Relazione del Progetto del corso di Sistemi Operativi

Autori del progetto:

Andrea Temin

- Matriclola: 5948774
- E-mail: andrea.temin@stud.unifi.it

Alessia Bertini

- Matricola: 6000523
- E-mail: alessia.beritini@stud.unifi.it

Ilaria Davascio

- Matricola: 5967871
- E-mail: ilaria.davascio@stud.unifi.it

Data di consegna: **28/06/2020**

Elemento Facoltativo	**Realizzato (SI/NO)**	**Metodo o file principale**
Utilizzo Makefile per compilazione	SI	Inserito nella cartella src
Organizzazione in folder, e creazione dei folder al momento della compilazione	SI	Makefile da eseguire make all
Riavvio di PFC1, PFC2, PFC3 alla ricezione di EMERGENZA		
Utilizzo macro nel Generatore Fallimenti per indicare le probabilità.		
PFC Disconnect Switch sblocca il processo se bloccato, e lo riavvia altrimenti.		
(*continua da sopra*) In entrambi i casi, il processo in questione deve riprendere a leggere dal punto giusto del file G18.txt.		

Descrizione Cartella

La cartella consegnata contiene al suo interno la cartella “src”, che conterrà a sua volta oltre ai file di codice, file utili e necessari per lo sviluppo e l’esecuzione dei programmi:

- G18.txt: contiene le coordinate del volo di un aereo
- makefile: racchiude i possibilità di esecuzione da usare con il comando make
  - “make all” consente da solo di settare organizzare e compilare tutto il necessario per eseguire il programma richiamando “setDir”, “clearall” e “compile”.
  - “make setDir” crea le cartelle log e tmp che verranno usate durante l’esecuzione del programma.
  - “clearall” esegue la pulizia di tutti i file non necessari all’inizio dell’esecuzione .

  Quest’ultimo si compone di
  - “make clean” che rimuove tutti i file di tipo “*.o” dalla medesima cartella (src).
  - “make clearlog” rimuove i file di tipo log all’interno della cartella log.
  - “make cleartmp” rimuove i file di tipo tmp all’interno della cartella tmp.
  - “make clearsrc” rimouve il file della socket.

La struttura del nostro codice consiste nella creazione di nove processi, tutti generati dal file “main”, alla quale a ciascuno di essi viene assegnato un id per identificare il tipo di processo, in questo modo:

Processo PFC1 avrà id uguale 1

Processo PFC2 avrà id uguale 2

Processo PFC3 avrà id uguale 3

Processo Transducers1 avrà id uguale 4

Processo Transducers2 avrà id uguale 5

Processo Transducers3 avrà id uguale 6

Processo Generatore Fallimenti avrà id uguale 7

Processo WES avrà id uguale 8

Processo PFC Disconnect Switch avrà id uguale 9

Successivamente, ogni processo scrive il proprio PID in un file, chiamato “filePID.log”, in questo modo, tutti i processi possono ricavare il PID degli altri processi accedendo al file.

A questo punto, tramite il comando “execvp” ogni processo esegue il proprio file di esecuzione.

readG18.c: sarebbe il file di esecuzione dei processi: PFC1, PFC2 e PFC3. I seguenti processi una volta al secondo leggono una riga del file "G18.txt", estraggono due coordinate del GPGLL, ricavano tutti i dati necessari per il calcolo della velocità (latitudine, longitudine e tempo) e dopo il calcolo, inviano il dato ai processi Transducers in base alle modalità previste dal progetto.

Nel nostro codice, abbiamo utilizzato un while, (termina quando è stato letto tutto il file "G18.txt") per leggere una volta al secondo le righe del file. Ogni volta, che troviamo una coordinata GPGLL, se la prima stringa è vuota, la salviamo al suo interno; altrimenti, la salviamo nella seconda stringa. Nel momento in cui entrambe le stringhe sono piene, possiamo andare a calcolare la velocita chiamando il metodo "speed".

All'interno di tale metodo, chiameremo le varie funzioni per il calcolo velocità. Tutte queste funzioni/metodi vengono implementate nel file "functions.c", che viene incluso nel file "readG18.c" tramite "functions.h".

Infine, dopo aver ricavato la velocità, il processo entrerà nel metodo "**dataTransfer**", la quale verrà passato in input anche id del processo, così da specificare l'invio del dato.

Il metodo "**dataTransfer**" è strutturato nel seguente modo:

Per PFC1, i dati vengono inviati tramite socket e prima della creazione, cioè dopo if, aggiungiamo una sospensione del processo di durata un secondo per coordinare PFC1 e Transducers1. Subito dopo, il dato viene inviato tramite il comando "send".

Per PFC2, i dati vengono inviati con pipe con nome. La pipe viene creata nella cartella "tmp", che dopo l'invio del dato, viene successivamente eliminata. Anche in questo caso, utilizziamo il comando "sleep(1)", dopo l'invio del dato, per coordinare i processi PFC2 e Transducers2.

Per PFC3, i dati vengono inviati tramite file condiviso. Il nome del file è "fileTransducers.log", che ogni volta stampiamo una riga con inserito: il numero della riga e la velocita. Il numero della riga ci servirà in Transducers3 per capire, se il valore inserito è: nuovo oppure no. Utilizziamo la sospensione di un secondo per coordinate PFC3 e Transducers3.

Al termine dell'invio del dato, il processo torna nel while per rileggere le nuove coordinate. Se il processo non viene fermato prima, dopo la lettura completa del file "G18.txt", i vari processi PFC inviano un segnale di sospensione ai processi Transducers e poi termina il processo PFC.

Sempre all'interno del file di esecuzione, i processi PFC devono gestire anche i seguenti segnali: SIGSTOP, SIGINT, SIGCONT e SIGUSR1; inviati dal processo Generatore Fallimenti. Nel nostro codice la gestione dei segnali viene effettuata tramite piccoli metodi, che in base al tipo di segnali ricevuto, fanno determinati procedimenti.

Dunque, avremo i metodi:

controlSignalSTOP_INT: vi entriamo, se riceviamo i segnali SIGSTOP o SIGINT, e tramite il metodo "writeStatus", incluso nel file "status.c", scriveremo nel file "status.tmp" lo stato del processo, che in questo caso è sospeso.

controlSignalCONT: se riceviamo un segnale SIGCONT, entreremo nel seguente metodo e come per il metodo **controlSignalSTOP_INT** andremo a scrivere nel file "status.tmp" lo stato del processo, che sarà: "Ripresa di esecuzione".

`controlSignalUSR:` per il segnale SIGUSR1, oltre a scrivere lo stato del processo, che specificherà di aver eseguito uno shift sulla velocità; avvertirà il processo tramite un comando boolean (shifSpeed), che se: sarà true, allora al calcolo successivo della velocità si esegue uno shift a sinistra di due bit; altrimenti non effettua lo shift.

Transducers.c: Per gestire la lettura dei dati inviati dai processi PFC, abbiamo creato tre processi Transducers, in modo tale, che ogni processo di questo tipo abbia solamente un metodo di lettura del dato. IL codice funziona in questo modo:

All'interno del "main" del file, tramite un if, dirigiamo i vari processi verso i metodi, che dovranno eseguire separatamente. Per esempio: readPFC1 per Transducers1, readPFC2 per Transducers2 e readPFC3 per Transducers3. Ogni processo esegue il proprio metodo una volta al secondo (gestito tramite comando sleep(1)).

A questo punto analizziamo ogni metodo:

`"readPFC1()"`: tramiti la socket trasferiamo il dato velocita da PFC1 e Transducers. Successivamente, copiamo l'array ricevuto in un altro array (tramite "strcopy"), così da poterlo scrivere nel file log: "speedPFC1.log". La scrittura viene effettuata tramite l'utlizzo dei metodi: **`"writeLog()"`** e **`"creatName()"`**.

`"readPFC2()":`l'invio del dato tra i due processi avviene tramite pipe con nome (nome della pipe: "mypipe", salvata nella cartella "tmp"). Anche qui riutilizziamo i metodi: **`"writeLog()"`** e **`"creatName()"`**, per scrivere all'interno del file "speedPFC2.log".

`readPFC3():` la comunicazione avviene tramite la lettura del file condiviso ("fileTransducers.log") e successivamente, confrontiamo la stringa letta con quella precedente, così da verificare se abbiamo letto un nuovo dato. Se il dato è nuovo, allora copiamo la stringa in un'altra per il confronto successivo e poi tramite due comandi "for" ricaviamo la velocità; altrimenti richiamiamo il metodo readPFC3() finché il dato non è aggiornato. Infine, tramiti i metodi: **`"writeLog()"`** e **`"creatName()"`** scriviamo il risultato nel file log: "speedPFC3.log".

`"creatName()":` generiamo la stringa, che corrisponderà al nome del file in cui dovremmo scrivere la velocità, cioè: speedPFC1, speedPFC2 e speedPFC3.

“writeLog()”: qui avviene la scrittura nei rispettivi file. Infatti, passiamo in input: la stringa, che contiene il nome del file, in cui andremo a scrivere e la stringa contenente la velocità.

I processi Transducers possono terminare in due modi: o tramite segnali di interruzione inviato dai PFC oppure tramite il segnale KILL da parte di PFC Disconnect Switch.

reaserchPid.c: viene richiamato dai vari processi per ricavare il pid di un determinato processo. I processi, che utilizzano questo file sono: i tre PFC e Generatore Fallimento. Viene passato in input l’id del processo, la quale, ci interessa sapere il pid e poi leggiamo ogni riga del file “filePID.log” per ricavare la riga contenente il pid interessato. Questo avviene tramite vari if, in cui si confronta: l’id passato in input e la prima posizione della riga, che contiene l’iniziale del processo. Per i processi PFC e Transducers essendo di tre tipi, si effettua anche un controllo sulla terza (per PFC) o sesta (per Transducers) posizione per ricavare il numero di PFC o Transducers. A questo punto, possiamo ricavare il pid, chiamando il metodo **`copyStr()`**, che prende in input la riga letta e partendo dal fondo di tale riga, ricaviamo il pid del processo interessato.

Infine, il file di esecuzione restituisce il pid del processo.

Functions.c: contiene tutti i metodi necessari per il calcolo della velocita. I metodi sono i seguenti:

`latitudine():` viene passato in input la coordinata letta precedentemente nel file “G18.txt”. Tramite un for scorro la stringa, finché non trovo la direzione della latitudine (Nord o Sud). Salvo la direzione, perché se è Nord la latitudine è positiva; altrimenti negativa. A questo punto chiamo il metodo **`calcoloLateLog()`** per effettuare il calcolo effettivo della latitudine.

`longitudine():` Simile al metodo precedente, solamente che ricaviamo la longitudine della coordinata passata per input. In questo caso, se la direzione è West, la longitudine è negativa; altrimenti positiva. Infine, si richiama **`calcoloLateLog()`** per effettuare il calcolo della longitudine.

`calcoloLateLog()`: prende in input: la coordinata, un puntatore (int i) e la direzione, per capire se si tratta di latitudine o longitudine. Siccome sia la latitudine, che la longitudine, sono strutturata così: ggmm.mmmm, cioè “g” uguale ai gradi, mentre “m” uguale a minuti; dobbiamo ricavare prima i minuti e poi i gradi. Questo avviene tramite un while, che partendo dal puntatore passato in input, andiamo indietro lungo la stringa per poi salvare i valori, di volta in volta, su un'altra stringa. Infine, la convertiamo in intero e dividiamo per 60 (perché un minuto è formato da 60 secondi) per convertirla in secondi. Lo stesso ragionamento lo facciamo per i gradi, solamente non li convertiamo in secondi e per la longitudine, i gradi sono composti da un solo valore (almeno quelli che consideriamo noi). Dopo aver ricavato i valori, il metodo restituisce la somma.

`tempoGLL()`: permette di ricavare il tempo presente nelle coordinate. Anche qui, dobbiamo ricavare: prima la posizione del tempo nella stringa e poi preleviamo il tempo salvandolo dentro una stringa. A questo punto, chiamiamo il metodo **`calcoloTmp()`**.

`calcoloTmp()`: prende in input la stringa contenente il tempo della coordinata e poi ricava singolarmente i seguenti valori: ore, minuti e secondi. Questo perché, il tempo è rappresentato così: hhmmss e dobbiamo

convertire tutto in secondi. Dunque, avremo: le ore moltiplicate per 3600 secondi, i minuti moltiplicati per 60 secondi e poi il tutto sommato con i secondi rimasti. Restituisce il risultato finale del tempo.

`calcuReadiusEarth():` calcola la latitudine in base alla latitudine della terra, che in questo caso corrisponde alla latitudine di Genova.

`distance():` passando in input i vari dati ricavati nei metodi precedenti, che sono: latitudine e longitudine di entrambe le coordinate. Calcoliamo la distanza tra di esse. La formula sarebbe:

```
dlat = (lat2 - lat1) * radiante (dr2)

dlong = (long2 - long1) * radiante

a = sin^2(dlat/2) + cos(lat1 * radiante) * cos(lat2 * radiante) * sin^2(dlon/2)

c = 2 * atan2( sqrt(a), sqrt(1-a))

d = raggioTerrestre(6371) * c
```

Infine, restituisce la distanza tra le due coordinate.

Generatore_Fallimenti.c: richiamato solamente dal processo Generatore Fallimenti ed ha il compito di inviare determinati segnali ai vari processi PFC. Il metodo è strutturato in modo tale, che all'inizio prende un valore random tra: 1, 2 e 3, che rappresenteranno i tre processi PFC, tramite il metodo "**randomNumber()"**; successivamente ricava il pid del processo ricavato da random e infine, calcola le probabilità dei vari segnali. Queste probabilità vengono gestite nel metodo: "**sedSignal()"**, che viene passato in input il pid del processo scelto e poi sempre tramite il metodo "**randomNumber()"** estraiamo un numero in un certo range:

Per il segnale SIGSTOP il range va da 1 a 100.

Per il segnale SIGINT il range va da 1 a 10000.

Per il segnale SIGCONT il range va da 1 a 10.

Per il segnale SIGUSR1 il range va da 1 a 10.

Se uno dei valori restituisce 1, allora il segnale verrà mandato al processo scelto all'inizio del codice, tramite il metodo **send()**, che dopo l'invio del segnali, chiamerà il metodo **writeOnLog()** per scrivere nel proprio file log la funzione da lui svolta. Il file viene nominato: "failures.log".

Status.c: scrive e legge gli stati dei vari processi PFC. Esso viene chiamato del processo PFC Disconnect Switch, quando una velocità è discorde rispetto alle altre. I metodi utilizzati sono: **writeStatus(),** per la scrittura dello status e prende in input: id per capire a quale processo PFC bisogna leggere lo status e l'array in cui viene specificato lo status; e **readStatus()** utilizzato per la lettura, che ha gli stessi input del metodo precedente, solamente all'interno del buffer viene salvato il risultato letto nel file.

Wes.c

- "main()" esegue il metodo "compareSpeeds(i)" (spiegato immediatamente sotto) che restiuirà un intero assegnato alla variabile "toSend" che, mediante uno switch, permetterà di costruire il messaggio da stampare su file e da mandare a video. "ToSend" sarà poi inviato con una namedPipe al file "discSwitch". "ToSend" può assumere solo i seguenti valori: 0, 1, 2 ,3, -1, -2.
- compareSpeeds(int posLine) questo metodo riceve come input l'intero che determinerà quale linea del file G18.txt bisognerà leggere, sarà dunque passato al metodo takeSpeed dopo esserersi settati dentro la cartella "log" mediante la funzione "goToDir()". Il metodo takeSpeed sarà dunque chiamato tre volte e restiuirà il valore float delle tre velocità rispettivamente in s1, s2 e s3. Di questi tre valori si verifica prima che siano tutto valori validi ovvero si controlla che takeSpeed non abbia letto una riga vuota altrimenti chi tra s1, s2 o s3 assumera' il valore arbitrario -2 e questo metodo restiutira al main il valore. Viene poi eseguito una serie di if annidati che sostanzialmente permettono di verificare se gli s1,s2,s3 abbiano lo stesso valore, se uno dei tre ha un valore diverso dagli altri 2 precisando quale o se i valori sono tutti diversi tra loro. Questo metodo restiuirà dunque:
  - 0 se tutti valori sono concordi.
  - 1 se s1 (il valore relativo speedPFC1) è discorde dagli altri 2
  - 2 se s2 (il valore relativo speedPFC2) è discorde dagli altri 2
  - 3 se s3 (il valore relativo speedPFC3) è discorde dagli altri 2
  - -1 se tutti i valori sono discordi tra loro
  - -2 se almeno uno due valori equivale a -2
- takeSpeed(char* nomeFile, int linePosition) riceve in input il neome del file a cui accedere, uno tra speedPFC1/2/3, e la linea del file da leggere. Questi file saranno letti in sola lettuara e ci si soffermerà al rigo indicato da linePosition. Nel in cui non vienga letto nessun valore il metodo restituirà arbitrariamente il valore -2.
- NamedPipe(int mess) ha il ruolo di comunicare al processo DiscSwitch il messaggio di eventuale errore e emergenza. Abbiamo deciso di comunicare con una named pipe per semplici motivi organizzativi, in questo modo da parallelizzare il lavoro il più possibile
- printMessage() Stampa il necessario a video e scrive sul file status.log

DiscSwitch.c

- Main() entra in un loop infinito
- readPipe() riveve i messaggi da wes.c mediante la named Pipe

- MenageError() controllo il messaggio ricevuto dal wes stampa su file switch.log il tipo di errore ricevuto e lo stato attuale del processo in errore in caso di emergenza arresta tutto il programma.