Net Raid 1 File System
გასაშვები ბრძანებები

პროექტის კომპილაცია:
make
სერვერის გაშვება:
./lux_server 127.0.0.1 10001 ./server1_dir
(make server1/server2/hotswap)
კლიენტის გაშვება:
./lux_client ./config
(make client)

პროექტის არქიტექტურა

კლიენტის გაშვებისას მიმდინარეობს კონფიგურაციის ფაილის პარსინგი. საიდან მიღებული მონაცემებით ვავსებთ client_info სტრუქტურას:

```
struct client_info {
  char path_to_error_log[256];
  unsigned long long cache_size;
  int cache_replacement;
  int timeout;
  int storage_count;
  struct storage_info* storages[24];

} _lux_client_info;
```

იქმნება ამ სტრუქტურის ერთი ინსტანცია, რომელიც ინახავს ზოგად ინფორმაციას: ლოგირების ფაილის მისამართს, timeout-ს.
შემდგომ fork()-ით იქმნება ახალი პროცესები. თითო storage-ზე თითო.
run_storage_raid_one() ფუნქციას გადაეცემა storage_info სტრუქტურა, რომელიც ინახავს სანახის სამართავად საჭირო ინფორმაციას:

```
 struct storage_info {
   char storage_name[256];          //სანახის სახელი
   char mountpoint[256];            //დასამაუნტებელი დირექტორიის path
   int raid;                        //RAID_ONE ან RAID_FIVE
   int server_count;                //მომსახურე სერვერების რაოდენობა (2). hotswap-ის
                                    //გარეშე
   struct lux_server* servers[24]; //სერვერების აღწერები
   struct lux_server* hotswap;      //hotswap სერვერის აღწერა
 };


struct lux_server {
  char server_ip[32];
  uint16_t port;
  time_t fail_time;
  int status;
  int socket_fd;
  struct sockaddr_in server_adress;
};
```

ძირითადი ფუნქციები და სტრუქტურები

```
int get_server_fd(struct lux_server *server)
```

ფუნქცია უკავშირდება სერვერს, რომლის მისამართი აღწერილია გადაცემულ სტრუქტურაში და აბრუნებს file descriptor-ს სოკეტზე წარმატების შემთხვევაში. სოკეტის დახურვა გამომძახებელს ევალება.
დაკავშირების წარუმატებლობის შემთხვევაში აბრუნებს -1-ს.

```
int get_live_server_fd()
```

აბრუნებს სოკეტის file descriptor-ს, რომელიც დაკავშირებული იქნება რომელიმე “ცოცხალ” სერვერთან. ინდექსით პირველი სერვერს პრიორიტეტი აქვს.
თუ ვერც ერთ სერვერთან ვერ მოხერხდა დაკავშირება აბრუნებს -1;

```
struct raid_one_live_sockets get_live_sockets(struct lux_server *servers[2])
```

დაბრუნებული სტრუქტურა შეიცავს file descriptor-ებს ორივე სერვერზე, თუ მოხერხდა მათთან დაკავშირება.

```
typedef struct raid_one_live_sockets {
  int count;
  int sock_fd[2];
} server_sockets;
```

count აღნიშნავს რამდენ სერვერთან მოხერხდა წარმატებით დაკავშირება.

```
struct raid_one_input generate_server_input(
  int command,
  const char *path,
  const char *char_buf,
  off_t offset,
  size_t size,
  mode_t mode,
  dev_t dev,
  const struct timespec *spec,
  int flags)
```

აგენერირებს სტრუქტურას (იგივე ველებით,რაც ამ ფუნქციას გადაეცემა), რომლის დანიშნულებაა სერვერისთვის აღწეროს შესასრულებელი დავალება. Command-ის შესაძლო ვარიანტები აღწერილია lux_commons.h ფაილში და უნდა იყოს შემდგომთაგან ერთერთი: DUMMY, GETATTR, READDIR, OPEN, READ, ACCESS, WRITE, TRUNCATE, RENAME, UNLINK, RMDIR, CREATE, UTIMENS, MKDIR. ფაილური სისტემის სისქოლების შესაბამისად. DUMMY აღწერს დასაიგნორებელ დავალებას.

```
int handle_error(int sock_fd, const char *path, int command);
int handle_return(int sock_fd, int error, const char *path, int command);
int handle_errors(struct raid_one_live_sockets socks, const char *path);
int handle_returns(struct raid_one_live_sockets socks, struct raid_one_response responses[2], const char *path)
```

ეს ფუნქციები უზრუნველყოფენ, რომ ფაილური სისტემის სისქოლის შესრულების შემდგომ დაიხუროს სოკეტები, ასევე იმას, რომ სისქოლებმა დააბრუნონ შესაბამისი სტატუსები შემდეგნაირად.
თუ სერვერზე შეცდომა მოხდა, გამოძახება დააბრუნებს სერვერის errno-ს, თუ კლიენტის მხარეს მოხდა შეცდომა დააბრუნებს კლიენტის errno-ს. სხვა შემთხვევაში 0-ს (read-ის შემთხვევაში size-ს).

```
int copy_file(struct lux_server *from, struct lux_server *to, const char path[256]);
```
აკოპირებს path-ით მითითებულ ფაილს სერვერ from-დან to-ში.

```
struct raid_one_response {
struct stat one_stat;
unsigned char recorded_hash[16];
unsigned char checked_hash[16];
int size;
int error;
};

struct raid_one_directories_response {
char filenames[32][32];
struct stat stats[32];
int size;
int error;
};
```

კლიენტ-სერვერის ურთიერთობა

ფაილურ სისტემაზე სისქოლის გამოძახებისას კლიენტი იღებს სერვერის(**getattr**, **readdir**, **read** ან **access**-ის შემთხვევაში) ან სერვერების(**open**, **write**, **truncate**, **rename**, **unlink**, **rmdir**, **create**, **utimens** ან **mkdir**-ის შემთხვევაში) სოკეტებს, შესაბამისად **get_live_server_fd()** ან **get_live_sockets()** გამოყენებით.
შემდგომ აგენერერირებს **raid_one_input** სტრუქტურას და უგზავნის სერვერს ან სერვერებს.
სერვერიდან ტასკის შესრულების შემდეგ ბრუნდება **raid_one_response** სტრუქტურა, რომელიც შეიცავს **stat** სტრუქტურას, რომელსაც **getattr** იყენებს.
Recorded_hash და **checked_hash**, შესაბამისად, შეიცავენ ფაილზე **write**-ს დროს მიღებულ ჰეშს და **open**-ის დროს ხელახლა გადათვლილ ჰეშს. მათი შედარების შემდგომ open აღადგენს დაზიანებულ ფაილს.
Error შეიცავს სერვერზე დავალების დამუშავებისას შეცდომის დაფიქსირების შემთხვევაში არსებულ **errno**-ს ან **0**-ს თუ წარმატებით შესრულდა დავალება.
Read რესფონსის მიღების შემდეგ კიდევ ერთ შეტყობინებას ელოდება, რომელიც ფაილის მონაცემებს მიიღებს.
Write რესფონსის მიღებამდე კიდევ ერთ შეტყობინებას აგზავნის, რომელიც ფაილის მონაცემებს შეიცავს.
Readdir რესფონსად იღებს **raid_one_directories_response** სტრუქტურას, რომელიც შეიცავს წაკითხული დირექტორიის ფაილების სახელებს და **stat**-ებს.
Open დაზიანებული ფაილის აღმოჩენის შემთხვევაში აკოპირებს სწორ ფაილს ერთი სერვერიდან და მისით ანაცვლებს მეორე სერვერზე დაზიანებულს **copy_file()** ფუნქციის მეშვეობით.

სერვერებთან კავშირის მართვა

ფუნქციები

```
int monitor_server(int server_index);
```
ფუნქცია უშვებს thread-ს, რომელიც გააკონტროლებს სერვერთან კავშირს, რომელის ინდექსი storage_info სტრუქტურაში არის server_index;

```
void *monitor_routine(void *args);

int copy_server_contents(struct lux_server *from, struct lux_server *to);
```
ფუნქციას გადააქვს ერთი სერვერის ყველა ფაილი და დირექტორია მეორე სერვერზე.

```
int copy_routine(char *p, struct lux_server *from, struct lux_server *to);
```
ფუნქცია გადაიტანს ერთი სერვერის ერთი დირექტორიის ყველა ფაილს მეორე სერვერზე. ყოველ დირექტორიაზე რეკურსიულად გამოიძახება.

არქიტექტურა

კლიენტის გაშვებისას ორივე სერვერისთვის გაეშვება **monitor_server()** ფუნქციას, რომელიც თავისებურად ახალ ნაკადში უშვებს **monitor_routine()** ფუნქციას. ეს ფუნქცია ტრიალებს ციკლში. 1 წამში ერთხელ ეცდება დაუკავშირდეს სერვერს.
თუ სერვერთან გაწყდა კავშირი, სერვერს სტატუს ეცვლება **STATUS_ALIVE**-დან **STATUS_DEGRADED**. შემდგომი timeout წამის განმავლობაში ფაილური სისტემის ქოლები არ ცდიან დეგრადირებულ სისტემასთან დაკავშირებას.
თუ კავშირი **timeout**-ზე მეტი ხანი ვერ აღდგება. დარჩენილი სერვერიდან მონაცემები გადაკოპირდება **hotswap** სერვერზე, შემდეგ ეს სერვერი ჩაანაცვლებს დაკარგულ სერვერს **storage_info** სტრუქტურაში.
თუ კავშირი დროულად აღდგება, ხდება მონაცემების აღდგენა მომუშავე სერვერიდან.
თუ რაღაც მომენტში არც ერთი სერვერი არაა მისაწვდომი კლიენტის პროგრამა დაასრულებს მუშაობას.

სინქრონიზაცია

სერვერთან კავშირის მართვა

```
sem_t replication_defender;
int copier_demands_freedom;
int pending_requests;
```

ეს ველები უზრუნველყოფენ კოპირების სისწორეს. როდესაც დადგება ერთი სერვერიდან მეორეზე მონაცემების კოპირების საჭიროება, მნიშვნელოვანია მოქმედებაში არ იყვნენ ფუნქციები, რომლებიც ცვლიან მონაცემებს (open, mkdir, write, rename etc.).
ამიტომ, კოპირებამდე კავშირის მაკონტროლებელი ნაკადი ცვლის სტატუს copier_demands_freedom-ს. ამის შემდეგ ზემოთხსენებული ფუნქციების ყოველი გამოძახება დააბრუნებს -EAGAIN შეცდომას და არ შეუშლის ხელს კოპირებას.
ასევე, ყველა ეს ფუნქცია გამოძახებისას ზრდის pending_requests ქაუნთერს და დასრულების შემდეგ ამცირებს. ამ ველის განულებას ელოდება ნაკადი და შემდგომ დაიწყებს კოპირებას.
კოპირების შემდგომ კვლავ ნულდება სტატუსი copier_demands_freedom, და ამ ფუნქციებს კვლავ შეუძლიათ მუშაობა.
სემაფორა უზრუნველყოფს ამ ველების ცვლის და წვდომის შესაბამის ატომურობას.
ამ დროს სხვა ფუნქციები, რომლებიც არ ცვლიან მონაცემებს (read, getattr etc.) ჩვეულებრივად მუშაობენ.

ფაილური სისტემის გამოძახებები

ბრძანებები, რომლებიც არ ცვლიან მონაცემებს არ საჭიროებენ დამცავ მექანიზმებს. სხვა ბრძანებებზე გათვალისწინებულია შემდგომი პროცესი:
lux_simple_list.c ინახავს მარტივ ნაკადს. ამ ნაკადში ინახება ყველა იმ path-ის ჰეში, რომელზეც კონკრეტულ მომენტში მიმდინარეობს “სახიფათო” პროცედურა.

შედეგად, ერთსა და იმავე path-ზე მიმართვები EAGAIN შეცდომას დააბრუნებს, თუმცა განსხვავებულ ფაილებზე მოქმედება არ გამოიწვევს შეცდომას ან დაყოვნებას. ამ ლისტთან ურთიერთობის ატომურობას უზრუნველყოფს ზემოთხსენებული სემაფორა.

სერვერი

სერვერზე ყოველი კლიენტის ბრძანების დაკავშირების შემდგომ ი-fork-ება და ცალცალკე ემსახურება კლიენტებს. თუმცა აქ არანაირი race condition-ის დამცავი მექანიზმი არ არის. კლიენტის მოვალეობაა სწორი მიმდევრობით აგზავნოს ბრძანებები.

შენიშვნა

ფაილური სისტემის multi-threading-ი აქ არ არის იმპლემენტირებული. თუმცა linux-ზე fuse-ს აქვს თავისი ნაკადებად დაყოფის მექანიზმი. ვაგრანტზე ვერ გავარკვიე იგივენაირადაა თუ მარტო ერთ ნაკადზე მუშაობს, თუმცა, ზემოთ ჩამოთვლილი პროცესების გამო ადვილი დასაიმპლემენტირებელი იქნება, თუკი საჭიროება მოითხოვს.
ასევე, ამიტომ გადავწყვიტე, რომ ყოველ ბრძანებას თავისი სოკეტი გავაკეთებინო და ისე ვეურთიერთო სერვერს, ასე მონაცემების გაცვლა მინიმალურ სინქრონიზაციის კონტროლს საჭიროებს და ეს გადაწყვეტილება ნაკლად არ მიმაჩნია.

ლოგირება

კონფიგურაციაში მითითებულ ფაილში ხდება მოვლენების ლოგირება, ეს მოვლენებია: კონფიგურაციის ფაილის პარსინგის დასრულება.

სერვერთან დაკავშირების წარმატება:
[Wed 2018-08-15 22:33:32] STORAGE1 127.0.0.1:10002 server status check successful.

სერვერთან კავშირის პირველი წარუმატებლობა:
[Thu 2018-08-16 13:08:48] STORAGE1 127.0.0.1:10001 server status check failed.

სერვერთან კავშირის გაგრძელებული წარუმატებლობა:
[Wed 2018-08-15 22:33:32] STORAGE1 127.0.0.1:10001 server status check failing for 5 seconds.

სერვერთან კავშირის საბოლოო გაწყვეტა:
[Thu 2018-08-16 13:08:56] STORAGE1 127.0.0.1:10001 server downtime exceeds timeout limit.

Hotswap-ზე მონაცემების კოპირების პროცესი:
[Thu 2018-08-16 13:08:56] STORAGE1 127.0.0.1:10001 replicating data to hotswap.

სერვერის ჩანაცვლება hotswap-ით
[Thu 2018-08-16 13:08:56] STORAGE1 127.0.0.1:10001 server replaced with hotswap.

სერვერთან კავშირის აღდგენა
[Thu 2018-08-16 13:08:43] STORAGE1 127.0.0.1:10001 connection with server reestablished after 5 seconds.

აღდგენილ სერვერზე მონაცემების კოპირების პროცესი:
[Thu 2018-08-16 13:08:43] STORAGE1 127.0.0.1:10001 restoring data to server.
[Thu 2018-08-16 13:08:43] STORAGE1 127.0.0.1:10001 restoration of data to server complete.

ასევე ლოგირდება ყოველი ფაილური სისტემის ბრძანების გამოძახება და დაბრუნებული სტატუსი.

[Thu 2018-08-16 13:08:43] STORAGE1 getattr called for path /
[Thu 2018-08-16 13:08:43] STORAGE1 getattr call for path / returned with success.
[Thu 2018-08-16 13:08:44] STORAGE1 getattr called for path /.git
[Thu 2018-08-16 13:08:44] STORAGE1 getattr call for path /.git returned with **error:** No such file or directory.