PROGETTO SISTEMI OPERATIVI 2017/2018

Elaborato 2: System calls
Consegna entro: 17 Giugno 2018; (23:55)

Obiettivo: mettere in pratica le conoscenze acquisite sulle system call durante il corso. In particolare è richiesto l'utilizzo di:

- System call di base (apertura file, exit, etc.)
- Memoria condivisa
- Semafori (protezione regione critica)
- Coda di messaggi
- Segnali e handler

Si vuole creare un programma in grado di trovare la chiavi utilizzate per codificare un set di stringhe. La figura sottostante riporta un esempio del file di input al programma contenente l'insieme di stringhe.

```
<plain_text1>;<encoded_text1>
<plain_text2>;<encoded_text2>
<plain_text3>;<encoded_text3>
…
```

Figura 1: `input.txt`

Ogni riga del file di input al programma rispetta il seguente pattern:

<plain_text>;<encoded_text>

plain_text è una stringa di testo con un numero di caratteri ASCII multiplo di 4 e con un numero massimo di caratteri uguale a 512. **encoded_text** è una stringa di massimo 512 caratteri ASCII generata codificando la stringa **plain_text** con un cifratore XOR a 32-bit (di uguale lunghezza).

La proprietà fondamentale del cifratore XOR è la seguente:

INPUT ⊕ KEY = CRYPT
(INPUT ⊕ KEY) ⊕ KEY = INPUT
CRYPT ⊕ KEY = INPUT

Si prenda come esempio la stringa **plain_text** "Wiki" (in binario 8-bit ASCII: **01010111 01101001 01101011 01101001**) e la chiave di 32-bit **0xF3F3F3F3** (in binario: **11110011 11110011 11110011 11110011**). La stringa *criptata* **encoded_text** è generata dal cifratore XOR come segue:

```
    01010111011010010110101101101001  <- ("Wiki")
    11110011111100111111001111110011^             <-
(0xF3F3F3F3)
    -------------------------------- =
    10100100100110101001100010011010   -> ("*")
```

In modo analogo, la *decriptazione* della stringa **encoded_text** "*" (in binario 8-bit ASCII: **10100100 10011010 10011000 10011010**) con la chiave di 32-bit **0xF3F3F3F3** (in binario: **11110011 0111110011 11110011 11110011 11110011**) genera il **plain_text** "Wiki" come segue:

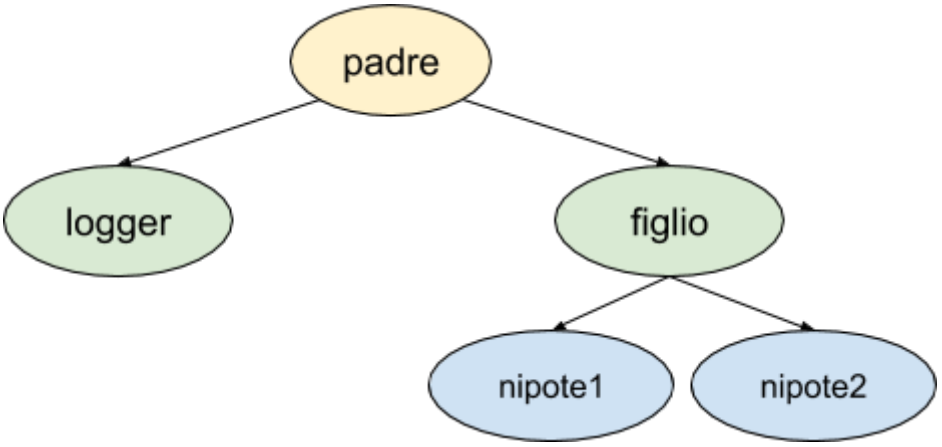
```
    10100100100110101001100010011010  <- ("*")
    11110011111100111111001111110011^             <-
(0xF3F3F3F3)
    -------------------------------- =
    01010111011010010110101101101001   -> ("Wiki")
```

Si ricorda che nel file di input il formato di codifica è little endian. Questo implica che l'ordine dei byte all'interno di una word di 32-bit sono processati in ordine opposto

```
"Cors" ->  0x73726F43
    (hexdump -C file_input.txt: 43 6F 72 73)
"*"    ->  0xB98CA5BD
    (hexdump -C file_input.txt: BD A5 8C B9)
KEY    ->  0xCAFECAFE
```

Specifiche del programma

1. Il programma deve avere il nome **`key_finder`**;
2. **`key_finder`** prende da riga di comando due argomenti:
    a. un file contenente l'insieme delle stringhe **`plain_text`** e **`encoded_text`** come riportato nell'esempio di Figura 1;
    b. un path ad un file non esistente, dove **`key_finder`** scriverà tutte le chiavi identificate (una per riga). L'ordine delle chiavi nel file di output deve rispettare l'ordine delle righe del file di input.
3. L'organizzazione dei sottoprocessi di **`key_finder`** è illustrata nella figura sottostante



padre: è il processo creato quando **`key_finder`** viene eseguito da terminale. Il processo padre deve:

1. Allocare un segmento **`S1`** in *memoria condivisa* sufficientemente grande per contenere il file di input e la seguente struttura:

```
struct Status {
    int  grandson;
    int  id_string;

};
```

2. copiare il file di input nel segmento di *memoria condivisa*;

3. allocare un segmento **`S2`** di *memoria condivisa* sufficientemente grande per contenere tutte le chiavi che saranno trovate (N.B. per ogni riga del file di input esiste una sola chiave di 32 bit);
4. generare il sottoprocesso logger;
5. generare il sottoprocesso figlio;
6. alla terminazione di entrambi i sottoprocessi logger e figlio, il processo padre
    - verifica che le chiavi trovate siano corrette per tutta la lunghezza dei messaggi;
    - salva le chiavi memorizzate sul segmento **`S2`** sul file di output (specificato dall'utente) in formato esadecimale;
    - rimuove i due segmenti di *memoria condivisa;*
    - terminare

logger: è il sottoprocesso creato dal processo padre. Il processo logger deve:

1. creare una *coda di messaggi*. Il messaggio depositato nella coda di messaggi dovrà rispettare la seguente struttura:

```
struct Message {
    long mtype;
    char text[128];

};
```

2. con ritardo di un secondo (polling), il processo logger legge la *coda di messaggi* stampando su stdout il campo **`text`** di ogni messaggio depositato **`(mtype != 1)`**;
3. se un messaggio di tipo 1 (**`mtype = 1`**) è stato depositato, il processo stampa su stdout il campo **`text`** di tutti i messaggi presenti nella coda non ancora letti, rimuove la coda di messaggi, infine termina.

figlio: è il sottoprocesso creato dal padre. Il processo figlio deve:

1. registrare la funzione ***status_updated*** come signal handler del segnale **SIGUSR1** (ricevuto dai nipoti). Quando invocata, la funzione ***status_updated*** deve stampare su stdout il seguente messaggio:
   `"Il nipote X sta analizzando la Y-esima stringa."`, dove `X` e `Y` sono rispettivamente il campo ***grandson e id_string*** della struttura ***Status*** presente in memoria condivisa **S1**;
2. crea un semaforo **P** utilizzato dai sottoprocessi nipote1 e nipote2;
3. generare i sottoprocessi nipote1 e nipote2;
4. attende la terminazione di entrambi i nipoti;
5. quando entrambi in nipoti sono terminati, deposita il messaggio **mtype=1**, **text="ricerca conclusa"** sulla *cosa di messaggi* creata dal processo logger, rimuove il semaforo **P**, ed infine termina.

nipote1 e **nipote2** sono i sottoprocessi creati dal processo figlio. Un processo nipoteX, *dove* X vale 1 o 2, deve:

1. definire una variabile *locale* di nome **my_string;**
2. utilizzando il semaforo **P**, attendere la possibilità di accedere in modo esclusivo alla struttura **Status** presente nel segmento di memoria condivisa **S1**
3. appena l'accesso in modo escluso a **Status** è possibile, il processo nipoteX:
   a. salva il valore corrente del campo **id_string** nella variabile locale **my_string;**
   b. se il valore di **my_string** è diverso dal numero di stringhe del file di input, il processo:

i. memorizza nel campo **grandson** di **Status** il valore X (identificativo del nipote);

ii. incrementa il valore del campo **id_string** di **Status;**

iii. invia il segnale **SIGUSR1** al processo figlio per notificare l'aggiornamento della struttura **Status;**

iv. esce dall'accesso esclusivo di **Status,** ovvero permette all'altro processo nipote di accedervi in modo esclusivo.

c. se il valore di **my_string** è uguale al numero di stringhe del file di input, il processo:

i. esce dall'accesso esclusivo di **Status;**

ii. infine termina, in quando tutte le stringhe sono già state analizzate;

4. se il processo non è terminato, allora una stringa **plain_text** e la corrispondente stringa **encoded_text** viene letta dalla memoria condivisa **S1**;
   N.B. il valore di **my_string** deve essere utilizzato come indice posizionale della stringa da leggere nel segmento di memoria condivisa **S1**. Esempio:

```
my_string = 0 ➜ <plain_text1>;<encoded_text1>
my_string = 1 ➜ <plain_text2>;<encoded_text2>
my_string = 2 ➜ <plain_text3>;<encoded_text3>
```

5. identifica la chiave **key** utilizzata per trasformare la stringa **plain_text** nella stringa **encoded_text**.
   N.B. La chiave può essere un valore qualsiasi da 0 a 2^{32}-1. Il processo nipote deve provare in modo esaustivo ogni valore possibile fino ad identificare la chiave corretta;

6. appena la chiave **key** è stata identificata, il processo salva **key** nel segmento di memoria condivisa **S2;**
   N.B. il valore di **my_string** deve essere utilizzato come indice della posizione della chiave nel segmento **S2**. Esempio:
   my_string = 0 ➜ key1 of <plain_text1>;<encoded_text1>
   my_string = 1 ➜ key2 of <plain_text2>;<encoded_text2>
   my_string = 2 ➜ key3 of <plain_text3>;<encoded_text3>

7. deposita il messaggio **mtype=2**, **text="chiave trovata in " + seconds** sulla *cosa di messaggi* creata dal processo logger, dove **seconds** e' il numero di secondi spesi per identificare la chiave;

8. riesegue dal punto 2 (vedi sopra) finché tutte le chiavi sono state trovate.

BONUS

Oltre alla versione attuale, implementare una variante di **key_finder** in cui il processo figlio genera due o più threads (thread1, thread2, threadN) al posto dei sottoprocessi nipote1 e nipote2.

CONSEGNA ELABORATO

Tutti i file sorgente dell'elaborato devono essere inclusi in una directory di nome **elaborato_syscall**. La directory **elaborato_syscall** deve essere compressa in un archivio di nome:

<matricola>_syscall.tar.gz

L'archivio deve essere creato con il comando **tar** (no programmi esterni). L'archivio **tar.gz** deve essere caricato nell'apposita sezione sul sito di e-learning.

Il codice del progetto deve essere organizzato nei seguenti file e funzioni:

- **`main.c`**: lancia la funzione `padre()`
- **`padre.c/padre.h`**: processo padre
  - `padre(...)` wrapper del processo padre
  - `attach_segments(...)` crea segmento di memoria condivisa
  - `detach_segments(...)` elimina segmento di memoria condivisa
  - `load_file(...)` carica il file di input
  - `save_keys(...)` salva le chiavi sul file di output
  - `check_keys(...)` controlla che le chiavi siano corrette per tutta la lunghezza delle stringhe
- **`logger.c/logger.h`**: processo logger
  - `logger(...)` wrapper del processo logger
  - `polling_receive(...)` scarica la coda di messaggi e la stampa su stdout
- **`figlio.c/figlio.h`**: processo figlio
  - `figlio(...)` wrapper del processo figlio
  - `status_updated(...)` signal handler
  - `send_terminate(...)` deposita il messaggio di terminazione nella coda di messaggi del processo logger
- **`nipote.c/nipote.h`**: processo nipote
  - `nipote(...)` wrapper del processo nipote
  - `load_string(...)` legge la stringa dal segmento S1
  - `lock(...)` blocca accesso esclusivo regione critica
  - `unlock(...)` sblocca accesso esclusivo regione critica
  - `find_key(...)` trova la chiave
  - `send_timeelapsed()` deposita il messaggio “chiave trovata/secondi” nella coda di messaggi del processo logger
  - `save_key(...)` salva la chiave nel segmento S2
- **`types.h:`** contiene le definizione delle strutture

Il progetto deve essere organizzato nel seguente modo:

- Il progetto deve contenere il **`makefile`** con i seguenti target:
  - **all**: compila tutti i target
  - **clean**: pulisce i file intermedi e l'eseguibile
  - **doc**: genera la documentazione
  - **help:** stampare l'elenco dei target possibili
  - **install**: copiare l'eseguibile nella cartella **bin**
  - **threads:** compilare la versione con le threads
- L'eseguibile ***key_finder*** deve essere creato nella sottocartella **bin**
- I file temporanei della compilazione devono essere riposti nella sottocartella **`build`**
- File header (.h) nella sottocartella **`include`** e documentati usando doxygen
- File sorgente (.c) nella sottocartella **`src`**
- La documentazione *html* generata da doxygen deve essere inserita nella sottocartella **`doc`**
- Il progetto deve contenere la configurazione per doxygen nel file **`doxygen.cfg`**

TUTTO CIÒ NON ESPRESSAMENTE SPECIFICATO NEL TESTO DEL PROGETTO E' A SCELTA DELLO STUDENTE

E' vietato l'uso di funzioni C invece di system calls (e.g. fopen, printf, etc)
Il progetto deve funzionare sui pc del laboratorio

Suggerimento:

Conversione di un array di caratteri in array di interi per facilitare la criptazione/decriptazione:

```
char* array = …
unsigned* array_unsigned = (unsigned*) array;
```