RELAZIONE PROGETTO SISTEMI OPERATIVI 2022/2023

Carrozzino Matteo - Chirio Giorgio

Sessione esami estiva 2023

Introduzione

Il progetto nasce con la stesura del file “master.c” il quale andrà in primis a generare tutti i propri child process quali:

- Il processo merci
- I vari processi porti
- I vari processi navi
- Il processo meteo

La funzione *check_inputs()*, viene usata per controllare che i dati messi in input nel file *env_var.h* seguano le richieste indicate nella consegna.

I file del progetto sono divisi nelle seguenti cartelle:

- SO cartella principale del progetto
- SO/bin cartella contenente i file eseguibili
- SO/src cartella contenente i codici sorgente della simulazione
- SO/lib cartella contenente tutte le librerie scritte da noi

Processo Master

Memorie condivise e semafori

Oltre a fare ciò il ruolo del master è quello di allocare le shared memory e i semafori di configurazione usati dalla simulazione. Nell'esattezza il numero di shared memory utilizzate sono 16. Segue descrizione nell'immagine:

```
/**
 * SHARED MEMORY
 *
 * parent pid + 1: lista Merce per simulazione
 * parent pid + 2: matrice indicante quali Merce possono essere richieste da ciascun porto
 * parend pid + 3: matrice indicante quali offerte possono essere richieste da ciascun porto
 * parent pid + 4: posizione porti
 * parent pid + 5: gestione generazione offerte e richieste dai porti
 * parent pid + 6: matrice di tutte le Merce richieste da tutti i porti
 * parent pid + 7: matrice di tutte le Merce offerte da tutti i poorti
 * parent pid + 8: array che conta il quantitativo di merci consegnate da tutte le navi
 * parent pid + 9: array che visualizza lo status delle navi nel corso della simulazione
 * parent pid + 10: array che salva tutti gli status delle merci in corso della simulazione
 * parent pid + 11: array che va a salvare i porti con l'offerta maggiore di una specifica merce
 * parent pid + 12: array che va a salvare i porti che hanno effettuato il maggior numero di richieste
 * parent pid + 13: array che contiene i pid delle navi
 * parent pid + 14: array che contiene i pid dei porti
 * parent pid + 15: array il quale contiene le statistiche merci all'interno di uno specifico porto
 * parent pid + 16: array dei porti che sono stati colpiti dalla mareggiata
*/

shm_merci_id = shmget(getpid() + 1, sizeof(Merce) * SO_MERCI, 0600 | IPC_CREAT);
tipi_merci = shmat(shm_merci_id, NULL, 0);

shm_richieste_id = shmget(getpid() + 2, sizeof(int) * ((SO_MERCI + 1)*SO_PORTI), 0600 | IPC_CREAT);
arr_richieste = shmat(shm_richieste_id, NULL, 0);

shm_offerte_id = shmget(getpid() + 3, sizeof(int) * ((SO_MERCI + 1) * SO_PORTI), 0600 | IPC_CREAT);
arr_offerte = shmat(shm_offerte_id, NULL, 0);

shm_pos_porti_id = shmget(getpid() + 4, sizeof(double) * (SO_PORTI * 3), 0600 | IPC_CREAT);
arr_pos_porti = shmat(shm_pos_porti_id, NULL, 0);

shm_porti_selezionati_id = shmget(getpid() + 5, sizeof(int), 0600 | IPC_CREAT);
porti_selezionati = shmat(shm_porti_selezionati_id, NULL, 0);

shm_richieste_global_id = shmget(getpid() + 6, sizeof(int) * ((SO_MERCI + 1) * SO_PORTI), 0600 | IPC_CREAT);
arr_richieste_global = shmat(shm_richieste_global_id, NULL, 0);

shm_offerte_global_id = shmget(getpid() + 7, sizeof(int) * ((SO_MERCI + 1) * SO_PORTI), 0600 | IPC_CREAT);
arr_offerte_global = shmat(shm_offerte_global_id, NULL, 0);

shm_merci_cosegnate_id = shmget(getpid() + 8, sizeof(int) * SO_MERCI, 0600 | IPC_CREAT);
merci_consegnate = shmat(shm_merci_cosegnate_id, NULL, 0);

shm_statusNavi_id = shmget(getpid() + 9, sizeof(int) * SO_NAVI * 6, 0600 | IPC_CREAT);
statusNavi = shmat(shm_statusNavi_id, NULL, 0);

shm_statusMerci_id = shmget(getpid() + 10, sizeof(int) * (SO_MERCI) * 5, 0600 | IPC_CREAT);
statusMerci = shmat(shm_statusMerci_id, NULL, 0);

shm_maxOfferte_id = shmget(getpid() + 11, sizeof(int) * SO_MERCI * 2, 0600 | IPC_CREAT);
maxOfferte = shmat(shm_maxOfferte_id, NULL, 0);

shm_maxRichieste_id = shmget(getpid() + 12, sizeof(int) * SO_MERCI * 2, 0600 | IPC_CREAT);
maxRichieste = shmat(shm_maxRichieste_id, NULL, 0);

shm_pidNavi_id = shmget(getpid() + 13, sizeof(pid_t) * SO_NAVI, 0600 | IPC_CREAT);
shmPidNavi = shmat(shm_pidNavi_id, NULL, 0);

shm_pidPorti_id = shmget(getpid() + 14, sizeof(pid_t) * SO_PORTI, 0600 | IPC_CREAT);
shmPidPorti = shmat(shm_pidPorti_id, NULL, 0);

shm_statusPorti_id = shmget(getpid() + 15, sizeof(int) * SO_PORTI * 6, 0600 | IPC_CREAT);
statusPorti = shmat(shm_statusPorti_id, NULL, 0);

shm_portiSwell_id = shmget(getpid() + 16, sizeof(int) * SO_DAYS, 0600 | IPC_CREAT);
portiSwell = shmat(shm_portiSwell_id, NULL, 0);
/*Fine allocazione delle shared memory*/
```

I semafori creati dal master sono due. Segue descrizione nell'immagine:

```
/**
 * SEMAFORI
 *
 * parent pid + 0: fine configuarazione iniziale
 * parent pid + 1: fine creazione matrice richieste/offerte dei porti
*/

sem_config_id = semget(getpid(), 1, 0600 | IPC_CREAT);
sem_set_val(sem_config_id, 0, (SO_NAVI + SO_PORTI + 1));

sem_offerte_richieste_id = semget(getpid() + 1, 1, 0600 | IPC_CREAT);
sem_set_val(sem_offerte_richieste_id, 0, 1);
/*Fine Sezione crezione semaforo per configuazione*/
```

Il primo, sem_config_id serve per sincronizzare la fine dell'inizializzazione dei processi figli, mentre sem_offerte_richieste_id è servito in quanto la nostra scelta implementativa per la gestione della scelta delle merci generabili presso i porti prevede due matrici:

- arr_offerte: serve a sapere quali merci possono essere offerte da uno specifico porto, identificato attraverso il suo PID. All'interno della matrice ogni riga rappresenta un porto e le colonne l'ID di tutte le merci.
- arr_richieste: serve a sapere quali merci possono essere richieste da uno specifico porto, identificato attraverso il suo PID. All'interno della matrice ogni riga rappresenta un porto e le colonne l'ID di tutte le merci.

In queste due matrici vengono riempite dalla funzione *gen_richiesta_offerta(),* con il supporto di *gen_offerta(), indicherà con un 1 le merci che possono essere offerte/richieste dal porto con uno 0 altrimenti*

```
void gen_richiesta_offerta(int * pidPorti, int * arr_richieste, int * arr_offerte, int print)
```

intestazione della funzione

Stampa giornaliera

Il report della simulazione viene fatto giornalmente dal master tramite la funzione *dailyPrint()*, e viene specificato metriche come le merci generate divise per tipo e stato e lo status delle navi.

```
void dailyPrint(int *, int *, int *, int *, int);
```

intestazione della funzione

Processo merci

Generazione delle merci

Per la generazione delle merci veniva richiesta la generazione casuale di vita e peso di ogni merce. Oltre a ciò abbiamo deciso di inserire il campo tipo, un valore numerico nel range da 1 a *SO_MERCI* usato per identificare il tipo di merce che si sta trattando. Questo array viene messo nella shared memory *getppid()+1* , in modo che ogni processo che ne abbia bisogno possa accedervi liberamente. Segue spezzone di codice contenente la generazione randomica dei parametri sopracitati:

```
for(i = 0; i < SO_MERCI; i++){
    tipi_merci[i].type = (i+1);
    tipi_merci[i].weight = ((rand() % SO_SIZE) + 1);
    tipi_merci[i].life = ((rand() % SO_MAX_VITA) + SO_MIN_VITA);
}
```

Ad ogni tipo di merce generata viene assegnato un identificatore, un peso ed una vita

La Mappa

Abbiamo deciso di implementare la mappa usando come punto di origine il centro del quadrato. Seguono alcuni importanti punti di riferimento con coordinate (x,y):

- Centro della mappa (0, 0)
- Angolo superiore sinistro (-SO_LATO/2, SO_LATO/2)
- Angolo superiore destro (SO_LATO/2, SO_LATO/2)
- Angolo inferiore sinistro (-SO_LATO/2, -SO_LATO/2)
- Angolo inferiore destro (SO_LATO/2, -SO_LATO/2)

Processi porto

Inizializzazione

L'inizializzazione dei processi porto comincia con l'attesa dei suddetti per la fine della generazione delle matrici contenenti le merci generabili da ogni porto. Una volta terminata l'attesa ad ogni porto viene assegnata una posizione sulla mappa: i primi quattro porti generati sono posizionati ai quattro angoli della mappa, mentre ai rimanenti *SO_PORTI - 4* porti viene assegnato una posizione casuale nei confini della mappa. Più porti possono trovarsi nella stessa posizione, in quanto non viene effettuato alcun controllo sulle posizioni generate. Le posizioni dei porti insieme ai loro PID vengono inserite nell'array presente in shared memory con id *getppid() + 4* per far si che tutte le navi sappiano la posizione dei porti. Dopo aver generato la propria posizione, il singolo porto genera un numero di banchine compreso tra 1 e SO_BANCHINE, che verrà usato per inizializzare un semaforo usato per regolare l'attracco delle barche nel porto. Una volta che il porto finisce la sua inizializzazione occupa il semaforo di configurazione ed entra nel loop principale attendendo un messaggio dalle navi.

Generazione giornaliera delle offerte e richieste

Ogni giorno il master invia ai porti un segnale che quando ricevuto avvia la procedura di generazione. Questa consiste innanzitutto con il passare del tempo e l'eliminazione delle merci scadute tra quelle offerte del porto. Dopo aver eliminato le merci non più idonee si passa alla generazione delle merci prima richieste poi offerte. Dato che ogni porto può generare solo una data quantità di merce ogni giorno questa viene divisa randomicamente tra merci richieste e offerte. Per tener traccia delle merci contenute in ogni porto utilizziamo due linked list implementate da noi nel file *list.c*

```c
if(listaOfferte.top != NULL){
    listSubtract(&listaOfferte, qta_merci_scadute, statusMerci, 1);
    /*La lista delle richieste scade? */
    /*listSubtract(&listaRichieste, qta_merci_scadute);*/
}

request_offer_gen(tipi_merce, porti_selezionati, matr_richieste, perc_richieste, 0);
request_offer_gen(tipi_merce, porti_selezionati, matr_offerte, 100 - perc_richieste, 1);
```

Codice per la distruzione delle merci scadute nelle offerte e le chiamate per la generazione di richiesta e offerta

Gestione messaggi

Quando il porto entra nel loop principale della simulazione si mette in attesa dei messaggi provenienti dalle navi che intendono attraccare. Quando viene ricevuto un messaggio, il porto decide cosa fare basandosi sul campo *operation* del suddetto. Questo campo può assumere valori tra -1 e 4, con ogni valore indicante una diversa operazione elencate di seguito:

- -1: Negazione della richiesta di attracco o rilascio della banchina
- 0: Richiesta da parte della nave di attracco al porto
- 1: Richiesta da parte della nave di scaricare la merce del tipo *messaggio.extra*
- 2: Richiesta da parte della nave di caricare la merce del tipo messaggio.extra
- 3: Messaggio contenente la vita rimanente della merce appena caricata
- 4: Richiesta da parte della nave di liberare la banchina andandosene dal porto

Il campo *pid_nave* dei messaggi indica al porto da quale nave è stato inviato il messaggio e serve per la corretta gestione dei messaggi di risposta.

Carico

Quando il porto riceve un messaggio con campo *operation* settato a 2 cominciano le operazioni di carico della merce del tipo specificato nel campo *extra*. Il porto elimina l'occorrenza "più vecchia" della merce richiesta, inviando alla nave la vita rimanente di quella specifica merce, cosicché la nave possa inserire nella propria stiva la merce.

```
case 2:
    Operation.operation = 3;
    Operation.type = (unsigned int)Operation.pid_nave;
    listRemoveToLeft(&listaOfferte, rem_life, Operation.extra);
    statusMerci[((Operation.extra - 1) * 5)] -= 1;     /*Tolgo al porto*/
    statusMerci[((Operation.extra - 1) * 5) + 1] += 1; /*Inserisco su nave*/
    Operation.extra = * rem_life;
    msgsnd(msg_porti_navi_id, &Operation, sizeof(int) * 2  + sizeof(pid_t), 0);
    statusPorti[(rigaStatus * 6) + 3] += 1;
    break;
```

Codice per la gestione dei carichi nel porto

Scarico

Quando il porto riceve un messaggio con campo *operation* settato a 1 cominciano le operazioni di scarico della merce del tipo specificato nel campo *extra*. Il porto elimina la prima occorrenza della merce scaricata inviando alla nave un messaggio sempre con *operation* settato a 1 per indicare il completamento dello scarico di quella merce.

```
case 1:
    listRemoveToLeft(&listaRichieste, NULL, Operation.extra);
    statusMerci[((Operation.extra - 1) * 5) + 2] += 1; /*Inserisco al porto*/
    statusMerci[((Operation.extra - 1) * 5) + 1] -= 1; /*Tolgo su nave*/
    Operation.type = (unsigned int)Operation.pid_nave;
    Operation.extra = 0;
    Operation.operation = 1;
    msgsnd(msg_porti_navi_id, &Operation, sizeof(int) * 2 + sizeof(pid_t), 0);
    statusPorti[(rigaStatus * 6) + 2] += 1;
    break;
```

Codice per la gestione degli scarichi nel porto

Processi nave

Inizializzazione

Il processo nave nasce in una posizione casuale nella mappa con una stiva vuota.

Gestione dei viaggi

Il viaggio viene gestito dalla funzione *calcoloDistanza()* e *travel()*. La prima serve a calcolare la distanza tra la posizione attuale della nave e il porto che è stato scelto casualmente. Questo calcolo viene effettuato usando il teorema di pitagora con l'appoggio della libreria *<math.h>*.
Travel() tutte le volte che verrà richiamata andrà ad effettuare un calcolo per la gestione dei tempi che verranno assegnati alla *nanosleep().*
Prima del viaggio la nave aggiorna i propri status per la corretta stampa del report giornaliero. Dopo il viaggio viene ricalcolata anche la stiva in quanto le merci presenti possono essere scadute. Tutto questo viene eseguito prima di un possibile attracco presso un porto.

```c
void travel(int * statusNavi, double distanza){
    struct timespec req, rem;
    int modulo;
    int i = 0;
    double nsec;
    rem.tv_nsec = 0;
    rem.tv_sec = 0;
    if(distanza < SO_SPEED){
        modulo = 0;
        nsec = (distanza / SO_SPEED) * CONVERSION_SEC_NSEN;
        req.tv_sec = (time_t)(modulo);
        req.tv_nsec = (long)nsec;
    }
    else{
        modulo = (int)distanza / SO_SPEED;
        nsec = ((distanza / SO_SPEED) - modulo) * CONVERSION_SEC_NSEN;
        req.tv_sec = (time_t)(modulo);
        req.tv_nsec = (long)(nsec);
    }
    while(req.tv_nsec != 0 || req.tv_sec != 0){
        if(nanosleep(&req, &rem) == -1){
            req.tv_nsec = rem.tv_nsec;
            req.tv_sec = rem.tv_sec;
        }
        else{
            req.tv_nsec = 0;
            req.tv_sec = 0;
        }
    }
    if(stiva.top != NULL){
        for(i = 0; i < modulo; i++){
            listSubtract(&stiva, merci_scadute, statusMerci, 0);
        }
    }
}
```

Codice per la gestione dei viaggi

Gestione messaggi e operazioni nel porto

Come specificato in precedenza nella sezione dei porti, le navi per eseguire l'attracco, lo scarico e il carico scambiano messaggi con i porti. L'imbarcazione dopo aver terminato il *travel()* invia un messaggio contenente la richiesta di attracco. Se quest'ultima ha esito positivo, la nave procederà con le operazioni di carico e di scarico descritte come segue.

Carico

La nave scorre l'array di merci offerte dal porto e le carica fino a che ha spazio nella stiva. Per ogni merce caricata invia un messaggio al porto con il tipo che intende caricare, e come risposta riceve la vita rimanente alla merce che ha caricato.

Scarico

La nave scorre la propria stiva confrontando le merci presenti con quelle accettate dal porto. Per ogni merce scaricata la elimina dalla sua stiva e invia un messaggio al porto contenente il tipo della merce da eliminare tra le richieste.

Processo meteo

Inizializzazione

Viene calcolato il tempo di attesa del Maelstrom che verrà utilizzato nel corpo principale. E' stato usato lo stesso criterio per il calcolo del tempo di attesa dei viaggi delle navi.

Cambiamenti del meteo nella simulazione e invio segnali

Ogni giorno il master invia al processo meteo un segnale che fa iniziare la procedura dailyDisaster(). Questa procedura sceglie casualmente una nave tra quelle non distrutte e un porto ed invia a loro il segnale *SIGUSR2* che richiama le funzioni *swellPause()* nei porti e *stormPause()* nelle navi mettendone in pausa l'esecuzione usando *nanosleep()*.

```
void dailyDisaster(){

    /**
     * SIGUSR2 -> NAVE ATTENDE ATTENDE MENTRE VIAGGIA / PORTO ATTENDE
     * SIGABRT -> MORTE DELLA NAVE
    */

    int pid_random;
    pid_random = rand() % SO_NAVI;
    while(pidNavi[pid_random] == -1){
        pid_random = rand() % SO_NAVI;
    }
    kill(pidNavi[pid_random], SIGUSR2);
    statusNavi[(pid_random * 6) + 5] += 1;

    pid_random = rand() % SO_PORTI;
    kill(pidPorti[pid_random], SIGUSR2);
    portiSwell[swellIndex] = pidPorti[pid_random];
    swellIndex += 1;
}
```

Codice per la gestione di STORM e SWELL

Ogni SO_MAELSTROM ore simulate una nave scelta casualmente viene distrutta con il segnale SIGTERM, e il suo PID nell'array dei pid in shared memory viene impostato a -1. Se tutte le navi sono state distrutte il meteo manda un segnale SIGABRT al master per segnalare la terminazione della simulazione.

Fine Simulazione

Quando sono passati SO_DAYS giorni il master invia a tutti i processi figlio ancora attivi il segnale SIGABRT che causa la loro terminazione. Il termine dei giorni non è l'unica condizione per la terminazione della simulazione, le altre due sono la distruzione totale delle navi e la completa soddisfazione di richiesta e offerta.
Una volta che tutti i figli sono terminati, il master stampa un resoconto totale della simulazione tramite la funzione finalReport(), che come già descritto per la stampa giornaliera stampa le metriche della simulazione

```
void finalReport(int *, int *, int *, int *, int *, int *, int *);
```

intestazione della funzione

Dopo *finalReport()* si provvede alla deallocazione di tutte le variabili IPC tra cui semafori, shared memory e code di messaggi.