Università di Cagliari
Facoltà di Scienze MM.FF.NN
Corso di Laurea in Informatica

TESINA CORSO SISTEMI OPERATIVI 1

Docente S. Carta (salvatore@unica.it)
Tutor L. Boratto (ludovico.boratto@unica.it)
A.A. 2011 - 2012

GRUPPO

STEFANO SIMONE MELIS	44790
SILVIA MARRAS	44927

specifica versione 1.1

Introduzione

Entrambe le versioni del progetto sono suddivise nei seguenti file:
space_main.c, space_game.h, space_game.c, space_model.h, space_model_c, space_comm.h, space_comm.c.

- **space_main.c:** come suggerisce il nome, è il **main** del programma e si occupa di **inizializzare** il gioco **creando** i task **produttori** (giocatore e astronavi nemiche) e il task **consumatore** (gestore della grafica e delle collisioni);
- **space_game.h**: contiene, oltre alle **dichiarazioni** delle funzioni, tutte le **define** utilizzate;
- **space_game.c**: contiene le **funzioni principali** eseguite in loop dai vari task, più qualche **procedura** atta a riutilizzare parti di codice.
- **space_model.h**: contiene, oltre alle **dichiarazioni** delle funzioni, anche quelle delle **matrici** di caratteri delle varie entità grafiche e tutte le **define** utilizzate;
- **space_model.c**: contiene le **inizializzazioni** delle matrici di caratteri delle varie entità grafiche e gli **algoritmi** di **disegnazione** e **cancellazione** delle stesse sul terminale;
- **space_comm.h**: contiene, oltre alle **dichiarazioni** delle funzioni di comunicazione, la **struct** dei messaggi e tutte le **define** utilizzate;
- **space_comm.c**: contiene le funzioni di **comunicazione**.

Abbiamo scelto di utilizzare uno stile di scrittura del codice molto simile a quello della programmazione ad oggetti, in cui i **nomi** di variabili sono dati in **camelCase** e i **tipi utente** in **UpperCamelCase**.

Documentazione versione 1

Descrizione struttura del programma (elenco delle funzioni e descrizione sommaria del main)

Il programma ha una struttura **produttore-consumatore** di tipo **molti a uno**: i produttori in questo caso sono le varie **entità grafiche** (l'astronave del giocatore, le astronavi nemiche, le bombe e i missili) che comunicano le proprie coordinate all'unico consumatore, iil main, il quale ha il compito di disegnarle e controllare le loro collisioni.

space_main.c

- ***void main()***: è il main del programma, si occupa di creare i vari task principali e le rispettive pipe di comunicazione.

space_game.c

- ***bombInit:*** invocata da un'astronave nemica, genera una bomba;
- ***bombLoop:*** loop vitale del task bomba, muore quando raggiunge il bordo inferiore dello schermo;
- ***checkCollision:*** invoca checkCoordinates passandogli due variabili EntityParams;
- ***checkCoordinates:*** controlla se le aree occupate dalle due EntityParams passategli si sovrappongono;
- ***enemyInit:*** inizializza correttamente la funzione enemyLoop a seconda di chi la invoca;
- ***enemyLoop:*** loop vitale del task astronave nemica. Prima di morire genera un task astronave nemica di livello superiore (fino al terzo, poi muore senza generare altri task);
- ***epilogue:*** procedura di chiusa, stampa a schermo l'esito della partita;
- ***isAlive:*** controlla se un determinato pid è maggiore di 1;
- ***mainLoop***: funzione-loop del consumatore. Il suo compito è quello di leggere i messaggi depositati nella mainPipe e, a seconda del mittente e del tipo di richiesta, eseguire una certa procedura/funzione. Ogni volta che un task invia le proprie coordinate, il mainLoop le salva in una lista basata su un array di EntityParams. Nella fase di inizializzazione ha il compito di creare un task timer usato per far abbassare le astronavi nemiche.
- ***missileLoop:*** loop vitale del task missile, muore quando raggiunge i bordi laterali e superiore dello schermo;
- ***missileInit:*** invocata dal giocatore, genera due missili e un timer;
- ***removeEntityFromList:*** pone a -1 i campi della k-esima posizione di un array di EntityParams;

- ***retrieveSizeInformation:*** recupera la dimensione in caratteri dell'entità passatagli e li salva su dei puntatori;
- ***starshipLoop***: funzione-loop dell'astronave controllata dal giocatore. Leggendo l'input da tastiera è in grado di spostare a destra e sinistra l'entità. Alla pressione della barra spaziatrice genera tre task: due missili e un timer che impedisce la generazione di altri missili fino alla sua morte.
- ***TimerFire***: se invocata da starshipLoop rallenta il rateo di fuoco dell'astronave del giocatore, se invocata da enemyLoop invia al task corrispondente l'ordine di aprire il fuoco dopo una pausa arbitraria;
- ***TimerMove***: invia a ripetizione l'ordine di spostarsi verso il basso, dopo una pausa arbitraria, a tutti i task astronave nemica.

space_graphics.c

- ***chloader:*** *r*estituisce un carattere di tipo chtype corrispondente leggendo un carattere di tipo char;
- ***moveEntity***: interfaccia per printEntity;
- ***moveShip***: simile a moveEntity, ma agisce in funzione del modello di astronave scelto dal giocatore;
- ***moveWeapon***: disegna/cancella l'entità missile/bomba per cui è stata invocata;
- ***placeEnemies***: algoritmo studiato per assegnare coordinate di partenza ai task astronavi nemiche;
- ***printEntity***: stampa il modello grafico dell'entità per cui è stata invocata.

space_comm.c

- ***createPipe:*** crea una pipe che può essere bloccante o non bloccante;
- ***receiveMessage***: scrive su una specifica pipe un messaggio di tipo entityParams;
- ***sendMessage***: legge su una specifica pipe un messaggio di tipo entityParams;

Descrizione architettura del programma

Descrizione della finalità e della implementazione di ciascun task

main: Viene generato all'avvio del gioco e la sua funzione iniziale è quella di inizializzare la partita generando tutti i processi principali (l'astronave del giocatore, le astronavi nemiche e il timer di spostamento verticale), le loro posizioni di partenza e le pipe di comunicazione; una volta entrato nel suo loop vitale si occupa di mantenere una lista aggiornata di tutte le entità grafiche presenti in campo e disegnarle ogni volta che riceve le coordinate aggiornate; inoltre gestisce le collisioni arma-entità, entità-entità, giocatore-entità, e le relative uccisioni degli altri processi generati. Termina quando ogni entità astronave nemica viene distrutta o quando l'astronave del giocatore viene colpita o quando un'entità astronave nemica raggiunge il bordo inferiore dell'area di gioco. E' suo compito anche gestire la chiusura del gioco e mostrare sul terminale l'esito della partita.

starship: Questo processo, creato nella parte più bassa dello schermo, viene comandato dall'utente: viene analizzato lo standard input per riconoscere la pressione dei tasti. Gli unici tasti consentiti sono sinistra, destra e barra spaziatrice, la quale serve per generare due missili. Il loop vitale di questo processo è un ciclo infinito: se viene colpito da una bomba invia una richiesta di terminazione al main, il quale invoca una kill sul pid corrispondente.

enemy: parte dalla parte più alta dello schermo; è in grado di muoversi a destra, a sinistra e verso il basso. Quando vuole spostarsi invia al main la nuova posizione e la direzione del moto (sinistra o destra): il main analizza questi dati e, se per caso è in prossimità o collide con un'entità alleata, il main gli ordina di tornare indietro finchè non esce dalla sovrapposizione. L'azione di fuoco è comandata da un timer generato alla nascita, che a intervalli regolari invia il permesso di aprire il fuoco. Alla morte del processo, che può avvenire solo per collisione su un missile generato dal giocatore, come ultima istruzione genera una nave di livello superiore, passandogli le stesse pipe utilizzate da esso, garantendo così una continuità nella comunicazione con il main.

missile: si muove in diagonale, verso destra o verso sinistra. La sua vita termina naturalmente quando raggiunge i bordi laterali e superiore dello schermo, altrimenti viene terminato dal main se viene rilevata una collisione su un'entità.

bomb: simile al missile, l'unica differenza è che si muove verticalmente verso il basso e termina appena raggiunge il bordo inferiore dello schermo e può collidere solo con l'entità astronave del giocatore.

timerMove: viene creato dal main nella fase di inizializzazione: gestisce lo spostamento verticale delle entità astronavi nemiche, inviando ad intervalli regolari l'ordine di muoversi di una coordinata più in basso.

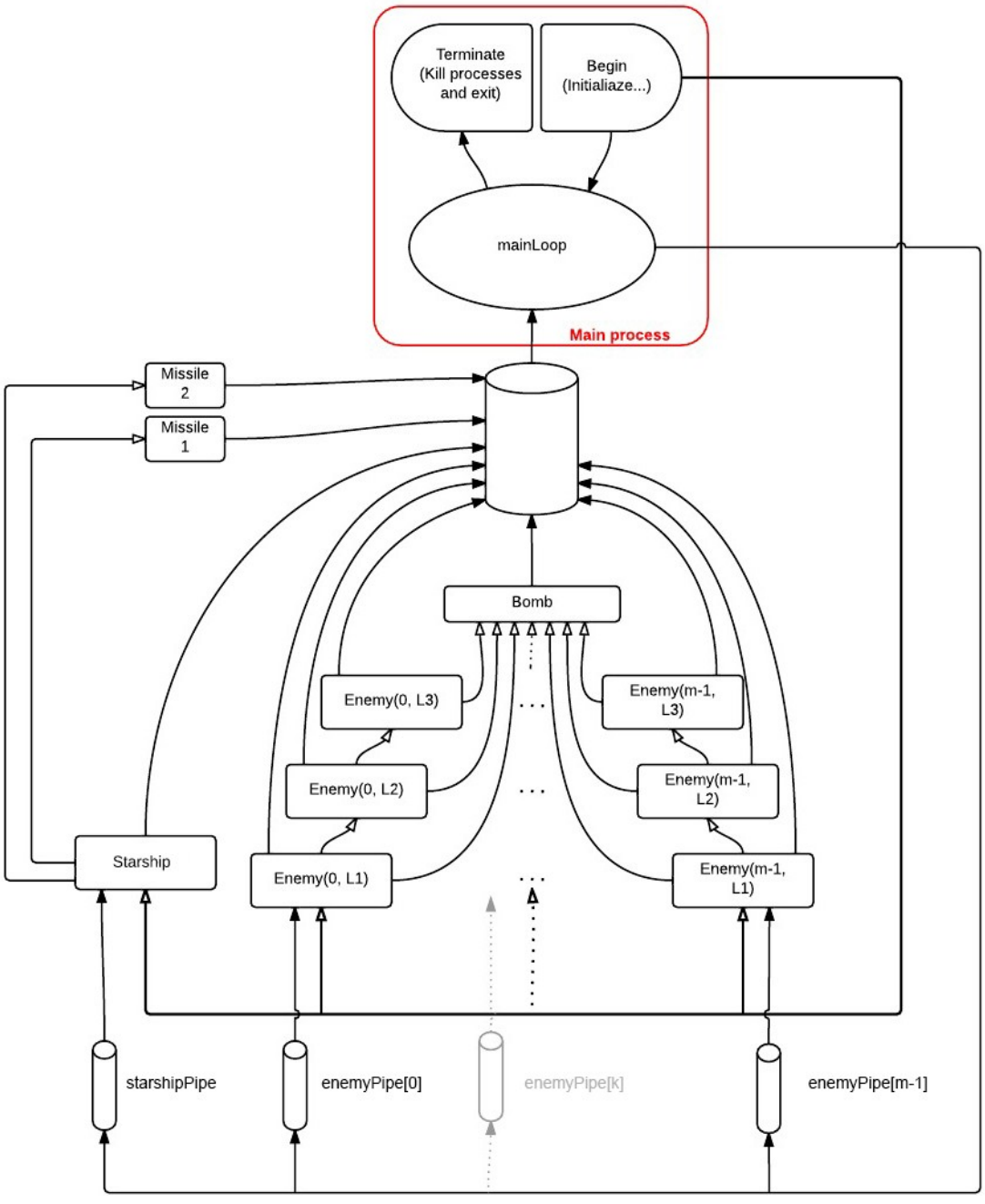
timerFire: quando invocato dallo starship invia un messaggio che blocca la possibilità di generare altri missili fino alla sua morte; quando invocato dall'enemy invia l'ordine di far fuoco a intervalli regolari.

Descrizione delle primitive di sincronizzazione utilizzate (che problemi risolvono e perchè)

In questo progetto abbiamo utilizzato le **linux-pipe** secondo la seguente configurazione: il CONSUMATORE dispone, in **lettura**, di un'**unica** "**mainPipe**" in cui confluiscono tutte le richieste da parte dei produttori e di **M+1** pipe in **scrittura** per inviare le notifiche di **collisione** alle altre entità principali (M è il numero di entità nemiche). A loro volta, i produttori sono consumatori nei confronti dei **timer**. Inoltre, le pipe in lettura delle varie entità sono state inizializzate come **non bloccanti** tramite funzioni della libreria **fcntl**, per garantire così una comunicazione **asincrona** durante le iterazioni dei propri loop vitali. La **scrittura** sulla pipe è realizzata tramite la funzione **write()**, mentre la **lettura** tramite la funzione **read()**, che ha anche come side-effect quello di **rimuovere** dalla pipe il messaggio letto.

Descrizione grafica della comunicazione fra i task

Per evitare un'alta complessità dello schema sono stati omessi task timer, i quali comunicano tramite pipe come tutti gli altri task.



Documentazione versione 2

Descrizione struttura del programma (elenco delle funzioni e descrizione sommaria del main)

La struttura di questa versione è analoga a quella precedente, con la differenza che al posto delle pipe abbiamo usato dei buffer.

space_main.c

- ***void main()***: è il main del programma, si occupa di creare i vari task principali e di inizializzare mutex, semaphore e contatori dei buffer.

space_game.c

- ***bombInit:*** invocata da un'astronave nemica, genera una bomba;
- ***bombLoop:*** loop vitale del task bomba, muore quando raggiunge il bordo inferiore dello schermo;
- ***checkCollision:*** invoca checkCoordinates passandogli due variabili EntityParams;
- ***checkCoordinates:*** controlla se le aree occupate dalle due EntityParams passategli si sovrappongono;
- ***enemyInit:*** inizializza correttamente la funzione enemyLoop a seconda di chi la invoca;
- ***enemyLoop:*** loop vitale del task astronave nemica. Prima di morire genera un task astronave nemica di livello superiore (fino al terzo, poi muore senza generare altri task);
- ***epilogue:*** procedura di chiusa, stampa a schermo l'esito della partita;
- ***mainLoop***: funzione-loop del consumatore. Il suo compito è quello di leggere i messaggi depositati nella mainPipe e, a seconda del mittente e del tipo di richiesta, eseguire una certa procedura/funzione. Ogni volta che un task invia le proprie coordinate, il mainLoop le salva in una lista basata su un array di EntityParams. Nella fase di inizializzazione ha il compito di creare un task timer usato per far abbassare le astronavi nemiche.
- ***missileLoop:*** loop vitale del task missile, muore quando raggiunge i bordi laterali e superiore dello schermo;
- ***missileInit:*** invocata dal giocatore, genera due missili e un timer;
- ***removeEntityFromList:*** pone a -1 i campi della k-esima posizione di un array di EntityParams;
- ***retrieveSizeInformation:*** recupera la dimensione in caratteri dell'entità passatagli e li salva su dei puntatori;
- ***starshipLoop***: funzione-loop dell'astronave controllata dal giocatore. Leggendo l'input da tastiera è in grado di spostare a destra e sinistra l'entità. Alla pressione della barra spaziatrice genera tre task: due missili e un timer che impedisce la generazione di altri missili fino alla sua morte.
- ***TimerFire***: se invocata da starshipLoop rallenta il rateo di fuoco dell'astronave del giocatore, se invocata da enemyLoop invia al task corrispondente l'ordine di aprire il fuoco dopo una pausa arbitraria;
- ***TimerMove***: invia a ripetizione l'ordine di spostarsi verso il basso, dopo una pausa arbitraria, a tutti i task astronave nemica.

space_graphics.c

- ***chloader:*** *r*estituisce un carattere di tipo chtype corrispondente leggendo un carattere di tipo char;
- ***moveEntity***: interfaccia per printEntity;
- ***moveShip***: simile a moveEntity, ma agisce in funzione del modello di astronave scelto dal giocatore;
- ***moveWeapon***: disegna/cancella l'entità missile/bomba per cui è stata invocata;
- ***placeEnemies***: algoritmo studiato per assegnare coordinate di partenza ai task astronavi nemiche;
- ***printEntity***: stampa il modello grafico dell'entità per cui è stata invocata.

space_comm.c

- ***insertBuffer:*** inserisce un nuovo item nel buffer passatogli come parametro;
- ***removeBuffer***: rimuove un item nel buffer passatogli come parametro.

Descrizione architettura del programma

Descrizione della finalità e della implementazione di ciascun task

main: Viene generato all'avvio del gioco e la sua funzione iniziale è quella di inizializzare la partita generando tutti i thread principali (l'astronave del giocatore, le astronavi nemiche e il timer di spostamento verticale), le loro posizioni di partenza e inizializzare tutti i mutex, semaphore e contatori usati nel gioco; una volta entrato nel suo loop vitale si occupa di mantenere una lista aggiornata di tutte le entità grafiche presenti in campo e disegnarle ogni volta che riceve le coordinate aggiornate; inoltre gestisce le collisioni arma-entità, entità-entità, giocatore-entità, e le relative uccisioni degli altri processi generati. Termina quando ogni entità astronave nemica viene distrutta o quando l'astronave del giocatore viene colpita o quando un'entità astronave nemica raggiunge il bordo inferiore dell'area di gioco. E' suo compito anche gestire la chiusura del gioco e mostrare sul terminale l'esito della partita.

starship: Questo thread, creato nella parte più bassa dello schermo, viene comandato dall'utente: viene analizzato lo standard input per riconoscere la pressione dei tasti. Gli unici tasti consentiti sono sinistra, destra e barra spaziatrice, la quale serve per generare due missili. Il loop vitale di questo processo è un ciclo infinito: se viene colpito da una bomba invia una richiesta di terminazione al main, il quale invoca una pthread_cancel sul thread-id corrispondente.

enemy: parte dalla parte più alta dello schermo; è in grado di muoversi a destra, a sinistra e verso il basso. Quando vuole spostarsi invia al main la nuova posizione e la direzione del moto (sinistra o destra): il main analizza questi dati e, se per caso è in prossimità o collide con un'entità alleata, il main gli ordina di tornare indietro finchè non esce dalla sovrapposizione. L'azione di fuoco è comandata da un timer generato alla nascita, che a intervalli regolari invia il permesso di aprire il fuoco. Alla morte del processo, che può avvenire solo per collisione su un missile generato dal giocatore, come ultima istruzione genera una nave di livello superiore che utilizzerà lo stesso buffer usato fino a quel momento, garantendo così una continuità nella comunicazione con il main.

missile: si muove in diagonale, verso destra o verso sinistra. La sua vita termina naturalmente quando raggiunge i bordi laterali e superiore dello schermo, altrimenti viene terminato dal main se viene rilevata una collisione su un'entità.

bomb: simile al missile, l'unica differenza è che si muove verticalmente verso il basso e termina appena raggiunge il bordo inferiore dello schermo e può collidere solo con l'entità astronave del giocatore.

timerMove: viene creato dal main nella fase di inizializzazione: gestisce lo spostamento verticale delle entità astronavi nemiche, inviando ad intervalli regolari l'ordine di muoversi di una coordinata più in basso.

timerFire: quando invocato dallo starship invia un messaggio che blocca la possibilità di generare altri missili fino alla sua morte; quando invocato dall'enemy invia l'ordine di far fuoco a intervalli regolari.

Descrizione delle primitive di sincronizzazione utilizzate (che problemi risolvono e perchè)

In questo progetto abbiamo utilizzato le **librerie p-thread** secondo la seguente configurazione: il consumatore dispone, usato in **lettura**, di un **unico** "**mainBuffer**" in cui confluiscono tutte le richieste da parte dei produttori e di **M+1** buffer usati in **scrittura** per inviare le notifiche di **collisione** alle altre entità principali (M è il numero di entità nemiche). Per quanto riguarda la **costruzione** dei buffer, abbiamo usato **array** dichiarati nello **scope globale**, in modo da renderli **visibili** agli altri thread. Dato che si tratta di spazi di memoria usati in **condivisione**, abbiamo opportunamente **protetto** l'accesso sia in lettura che in scrittura per mezzo di **mutex** per garantire la mutua esclusione, mentre per ogni lettura abbiamo utilizzato anche i **semaphore**, così da mettere in **attesa** il thread che legge da buffer **vuoto**. Dato che le librerie **ncurses** usate in congiunzione con le librerie **p_thread** si sono rivelate abbastanza farraginose, in quanto **non thread-safe**, abbiamo usato i mutex ogni qualvolta si doveva scrivere sullo schermo del terminale, in modo tale da evitare (o quanto meno limitare) **bug grafici**.

Descrizione grafica della comunicazione fra i task

Per evitare un'alta complessità dello schema sono stati omessi task timer, i quali comunicano tramite buffer come tutti gli altri task.