Számítógép labor V.

# Egyszerű Web szerver

Dokumentáció

**Készítette:** Ács Gergely (K4C03M)

2003.04.29

# Egyszerű Web szerver

**Feladat:** Egyszerű Web szerver
**Feladat sorszám:** 17
**Leírás:** Készítsen egy egyszerű Web szervert, amely képes statikus html oldalakat kiszolgálni. A szerver egy konfigurálhatóan megadható könyvtárban elhelyezett html formátumú oldalakat legyen képes egy adott porton keresztül kiszolgálni.

## A szerver használata

A szerver futtatása szokványos UNIX-os környezetben:
`./commanche`
Ezután a szerver démonként fut tovább a háttérben, így a kommunikációja a külvilággal egy opcionálisan megadott konfigurációs állományon keresztül zajlik. A szerver első lépésként feldolgozza a saját könyvtárában levő `commanche.cf` konfigurációs állományt. Ha feldolgozás során hibába ütközik, akkor a saját vagy alapértelmezett értékeit használja, vagy kilép.

**A commanche.cf állományban beállítható paraméterek**:

- `Server <sztring>`
  *Használat:* A <sztring> lesz a szerver azonosító sztringje.
  *Alapérték:* "Commanche v.0.01"

- `MaxPendingConnections <portszám>`
  *Használat:* Megadható vele a még el nem fogadott várakozási sorban levő kérelmek maximális száma.
  *Alapérték:* 10

- `MaxAliveConnections <szám>`
  *Használat:* Megadható vele az egyidejű kapcsolatok maximális száma, vagyis a szálak száma.
  *Alapérték:* 20

- `HTMLRootDirectory "<sztring>"`
  *Használat:* A HTML illetve egyéb objektumok, melyek a klienssel a szerver által elérhetőek.
  *Alapérték:* “./htdocs”

- `Port <szám>`
  *Használat:* A szerver kommunikációs portját adja meg. Érdemes 1024-nél nagyobb de 60000-nél kisebb (Solaris alatt).
  *Alapérték:* “5555”

- `LogFile "<sztring>"`
  *Használat:* A log állomány elérési útja és neve adható meg. Ide kerül minden a

működéssel kapcsolatos (hiba)üzenet.
*Alapérték:* "./commanche.log"

# Hibakezelés

A szerver a GET és HEAD http protokoll parancsokat valósítja meg. A hibaüzenetek egy része lehet a szabványban rögzített hibajelenség, vagy valamilyen súlyosabb rendszerhiba (nincs elég memória, nem található a működéshez szükséges állomány). A rendszerszintű hibákat az `error()` függvény kezeli le, aminek paraméter értékéül még megadható a programból való kilépés is.

**Protokoll hibajelenségek:**

- 400 Bad Request
  Ha a nem értelmezhető a küldött parancs.
- 403 Forbidden
  Az állomány létezik, de a hozzáférés nem engedélyezett.
- 404 Not Found
  A kért objektum nem létezik.
- 501 Not Implemented
  A parancs nincs még implementálva.

**Rendszerszintű hibák:**

- `#define ERR_THREAD 0x01`
  A szálat nem sikerült létrehozni.
- `#define ERR_NOMEM 0x02`
  Nincs elég memória.
- `#define ERR_SEND 0x03`
  Nem sikerült adatot küldeni a socketen keresztül.
- `#define ERR_DETACHED_MODE 0x04`
  A szálat nem sikerült DETACHED módban futtatni.
- `#define ERR_MAXLIMIT 0x05`
  Nem hozható létre több szál.
- `#define ERR_LOG_FILE 0x06`
  Nem sikerült a log file létrehozása.
- `#define ERR_CF_FILE 0x07`
  Nem sikerült a konfigurációs file feldolgozása, vagy megnyitása

# A protokoll ismertetése

A http 1.0 verziójának GET és HEAD parancsa lett megvalósítva. Más parancsok fogadásakor *Bad request* vagy *Not Implemented* hibákat küld a szerver a kliensnek.

**GET parancs:**
A GET parancs különböző objektumok letöltésére szolgáló parancs. Két fajtája lehetséges. Az egyszerű GET parancs csak egy sorból áll, amire válaszul azonnal küldhető a kért entitás. Teljes GET parancs esetén a GET után a kért

entitás URL-je, majd azt követően a protokoll verziószáma kerül definiálásra. A többi sor definiálása opcionális. A GET kérést 2 darab újsor jel zárja. A szerver ezután válaszol kötött formátumú státuszjelentéssel a kért entitásról, majd küldi az entitást ha ez küldhető.

A http válasz formátuma:

```
"HTTP/1.0 %d %s\r\n"//
"Server: Commanche v.0.01\r\n"//
"Content-Type: %s\r\n\r\n"
```

ahol a *%d* jelenti a visszaadott státuszkódot (ha kérés kiszolgálható akkor ennek értéke 200), az első *%s* az üzenetet (ha nincs hiba akkor *OK!* az értéke). Végül utolsó paraméterként adható meg az entitás típusa, ami lehet *text, html, gif, jpeg*.

**HEAD parancs:**
A HEAD protokoll parancs nagyon hasonló a GET-hez. A különbség annyi, hogy a HEAD csak a fent részletezett http választ küldi, magát az entitást semmilyen esetben sem küldi el. Így csak a kért entitás típusáról, elérhetőségéről szerezhetünk információt.

# A szerver felépítése és működése

**Felépítés:**
A szervert két állomány alkotja:

- `main.c`
  Ez tartalmazza a szerver összes függvényeit, maga az implementáció. Itt vannak a függvények prototípusaik is.
- `commanche.h`
  Csak a szerverre jellemző típusdefiniálások, makrók.

Valamint a fordításhoz szükséges `Makefile`. A fordítás a make parancs kiadásával lehetséges, ami előállít egy `commanche` nevű végrehajtható állományt.

Maga a szerver ***szálak*** létrehozásával működik. Ez azt jelenti, hogy minden kérés kiszolgálásához egy külön szálat indít. Ha kérést az adott szál kiszolgálta, akkor megszűnik. A főprogram nem várja meg a szál befejeződését, vagyis nincs szinkronizáció a szálak között (DETACHED módban futnak). A szálak alkalmazásával a szerver gyorsabb, erőforrás takarékosabb, hatékonyabb, mivel az operációs rendszer magja a szálak közötti kontextus váltás során kevesebb adatot mozgat, hiszen az adatszerkezetek közösek.

**Működés:**
A program elején a szerver egy *fork* hívással éri el, hogy démonként fusson tovább. Még a program elején a SIGPIPE blokkolásra került, hogy rossz kommunikáció esetén a szerver ne lépjen ki indokolatlanul. Ezekután a *server_main* függvény elindítja a szervert. Ezután értelmezzük a konfigurációs file-t a *parse_cf_file* függvény segítségével. Ez feltölti a **struct** config_t típusú cf struktúrát a megfelelő szerverbeállításokkal. Ezekután kerül sor a log állomány

megnyitására, és standard kimenet és hibakimenet átirányítására a log file-ba. Lefoglaljuk a memóriát a szálak és azok adatszerkezeteinek számára.

***Szálak adatszerkezete:***

```
struct thread_param_t
{
        // Kapcsolat file-leirója
        int fd;
        // Kapcsolat azonosítója (indexe)
        int idx;
        // Kliens címe
        int addrlen;
        // Állapot (FREE->nem megy a szál)
        volatile uchar state;

        // Ez a közösen használt változó melyet mindegyik szál írhat is
        int volatile *next_free_idx;
        pthread_mutex_t *free_idx_lock;

        // Kliens cime
        struct sockaddr_in remoteaddr;
        // HTTP root könyvtár
        char ht_root[HT_ROOT_SIZE];
        // Log file deszkriptor
        int log_fd;
};
```

***A szálak szinkronizálása:***

Miután egy kapcsolódási kérelem *accept*-el elfogadásra került, a szerver létrehoz egy szálat a *pthread_create* utasítással a kérés kiszolgálására. Az aktív szálak aktivitásukat a saját adatstruktúrájukban a state változóval jelzik (lehet RESERVED vagy FREE értelemszerűen). Ha egy szál lefutott, akkor a *next_free_idx* változót beállítja a saját *idx* változójára. Ezzel jelzi, hogy ő már szabad, valamint beállítja az állapotát FREE-re. Így a főszál mindig a *next_free_idx* változóval jelzett helyre hoz létre új szálat. Ha létrehozott egy szálat, akkor lineáris kereséssel megkeresi a következő szabad bejegyzést. Így ha már nincs szabad bejegyzés, akkor ez így kimutatható. Mivel a *next_free_idx* egy közösen írt és olvasott változó a szálak által, így ennek írását egy *mutex* (bináris szemafor) segítségével végezzük. Ezt jelzi a *free_idx_lock* pointer. Az írása ennek a változónak a *set_free_idx* függvénnyel lehetséges. Így a kölcsönös kizárás erre a változóra biztosított. Az *idx* a *conns_data* (szálak adatszerkezeteinek tömbje) és a thread_serving (szálak tömbje) tömbök indexét jelenti, úgy ahogy a *next_free_idx* is.

***A főszál működése:***

A főszál feladata az előredefiniált porton egy kommunikációs socket létrehozása, és a kapcsolódási kérelmek figyelése ezen a socketen, valamint ezen kérelmek kiszolgálására egy szál elindítása. A *listener* socket létrehozása a *socket* függvénnyel történik. Ezután a *setsockopt* függvénnyel beállítjuk a socket opcióit (socket azonnali elengedése kilépés után). Majd beállítjuk egy **`struct`** `sockaddr_in` struktúrában a szerverünk címét, a kívánt IP protokollt, portot.

Ezekután a rendszercímet hozzárendeljük a sockethez a *bind* függvény segítségével. A *listen* hívással engedélyezzük a socketet, és a függő kapcsolatok maximális számát beállítjuk. Ezután egy végtelen ciklusban várakozunk új kapcsolat felépítésére az *accept* hívással. Mivel az *accept* blokkolva van amíg nincs kapcsolódási kérelem, így CPU polling nem lép fel. Az accept visszaadja az újonnan felépített kapcsolat file-leíróját (socket), amelyet átadva egy szintén ekkor létrhozott szálnak, a kliens kérelmét az új szál lekezeli, majd megszűnik. A szál létrehozása előtt feltöltjük a létrehozandó szál adatstruktúráját, majd megkeressük a következö szál szabad indexét.

***A kérést kezelő szálak működése:***
A kérést kezelő szálak feladata a paraméterként átadott socketen keresztül az érvényes http kérések kiszolgálása, hibaüzenetek küldése, majd kilépés. A *thread_serve* függvény jelenti magát a szálat. Kezdetben lefoglaljuk a bejövő adatoknak a buffert (*in_buf*), majd várjuk a *recv* rendszerhívás segítségével a bejövő adatokat. Ez http protokoll révén egy kérés lesz (GET vagy HEAD), melyet a *get_params* függvény segítségével bontunk fel parancs nevére (*comm_name*), argumentumaira (URL, *comm_args*) és a protokoll azonosítójára (*protocol_id)* ha van ilyen (ebben az esetben beszélünk teljes http kérésről). Ha felbontás nem sikerült, akkor hibát küldünk, majd az erőforrások elengedése után kilépünk. Ezekután feltöltjük a *struct command_t* típusú params struktúrát, amely felépítése a következő:

```
struct command_t
{
        // Egyszerű vagy teljes HTTP kérésről van-e szó (SIMPLE,FULL)
        method_t method;
        // GET vagy HEAD vagy UNSUPPORTED
        procedure_t procedure;

        // Kliens socket
        int client_sck;
        // Kért URL
        char rel_path[COMM_ARGS_SIZE];
        // HTTP Root könyvtár
        char ht_root[HT_ROOT_SIZE];
};
```

A *process_command* eljárás feldolgozza a küldött parancsot, és hibakódot ad vissza a művelet sikerességéről. Ez lehet
- GET_NOFILE: Nem találta meg a kért entitást.
- GET_NOMEM: Nincs elég memória.
- GET_ERR_SEND: Hiba történt az entitás küldése során.
- GET_ERR_UNSUPPORTED: Érvénytelen parancs.
- OK: Az entitás sikeresen el lett küldve.

A kliens kiszolgálása után a kapcsolatot bontjuk, a szálat felszabadítjuk, és beállítjuk a következő szabad szál indexét az éppen megszűnő szál indexére.
A *process_command* feldolgozza az aktuális parancsot. Először létrehozza a megfelelő elérési útvonalat (a kért URL-t a HTML root könyvtár után fűzi). Ezután s megnyitja a kért állományt. Ha ez nem sikerül, akkor vagy egy *Forbidden* vagy

*Not Found* http választ küld a kliensnek, és visszatér a hívó eljáráshoz. Ha a kérés teljes, akkor egy http válasz formájában tájékoztatunk a sikerességéről, és elküldjük az entitás típusát is. Ezt a *get_filetype* eljárás végzi, mely az adott URL-ből megállapítja a file kiterjesztése alapján, hogy milyen típusú állományról van szó. A következö lépésben ha GET parancsról van szó akkor elküldjük a kért entitás tartalmát, ha HEAD-ről van szó akkor végeztünk.

***Biztonság:***
A fejlesztés során próbáltam figyelni arra, hogy a körülményekhez képest biztonságos legyen a szerver. Így például minden buffer-túlcsordulási lehetőséget próbáltam kizárni. Az *strcpy* helyett *strncpy*-t használtam (ahol ez indokolt volt), valamint a pointer műveleteknél is próbáltam a túlcímzés lehetőségét elkerülni. Így most akármilyen hosszú címet megadva nem okozható buffer-túlcsordulás.

***URL formátuma:***
A szerver a szabványban rögzített URL-eket kezeli, így lekezeli a '+' és %<kód> karaktereket is. Ha URL-ként csak egy '/' jelet adunk meg, akkor ezt automatikusan az “/index.html”-re alakítja, és azzal dolgozik tovább.

# Függvények és struktúrák listája

**Függvények:**

| *Név* | *Leírás* |
|---|---|
| `int server_main();` | Maga a szerver. Ez csak fork hívás miatt került bele, áttekinthetőséget növeli. |
| `void* thread_serve(void *);` | A kéréseket kiszolgáló szálak függvénye. |
| `void error(uchar,uchar);` | Hibakiírást és opcionálisan (2. paraméter) kilépést valósít meg. |
| `int process_command(struct command_t*);` | A kért HTTP parancsot hajtja végre. A kérés paramétereit a struct command_t tartalmazza. |
| `void get_filetype(char *, char*);` | Az állomány típusát állapítja meg. |
| `void log_event(int,const struct thread_param_t*,const char*);` | Log-bejegyzést generál és ír a log-állományba. |
| `void send_http_resp(int,int,char *);` | HTTP válasz küldése, protokollban rögzített. Csak teljes kérés esetén küldünk. |
| `void set_free_idx(struct thread_param_t*,int);` | A szabad index (next_free_idx) beállítása, kölcsönös kizárással. |
| `int parse_cf_file(struct config_t*);` | Konfigurációs állomány feldolgozása, alapértékek beállítása (struct config_t feltöltése) |
| `int get_params(const char*,char*,char*,char*);` | Paraméterek kinyerése a küldött HTTP parancsból. Felbontja parancsnévre, URL-re, protokoll azonosítóra. |

***Struktúrák:***

| *Név* | *Leírás* |
|---|---|
| **struct** `thread_param_t` | A szálak adatstruktúráját írják le, amit a a főszáltól kapnak. Benne van a kliens címe, indexek, kliens socket leíró, szemafor, log file leíró, HTML root könyvtár. |
| **struct** `command_t` | A HTTP kérés paramétereit tartalmazza (módszer, eljárás, kliens socket leíró, elérési útvonalak). |
| **struct** `config_t` | Konfigurációs állomány tartalma struktúrába képezve. |

# Mellékletek

Mellékletben szerepelnek a következö állományok:

- `docs/main.c.pdf`: A `main.c` forrásállomány pdf verziója.
- `docs/commanche.h.pdf`: A `commanche.h` forrásállomány pdf verziója.
- `docs/web_server_doc.pdf`: Jelen dokumentáció pdf verziója.
- `docs/web_server_doc.rtf`: Jelen dokumentáció rtf verziója.
- `src/commanche.c`: Forráskód.
- `src/main.c`: Forráskód.
- `src/Makefile`: Makefile a fordításhoz.
- `src/htdocs.all/`: Minta HTML root könyvtár.
- `src/commanche.cf`: Minta konfigurációs állomány.