Relazione progetto Sistemi Operativi

Marco Edoardo Santimaria (912404)
Nicolò Vanzo (912759)

28 dicembre 2020

Indice

1 Compilazione ed Esecuzione

Per compilare il progetto, recarsi nella cartella del progetto e digitare il comando

```
$  make
```

Ciò procederà a compilare il codice e ad avviarne l'esecuzione con inserimento di parametri simulazione assistito.
Per avviare la simulazione, senza passare dalla modalità di inserimento assistito è possibile eseguire il seguente comand (previa compilazione):

```
$ ./master <SO_TAXI> <SO_SOURCES> <SO_HOLES> <SO_CAP_MIN> <SO_CAP_MAX> <SO_TIMENSEC_MIN> <SO_TIMENSEC_MAX>

<SO_TOP_CELLS> <SO_TIMEOUT> <SO_DURATION>
```

In entrambi i casi, successivamente, verrà mostrato un riepilogo dei parametri inseriti e, se non soddisfatti sarà possibile ignorare i parametri inseriti e procedere nuovamente a un nuovo inserimento. Per ulteriori target, si rimanda alla visione del `MAKEFILE`.

2 Struttura

Il progetto è composto dai seguenti files:

- master.c
- source.c
- taxi.c
- utils.c
- include-main.h
- ipcKey.h
- makefile

2.1 master.c

Il file master.c contiene il codice in cui viene creata la mappa di gioco e l'interazione con l'utente attraverso la richiesta di inserimento dati da terminale (seguono dettagli). Al momento della creazione della mappa le celle vengono create tutte uguali, per poi essere modificate (tramite le funzioni spawnBlockse spawnSources) in celle di tipo SOURCE o HOLES (da noi chiamate celle BLOCK).

Il file master.c è responsabile della stampa delle statistiche dei taxi e delle corse, in particolare: grazie alla funzione stampaStatistiche l'utente visualizza a schermo la situazione della mappa, il numero di taxi che è arrivato a destinazione (una volta letto un messaggio dalla coda di messaggi), il numero di messaggi presenti in coda che attendono di essere serviti da un taxi (pending rides) e il numero di taxi che, una volta raccolta una richiesta non ha terminato la corsa.

Oltre alle statistiche sopra citate durante l'esecuzione del programma è possibile visualizzare a schermo

- rappresentazione grafica della mappa della città con numero di taxi presenti in ogni cella in un determinato momento;
- Durata totale della simulazione;
- Pid del taxi che ha attraversato più celle;
- pid del taxi che ha passato più tempo in strada con clienti a bordo;
- pid del taxi che ha servito più clienti;
- le SO TOP CELLS stampate alla fine della simulazione;

2.1.1 Stampa

L'aggiornamento della mappa viene effettuato ogni secondo, bloccando il processo di stampa con la syscall `sleep(1);` Questo offre una buona approssimazione di stampa ogni secondo, anche se effettivamente tra una stampa e un'altra passa poco di più di 1 secondo.

Per potere codificare più informazioni nella mappa, abbiamo deciso di stampare la mappa a colori. Questo viene ottenuto grazie alla possibilità di cambiare il colore con cui il terminale stampa sia il testo che lo sfondo del testo, "stampando" a terminale la sequenza di escape `\033[` seguita dal codice di colore più consono (si veda `colorPrintf()`.

La stampa con effetto "aggiornamento continuo" viene ottenuta con un artefatto: non vengono indicizzati i singoli caratteri per l'aggiornamento (anche se si potrebbe fare con la libreria ncurses, della quale comunque non abbiamo approfondito) ma vengono calcolati quanto `\n` devono essere stampati ogni volta in modo che solo la mappa e le sue statistiche siano presenti a terminale.

Per ottenere questo effetto abbiamo la necessità di conoscere a priori la dimensione in caratteri del terminale. Ciò è possibile con una api POSIX (unica usata in tutto il progetto), chiamata `ioctl();`. Questa api, ci consente di ottenere, con i parametri illustrati nel codice sorgente una struttura di tipo `struct winsize size;` contenente le dimensioni sia in pixel che in caratteri del terminale corrente. Una volta ottenuta la dimensione del terminale basterà stampare abbastanza `\n`in modo da mostrare solo la mappa.

Una precedente implementazione, non faceva utilizzo di questa API, ma chiedeva all'utente di ridimensionare il terminale, in modo da mostrare solamente la mappa.
In conclusione l'API `ioctl();` non è fondamentale al funzionamento del software, ma è solo un vezzo grafico.

Sono disponibili due tipi di stampe: la prima (quella di default) stampa la mappa in forma estesa con molte raffinatezze grafiche. La seconda invece, viene usata qualora la larghezza del terminale non fosse sufficente alla stampa della prima versione.

2.1.2 Colori impiegati nella stampa

- **Magenta**: Una cella di tipo magenta è una cella di tipo `SOURCE` ; Nella stampa normale lo sfondo è magenta, nella stampa compatta, il testo è in magenta;
- **Bianco**: Una cella di tipo bianca è una cella di tipo `ROAD`. Nella stampa normale lo sfondo è bianco, nella stampa compatta, il testo è bianco;
- **Nero**: Una cella senza testo e con sfondo nero è ti tipo `BLOCK(hole)` ;
- **Giallo**: Una cella di tipo `SOURCE` o di tipo `ROAD` può essere evidenziata in giallo, a fine esecuzione del progranna, se rientra nelle `SO_TOP_CELLS` .
  - **NOTA**: nella stampa finale, se una cella è evidenziata in giallo con testo nero, significa che la cella era nelle `SO_TOP_SOURCES` ed era di tipo BLOCK. Se il testo invece è di colore magenta, significa che la cella è contemporaneamente in `SO_TOP_CELLS` e che il tipo di cella è `SOURCE`;

2.2 source.c

Il file source.c è il responsabile della creazione dei messaggi e del loro inserimento nella coda che verrà utilizzata dai taxi per ottenere le richieste dei clienti. Questo processo viene creato dal processo master.
E' possibili forzare la creazione di una richiesta manualmente, inviando al processo source, il segnale SIGALARM nel seguente modo:

```
$ kill -SIGALRM <source_pid>
```

Le richieste vengono generate ogni `(rand()% 5)+1)` secondi, se non si inviano segnali al processo taxi.

2.3 taxi.c

Il file taxi.c viene creato dal processo Master attraverso una fork e da una successiva execlp. Il taxi eseguirà le seguenti azioni fino allo scadere del tempo:

1. Il taxi viene posizionato all'interno di una cella accessibile;

2. Il taxi comincia a muoversi verso la cella di tipo SOURCE più vicina a lui (questo punto è frutto di interpretazione del testo dell'esercizio da parte nostra, visto che veiene lasciata libera scelta sul comportamento che deve assumere il taxi una volta posizionato all'interno di una cella casuale);
3. Arrivato sulla cella di tipo SOURCE più vicina si mette in ascolto di un tipo di messaggio che dipende dalla SOURCE su cui è posizionato (ad esempio: se la SOURCE in cui siamo seduti è la prima a comparire nella mappa il taxi attende che source.c generi messaggi di tipo 1);
4. Il taxi accoglie la richiesta generata e procede muovendosi verso la sua destinazione, codificata nel messaggio;
5. Torna al punto 2 e riesegue;

2.3.1 Deadlock

Per tentare di arginare il fenomeno di deadlock il più possibile, si è scelto di utilizzare il seguente algoritmo:

- Se un taxi si vuole spostare a destra, richiede prima il mutex della cella in cui si trova e successivamente il mutex della cella di destinazione;
- Se un taxi si vuole spostare a sinistra, richiede prima il mutex della cella di destinazione e successivamente della cella in cui si trova;
- Se un taxi si vuole spostare in alto, richiede prima il mutex della cella di destinazione e successivamente della cella in cui si trova;
- Se un taxi si vuole spostare in basso, richiede prima il mutex della cella in cui si trova e successivamente della cella di destinazione;

Questo algoritmo risolve il deadlock in qualsiasi possibile combinazione di spostamento ad eccezione di due casi:

1. Due taxi si vogliono spostare contemporaneamente in una cella e uno proviene a sinistra della cella di destinazione e l'altro dal basso;
2. Due taxi si vogliono spostare contemporaneamente in una cella e uno proviene a destra e uno dall'altro;

Per risolvere questi ultimi due casi di deadlock, viene usato il parametro `SO_TAXI_TIMEOUT` in modo da uccidere uno dei due taxi (quello che attende da più tempo) e procedere al rilascio delle risorse e al contino dello spostamento dell'altro taxi. In ogni caso, un taxi, in qualsias cella si trovi, se trascorre troppo tempo senza spostarsi, termina la sua esecuzione, e si procede a crearne uno nuovo, posizionandolo sempre casualmente sulla mappa.

2.4 utils.c

Il file Utils.c contiene 4 funzioni che verranno utilizzate in master e in taxi, queste funzioni sono:

- **ColorPrintf()**: Funzione usata per rendere più agevole la stampa a colori su terminale;
- **P()**: è una funzione usata per eseguire una P su un semaforo;
- **V()**: è una funzione usata per eseguire una V su un semaforo;
- **VwaitForZero()**: è la funzione utilizzata dai taxi per far sì che tutti I taxi vengano posizionati nella mappa prima di continuare l'esecuzione, facendo appunto, una `waitForZero()`;

2.5 include-main.h

Il file `include_main.h` contiene le struct utilizzate per la realizzazione della mappa, delle statistiche, delle celle che formano la mappa e per la struttura dei messaggi presenti in coda (vedi sezione Strutture richieste).

2.6 ipcKey.h

Il file contiene testo per la realizzazione di chiavi univoche tramite l'utilizzo della funzione ftok(), utilizzata per generare chiavi con cui ottenere gli identificatori delle struttura IPC.

2.7 makefile

File per il comando make, contenente le specifiche ed I target di compilazione (-std=c89 –pedantic).

3 Scelte Implementative

3.1 Strutture utilizzate

Abbiamo deciso di utilizzare quattro strutture fondamentali:

- Una struttura dati che contiene la matrice di `SO_HEIGHT` e `SO_WIDTH`, un semaforo mutex per garantire la mutua esclusione, un semaforo aspettaTutti utilizzato dai taxi per aspettare che tutti I taxi siano posizionati prima di cominciare a muoversi e infine due variabili intere: `aborted_rides` e `successes_rides` usate nel master per stampare il numero di corse non andate a buon fine o corse andate a buon fine. La modifica delle due variabili intere avviene in mutua esclusione, motivo per cui abbiamo dovuto usare semafori di tipo binari, chiamati mutex;
- Una ADT (Abstract Data Type) che identifica una cella della mappa e che contiene le seguenti variabili: il tipo di cella (cellType) che può variare tra ROAD, HOLES, SOURCE; availableSpace è una variabile di tipo semaforo che indica la quantità di taxi che possono stare in un cella; taxiOnThisCell è una variabile intera che indica il numero di taxi presenti in una cella; availableForHoles è una variabile utilizzata per facilitare l'implementazione del posizionamento dei buchi nella mappa; totalNumberOfTaxiPassedHere è una variabile intera usata per indicare quanti taxi sono passati in quella cella e che viene usata per determinare le celle da evidenziare a fine simulazione; timeRequiredToCrossCell è una variabile di tipo intero che indica quanto ci metterà il taxi a passare attraverso quella cella; isIntopCell è una variabile di tipo booleano usata nel processo di evidenziamento delle celle a fine gioco; mutex viene utilizzato per garantire la mutua esclusione;
- Una struct che contiene la struttura dei messaggi che vengono inseriti all'interno della coda di messaggi, all'interno ci sono: una variabile di tipo intero mtype che indica il tipo di messaggio che viene inserito in coda; xDest che specifica la coordinata x verso cui deve andare il taxi e yDest la coordinata y verso cui deve andare il taxi;
- Una struct "taxiStatistiche" utilizzata per l'aggiornamento del pid del taxi da stampare;

3.2 Oggetti IPC utilizzati

Gli oggetti IPC utilizzati sono I seguenti:

- Una coda di messaggi che viene creata dal processo master e a cui si attaccano I processi taxi e I processi source usata come mezzo per consegnare richieste ai taxi;
- una struttura presente in memoria condivisa contenente la mappa della città e alcune statistiche;
- Una struct contentente le statistiche relative ai PID dei processi;
- una serie di semafori, utilizzati da tutti i processi;

3.3 Creazioni Richieste

Una volta che il master genera I figli SOURCE il programma agirà autonomamente, quando I processi source vengono generati comincieranno a inserire messaggi nella coda di messaggi che vengono estratti dai taxi e usati per recersi verso la loro destinazione.

3.4 Comportamento dei taxi

I processi taxi durante il loro ciclo di vita svolgono le stesse azioni a ripetizione, che sono le seguenti:

1. Il processo di posizionamento del taxi viene gestito dalla funzione spawn(), all'interno della quale viene anche gestita l'attesa del posiionamento dei taxi prima della partenza;
2. Dopo il posizionamento I taxi cercano la cella di tipo SOURCE più vicina tramite la funzione closestSource();
3. Individuata la SOURCE procedono con il movimento verso di essa;
4. Posizionati sulla SOURCE prelevano un messaggio così ottenendo le informazioni sulla loro destinazione;
5. Procedono verso la destinazione e ripetono dal punto 2;

1.

4 Note sul progetto

1. Qualora si trovino coordinate X o Y nel codice, è bene tenere a mente che nono sono X e Y cartesiane ma, X = numero di riga, Y = numero di colonna;

2. I colori utilizzati nella stampa, potrebbero variare leggeremente se il terminale dell'utente ha delle personalizzazioni che vanno a modificare l'aspetto delle assegnazioni di default dei colori. In ogni caso, eventuali modifiche variano minimamente l'aspetto della stampa;