# Relazione secondo progetto 2016 **MemBox**

Matteo Giorgi & Quarta Andrea

05/01/2016

# Indice

# Capitolo 1

# Struttura codice

Il progetto **MemBox** implementa un server che gestisce un repository contenente generici oggetti (nel caso specifico *char**) biunivocamente associati a generiche chiavi (qui *unsigned long*).

Il server (**main**), dopo aver creato e resa operativa una socket passiva AF_UNIX con le quattro syscall *socket()*, *bind()*, *listen()* e *accept()*, crea un pool di thread (**lavoratori**) che si occuperanno di eseguire sul repository le operazioni richieste dai vari clienti che si connettono alla peer-socket.

## 1.1 Parser file di configurazione

Il server deve conoscere i parametri necessari per accendere la socket e creare il pool dei lavoratori: questi vengono estrapolati da uno dei file di configurazione presenti nella sottodirectoy **/DATA** della directory corrente con la funzione parser. La funzione apre il file corrispondente alla stringa passatagli come argomento, scandisce ciascuna riga, escludendo quelle marcate con il carattere #, quelle vuote o contenenti soli spazi. Delle righe rimanenti controlla se contengono un solo = (0 o più di due = creerebbero ambiguità), taglia dunque i caratteri spazio in testa ed in coda alle due sottostringhe separate dall'= e salva il valore trovato (sottostringa alla destra dell'=) in una struttura appositamente creata, contenente i nomi dei parametri standard.

```
struct config config_vars = {
    .nomi = {
        /* 0 */ [UNIXPATH] = "UnixPath",
        /* 1 */ [STAT_FILE_NAME] = "StatFileName",
        /* 2 */ [MAX_CONNECTIONS] = "MaxConnections",
        /* 3 */ [THREADS_IN_POOL] = "ThreadsInPool",
        /* 4 */ [STORAGE_SIZE] = "StorageSize",
        /* 5 */ [STORAGE_BYTE_SIZE] = "StorageByteSize",
        /* 6 */ [MAX_OBJ_SIZE] = "MaxObjSize",
        /* 7 */ [THE_END] = NULL
    },
    .valori = {
        /* 0 */ [UNIXPATH] = (char*) alloca(M*sizeof(char)),
        /* 1 */ [STAT_FILE_NAME] = (char*) alloca(M*sizeof(char)),
        /* 2->6 */ [MAX_CONNECTIONS ... MAX_OBJ_SIZE] = (char*)(long) 0,
        /* 7 */ [THE_END] = NULL
    }
};
```

Sopra è riportata la struttura delle variabili di configurazione *config_vars* allocata nello stack del **main** e passata per indirizzo al parser così che la riempia. Da notare che le stringhe in *config_vars.valori* sono allocate staticamente con *alloca()* così da ottenere una struttura interamente statica che non necessita di essere liberata (come specificato in *man*, usando noi *gcc* con il flag *-std=c99*, si è incluso l'header *alloca.h* per evitare comportamenti inadeguati a tempo di compilazione). Si noti inoltre che gli array *config_vars.nomi* e *config_vars.valori* sono null-terminated perchè di più facile utilizzo.

Il parser adotta infine una politica piuttosto flessibile riguardo i nomi delle variabili da scandire: se si trovasse un nome di variabile non presente in *config_vars.nomi* questo, assieme al suo corrispondente valore scansionato, verrà salvato in una struttura di tipo *struct config* poi

restituita dalla funzione parser. Il server potrà quindi utilizzare questi valori extra come meglio crede.

## 1.2 Strutture utilizzate per la comunicazione fra thread

Nel file *membox.h* sono state inserite diverse variabili e strutture utitli al server:

**lavoratore_t**: i thread-lavoratore che si occuperanno di rispondere ai clienti sono stati implementati come una struttura contenente il proprio tid, valore di ritorno e un flag booleano indicante lo stato.

**array_lavoratori**: i thread-lavoratore sono contenuti in un array globale allocato dinamicamente così che i singoli lavoratori siano rappresentabili con l'indice di posizione nell'array.

**coda_cfd**: i file descriptor delle peer-socket ottenuti dalla syscall *accept* vengono messi dal main in una coda alla quale accederanno in mutua esclusione i lavoratori per estrarre una connessione ed eseguire le operazioni che il cliente scrive.

**stop_sigint**: flag che segnala l'arrivo di un **SIGINT**, **SIGQUIT** o **SIGTERM**. La variabile, inizialmente 0, verrà settata ad 1 dal thread-gestore dei segnali e verrà osservata sia dal main all'inizio del *for(;;)* dove compie le *accept*, sia dai lavoratori all'inizio del loro *for(;;)* principale dove estraggono connessioni dalla coda e nel *for(;;)* secondario dove eseguono le richieste del cliente.

**stop_sigusr2**: flag analogo a **stop_sigint** ma che segnala l'arrivo di un **SIGUSR2**.Il funzionamento è il medesimo del precedente, ad eccezione che non viene controllato dai clienti nel *for(;;)* secondario.

**operativi**: variabile globale che indica il numero di lavoratori attivi. La variabile servirà per capire se occorre rifiutare una connessione ottenuta dalla accept in caso il numero di connessioni in coda superi MaxConnections-ThreadsInPool.

**mboxStats**: struttura contenente le variabili per le statistiche.

**lock**: variabile di mutua esclusione usata dai thread del processo per accedere alle strutture condivise quali **coda_cfd**, **stop_sigint**, **stop_sigusr2**, **operativi**, **mboxStats**.

**dormi**: variabile di condizione usata in coppia con **lock**, impiegata dai lavoratori per sospendersi in caso trovino la **coda_cfd** vuota.

## 1.3 Arbitro

Importante è menzionare il ruolo di arbitro che il **main** ha nei confronti dei lavoratori: egli, dopo aver creato il pool, compie un *for(;;)* nel quale controlla le variabili **stop_sigint** e **stop_sigusr2** e si mette in ascolto sulla *accept()*. Ottenuto il file descriptor della peer-socket lo deve inserire in **coda_cfd**:

```
/* ========== Lock ========== */
if( ris=pthread_mutex_lock(&lock) )
    err_exit_en(ris, "lock fallita");
    if( (mcn==tip && operativi==tip) || !insert(coda_cfd, cfd, FALSE) ){
        op_t op = OP_FAIL;
        membox_key_t key = 0;
        for(ur=sizeof(int), ptr=&op;
                (wr=write(cfd, ptr, ur))<ur; ur-=wr, ptr+=wr)
            if(wr<=0) err_exit("errore in scrittura");
        for(ur=sizeof(unsigned long), ptr=&key;
                (wr=write(cfd, ptr, ur))<ur; ur-=wr, ptr+=wr)
            if(wr<=0) err_exit("errore in scrittura");
        if( (ris=close(cfd))<0 )
            err_exit_en(ris, "close fallita");
        /* ========== Unlock ========== */
        if( ris=pthread_mutex_unlock(&lock) )
            err_exit_en(ris, "unlock fallita");
        continue;
    }
    else pthread_cond_signal(&dormi);
/* ========== Unlock ========== */
if( ris=pthread_mutex_unlock(&lock) )
    err_exit_en(ris, "unlock fallita");
```

Come si evince dal frammento, il `main`, prima di inserire in coda controlla che, nel caso i lavoratori siano tutti attivi, il numero di connessioni nella coda non superi MaxConnections-ThreadsInPool (come riportato nelle specifiche del progetto). In caso di fallimento egli scriverà subito il messaggio di risposta al cliente, chiuderà la peer-socket ed inizierà un nuovo ciclo, diversamente sveglierà un lavoratore sospeso sulla variabile di condizione `dormi`.

## 1.4 Lavoratore

Come preannunciato il compito di seguire le operazioni richieste dal cliente spetta ai thread lavoratori. Essi compiono un *for(;;)* dove inizialmente modificano il proprio stato e la variabile `operativi`, controllano le variabili `stop_sigint` e `stop_sigusr2` ed estraggono una connessione da `coda_cfd`. In caso di fallimento si sospendono sulla variabile `dormi`, altrimenti iniziano un secondo *for(;;)* dove leggono un *message_hdr_t* ed un *message_data_t* dalla peer-socket ed entrano in uno *switch* dove eseguono l'operazione richiesta. Il lavoratore uscirà da questo secondo *for(;;)* quando fallirà la lettura della *message_hdr_t* tornando in testa al primo *for(;;)* per iniziare un nuovo ciclo.

## 1.5 Segnali

La gestione dei segnali è stata realizzata senza ricorrere ad un signal handler, ma creando un thread-gestore dei segnali che si mette in attesa sulla syscall *sigwaitinfo()* dei segnali `SIGINT`, `SIGQUIT`, `SIGTERM`, `SIGPIPE`, `SIGUSR1`, `SIGUSR2` già mascherati prima della sua creazione dal main.

Il gestore, in base al segnale ricevuto tra `SIGINT`, `SIGQUIT`, `SIGTERM`, `SIGUSR2`, modifica una delle due variabili globali *stop_sigint* o *stop_sigusr2* settandola a 1, chiude la socket facendo fallire la *accept()* sulla quale è eventualmente sospeso il `main` che dunque inizierà un nuovo ciclo, per poi terminare. Con questa operazione il main ed i lavoratori attivi controlleranno in mutua escusione le variabili *stop_sigint* e *stop_sigusr2* e quindi usciranno dal loro ciclo principale: i lavoratori attivi terminano ed attendono la *join()* dal main, mentre quest'ultimo risveglierà con una *pthread_cond_broadcast()* tutti i lavoratori sospesi sulla variabile di condizione `dormi` che controlleranno a loro volta le variabili *stop_sigint* e *stop_sigusr2* e termineranno.

Al main non resta dunque che compiere la *join()* su lavoratori e gestore, invocare la *pthread_cleanup_pop()* che lancerà la funzione *spazino()* (dichiarata in *membox.h*) per pulire la memoria e cancellare la socket.

# Capitolo 2

# Librerie

## 2.1 Repository e Lettori-Scrittori-Bloccatori

La scelta fatta prevede di poter utilizzare un qualsiasi repository, da qui l'idea di implementare il protocollo di accesso con una libreria del tipo lettori-scrittori ed una unica lock per tutto il repository.

Le operazioni richieste nelle specifiche sono state suddivise in operazioni di lettura (GET_OP), o di scrittura (PUT_OP), o di lettura/scrittura (UPDATE_OP e REMOVE_OP) o di bloccaggio (LOCK_OP).

Il lavoratore dovrà quindi richiedere di poter leggere, scrivere o bloccare il repository in base all'operazione che deve compiere con le funzioni lockRepo(), unlockRepo(), startRead(), doneRead(), startWrite(), doneWrite() presenti nella libreria *read_write.h*.

Questa scelta permette di poter utilizzare una qualsiasi libreria per l'implementazione del repository ed allo stesso tempo permette una discreta concorrenza fra i vari thread. In particolare, ai fini di rendere tutto il più generico possibile è stata creata una struttura repository

```
struct repository{
    void *archivio;
    void* (*crea_repo) (int, unsigned int (*funzione_hash) (void*),
              int (*compara) (void*, void*));
    int (*distruggi_repo) (void*, void (*libera_key)(void*),
                  void (*libera_data)(void*));
    void* (*cerca) (void*, void*);
    void* (*inserisci) (void*, void*, void*);
    void* (*aggiorna) (void*, void*, void*, void**);
    int (*cancella) (void*, void*, void (*libera_key) (void*),
              void (*libera_data) (void*));
    unsigned int (*funzione_hash) (void*);
    int (*compara) (void*, void*);
    void (*libera_key)(void*);
    void (*libera_data)(void*);
} repo;
```

che contiene il puntatore alla struttura e tutte le funzioni necessarie per il suo utilizzo.

## 2.2 Gestione errori

La trattazione degli errori è stata affrontata con una libreria apposita che permette un utilizzo eclettico ed efficiente delle funzioni di errore. Nella libreria *errors.h* è presente un elenco dei possibili valori di errno (sottoforma di array di stringhe) e delle funzioni per la terminazione con *exit()*, *_exit* o *abort()* con opzioni come la stampa di un messaggio di errore da parte dell'utente e la possibilità di flushare lo stderr. Una particolarità è rappresentata dalla possibilità di abortire il processo

```
static void
termina( bool exit_3 )
{
    char *s;
    if((s=getenv("EF_DUMPCORE")) && *s!='\0') abort();
```

```
        if(exit_3) exit(EXIT_FAILURE);
        else _exit(EXIT_FAILURE);
    }
```

settando una variabile di ambiente fittizia.

## 2.3 Code

La libreria delle code (*queue.h*), necessaria ai fini del server, implementa una coda di interi con un array circolare ridimensionabile.

Viene mantenuto un puntatore al primo e all'ultimo elemenro della coda limitandosi a spostarli in caso di inserimento o estrazione: questo metodo risulta assai efficiente (rispetto ad una classica implementazione con lista) perchè non necessita di allocazioni multiple di memoria.

Una particolarità (non utilizzata nel server) sta nel fatto che l'utilizatore può scegliere di inserire forzatamente un elemento anche quando la coda è piena: nella funzione di inserimento c'è un flag booleono preposto allo scopo.