UNIVERSITETI I PRISHTINËS
HASAN PRISHTINA

FAKULTETI I INXHINIERISË
ELEKTRIKE DHE KOMPJUTERIKE

PROGRAMIMI SISTEMOR

Detyra e tretë

ABSTRAKTI

Temat që do të shtjellohen në detyrën e dytë janë:
i) sockets.

AFATI I DORËZIMIT TË DETYRËS
E premte, më 02.08.2019

ARSIMTARI I LËNDËS
ASS. PROF. DR. IDRIZ SMAILI

PRISHTINË, KORRIK 2019

PËRMBAJTJA

LISTA E FIGURAVE

ORGANIZIMI

Detyra e dytë ka të bëjë me këtë tematikë:

- **Proceset**: krijimi i proceseve përmes fork,
- **Semaforët**: sinkronizimi i qasjes në resurset e përbashkëta,
- **Pipes**: krijimi i pipes për shkëmbim të informatave në mes të proceseve,
- **Sockets**: përmes sockets bëhet shkëmbimi i informatave për pipes, dhe
- **Shared Memory**: krijimi i shared memory për shkëmbim të informatave në mes proceseve.

Detyra konsiderohet e kryerë nëse i) janë të implementuara të gjitha kërkesat (funksionale dhe jofunksionale), ii) nëse është dokumentuar secili funksion (public dhe “private/protected” ose

static në gjuhën programuese C), iii) është bërë përshkrimi se si duhet të përdoret/testohet aplikacioni (detyra), si dhe iv) detyra është mbrojtur me sukses para arsimtarit të lëndës[1].

Numri maksimal i pikëve nga absolvimi me sukses është **pesëmbëdhjetë** (**10**). Në rast se nuk plotësohen kërkesat (jo- dhe funksionale), atëherë do të ketë pikë negative si vijon:

- -1 pikë: ***nFncReq 1***,
- -1 pikë: ***nFncReq 2***,
- -1 pikë: ***nFncReq 3,*** dhe
- -n pikë: Σ(***FncReq_të_plotësuara)/*** Σ(***FncReq)*15***.

Në rast se detyra e dorëzuar nuk është e saktë, që d.t.th. se njëra nga kërkesat funksionale nuk është plotësuar, atëherë detyra duhet dorëzuar përsëri, mirëpo numri maksimal i pikëve në këtë rast do të dekrementohet për **dy** (**2**), që d.t.th. që numri maksimal i pikëve pas iteracionit të parë është **tetë** (**08**).

FILE STRUKTURA

File struktura e detyrës duhet të jetë e ngjajshme me file strukturën e shembullit të parë (exercise_1) gjatë ligjëratave, që d.t.th. që në root directoriumin e detyrës, i cili duhet të emërohet `emri_mbiemri_detyra2` duhet ketë vetëm tre direktoriume, `Makefile` dhe `readme.txt`. Direktoriumet e lejuara janë:

- **`src`**: në këtë direktorium duhet të ruhen vetëm source file-at, q.d.th. *.c,
- **`include`**: në këtë direktorium duhet të ruhen vetëm include file-at,
- **`doc`**: file-i i vetëm që duhet të ruhet në këtë direktorium është exc1.doc, i cili përdorët për konfigurim të `doxygen`-it.

Makefile duhet të ndërtohet asisoi që gjatë procesit të make të krijohet një direktorium **`output`**, në të cilin do të krijohen këto direktoriume:

- **`build`**: në këtë direktorium do të gjenerohen object files (*.o), të cilët krijohen nga gcc compiler-i gjatë compile-imit të source file-ave,
- **`exe`**: në këtë direktorium do të gjenerohet executable pasi të jenë linkuar të gjithë object files, dhe
- **`docu`**: është direktoriumi në të cilin do të gjenerohet online dokumentacioni nga `doxygen` tool.

Direktoriumet e mësipërme do të krijohen gjatë make procesit pëmes komandave:

```
make all
```

[1] Skype meetings do të përcaktohen nga arsimtari i lëndës.

```
make dox
```

Në file-in `readme.txt` duhet ta përshkruani me anë të një paragrafi aplikacionin (detyrën) si dhe t'i specifikoni hapat që shfrytëzuesi duhet t'i bëjë në mënyrë që të mund ta testoj aplikacionin e juaj.

DORËZIMI I DETYRËS

Detyra duhet të dorëzohet përmes emailit më së largu gjerë të premten e fundit të muajit maj, më **02.08.2019** para orës **24:00**. Detyrat e dorëzuara me vonësë nuk do të merren parasysh, dhe numri i pikëve do të konsiderohet i barabartë me **zero**.

Është shumë me rëndësi të cekët që para dorëzimit të detyrës, ju duhet patjeter t'i pastroni direktoriumet e gjeneruara përmes komandës:

```
make clean
```

File-i me këtë emërtim `emri_mbiemri_detyra2.tgz`. duhet t'i bashkangjitt emaili gjatë dorëzimi të detyrës. Krijimi i këtij file-i bëhet përmes këtyre komandave:

```
cd emri_mbiemri_detyra2
make clean
tar czf ~/emri_mbiemri_detyra2.tgz detyra1
```

SPECIFIKACIONI

KËRKESAT FUNKSIONALE

Të zhvillohet sistemi client/server i cili përdoret për *logging* (shife Figure 1) të shënimeve (info, warning & error) në një file. Sistemi duhet përbëhet nga dy pjesë: i) server-i, dhe ii) client-ët.

SERVERI

Server-i duhet të zhvillohet si aplikacion mulittasking. Server-i duhet të përbëhet nga *tre* (*3*) lloje të ndryshme të proceseve: i) `main`, ii) `record`, dhe `conn_handler`, si dhe të ketë një `message queue` dhe një `shared memory`. *Sockets përdoren nga procesi main përmes së cilit clientët e kontkojnë serverin.* Shared memory përdoret për shkëmbimin e shënimeve të pranuara nga clientët përmes `conn_handler`, prej nga ku procesi `record` i shkruan shënimet në file-in e përbashkët të specifikuar përmes opsionit `-f logFile.txt`.

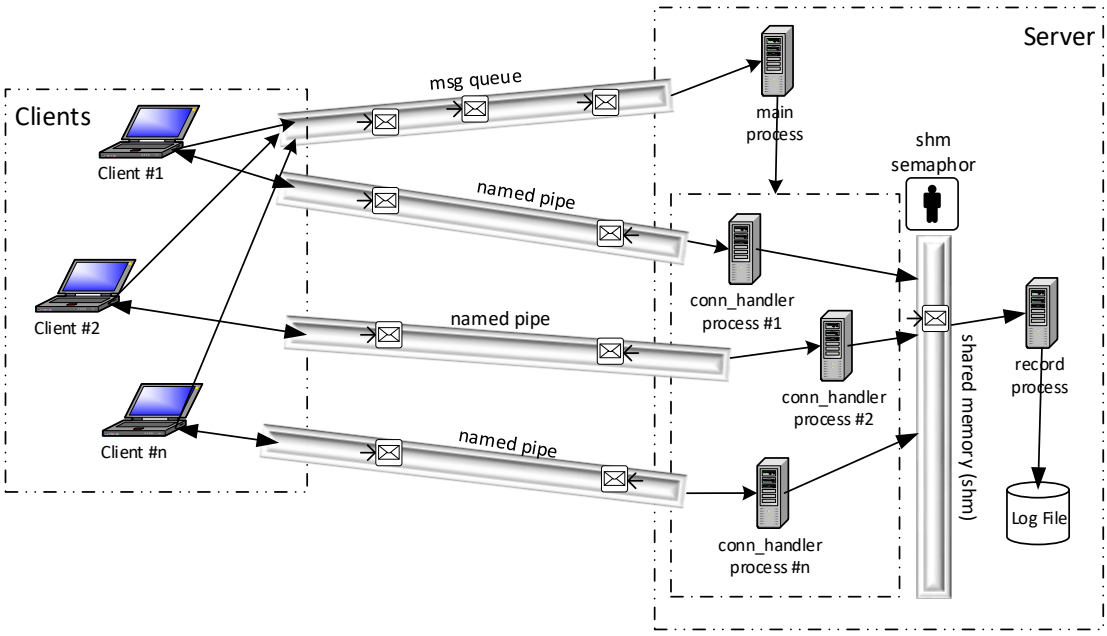


Figure 1: Sistemi i shpërndar për logging

Procesi `main`: *Ky proces "ndëgjon" në socket, përmes së cilit client-ët e kontaktojnë Server-in me kërkesën për connect.* Ky process duhet të bllokohet[2] gjatë leximit nga *socket* (*block on read*). Client-ët duhet ta dërgojnë një element në *message queue*, përmes së cilit client-ët shprehin kërkesën për kyçje në server. Elementi duhet të përmbajë një string, i cili në fakt tregon emrin e *named pipes*, e cila do të krijohet nga procesi i ri `conn_handler`, përmes së cilës client-i do të komunikojnë server-in.

Procesi `conn_handler`: Për secilin client do të krijohet një proces i ri i tipit `conn_handler`, i cili komunikon me client-in përmes *named pipe*, emrin e së cilës client-i ia ka dërguar serverit si element në *message queue* gjatë krijimit të koneksionit. Ky proces do të konkuroi me proceset tjera të njëjta, që d.t.th. client tjerë, për qasje në *shared memory* për ruajtjen e shënimeve në file-in e përbashkët të specifikuar përmes opsionit `-f logFile.txt`.

Procesi `record`: Detyra e këtij procesi është t'i marrë shënimet e vendosura në *shared memory* nga proceset `conn_handler` dhe t'i ruaj ato në file-in e përbashkët të specifikuar përmes opsionit `-f logFile.txt`. Qasja konkurrente në mes të `conn_handler` në njëren anë (në mes vete) si dhe në anën tjetër `record` procesit në shared memory duhet të sinkronizohet përmes `shm_sempahor`-it (shife Figure 1).

Message Queue: përdoret për pranimin e kërkesës për kyçje nga ana e client-ëve. Secili client duhet t'a dërgoi një element në message queue në mënyrë që të mundësoi kyçjen në server. Secili element në *message queue* duhet të jetë e tipit `msg_queue_t` (shife Figure 2).

[2] Gjatë hapjes së *message queue* ***nuk guxoni*** ta përorni opsionin `O_NONBLOCK`!

```c
#define MSQ_LEN 30
typedef struct _msq_elm
{
  int  p_id;          /* client's process id              */
  int  len;           /* length in bytes of the msg       */
  char msg [MSQ_LEN]; /* the message, i.e. the pipe's name */
} msq_elm_t;
```

Figure 2: Një element në message queue

Emrin e *shared pipe* duhet ta përcaktoi client-i dhe këtë t'ia dërgoi server-it përmes *message queu*.

Shared Memory: përdoret për shkëmbim të shënimeve në mes të `conn_handler` proceseve të cilët i pranojnë shënimet nga client-ët dhe `record` procesit, i cili i merrë shënimet nga shared memory dhe i ruan në file-in e përbashkët. Secili client duhet t'a dërgoi një element në shared memory në mënyrë që të mundësoi kyçjen në server.

```c
/* defines the shared memory message length in bytes */
#define SHM_MSG_LEN      (int)  512

/* defines the shared memory states */
#define SHM_EMPTY        (char) 0x01
#define SHM_FULL         (char) 0x02

/* define the shared memory structure */
typedef struct _shm_elm
{
  char  state;             /* shared memory element's state  */
  int   len;               /* shared memory element's length */
  char  msg [SHM_MSG_LEN]; /* shared memory message          */
} shm_elm_t;
```

Figure 3: Një element në shared memory

Procesi `conn_handler` guxon të shkruaj në shared memory vetëm nëse procesi `record` i ka marre shënimet nga shared memory, që d.t.th. `state == SHM_EMPTY` (shife Figure 3)! Procesi `conn_handler` do ta ndryshon `state == SHM_FULL`, pasi ta ketë kopjuar në *shared memory* mesazhin e pranuar nga client-i përmes *named pipe*.

Named Pipe: përdoret për shkëmbim të shënimeve në mes të client-it dhe `conn_handler` procesit, i cili e menaxhon lidhjen me client-in. Sinkronizimi në mes të client-ëve dhe `conn_handler` procesit bëhet në mënyrë automatike, nëse ju gjatë hapjes së *named pipe* **nuk e përdorni** opsionin `O_NONBLOCK`.

```c
#define NAMED_PIPE_MSG_LEN 512
typedef struct named_pipe_
{
  int  len;                       /* length of message to be stored */
  char msg [NAMED_PIPE_MSG_LEN]; /* message to be stored           */
} named_pipe_t;
```

Figure 4: Një element në named pipe

Synopsis i serverit duhet të duket si vijon:

```
[ISM3WI@WI-Z11294]  $ ./server.exe -h

The option -h [-h]
server - The server tool

SYNOPSIS
        exc1 [OPTION] ...
         -f arg, (mandatory) the file to record messages
```

Figure 5: Synopsis i server-it

CLIENT-I

Clienti duhet ta kontaktoi serverin përmes message queue duke e dërguar nje message përmes struktures `msg_queue_t`. Për leximin e process-ID duhet të përdoret *system function call* (API) `pid_t getpid()`. Gjithashtu client-i duhet ta përcaktoi emrin e named pipe, i cili duhet të jetë unik në makinë (computer), si p.sh. "`/tmp/nmpiped_pid`"; pid është process-id e lexuar nga `getpid()` *system call*. Client-i duhet ta krijoi këtë named pipe me këtë emër para se t'ia dërgoi server-it këtë message përmes *message queue*. Pasi të jetë krijuar lidhja me server-in, atëherë clienti e lexon përmbajtjen e file-it të përcaktuar përmes opsionit `-f msg_file.txt`. Secili rresht në këtë file duhet të dergohet tek server-i si një message element përmes named pipe. Pas çdo message-i clienti duhet të pretë n_sec sekonda, i cili përcaktohet përmes opsionit `-t n_secs`.

```
/**************************************************************************//**
 * @brief Writes an input string into the file
 *
 * First the four (4) bytes will be written indicating the length of the
 * string to be written, and then the string itself will be written.
 *
 * @param[in,out] fp - file pointer
 * @param[in] str - the input string
 *
 * @retval 0  in case an error was occured
 * @retval >0 number of bytes written in the file
 *****************************************************************************/
int str_write (FILE *fp, const char *str)
{
  int status = 0;
  int length = 0;

  length = str_len (str);

  status = (int) fwrite ((const void *) &length, (size_t) 1,
                         (size_t) SER_INT_LEN, fp);

  if (status == 0)
    {
       printf ("\nError writing length of string '%s' to the file", str);
       return status;
    }

  status = (int) fwrite ((const void *) str, (size_t) 1,
                         (size_t) length, fp);

  if (status == 0)
    {
       printf ("\nError writing string '%s' to the file", str);
    }

  return status;
}
```

Figure 6: Stili i kodimit (coding style)

Synopsis i client-it duhet të duket si vijon:

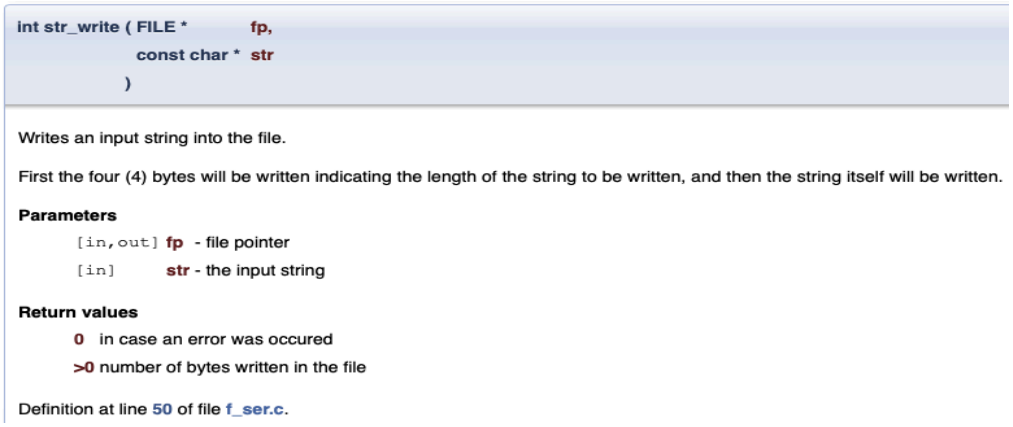
```
[ISM3WI@WI-Z11294]  $ ./client.exe -h

client - The client tool

SYNOPSIS
        exc1 [OPTION] ..I
         -f arg, (mandatory) the file which contains messages
         -t,     (optional)  the number of seconds to wait
```

Figure 7: Synopsis i client-it

KËRKESAT JOFUNKSIONALE

nFncReq 1: Coding style i paraqitur si në Figure 6 duhet të aplikohet gjatë implementimit të sistemit.

nFncReq 2: Dokumentimi online i paraqitur si në **Error! Reference source not found.** duhet të aplikohet gjatë implementimit të secilit funksion të sistemit. Dokumentimi i secilit funksion duhet të përmbaj: i) brief – një përshkrim i shkurtër i funksionit, ii) përshkrim i shkurtër i secilit argument të funksionit në formatin e paraqitur në figurën e mësipërme, si dhe iii) dokumentimi i vlerave kthyese të funksionit.



```
int str_write ( FILE *        fp,
                const char *  str
              )
```

Writes an input string into the file.

First the four (4) bytes will be written indicating the length of the string to be written, and then the string itself will be written.

Parameters

[in,out] **fp** - file pointer

[in] **str** - the input string

Return values

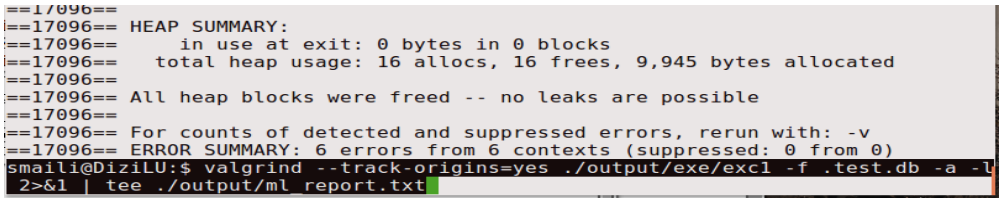
0 in case an error was occured

>0 number of bytes written in the file

Definition at line 50 of file f_ser.c.

Figure 8: Dokumentimi i funksionëve përmes doxygen

nFncReq 3: Sistemi duhet të egzekutohet *free of memory leaks*. Kontrolli për *memory leaks* duhet të bëhet duke e shfrytëzuar tools-in `valgrind` dhe opsionin e aktivizuar `--track-origins=yes`. Sistemi konsiderohet *free of memory leaks* nëse rezultati përmban këtë fjali *„All heap blocks were freed -- no memory leaks are possible*“.



```
==17096==
==17096== HEAP SUMMARY:
==17096==     in use at exit: 0 bytes in 0 blocks
==17096==   total heap usage: 16 allocs, 16 frees, 9,945 bytes allocated
==17096==
==17096== All heap blocks were freed -- no leaks are possible
==17096==
==17096== For counts of detected and suppressed errors, rerun with: -v
==17096== ERROR SUMMARY: 6 errors from 6 contexts (suppressed: 0 from 0)
smaili@DiziLU:$ valgrind --track-origins=yes ./output/exe/exc1 -f .test.db -a -l
 2>&1 | tee ./output/ml_report.txt
```

Figure 9: Kontrollimi i memory leaks