sock, void *ud, bool enable) {
60      struct kevent ke;
61      EV_SET(&ke, sock, EVFILT_WRITE, enable ? EV_ENABLE :
    EV_DISABLE, 0, 0, ud);
62      if (kevent(kfd, &ke, 1, NULL, 0, NULL) == -1) {
63          // todo: check error
64      }
65  }
66
67  static int
68  sp_wait(int kfd, struct event *e, int max) {
69      struct kevent ev[max];
70      int n = kevent(kfd, NULL, 0, ev, max, NULL);
71
72      int i;
73      for (i=0;i<n;i++) {
74          e[i].s = ev[i].udata;
75          unsigned filter = ev[i].filter;
76          e[i].write = (filter == EVFILT_WRITE);
77          e[i].read = (filter == EVFILT_READ);
78      }
79
80      return n;
```

```
81 }
82
83 static void
84 sp_nonblocking(int fd) {
85     int flag = fcntl(fd, F_GETFL, 0);
86     if ( -1 == flag ) {
87         return;
88     }
89
90     fcntl(fd, F_SETFL, flag | O_NONBLOCK);
91 }
92
93 #endif
```

z:\game\ff_server\skynet\skynet-src\socket_poll.h

```
1  #ifndef socket_poll_h
2  #define socket_poll_h
3
4  #include <stdbool.h>
5
6  typedef int poll_fd;
7
8  struct event {
9      void * s;
10     bool read;
11     bool write;
12 };
13
14 static bool sp_invalid(poll_fd fd);
15 static poll_fd sp_create();
16 static void sp_release(poll_fd fd);
17 static int sp_add(poll_fd fd, int sock, void *ud);
18 static void sp_del(poll_fd fd, int sock);
19 static void sp_write(poll_fd, int sock, void *ud, bool
   enable);
20 static int sp_wait(poll_fd, struct event *e, int max);
21 static void sp_nonblocking(int sock);
22
23 #ifdef __linux__
24 #include "socket_epoll.h"
25 #endif
26
27 #if defined(__APPLE__) || defined(__FreeBSD__) ||
   defined(__OpenBSD__) || defined (__NetBSD__)
28 #include "socket_kqueue.h"
29 #endif
30
31 #endif
```

z:\game\ff_server\skynet\skynet-src\socket_server.h

```
1  #ifndef skynet_socket_server_h
2  #define skynet_socket_server_h
3
4  #include <stdint.h>
5
6  #define SOCKET_DATA 0
7  #define SOCKET_CLOSE 1
8  #define SOCKET_OPEN 2
9  #define SOCKET_ACCEPT 3
10 #define SOCKET_ERROR 4
11 #define SOCKET_EXIT 5
12 #define SOCKET_UDP 6
13
14 struct socket_server;
15
16 struct socket_message {
17     int id;
18     uintptr_t opaque;
19     int ud; // for accept, ud is new connection id ; for data,
   ud is size of data
20     char * data;
21 };
22
23 struct socket_server * socket_server_create();
24 void socket_server_release(struct socket_server *);
25 int socket_server_poll(struct socket_server *, struct
   socket_message *result, int *more);
26
27 void socket_server_exit(struct socket_server *);
28 void socket_server_close(struct socket_server *, uintptr_t
   opaque, int id);
29 void socket_server_shutdown(struct socket_server *,
   uintptr_t opaque, int id);
30 void socket_server_start(struct socket_server *, uintptr_t
   opaque, int id);
31
32 // return -1 when error
33 int64_t socket_server_send(struct socket_server *, int id,
   const void * buffer, int sz);
34 void socket_server_send_lowpriority(struct socket_server *,
```

```
int id, const void * buffer, int sz);

// ctrl command below returns id
int socket_server_listen(struct socket_server *, uintptr_t opaque, const char * addr, int port, int backlog);
int socket_server_connect(struct socket_server *, uintptr_t opaque, const char * addr, int port);
int socket_server_bind(struct socket_server *, uintptr_t opaque, int fd);

// for tcp
void socket_server_nodelay(struct socket_server *, int id);

struct socket_udp_address;

// create an udp socket handle, attach opaque with it . udp socket don't need call socket_server_start to recv message
// if port != 0, bind the socket . if addr == NULL, bind ipv4 0.0.0.0 . If you want to use ipv6, addr can be "::" and port 0.
int socket_server_udp(struct socket_server *, uintptr_t opaque, const char * addr, int port);
// set default dest address, return 0 when success
int socket_server_udp_connect(struct socket_server *, int id, const char * addr, int port);
// If the socket_udp_address is NULL, use last call socket_server_udp_connect address instead
// You can also use socket_server_send
int64_t socket_server_udp_send(struct socket_server *, int id, const struct socket_udp_address *, const void *buffer, int sz);
// extract the address of the message, struct socket_message * should be SOCKET_UDP
const struct socket_udp_address *
socket_server_udp_address(struct socket_server *, struct socket_message *, int *addrsz);

struct socket_object_interface {
    void * (*buffer)(void *);
    int (*size)(void *);
    void (*free)(void *);
};

// if you send package sz == -1, use soi.
void socket_server_userobject(struct socket_server *, struct
```

```
socket_object_interface *soi);

#endif
```

z:\game\ff_server\skynet\skynet-src\spinlock.h

```
#ifndef SKYNET_SPINLOCK_H
#define SKYNET_SPINLOCK_H

#define SPIN_INIT(q) spinlock_init(&(q)->lock);
#define SPIN_LOCK(q) spinlock_lock(&(q)->lock);
#define SPIN_UNLOCK(q) spinlock_unlock(&(q)->lock);
#define SPIN_DESTROY(q) spinlock_destroy(&(q)->lock);

#ifndef USE_PTHREAD_LOCK

struct spinlock {
    int lock;
};

static inline void
spinlock_init(struct spinlock *lock) {
    lock->lock = 0;
}

static inline void
spinlock_lock(struct spinlock *lock) {
    while (__sync_lock_test_and_set(&lock->lock,1)) {}
}

static inline int
spinlock_trylock(struct spinlock *lock) {
    return __sync_lock_test_and_set(&lock->lock,1) == 0;
}

static inline void
spinlock_unlock(struct spinlock *lock) {
    __sync_lock_release(&lock->lock);
}

static inline void
spinlock_destroy(struct spinlock *lock) {
    (void) lock;
}

#else
```

```
41
42 #include <pthread.h>
43
44 // we use mutex instead of spinlock for some reason
45 // you can also replace to pthread_spinlock
46
47 struct spinlock {
48     pthread_mutex_t lock;
49 };
50
51 static inline void
52 spinlock_init(struct spinlock *lock) {
53     pthread_mutex_init(&lock->lock, NULL);
54 }
55
56 static inline void
57 spinlock_lock(struct spinlock *lock) {
58     pthread_mutex_lock(&lock->lock);
59 }
60
61 static inline int
62 spinlock_trylock(struct spinlock *lock) {
63     return pthread_mutex_trylock(&lock->lock) == 0;
64 }
65
66 static inline void
67 spinlock_unlock(struct spinlock *lock) {
68     pthread_mutex_unlock(&lock->lock);
69 }
70
71 static inline void
72 spinlock_destroy(struct spinlock *lock) {
73     pthread_mutex_destroy(&lock->lock);
74 }
75
76 #endif
77
78 #endif
```