`sfat`: un filesystem unix semplificato basato su FAT32

Progetto finale del corso di LCS 2006/07

Indice

1 Introduzione

Il corso di Laboratorio di Programmazione Concorrente e di Sistema (AA538 – 6 crediti) prevede lo svolgimento di due suite di esercizi intermedi (frammento1 e frammento 2) e di un progetto finale individuale. Il progetto finale è descritto in questo documento.

Il progetto consiste nello sviluppo del software relativo ad un sistema client-server che realizza `sfat` (Simplified File Allocation Table), un filesystem semplificato implementato in spazio utente e della relativa documentazione generata utilizzando gli strumenti presentati durante il corso.

1.1 Materiale in linea

Tutto il materiale relativo al corso puó essere reperito sul sito Web:

`http://www.cli.di.unipi.it/doku/doku.php/lcs/lcs07/start`

Il sito verrà progressivamente aggiornato con le informazioni riguardanti il progetto (es. FAQ, suggerimenti, avvisi), sul ricevimento e simili. Il sito è un Wiki e gli studenti possono registrarsi per ricevere automaticamente gli aggiornamenti delle pagine che interessano maggiormente. In particolare consigliamo a tutti di registrarsi alla pagina delle 'FAQ' e degli 'avvisi urgenti'.

Eventuali chiarimenti possono essere richiesti consultando i docenti di LCS durante l'orario di ricevimento, le ore in laboratorio e/o per posta elettronica.

1.2 Struttura del progetto e tempi di consegna

Il progetto deve essere sviluppato da un singolo studente individualmente e può essere consegnato entro il 1 Febbraio 2008.

La consegna del progetto avviene *esclusivamente* per posta elettronica, attraverso il target `consegna` del `Makefile` contenuto nel kit di sviluppo del progetto.

I progetti che non rispettano il formato o non consegnati con il target `consegna` non verranno accettati.

La data ultima di consegna dei due frammenti e del progetto finale è il 01/02/08. Dopo questa data gli studenti dovranno svolgere la prova prevista per il corso 2007/08.

Inoltre, gli studenti che consegnano una versione sufficiente del progetto finale entro il 20 luglio 2007 accumuleranno il terzo bonus di 2, che contribuisce

al voto finale con le modalità illustrate nelle slide introduttive del corso (vedi sito).

1.3 Valutazione del progetto

Al progetto viene assegnato un punteggio da 0 a 26 di cui 6 punti esclusivamente determinati dalla *qualità della documentazione allegata*. La valutazione del progetto è effettuata in base ai seguenti criteri:

- motivazioni, originalità ed economicità delle scelte progettuali
- strutturazione del codice (suddivisione in moduli, uso di makefile e librerie etc.)
- efficienza e robustezza del software
- modalità di testing
- aderenza alle specifiche
- qualità del codice C e dei commenti
- chiarezza ed adeguatezza della relazione (vedi Sez. 12.3

La prova orale tenderà a stabilire se lo studente è realmente l'autore del progetto consegnato e verterà su tutto il programma del corso. Il voto dell'orale (ancora da 0 a 26) fa media con la valutazione del progetto per delineare il voto finale. In particolare, l'orale comprenderà

- una discussione delle scelte implementative del progetto e dei frammenti
- l'impostazione e la scrittura di script bash e makefile
- l'impostazione e la scrittura di programmi C + PosiX non banali (sia sequenziali che concorrenti)
- domande su tutto il programma presentato durante il corso.

Casi particolari Gli studenti lavoratori iscritti alla laurea triennale in Informatica possono consegnare i due frammenti e il progetto in un'unica soluzione in qualsiasi momento dell'anno ed essere valutati con votazione da 0 a 30. In questo caso è necessaria la certificazione da consegnare al docente.

Gli studenti che svolgono il progetto per abbreviazioni delle nuove lauree specialistiche sono invitati a contattare il docente.

Gli studenti iscritti ai vecchi ordinamenti (nei quali il voto di Lab. 4 contribuiva al voto di SO) avranno assegnato un voto in trentesimi che verrà combinato con il voto del corso di SO corrispondente secondo modalità da richiedere ai due docenti coinvolti.

2 Il progetto: `sfat`

Lo scopo del progetto è lo sviluppo di un sistema client server che realizza `sfat`, un filesystem Unix semplificato bastato su FAT32[1].

Il filesystem FAT32 (e le sue evoluzioni quali il VFAT), é stato utilizzato dalle versioni consumer della famiglia dei sistemi operativi Windows sino al 2000. Nel seguito di questo documento verranno fornite sia una definizione generale del filesystem FAT32, sia i requisiti funzionali del sistema che deve essere realizzato.

3 `sfat`: Simplified FAT32

Il filesystem FAT32 suddivide il device che lo ospita (ad esempio un disco) in tre zone denominate *Boot Sector*, *File Allocation Table* e *Data Region* (Fig 1). Il Boot Sector e la File Allocation Table contengono i filesytem metadata, mentre la Data Region contiene i file e le directory. La Data Region è una sequenza di blocchi aventi tutti la stessa dimensione. Tale dimensione può variare dai 512Bytes ai 64KB.

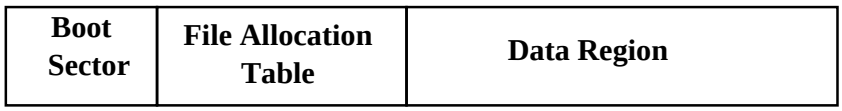

Figura 1: Organizzazione di un device per il filesystem FAT32

Oltre al Boot Sector ed alla File Allocation Table, FAT32 utilizza un terzo tipo di filesystem metadata, detta Directory Table. La Directory Table è memorizzata all'interno della Data Region, ed ha lo scopo di mantenere le informazioni relative ai file ed alle sottodirectory contentute all'interno di ciascuna directory.

Ciascun blocco nella data region viene identificato da un intero (`unsigned`) a 32 bit, da cui deriva il numero 32 concatenato alla fine del nome FAT32.

Il filesystem `sfat` utilizza come *device* un normale file unix e segue gli stessi principi del filesystem FAT32, mentre risultano semplificati i filesystem metadata e le operazioni che possono essere effettuate sul filesystem stesso.

Descriveremo le varie componenti di `sfat` nelle sezioni successive.

3.1 `sfat` Boot Sector

Il Boot Sector è allocato sul primo blocco del device e contiene le informazioni necessarie alla gestione del filesystem (ad esempio per effettuare l'operazione di *mounting*). Le informazioni contenute nel Boot Sector sono:

- L'identificatore del filesystem. Per SFAT tale identificatore è 0x46 (il carattere `'F'`),
- Il numero totale di blocchi disponibili nella data region,
- La dimensione di ciascun blocco. `sfat` ammette blocchi di 4 dimensioni diverse: 128 Bytes, 1KB, 2KB, 4KB

[1] `http://en.wikipedia.org/wiki/File_Allocation_Table`

3.2 `sfat` File Allocation Table

La File Allocation Table (o semplicemenet FAT) è un array che contiene un elemento per ogni blocco presente nella Data Region. I blocchi sono numerati a partire da 0 (zero). Si ricorda che il numero di blocchi disponibili è memorizzato nel Boot Sector. Il blocco di indice 0 è il primo blocco del device che segue la FAT e definisce l'inizio della Data Region. Ciascuna elemento della FAT contiene un intero (`unsigned`) a 32 bit che descrive lo stato di un blocco. I possibili stati che può assumere un blocco, e conseguentemente i possibili valori che possono essere assunti da ciascun elemento della FAT, sono:

- `0x00000000` indica un blocco libero. Inizialmente (ossia al termine del processo di formattazione) tutti gli elementi della FAT sono marcati come blocchi liberi.
- I valori compresi tra `0x00000001` e `0xFFFFFFFE` indicano il blocco successivo in cui è memorizzato un file.
- Il valore `0xFFFFFFFF` indica che questo è l'ultimo blocco utilizzato per memorizare un file.

La FAT ha il compito di mantenere la corrispondenza fra i blocchi logici del file ed i blocchi fisici in cui questi vengono memorizzati. Supponiamo ad esempio che un file F sia memorizzato in tre 3 blocchi fisici il primo di indice 1, il secondo di indice 8 ed il terzo (ed ultimo) di indice 4. La FAT mantiene traccia del file F come riportato in Fig. 2.

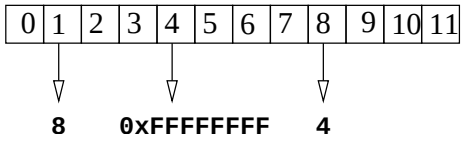

Figura 2: Un esempio di FAT.

L'indice del primo blocco del file F è contenuto nella directory table della directory che contiene F.

3.3 `sfat` Directory Table

La Directory Table (DT nel seguito) contiene le informazioni che consentono di navigare all'interno del filesystem `sfat` e di rintracciare i blocchi in cui sono memorizzati i file e le directory. Esiste una DT per ciascuna directory presente nel filesystem. Ogni DT contiene una *entry* (riga) per ciascun file o sottodirectory in essa contenuta. Ogni entry comprende i seguenti campi:

- `used`: un byte che indica se la entry è in uso (valore `0x01`) oppure è libera (valore `0x00`).
- `Name`: 8 byte per indicare il *nome* del file o della directory a cui la entry si riferisce. Quindi il nome di un file o directory può essere lungo al massimo 8 caratteri

- `Attribute`: un byte utilizzato per indicare se la entry è relativa ad un file (valore `0x02`) o una directory (valore `0x03`).
- `Index` : un unsigned a 32 bit che rappresenta l'indice del blocco a partire da cui il file o la directory è memorizzata.
- `Len` : intero a 32 bit che rappresenta la lunghezza (in byte) del file o della directory.

Ogni directory table viene gestita come un normale file, ed un nuovo blocco viene allocato ogni volta che occorre inserire una nuova entry e non esiste spazio disponibile nel blocco corrente.

La Directory Table della root directory (/) viene sempre memorizzata nel blocco `0`. Tutte le directory table contengono almeno due sottodirectory che sono "." (dot) e ".." (double dot). Nel caso della root directory ".." punta al blocco 0 in quanto non esiste un antenato della root directory.

Ogni volta che viene creata una nuova directory viene allocato un nuovo blocco che viene inizializzato per contenere le due sottodirectory "." e "..". Supponiamo ad esempio che si voglia creare la directory "/DirName1". La creazione di tale directory richiede le seguenti operazioni:

- A partire dalla root directory (blocco 0) si naviga il filesystem seguendo le directory entry che compongono il path che indica dove deve essere creata la nuova directory. In questo caso dobbiamo fermarci alla root directory ("/"). Identifichiamo il blocco BE in cui inserire la nuova entry relativa alla sottodirectory DirName1. Osserviamo che potrebbe essere necessario allocare un nuovo bocco alla directory table che ospita la nuova sottodirectory.
- Si determina un blocco libero BF a partire dal quale verrà memorizzata la Directory Table relativa alla nuova directory DirName1.
- Il blocco BF viene inizializzato con le entry relative alle directory "." e "..".
- Viene preparata la entry da aggiungere al blocco BE. In particolare, al campo Name viene assegnato DirName1, mentre al campo Index viene assegnato il valore dell'indice del blocco BF che contiene la prima porzione della Directory Table relativa a DirName1.

3.4 `sfat` Data Region

La Data Region contiene in sequenza tutti i blocchi che possono essere utilizzati per memorizzare i file e le Directory Table. Come discusso in precedenza, le directory sono trattate dal sistema esattamente come file, cosè che non vengono imposte limitazioni sul numero di file o sottodirectory che possono essere contenuti in una directory.

4 `sfat` come sistema client-server

Un filesystem `sfat` viene creato (formattato) tramite il comando `sfat_create`. Un filesystem esistente viene gestito da un processo server sempre in attesa di

connessioni su un socket di nome fissato. Le operazioni sul file system possono essere invocate tramite un insieme di comandi che funzionano da client.

Nelle prossime sezioni descriveremo in dettaglio le diverse funzionalità.

5 Il comando `sfat_create`

Il comando

```
bash:~$ sfat_create device_name nblocks nsize
```

crea un file di nome `device_name` da usare come device per un filesystem `sfat` e lo formatta in modo che la data Region contenga `nblocks` blocchi di ampiezza `size`. Ricordiamo che la size può assumere uno dei 4 valori: 128, 1K, 2K, 4K.

Se il file esiste, il comando `sfat_create` termina con errore. Un file system non più in uso può essere rimosso con una normale `rm`.

6 Il server

Il processo server viene attivato da shell

```
bash:~$ sfat_server device
```

dove `device` corrisponde a un filesystem `sfat` già precedentemente creato.

Una volta attivato, il server crea il socket su cui riceverà le richieste dei client ed effettua il mounting del device. Il socket di connessione è un socket AF_UNIX di path[2]

```
./tmp/sfatsock
```

su cui il server attende le richieste dei client.

Il server può essere terminato *gentilmente* in ogni momento della sua esecuzione inviando un segnale `SIGINT` o `SIGTERM`. All'arrivo del segnale il server stampa

```
server: signal 2 detected
```

in caso di `SIGINT` o

```
server: signal 15 detected
```

in caso di `SIGTERM` e termina dopo aver cancellato il socket di connessione ed aver portato a termine tutte le richieste pendenti.

Il server è organizzato secondo il modello multithreaded: all'attivazione viene attivato un thread *dispatcher* che riceve richieste di connessione (`connect`) da parte dei client sul socket `./tmp/sfatsock`. Alla ricezione di una connessione il dispatcher crea un nuovo thread T dedicato a servire le richieste relative a quella connessione. Come prima cosa, T si mette in attesa delle richieste da parte dei client. Ci sono 5 tipi di richieste possibili: creazione file (`MSG_MKFILE`), creazione

[2]la creazione in una directory locale è necessaria per facilitare il testing sulle macchine del cli da parte dei diversi gruppi. Una scelta più adeguata in sistemi reali sarebbe l'uso di /tmp o di indirizzi AF_INET.

directory (MSG_MKDIR), append su file (MSG_FWRITE), lettura di file (MSG_FREAD) e listing del contenuto di una directory (MSG_LS).

Ogni richiesta viene eseguita e viene inviato un messaggio di risposta al client prima di leggere la successiva. T termina quando il client chiude la connessione.

Il protocollo di interazione ed il formato delle richieste è descritto dettagliatamente nella Sezione 9.

7 I comandi client

I client sono comandi Unix descritti nei paragrafi successivi. Tutti i path accettati dai comandi client sono esclusivamente *assoluti*.

7.1 sfat_mkdir

Il comando

```
bash:~$ sfat_mkdir /dir1/.../dirk/dir
```

crea una directory `dir` nella directory `/dir1/.../dirk`, se non esiste alcun file o directory con lo stesso nome. Tutte le directory `dir1...dirk` devono esistere.

All'invocazione il client controlla i parametri e contatta il server inviando una richesta di connessione sul socket `./tmp/sfatsock`. Se la connessione ha successo il client invia una richiesta di `MSG_MKDIR` secondo il formato in Sezione 9. Ad esempio (stdout):

```
bash:~$ sfat_mkdir /D2
sfat_mkdir: /D2 created!
bash:~$
```

gli errori sono invece riportati tutti su standard error.

7.2 sfat_mkfile

Il comando

```
bash:~$ sfat_mkfile /dir1/.../dirk/file
```

crea un file `file` nella directory `/dir1/.../dirk`, se non esiste alcun file o directory con lo stesso nome. Tutte le directory `dir1...dirk` devono esistere.

All'invocazione il client controlla i parametri e contatta il server inviando una richesta di connessione sul socket `./tmp/sfatsock`. Se la connessione ha successo il client invia una richiesta di `MSG_MKFILE` secondo il formato in Sezione 9. Ad esempio (stdout):

```
bash:~$ sfat_mkfile /D2/ff3
sfat_mkfile: file /D2/ff3 created!
bash:~$
```

gli errori sono invece riportati tutti su standard error.

7.3 sfat_ls

Il comando

```
bash:~$ sfat_ls dirpath
```

stampa il listing di tutti i file e le directory in `dirpath`, separati da TAB.

All'invocazione il client controlla i parametri e contatta il server inviando una richesta di connessione sul socket `./tmp/sfatsock`. Se la connessione ha successo il client invia una richiesta di `MSG_LS` secondo il formato in Sezione 9. Ad esempio (stdout):

```
bash:~$ sfat_ls /
sfat_ls: listing /
 .      ..      D2
bash:~$
```

gli errori sono invece riportati tutti su standard error.

7.4 sfat_append

Il comando

```
bash:~$ sfat_append file string
```

appende la stringa `string` (SENZA il terminatore `'\0'` finale) alla fine del file `file`.

All'invocazione il client controlla i parametri e contatta il server inviando una richesta di connessione sul socket `./tmp/sfatsock`. Se la connessione ha successo il client invia una richiesta di `MSG_FWRITE` secondo il formato in Sezione 9. Ad esempio (stdout):

```
bash:~$ sfat_append /D2/ff3 "pippo e pluto"
sfat_append: appended "pippo e pluto"
bash:~$
```

gli errori sono invece riportati tutti su standard error.

7.5 sfat_read

Il comando

```
bash:~$ sfat_read file offset len
```

legge al più `len` byte dal file `file` a partire dall'offset `offset`. All'invocazione il client controlla i parametri contatta il server inviando una richesta di connessione sul socket `./tmp/sfatsock`. Se la connessione ha successo il client invia una richiesta di `MSG_FREAD` secondo il formato in Sezione 9. Il numero dei byte letti può essere minore di `len` se viene incontrato prima EOF. Ad esempio (stdout):

```
bash:~$ sfat_read /D2/ff3 0 256
sfat_read: read /D2/ff3 "pippo e pluto"
bash:~$
```

gli errori sono invece riportati tutti su standard error.

7.6 `sfat_cp`

Il comando

```
bash:~$ sfat_cp file1 file2
```

crea il file `file2` e vi ricopia il contenuto del file `file1`. All'invocazione il client controlla i parametri contatta il server inviando una richesta di connessione sul socket `./tmp/sfatsock`. Se la connessione ha successo il client invia le opportune richieste di creazione lettura e scrittura per portare a termine la copia.

```
bash:~$ sfat_cp /D2/ff3 /D2/gg3
sfat_cp: /D2/gg3 written
bash:~$
```

gli errori sono invece riportati tutti su standard error.

8 La libreria `libfat`

Tutte le funzioni relative alla manipolazione del file system si trovano all'interno della libreria `libfat`. Nei file di include presenti nel kit si trovano i prototipi di tutte le funzioni da implementare nella libreria e le specifiche del comportamento e degli errori ritornati.

Per i test della libreria è fornita una suite di test `cunit`[3].

9 Protocollo di interazione client-server

Server e client interagiscono utilizzando i *socket* AF_UNIX. Il server al momento della sua attivazione crea un server socket su cui ricevere le richieste di connessione dei client. Quando una richiesta di connessione è accettata viene automaticamente creato un nuovo socket che verrà poi usato per tutte le comunicazioni con il client (bidirezionale).

9.1 Formato dei messaggi

I messaggi scambiati fra server e client hanno la seguente struttura:

```
typedef struct {
   char type;
   unsigned int length;
   char* buffer;
} message_t;
```

Il campo `type` é un `char` (8 bit) che contiene il tipo del messaggio spedito. `type` puó assumere i seguenti valori:

```
#define MSG_MKDIR        'D'
#define MSG_LS           'L'
#define MSG_MKFILE       'F'
```

[3] Vedi `http://cunit.sourceforge.net/`

```
#define MSG_FREAD       'R'
#define MSG_FWRITE      'W'
#define MSG_ERROR       'E'
#define MSG_OK          'O'
```

Il campo `length` é un `unsigned int` (32 bit) che indica la dimensione del campo `buffer`. Se buffer e una stringa il suo valore comprende anche il terminatore di stringa '\0' finale che deve essere presente nel buffer. Il campo `length` Vale `0` nel caso in cui il campo `buffer` non sia significativo. Il campo `buffer` é un puntatore a un buffer di caratteri.

9.2 Messaggi da Client a Server

Nei messaggi spediti dal Client al Server, il campo `type` puó assumere i seguenti valori:

`MSG_MKDIR` Messaggio di creazione directory, spedito dal client per effettuare la creazione di una directory. In questo caso il campo `buffer` contiene il path della directory da creare. Il Server risponderá con un messaggio di errore (type `MSG_ERROR` e buffer che contiene il codice di errore) se si è verificato un problema durante la creazione oppure con un messaggio di ok (type `MSG_OK` e buffer vuoto) se tutto è andato bene.

`MSG_MKFILE` Messaggio di creazione file, spedito dal client per effettuare la creazione di un file regolare (byte di dati). In questo caso il campo `buffer` contiene il path del file da creare. Il Server risponderá con un messaggio di errore (type `MSG_ERROR` e buffer che contiene il codice di errore) se si è verificato un problema durante la creazione oppure con un messaggio di ok (type `MSG_OK` e buffer vuoto) se tutto è andato bene.

`MSG_LS` Messaggio di "list", spedito dal client per richiedere il listing di una directory. In questo caso il campo `buffer` contiene il path della directory da listare. Il Server risponderá con un messaggio di errore (type `MSG_ERROR` e buffer che contiene il codice di errore) se si è verificato un problema durante il listing oppure con un messaggio di ok (type `MSG_OK` e buffer contenente i nomi di file e directory separati da TAB e terminati da \0) se tutto è andato bene.

`MSG_FWRITE` Messaggio di richiesta di append (scrittura alla fine) ad un file `Path` dei caratteri contenuti in un buffer (`Buf`) di lunghezza `Size`. In questo caso, il buffer del messaggio contiene

`Path\0Buf`

mentre la lunghezza è uguale a `strlen(Path)+1+Size`. Il Server risponderá con un messaggio di errore (type `MSG_ERROR` e buffer che contiene il codice di errore) se si è verificato un problema durante la scrittura oppure con un messaggio di ok (type `MSG_OK` e buffer vuoto) se tutto è andato bene.

`MSG_FREAD` Messaggio di richiesta di lettura da un file `Path` di `Size` byte a partire dall'offset `Offset`. In questo caso, il buffer del messaggio contiene

`OffsetSizePath`

mentre la lunghezza è uguale a `sizeof(int)*2+Size`. Il Server risponderá con un messaggio di errore (type `MSG_ERROR` e buffer che contiene il codice di errore) se si è verificato un problema durante la lettura oppure con un messaggio di ok (type `MSG_OK` e buffer contenente i caratteri letti) se tutto è andato bene.

9.3 Messaggi da Server a Client

Nei messaggi spediti dal Server a ciascun Client, il campo `type` puó assumere i seguenti valori:

`MSG_ERROR` Messaggio di errore. Questo tipo di messaggio viene spedito quando si riscontra un errore. Il campo `buffer` contiene l'errore riscontrato (secondo la definizione in `fat_defs.h`).

`MSG_OK` Con eventuale messaggio che dipende dalla richiesta effettuta.

10 Istruzioni

10.1 Materiale fornito dai docenti

Nel kit del progetto vengono forniti

- funzioni di test e verifica per libsfat
- `makefile` per test e consegna
- file di intestazione (.h) con definizione dei prototipi e delle strutture dati
- vari `README` di istruzioni

10.2 Cosa devono fare gli studenti

Gli studenti devono:

- leggere *attentamente* i README e capire il funzionamento del codice fornito dai docenti
- implementare le funzioni richieste e testarle
- verificare le funzioni con i test forniti dai docenti (attenzione: questi test vanno eseguiti su codice gia' corretto e funzionante altrimenti possono dare risultati fourvianti o di difficile interpretazione)
- preparare la *documentazione*: vedi la Sez. 12.
- sottomettere il progetto esclusivamente utilizzando il makefile fornito e seguendo le istruzioni nel README.

11 Parti Opzionali

Opzionalemnte possono essere realizzate le funzionalità di: `sfat_write` (write generalizzata), `sfat_rm` (rimozione di file), `sfat_rmdir`(rimozione di directory vuota).

Per questi comandi deve essere decisa un'opportuna sintassi da riportare nella relazione.

12 Codice e documentazione

In questa sezione, vengono riportati alcuni requisiti del codice sviluppato e della relativa documentazione.

12.1 Vincoli sul codice

Makefile e codice devono compilare ed eseguire CORRETTAMENTE su (un sottinsieme non vuoto del) le macchine del CLI. Il README deve specificare su quali macchine è possibile far girare correttamente il codice. Inoltre, se si usano software e librerie non presenti al CLI: (1) devono essere presenti nel tar TUTTI i file necessari per l'istallazione in locale del/i tool e (2) devono essere presenti nel makefile degli opportuni target per effettuare automaticamente l'installazione in locale. Se questa condizione non è verificata il progetto non viene accettato per la correzione.

Inoltre la stesura del codice deve osservare i seguenti vincoli:

- la compilazione del codice deve avvenire definendo delle regole appropriate nella parte iniziale del makefile contenuto nel kit;
- il codice deve compilare senza errori o warning utilizzando le opzioni `-Wall -pedantic`
- NON devono essere utilizzate funzioni per la manipolazione delle stringhe che *non* limitano il numero di caratteri scritti/manipolati, ad esempio la `strcpy()` deve essere evitata a favore della `strncpy()` in cui è possibile fissare il massimo numero di caratteri copiati (per le motivazioni consultare il man in linea)
- NON devono essere utilizzate funzioni di temporizzazioni quali le `sleep()` o le `alarm()` per risolvere problemi di race condition o deadlock fra i processi. Le soluzioni implementate devono necessariamente funzionare qualsiasi sia lo scheduling dei processi coinvolti
- DEVONO essere usati dei nomi significativi per le costanti nel codice (con opportune `#define` o `enum`)

12.2 Formato del codice

Il codice sorgente deve adottare una convenzione di indentazione e commenti chiara e coerente. In particolare deve contenere

- una intestazione per ogni file che contiene: il nome ed il cognome dell'autore, la matricola, il nome del programma; dichiarazione che il programma è, in ogni sua parte, opera originale dell'autore; firma dell'autore.
- un commento all'inizio di ogni funzione che specifichi l'uso della funzione (in modo sintetico), l'algoritmo utilizzato (se significativo), il significato delle variabili passate come parametri, eventuali variabili globali utilizzate, effetti collaterali sui parametri passati per puntatore etc.
- un breve commento che spieghi il significato delle strutture dati e delle variabili globali (se esistono);
- un breve commento per i punti critici o che potrebbero risultare poco chiari alla lettura
- un breve commento all'atto della dichiarazione delle variabili locali, che spieghi l'uso che si intende farne

12.3 Relazione

La documentazione del progetto consiste nei commenti al codice e in una breve relazione (massimo 10 pagine) il cui scopo è quello di descrivere la struttura complessiva del lavoro svolto. La relazione *deve rendere comprensibile il lavoro svolto ad un estraneo, senza bisogno di leggere il codice se non per chiarire dettagli implementativi.* In pratica la relazione deve contenere:

- le principali scelte di progetto (strutture dati principali, algoritmi fondamentali e loro motivazioni)
- la strutturazione del codice (logica della divisione su più file, librerie etc.)
- la struttura del server e del client
- la struttura dei programmi di test
- le difficoltà incontrate e le soluzioni adottate
- quanto altro si ritiene essenziale alla comprensione del lavoro svolto
- README di istruzioni su come compilare/eseguire ed utilizzare il

La relazione deve essere in formato PDF.