Università degli Studi di Torino (2023), 1–3

ARTICLE TYPE

Relazione Progetto di Sistemi Operativi

Oskar Heise

Matricola: 976322, email: oskar.heise@edu.unito.it, Turno: T2

Consegna

Si intende simulare il traffico di navi cargo per il trasporto di merci di vario tipo, attraverso dei porti. Questo viene realizzato tramite i processi *master*, *mave* e *porto*.
Nella simulazione esistono diversi tipi di merce. Queste vengono generate presso i porti. Le navi nascono da posizioni casuali e senza merce e il loro spostamento viene realizzato tramite una *nanosleep*. I porti hanno una certa domanda e offerta e le proprie banchine vengono gestite tramite *semafori*.

Master.c

Il processo Master è quello incaricato di creare gli altri processi e gestire la simulazione. Viene inoltre creato e inizializzato il semaforo nominato *semaforo_master*, indispensabile per la corretta sincronizzazione dei processi *nave* e *porto*. Prima della creazione di questi ultimi, viene aperta una memoria condivisa, necessaria per il passaggio delle informazioni da porti a navi.
La creazione dei processi figli avviene tramite due cicli *for* distinti, ecco come funziona:

```
43      for(i = 0; i < SO_PORTI; i++){
44          sem_wait();
45          switch (){
46              case -1:
47                  /*errore*/
48              case 0:
49                  /*execvp*/
50              default:
51                  /*inserimento array shm*/
52          }
53      }
```

Subito dopo l'ingresso nel ciclo, il semaforo, inizializzato a 1, viene diminuito di valore, attraverso un'operazione di *sem_wait*. I processi vengono creati attraverso l'operazione di *fork()* e uno *switch* si occupa di compiere le varie operazioni in base al valore ritornato:

- case -1: si è verificato un errore;
- case 0: accedo al processo porto attraverso una *execvp*;
- default: inserisco l'array di strutture contenente la merce nella memoria condivisa.

Per la creazione dei processi *nave* viene fatto lo stesso identico procedimento, con l'aggiunta di un'altra memoria condivisa necessaria per il conteggio delle statistiche delle navi.
Al termine del processo *master*, mi preoccupo di deallocare le memorie condivise e di rimuovere i semafori nominati.

Porti.c

I porti vengono localizzati in una certa posizione nella mappa e gestiscono un numero casuale di banchine. I primi quattro porti vengono sempre generati agli angoli della mappa. Questo viene fatto attraverso *generatore_posizione_iniziale_porto()*, presente nella libreria personalizzata *header.h* di cui parlerò in seguito.
All'inizio del codice dei porti ricevo l'array della memoria condivisa, e procedo a generare tutte le informazioni dei porti. tutte queste informazioni vengono inserite in un array di strutture e inserite nella memoria condivisa. Nel processo viene inoltre gestito *semaforo_master* e incrementato di 1 attraverso l'operazione di *sem_post*.
L'operazione più particolare presente nel codice è la seguente:

```
19      /*setto indici*/
20      pid_di_stampa = (getppid()) / (getpid()
            - SO_PORTI + 1);
```

Questa operazione sarà anche presente nel processo *nave*, e mi è risultata utile per gestire l'operazione di stampa del risultato finale. Il calcolo che viene fatto dà come risultato 0 ad un solo processo e 1 a tutti gli altri. In questo modo, solamente un processo si occuperà della pubblicazione dei report giornalieri e finale, e si eviteranno così ripetizioni indesiderate.
All'interno dei processi porto, *pid_di_stampa* viene utilizzato per la creazione della coda di messaggi.
L'altra operazione dubbiosa presente nel codice è la seguente:

```
56  shared_memory_porto[(getpid() - getppid())
        -1] = porto;
```

In questa linea di codice, utilizzo nuovamente i *pid* dei processi per utilizzarli come indici di un array, ed inserire in maniera corretta le informazioni di ciascun porto all'interno della memoria condivisa che utilizzerò successivamente nei processi *nave*.

Navi.c

Le navi sono i processi più importanti dell'intero programma, ed è qui che gestisco la maggior parte delle operazioni del Progetto, ovvero:

- Movimento delle navi;
- Scambio delle merci;
- Stampa dei report.

Una buona parte del codice è dedicata alla memorizzazione delle variabili utili per le statistiche intermedie e finali, e la prima cosa che faccio nel codice è proprio quella di resettarle. Successivamente, mi occupo di ricevere i diversi array dalla memoria condivisa, in particolare le informazioni dei porti. Dopo, genero tutte le informazioni della nave, in particolare la posizione iniziale della nave, attraverso la funzione *generatore_posizione_iniziale_nave()*.
Una parte fondamentale del processo *nave* è quella della gestione del tempo. Ho deciso di gestire il timer attraverso un ciclo while, dedicato esclusivamente a questa funzione:

```
while((numero_giorno < SO_DAYS+1) || (
    numero_offerta() == 0 &&
    numero_richiesta() == 0))
```

La funzione chiamata all'interno del ciclo while è presente prima del *main* e si occupa interamente della stampa dei report. Al termine della simulazione viene chiamata un'ulteriore funzione che stampa il report finale, ed infine vengono eliminati tutti i processi, attraverso la funzione *kill()*:

```
for(i = (getpid() - SO_NAVI); i <= getpid()
    ; i++){
    pid_kill = i;
    kill(pid_kill, SIGKILL);
}
```

La stampa viene effettuata solamente da un processo, scelto attraverso l'operazione spiegata in nel capitolo precedente e contenuta nella variabile *pid_di_stampa*. L'unità di misura del secondo viene gestita attraverso una *sleep()*, che "addormenta" il processo per un secondo prima di ricominciare a stampare il report. Quando i giorni sono terminati, oppure quando sia la richiesta che l'offerta sono a 0, viene stampato il report finale.
Tornando nel *main*, attraverso un ciclo while, comincia la simulazione vera e propria. Ho diviso la simulazione in diversi step:

1. Controllo della scadenza della merce nelle navi e nei porti;
2. Scelta del porto in cui sbarcare;
3. Spostamento della nave;
4. Memorizzazione dei valori per le statistiche parziali e finali;
5. Caricamento e scaricamento delle navi;

Dopo aver controllato la scadenza della merce, attraverso dei semplici cicli *for*, mi occupo di scegliere il porto in cui far sbarcare la nave:

```
if((informazioni_porto[i].
    merce_richiesta_id == merce_nella_nave[
    indice_nave].id_merce || tappe_nei_porti
     == 0 || merce_nella_nave[indice_nave].
    tempo_vita_merce < 0) &&
    distanza_minima_temporanea >
    distanza_nave_porto(nave.posizione_nave,
     informazioni_porto[i].posizione_porto_X
    , informazioni_porto[i].
    posizione_porto_Y) && informazioni_porto
    [i].numero_banchine_libere != 0  &&
    informazioni_porto[i].numero_lotti_merce
     > 0)
```

Questa condizione, sceglie semplicemente il porto più vicino, qualora la nave fosse priva di carico, ma, in caso contrario, controllando anche se ci sono delle banchine libere e se la richiesta è compatibile con il carico dell'imbarcazione.
Successivamente la nave viene "teletrasportata" al porto dopo un certo numero di tempo, grazie alla funzione *nanosleep* e calcolato attraverso la formula:

$$\frac{\textit{distanza tra posizione di partenza e arrivo}}{\textit{velocit di navigazione}}$$

Una volta arrivata al porto, l'accesso è controllato attraverso un semaforo apposito dichiarato ad inizio codice e qualora lo spazio nelle banchine fosse presente, la nave è pronta a scaricare e caricare la nuova merce.
Una volta fatto ciò, la nave è pronta a ripartire.

header.h

La libreria *header.h* contiene tutte le librerie utilizzate nel codice, alcune costanti, strutture e variabili utilizzate da molti dei processi utilizzati e, ultime ma non meno importanti, tutte le funzioni impiegate nello sviluppo del codice.
Le funzioni più interessanti sono probabilmente quelle dedicate alla gestione delle risorse di *Intern Process Comunication*, in particolare la memoria condivisa e le code di messaggi. In tutte queste funzioni ho inserito un controllo, che avvisa l'utente attraverso *errno*. In caso di errore, le memorie condivise, per esempio, vengono eliminate attraverso la funzione *shmctl* e il comando *IPC_RMID*.
Le altre due funzioni interessanti presenti nella libreria personalizzata, sono ovviamente quelle incaricate della stampa dei report giornalieri e finali:

```
void print_report_finale(struct
    struct_conteggio_nave *conteggio_nave,
    struct struct_merce *merce_nella_nave,
    int numero_giorno, struct struct_porto *
    informazioni_porto, int *
    somma_merci_disponibili, int *
    conteggio_merce_consegnata, int*
    totale_merce_generata_inizialmente, int
    *merce_scaduta_in_nave, int *
    merce_scaduta_in_porto, int
    tappe_nei_porti)
```

Queste due funzioni ricevono un gran numero di dati, e si occupano di stamparli, oltre a fare su alcuni di essi una serie di ulteriori calcoli.

makefile

Make è un'*utility* che automatizza il processo di compilazione ed esecuzione dei programmi. Come da consegna ho aggiunto le seguenti opzioni di compilazione:

```
compexec: compile execute

compile:
    gcc -std=c89 -Wpedantic Master.c -o
        Master
    gcc -std=c89 -Wpedantic Porti.c -o
        Porti
    gcc -std=c89 -Wpedantic Navi.c -o Navi

execute:
    ./Master

gdb:
    gcc -std=c89 -Wpedantic -g -00 Master.c
         -o Master
    gcc -std=c89 -Wpedantic -g -00 Porti.c
        -o Porti
    gcc -std=c89 -Wpedantic -g -00 Navi.c -
        o Navi
    gdb Master
```

Difetti e Conclusione

Per onestà intellettuale, è doveroso elencare i difetti e i malfunzionamenti del codice da me prodotto. Questi problemi si concentrano maggiormente sul calcolo delle statistiche dei report giornalieri e finale. Talvolta, infatti, i risultati finali e intermedi risultano leggermente errati e spesso alcuni porti non vengono popolati di merci. Risulta inoltre difficoltoso eseguire il programma con un elevato numero di navi e merci.