Documentazione Progetto FORZA 4

Pietro BIANCHEDI VR455975; Niccolò ISELLE VR456714

A.A. 2022/2023

1 Introduzione

Il progetto riguarda lo sviluppare il gioco "Forza 4". Il gioco si svolge tra due giocatori, su un campo che da dimensione minima di 5x5 dove ogni giocatore deve decidere una colonna e lasciar cadere il proprio "gettone" dentro la colonna scelta che cadrà sopra un altro gettone posato oppure si poserà semplicemente sul fondo. Il giocatore vince quando è riuscito ad allineare 4 propri gettoni senza interruzione. Per semplicità abbiamo scelto di considerare validi soltanto gli allineamenti verticali e orizzontali tralasciando gli allineamenti diagonali. Le system call da utilizzare obbligatoriamente sono:

- Gestione dei processi figli
- Memoria condivisa
- Segnali e semafori tra processi

Il gioco deve essere implementato con due eseguibili:

- F4Server che si occupa di inizializzare il gioco e di arbitrare la partita.
- F4Client che si occupa di gestire la mossa del giocatore e visualizzare il tabellone di gioco.

2 Implementazione System Call

2.1 Gestione dei processi figli

Abbiamo utilizzato la funzione `getpid()` per recuperare i pid dei processi che sono coinvolti. I pid vengono salvati in una struttura chiamata `shared_pid`.

Per la terminazione del giocatore che abbandona durante la partita abbiamo utilizzato la funzione `exit()`.

2.2 Memoria Condivisa

Sono state definite tre shared memory. La prima (`shared_board`) per la gestione di una matrice di caratteri che funge da tabellone di gioco, la seconda struttura condivisa (`shared_pid`) si occupa della condivisione del pid dei vari giovatori ed infine, la terza (`winning`) si occupa della gestione della vittoria e della sconfitta dei due giocatori.

shared_board - La gestione del tabellone di gioco

La struttura è definita come segue:

```
struct shared_board {
  char board[100][100];
  int rows;
  int cols;
};
```

La struttura contiene la matrice che funge da tabellone e il numero di righe e colonne di tale matrice.

Successivamente, nel `main` del file `F4Server.c`, questa struttura viene collegata al puntatore `ptr_gb`:

```
/* Puntatore dichiarato globalmente */
int shBoardID
struct shared_board *ptr_gb;

int main () {
  key_t boardKey = 5090;

  size_t boardSize = sizeof(struct shared_board);
  shBoardID = shmget(boardKey, boardSize, IPC_CREAT | S_IRUSR |
     S_IWUSR);
  if (shBoardID == -1) {
    errExit("shmget board failed");
  }

  ptr_gb = shmat(shBoardID, NULL, 0);
}
```

shared_pid - Gestione delle info dei giocatori

La struttura è definita come segue:

```
struct shared_pid {
  bool first;
  char player1Name[100];
  pid_t player1;
  char player1Token;
  char player2Name[100];
  pid_t player2;
  char player2Token;
```

```
  pid_t serverPid;
};
```

In questa struttura vengono salvate tutte le informazioni necessarie al server e ai client per interagire tra loro e gestire correttamente la partita. La voce booleana *first* si occupa di determinare se il client connesso è il primo, che in quel caso attenderà su un semaforo o se è il secondo, che in quel caso sbloccherà il semaforo su cui attende il server. Le altre variabili sono dei semplici contenitori di nome, gettone e pid dei giocatori e del pid del server.

Nel main la memoria condivisa viene creata e inizializzata:

```
struct shared_pid *ptr_playersPid;
int shPidID;

int main() {
  key_t pidKey = 6050;

  size_t winSize = sizeof(struct shared_pid);
  shPidID = shmget(pidKey, pidSize, IPC_CREAT | S_IRUSR |
    S_IWUSR);
  if (shPidID == -1) {
    errExit("shmget players failed");
      }

  ptr_playersPid = shmat(shPidID, NULL, 0);

  /* inizializzazione campi */
  ptr_playersPid->player1 = -1;
  ptr_playersPid->player2 = -2;
  ptr_playersPid->first = true;
  ptr_playersPid->serverPid = getpid();
}
```

winning - Gestione del fine partita

La struttura `winning` è quella che si occupa della gestione della vittoria e della sconfitta dei giocatori. Viene definita nel seguente modo:

```
struct winning {
  int playerLeft;
  bool player1Win;
  bool player2Win;
  bool full;
  bool end;
};
```

Il campo *playerLeft* viene usato dal signal handler che si occupa di intercettare la pressione ripetuta dei tasti `CTRL-C`, alla chiusura del processo viene inserito il pid del giocatore che ha abbandonato la partita. I campi *player1Win* e *player2Win* sono usati durante il controllo dello stato della partita, tra un turno e l'altro quando il controllo torna al server. Se quest'ultimo riscontra una vittoria imposterà i valori a `true` e `false` di conseguenza,

notificando i client che la partita è terminata. Questo viene fatto attraverso il campo *end* che viene posto a `true`. L'ultimo campo, *full*, viene usato per determinare se non ci sono più mosse disponibili, ovvero quando la partita finisce in parità.

Questa struttura viene collegata nel main come segue:

```
struct winning;
int shWinID;

int main() {
  key_t winKey = 4070;

  size_t winSize = sizeof(struct winning);
  shWinID = shmget(winKey, winSize, IPC_CREAT | S_IRUSR |
    S_IWUSR);
      if (shWinID == -1) {
            errExit("shmget winning failed");
      }

  ptr_winCheck = shmat(shWinID, NULL, 0);

  /* inizializzazione campi */
  ptr_winCheck->player1Win = false;
  ptr_winCheck->player2Win = false;
  ptr_winCheck->full = false;
  ptr_winCheck->end = true;
}
```

3 Segnali

Abbiamo utilizzato tre segnali: `SIGINT, SIGUSR1 e SIGUSR2`.

3.1 SIGINT

Questo segnale serve per chiudere un processo con la combinazione di tasti `CTRL-C`. Per il regolamento del gioco, il giocatore che intende abbandonare deve premerlo due volte. Abbiamo implementato l'handler per gestire questa regola nel seguente modo.

Lato SERVER

```
int count_sig = 0;

void sigHandler(int sig) {
    if (signal(SIGINT, sigHandler) == SIG_ERR) {
        errExit("signal handler failed");
    }
    if (count_sig == 0) {
```

```
8           printf("\n<Server> Attenzione pressione CTRL+C rilevata
    . Un'ulteriore pressione comporta la chiusura del gioco!\n"
    );
9           count_sig++;
10      }
11      else if (count_sig == 1) {
12          printf("\n<F4Server> Gioco terminato dal Server.\n");
13
14          /* Chiusura di shared memory e semafori */
15          closeGameProcedure();
16
17          /* Invio del segnale di chiusura ai processi F4Client
    */
18          if (kill(ptr_playersPid->player1,SIGKILL) == -1 || kill
    (ptr_playersPid->player2,SIGKILL) == -1 ) {
19              errExit("kill failed");
20          } else {
21              printf("<F4Server> Tutti i giocatori sono usciti
    dalla partita!\n");
22              printf("<F4Server> Partita terminata!");
23          }
24          exit(0);
25      }
26  }
27
28  /* Intercettazione del segnale nel main */
29  int main() {
30    /* ... */
31    if (signal(SIGINT, sigHandler) == SIG_ERR) {
32              errExit("change signal SIGINT handler failed");
33        }
34        /*...*/
35  }
```

La variabile globale `count_sig` che conta quante volte viene intercettato il segnale `SIGINT`. Quando la variabile è uguale a 0 viene stampato a schermo che se il giocatore preme un'altra volta `CTRL-C` il server chiude il gioco. Se lo preme un'altra volta il client si chiude con `exit(0)`. Prima dell'effettiva chiusura viene inviato un segnale di terminazione ai due processi client.

Lato CLIENT

```
1  int count_sig = 0;
2
3  /* Handler del segnale di interruzione Ctrl+C */
4  void sigHandler(int sig) {
5      if (signal(SIGINT, sigHandler) == SIG_ERR) {
6          errExit("signal handler failed");
7      }
8
9      if (count_sig == 0) {
```

```
10          printf("\n<F4Server> Attenzione pressione CTRL+C
    rilevata. Un'ulteriore pressione comporta la chiusura del
    gioco!\n");
11          count_sig++;
12      } else if (count_sig == 1) {
13          printf("\n<F4Server> Gioco terminato dal Server.\n");
14
15          /* Informo il server che il processo client corrente ha
     abandonato */
16          ptr_winCheck->playerLeft = getpid();
17          if (kill(ptr_playersPid->serverPid, SIGUSR1) == -1) {
18              errExit("kill SIGUSR1 failed");
19          }
20
21          exit(0);
22      }
23  }
24
25  int main() {
26    /* Intercettazione del segnale nel main */
27    if (signal(SIGINT, sigHandler) == SIG_ERR) {
28                errExit("signal handler failed");
29        }
30  }
```

Funziona con la stessa modalità del server ma cambia la gestione dell'uscita della partita. Quando viene premuto per la seconda volta `CTRL-C` viene aggiornata la struttura dati che si occupa della vittoria e viene scritto sulla variabile `playerLeft` il pid del client che abbandona. Successivamente viene mandato al server il segnale `SIGUSR1` che notifica l'abbandono e il client si chiude con la funzione `exit(0)`.

3.2 SIGUSR1

Questo segnale si occupa della gestione di norifica dell'abbandono di uno dei due client. Viene inviato al server quando un client abbandona la partita. L'handler del segnale è stato implementato come segue:

```
1   void sigHandlerPlayerLeft(int sig) {
2
3       if (ptr_playersPid->player1 == ptr_winCheck->playerLeft) {
4           printf("\n<F4Client> Hai vinto per abbandono di %s.\n",
     ptr_playersPid->player1Name);
5       } else {
6           printf("\n<F4Client> Hai vinto per abbandono di %s.\n",
     ptr_playersPid->player2Name);
7       }
8
9       exit(0);
10  }
11
12  int main() {
```

```
  /* Handler abbandono partita di un giocatore */
     if (signal(SIGUSR2, sigHandlerPlayerLeft) == SIG_ERR) {
          errExit("signal handler failed");
   }
}
```

Quando avviene la notifica dell'abbandono viene mandato al giocatore ancora in partita il segnale `SIGUSR2` che notifica la vittoria e si occupa di tutta la questione relativa alla chiusura delle shared memory e del set di semafori invocando la funzione `closeGameProcedure()`.

3.3 SIGUSR2

Questo segnale si occupa di notificare la vincita al player che è rimasto ancora in partita. Viene mandato dal server e lo riceve il client vincitore. L'handler è stato implementato come segue:

```
void sigHandlerPlayerLeft(int sig) {

    if (ptr_playersPid->player1 == ptr_winCheck->playerLeft) {
        printf("\n<F4Client> Hai vinto per abbandono di %s.\n",
    ptr_playersPid->player1Name);
    } else {
        printf("\n<F4Client> Hai vinto per abbandono di %s.\n",
    ptr_playersPid->player2Name);
    }

    exit(0);
}

int main() {
  /* Intercettazione del segnale SIGUSR2 */
      if (signal(SIGUSR2, sigHandlerPlayerLeft) == SIG_ERR) {
           errExit("signal handler failed");
    }
}
```

Avviene un controllo del pid e in base a chi ha vinto stampa a schermo la frase di vittoria e il player che ha abbandonato. Anche in questo caso la chiusura avviene attraverso la funzione `exit(0)`.

4 Semafori

I semafori sono stati implementati seguendo questo schema.

```
bool S = 0, C1 = 0, C2 = 0; // 3 semafori inizializzati a 0
```

Server

```
int Server() {
  /* Creazione ed inizializzazione handler dei segnali
    * e delle varie strutture nella memoria condivisa */

  P(S);      // attesa dei client
  while (!win) {

    V(C1);    // concessione del turno al primo client
    P(S);   // attesa

    /* Verifica dello stato di gioco, se nessuno
      * ha vinto passo il turno al giocatore 2 */

    V(C2);    // turno al secondo client
    P(S);   // attesa

    /* Verifica dello stato di gioco, se nessuno
     * ha vinto passo il turno al giocatore 1 */

  }
}
```

Client

```
int Client() {
  /* Ottenimento dell'accesso alle strutture
   * della memoria condivisa e inizializzazione
   * degli handler dei segnali. */

   if (ptr_playersPid->first == true) {
    P(C1);    // attesa del secondo client
   } else {
    P(C2);    // attesa del turno
   }

  do {
    if (player1 && !endGame) {
      /* Turno del client 1 */
      V(S);   // libero il server
      P(C1);    // attendo il prossimo turno
    }

    if (player2 && !endGame) {
```

```
20        /* Turno del client 2 */
21        V(S);   // libero il server
22        P(C2);    // attendo il prossimo turno
23      }
24
25      /* Verifica se il server ha comunicato un vincitore */
26    } while (!endGame);
27  }
```

5 Struttura del codice

5.1 Codice del Client

Sono state definite alcune funzioni per ottimizzare la leggibilità del codice e per la gestione dei segnali. Le funzioni sono:

- `bool checkValidity`: si occupa di gestire l'inserimento del gettone del player. Ritorna true se il gettone viene inserito, false altrimenti. I parametri sono: `struct shared_board *ptr_sh`, ovvero la struttura contenente il tabellone di gioco; `int col`, la variabile su cui è salvata la colonna scelta dall'utente ed infine, `char token`, il gettone del giocatore attivo.
- `void printBoard`: procedura che si occupa di stampare a video la matrice che funge da campo di gioco. Il suo unico parametro è `struct shared_board *ptr_sh`.
- `sigHandler`: handler di gestione della pressione del `CTRL-C`, come spiegato nel capitolo sui segnali.
- `sigHandlerPlayerLeft`: handler di gestione dell'abbandono della partita, spiegato nel capitolo sui segnali.
- `main`: nel main sono presenti la gestione delle shared memory dove viene effettuato l'attachment ai puntatori; la gestione dei turni, effettuata tramite i semafori e la richiesta di input.

6 Codice del Server

Le funzioni e le procedure definite nel server sono:

- `closeGameProcedure`: funzione che si occupa di gestire in modo coerente la chiusura di tutte le shared memory e dei semafori utilizzati, effettuando la detach e la remove delle shared memory e la rimozione del set di semafori creati.

- `checkGameStatus`: è la funzione di verifica vittoria o sconfitta. Controlla, dopo ogni turno, se si è creata la condizione di vittoria del giocatore che ha appena inserito la giocata.
- `sigHandler`: funzione di gestione della pressione di `CTRL-C`. Viene spiegato il suo funzionamento nel capitolo sui segnali.
- `sigPlayerLeft`: gestisce la ricevuta di notifica di abbandono di uno dei due giocatori. Spiegata nel dettaglio sotto la voce segnali.
- `main`: qui è presente la gestione delle shared memory. Vengono create le memorie condivise e agganciate ai puntatori. È presente anche la creazione e l'inizializzazione del set di semafori che servono per la gestione dei turni.