Relazione Progetto Sistemi Operativi 2020/21

Questo documento (markdown convertito in formato .pdf) riassume indicazioni e scelte implementative per il progetto finale del laboratorio di Sistemi Operativi (12 CFU), appartenente al Corso di Laurea Triennale in Informatica dell'Università degli Studi di Torino (a.a. 2020/2021).

Dario Ferrero
Sara Pangrazi
Beatrice Zicola

1. Compilazione ed Esecuzione

1.1 Makefile

Il Makefile presente nella directory principale permette di compilare i file suddivisi nelle diverse directory nei rispettivi eseguibili, gestendo le dipendenze.
I comandi forniti sono i seguenti :

- *make all* : Compila tutti i file necessari e genera gli eseguibili finali alla simulazione nella directory **out/**
- *make run* : Come *make all*, ma esegue anche il codice
- *make clear* : Elimina ogni file generato dai due comandi precedenti

1.2 Suddivisione del codice

La suddivisione in sottodirectory raggruppa tra di loro file sorgente (**src/**), dichiarazioni e documentazioni (**lib/**), file di configurazione (**conf/**) e prodotti della compilazione (**out/**).
I seguenti nomi definiscono i vari processi presenti nella simulazione o le librerie usate :

- **master** : processo iniziale che si occupa di inizializzare le strutture dati principali per la simulazione, creare gli altri processi e gestirne la terminazione / raccoglierne le statistiche.
- **sorgente** : uno dei processi situati idealmente su una cella della griglia ed incaricati di generare le richieste per i **taxi**.
- **taxi** : uno dei processi erranti tra le celle della griglia al fine di soddisfare le richieste create dalle **sorgenti**.
- **printer** : processo inizialmente creato dal **master** ed incaricato di stampare ad output lo stato di occupazione della griglia ad ogni secondo della simulazione.
- **gridprint** : piccolo modulo contenente le funzioni di stampa della griglia. È usato sia dal **master** che dal **printer**.

1.3 Modifica dei parametri di simulazione

La directory **conf/** ospita i file di configurazione : ogni riga di un file .conf è una coppia nome-valore separata dal carattere '=', rappresentante i parametri della simulazione che saranno letti a run-time. Sono presenti due file, **dense.conf** e **large.conf**, rappresentanti le configurazioni fornite dalla traccia del progetto, più un file "test.conf" di configurazione alternativa.

- Per modificare quale file venga letto a run-time occorre modificare la macro *PARAMS_FILE* presente nel file **master.h** nella directory **lib/**
- Oltre a quelli letti a runtime dai file .conf, due importanti parametri sono definiti tramite macro (e modificabili) nel file **common.h** della directory **lib/** : *SO_WIDTH* e *SO_HEIGHT*, le dimensioni della griglia di simulazione.

Ogni valore di parametro è liberamente modificabile, va tenuto però conto che, per alcuni di questi, l'esecuzione tramite *make run* terminerà immediatamente stampando ad output il problema.

2. Scelte Implementative

2.1 Sincronizzazione iniziale (utilizzo dei semafori)

La sincronizzazione tra i processi prima dell'avvio della simulazione è garantita dai due semafori *SEM_KIDS* e *SEM_START*, dove il primo è usato per sincronizzare il master con un certo tipo di processi figli, mentre il secondo è usato perchè ogni processo creato attenda l'inizio della simulazione. Il flusso (semplificato) iniziale è il seguente.

1. Il master inizializza le proprie variabili e risorse IPC (griglia, coda per statistiche, array di semafori, ...). Come valori iniziali, *SEM_KIDS* = 0 e *SEM_START* = 1.
2. Il master crea SO_SOURCES processi sorgente ed aspetta, tramite una semop di -SO_SOURCES su *SEM_KIDS*, che questi si siano inizializzati.
3. Ogni sorgente, quando ha finito di inizializzarsi, incrementa di 1 il valore di *SEM_KIDS* ed aspetta l'inizio della simulazione con una "wait for zero" sul semaforo *SEM_START*. Quando ogni processo sorgente avrà incrementato *SEM_KIDS*, il master sarà sbloccato.
4. Il master ripete i punti 2. e 3. per creare rispettivamente i *SO_TAXI* processi taxi e per il singolo processo printer, sempre sfruttando *SEM_KIDS*.
5. Ora che ogni processo creato sarà in waiting state su *SEM_START*, il master può farli "partire", assieme alla simulazione, decrementando di 1 *SEM_START*.

2.2 Svolgimento della simulazione

Durante i *SO_DURATION* secondi della simulazione, ogni processo entra in un ciclo infinito in cui esegue un certo tipo di attività :

- **master** : gestisce la terminazione per timeout dei taxi, collezionandone le statistiche e creando un nuovo processo taxi per ognuna di queste.
- **sorgente** : rimane inattivo per un certo lasso di tempo (*BURST_INTERVAL*) per poi generare un certo numero di richieste (*REQS_RATE*). Può svegliarsi in anticipo per soddisfare una richiesta fatta da terminale (sez. 2.3).
- **taxi** : si sposta sulla griglia per raccogliere e poi soddisfare le richieste generate dai processi sorgente, secondo un criterio specificato nella sez. 2.4.
- **printer** : stampa ogni secondo lo stato di occupazione della griglia (da parte dei taxi). La sincronizzazione con questi avviene tramite il semaforo *SEM_PRINT* : quando la stampa sta per avvenire, il printer incrementa di 1 il valore di *SEM_PRINT*.
  Così facendo, ogni taxi intenzionato ad attraversare una cella tramite la funzione access_cell() attenderà, tramite un'istruzione di "wait for zero", che il printer abbia finito di stampare, e che quindi decrementi *SEM_PRINT* per sbloccare i taxi.

2.3 Richieste da terminale

Durante la simulazione è possibile generare una richiesta per un certo processo sorgente inviandogli il segnale *SIGUSR1*. Questo segnale sarà gestito dalla funzione **handle_signal** del processo, permettendo la generazione di una richiesta ed il suo inserimento sulla coda di messaggi del processo sorgente.
Per poter permettere all'utente di consultare i process ID delle sorgenti, una volta che la simulazione sarà pronta a partire (dopo l'esecuzione tramite *make run*) verrà stampato un prompt a terminale ed il programma attenderà senza limitazioni di tempo un input da tastiera.
Il messaggio di prompt indica come consultare i process ID necessari e come creare richieste da un ulteriore finestra di terminale (tramite comando *kill*).
L'utente, una volta pronto, potrà avviare la simulazione premendo il tasto ENTER oppure annullarla premendo Ctrl-C.

2.4 Movimento dei taxi

L'attività di un processo taxi durante il ciclo di simulazione è riassumibile nei seguenti passi :

1. Il taxi cerca, tra le posizioni delle celle sorgente, quella più vicina a sè (stimata tramite la Distanza di Manhattan come numero di celle da attraversare).
2. Il taxi si reca sulla cella sorgente più vicina.

3. Il taxi cerca di raccogliere una richiesta dalla coda di messaggi della sorgente (il cui ID è scritto nella struttura dati della cella stessa) :
    1. Se nessuna nuova richiesta è presente sulla coda, torna al passo 1. (escludendo tra la ricerca di sorgenti quella su cui si trova).
4. Il taxi si sposta dalla cella sorgente alla cella destinazione della richiesta.
5. Una volta soddisfatta la richiesta, torna al passo 1.

Il requisito di timeout dei taxi è gestito principalmente da un timer real-time inizializzato a SO_TIMEOUT secondi e resettato esclusivamente quando un taxi si sposta con successo da una cella ad un'altra.
Il timeout avviene dunque con la ricezione di un SIGALRM che, gestito dal processo, porterà all'invio delle proprie statistiche al processo master sulla apposita coda di messaggi ed alla terminazione del taxi.

Per quanto riguarda l'implementazione del criterio di spostamento verso una sorgente / destinazione, questo è realizzato dalle quattro funzioni **drive_diagonal**, **drive_straight**, **circle_hole** ed **access_cell**. (per informazioni più dettagliate su di queste consultare i file **taxi.h** e **taxi.c**)

- **drive_diagonal** : in primo luogo, un taxi si trova molto probabilmente in una posizione non allineata con la sua prossima destinazione. Dal momento che il taxi può spostarsi al massimo in quattro direzioni (su, giù, sinistra, destra, senza uscire dai bordi) sappiamo che solo due di queste lo avvicineranno al suo obbiettivo. Dunque, finchè non allineato con la cella da raggiungere, il taxi sorteggia una delle due strade migliori e prova ad accedere alla cella in quella direzione tramite **access_cell**. La possibilità di scelta tra due strade, assieme all'assenza di "barriere" di celle inaccessibili, garantiscono che questa funzione eviti implicitamente le celle inaccessibili sul proprio cammino poichè ci sarà sempre una cella accessibile che permetta di avvicinarsi alla destinazione.
- **drive_straight** : supponendo che ora il taxi sia allineato col suo obbiettivo, non gli resta che spostarsi in linea retta verso questo, accedendo una cella alla volta in quella direzione. L'unico ostacolo possibile sarebbe una cella inaccessibile incontrata sul percorso : tramite la funzione **circle_hole** questa viene evitata ed il taxi può continuare il suo viaggio nel ciclo di spostamento.
- **circle_hole** : data la direzione della destinazione rispetto alla posizione del taxi, la cella adiacente per quella strada non è accessibile. È quindi necessario "circumnavigare" almeno parzialmente la cella : prima il taxi si sposta di una cella in una delle due direzioni perpendicolari a quella data, e subito dopo prova ad accedere alla cella adiacente nella direzione originale. Su successo della funzione, il taxi ora si troverà "di fianco" alla cella inaccessibile : l'esecuzione tornerà prima a **drive_straight** e poi al ciclo principale di spostamento, dove in una nuova iterazione verrà chiamata la funzione **drive_diagonal**.
- **access_cell** : regola l'accesso del taxi ad una singola cella. Questo è possibile se il semaforo che rappresenta la capacità attuale della cella ha un valore maggiore di zero. In caso contrario, il taxi attende sul semaforo per un tempo lungo fino al massimo *SO_TIMEOUT* secondi (ed eventualmente terminerà ricevendo un SIGALRM dovuto allo scadere del timer del processo). Una volta "prenotato" il proprio spostamento sulla cella decrementando il semaforo, il taxi dovrà ancora raggiungerla simulando il viaggio con una nanosleep di *Cell.cross_time* nanosecondi (il timer

potrebbe scattare anche durante questo "viaggio"). Una volta raggiunta la cella, il taxi resetta il timer a SO_TIMEOUT secondi, aggiorna le proprie statistiche e ritorna dalla funzione, continuando il suo spostamento.

Il ciclo di spostamento di un taxi è generalizzabile dal seguente esempio :

```
    while (taxi_pos != dest_pos) {
        drive_diagonal(dest_pos);
        drive_straight(dest_pos);
    }
```

- La postcondizione di **drive_diagonal** è che *taxi_pos* sia allineata con *dest_pos*, che è anche la precondizione di **drive_straight**.
- Il risultato di **drive_straight** (*taxi_pos* uguale o meno a *dest_pos*) non impedisce al taxi di continuare a spostarsi nella seguente iterazione di questo ciclo.

3. Altre interpretazioni della traccia

- È stato scelto di avere una coda di messaggi per ogni processo sorgente, l'ID della quale è mantenuto nella struttura dati della cella stessa. Questo implica che un taxi, per poter raccogliere una richiesta, abbia bisogno di trovarsi su di una cella sorgente.
- Le informazioni necessarie alla stampa finale sono collezionate dal master tramite una coda di messaggi apposita, sulla quale vengono inviate da parte dei processi taxi e sorgente prima che terminino.
- Il numero di richieste generate ogni *BURST_INTERVAL* secondi da un processo sorgente è impostato come il rapporto tra il numero di taxi ed il numero di sorgenti nella simulazione, al fine di bilanciare il carico tra queste ultime.
- Il numero di richieste "abortite" al termine della simulazione è determinato dalla somma del numero di taxi terminati (per timeout o per fine simulazione) mentre stanno soddisfacendo una richiesta già raccolta. Questo stato è rappresentato dal campo *mtype* delle statistiche che il taxi invierà al master.