Università degli Studi di Torino

Project report

Operating systems - Sistemi operativi

Nome: Edvinas Nomeika
Matricola: 895757
Corso: B, Turno: T3

Contents

1 The processes

The project can create a number of processes in it's lifetime, but 3 of them are unique. Each of these processes are required to be compiled with "**common.o**" and "**semaphoreSO.o**"

1.1 Master

The master process is the first process to come alive. It's job is to create shared memory segments, create semaphores, syncronize players/pawns, place flags, scans the board for flags, print the Chessboard and individual scores and lastly, to terminate all player processes (which in turn, terminate their pawns) and deallocate the various IPC segments created. When creating a player process, it is assigned a turn (from 1 to N), this value is required to place pawns in an orderly fashion an as a unique key to create a specific shared memory segment with the player.

1.2 Player

The player process is created by the master, it has a simple job of creating pawns (in order and in turns), coordinating them and eventually, terminate them. The strategy the player uses is assigning a flag to a closest pawn and ONLY one pawn per player (the pawns are meant to work together not against eachother). The Player ones does it once per round, assuming that all reachable flags have been assigned to a pawn and then it goes to sleep. The Player creates it's own personal shared memory segment called "**MyTarget**", it is shared between one player and all of it's pawns. It's defined as:

```
struct Destination{
  int Distance; /* Distance to flag */
  int DestinationCol;
  int DestinationRow;
  int Fuel; /* Remaining moves */
  int SourceRow;
  int SourceCol;
};

```

Similarly to the master, the player process assigns a turn (from 0 to N) to each Pawn which is used as an "**Index**" value for the "**MyTarget**" array.

1.3 Pawn

The pawn is created by the player process, it's main objective is to wait for a destination (which is given by the player). A pawn has a limited amount of moves, it should **NEVER** pursue a flag that's too far and it cannot access a cell that's already occupied by another pawn. The movement of a pawn is simple, it moves vertically until it's on the same row as it's destination, then moves horizontally, avoiding obstacles if possible. When a pawn captures a flag (originally it sent an signal to the Master, but due to many signals being sent at once, some were lost causing syncronization errors) , it updates its parent's score and then checks if it's close to another flag, if yes, it also checks if it's THE closest pawn, if that's also true, the original pursuer pawn is told to stop moving and the new pawn begins pursuing the flag. Otherwise, it sleeps until a new round starts.

2 IPCs used

In this project, only the shared memory and semaphores were used, message passing was not.

2.1 Shared memory

2.1.1 Created by master

The master process begin its life by creating two main shared memory segments, known as "**Chessboard**" and "**Scoretable**" which are used among all processes to share locations of flags/pawns and to know the score of each player.

2.1.2 Created by a player

Each player creates its own shared memory segment, which is used to coordinate with the pawns, its structure was shown before.

2.2 Semaphores

The Master also has the job of creating two main semaphore segments "**ChessboardSemaphoresID**" and "**SemID**", the latter is required to syncronize the Players and the Pawns to avoid overwriting necessary data and to keep them orderly. The "**ChessboardSemaphoresID**" have dedicated semaphores for each Cell in our "**Chessboard**", which is required to let other pawns know if the Cell they're trying to access is occupied or not.

3 The chessboard

The chessboard is not defined just by the semaphores, which are required to ONLY know if a specific cell is occupied or not, what we also need to know is what's in the cell and it's attributes.

3.1 Chessboard structure

Therefore, it's no simple "Char" matrix, it is infact a structure defined as:

```c
struct Pawn{
  int PIDParent;
  int PIDPawn;
};

union Attribute{
  int Points; /* For flag , is 0 or NULL when empty */
  struct Pawn pawn; /* For Pawn */
};

struct Cell{
  char Symbol; /* Can be either a F for Flag , or P for Pawn */
  union Attribute Att;
};

```

This structure has the advantage of carrying multiple data within itself without requiring to create abundant segments for information exchanging. It is closely used with the "**ChessboardSemaphoreID**", which as discussed previously, is required to let each pawn know which cell is occupied.

3.2 Printing the chessboard

The chessboard is printed by the master processes and it's done every 0.1 seconds. The empty spaces in the chessboard are filled with dots ".", while the flags are recognized by the symbol "**F**" and are always colored red. The Pawns are more tricky, The first 10 players use the first capital letter of the alphabet as their symbol, while the next 10 use the 2nd letter and so on. Despite the pawns using the same letter on the chessboard, they are colored differently, otherwise it would nearly be impossible to recognize which pawn belongs to which player.

Above the chessboard, the master also prints the "**Leaderboard**", which indicate the Player color, it's score and the amount of moves used in it's life time.

4 Signals

Originally the project was heavily dependent on the master receiving a signal every time a pawn captures a flag, this was proven to be too problematic. Instead, the signals are mostly used to detect either an interrupt sent by the user, which in turn terminates all other processes and deallocate all of the IPCs segments or an error in general (a segmentation fault).

4.1 Pause

When a pawn doesn't have a destination, it is put to sleep with the function "**pause()**". This is useful to not waste the processor usage, it greatly increases the execution speed. A process can be awaken by sending any signal to it.

4.2 Alarm

The game ends once the players were unsuccessful at capturing a flag in a specific amount of time in a round. In order to achieve it, the project uses the function "**alarm()**", which sends a signal to the master after a specific amount of time, this alarm resets every round. The time can be modified in the "**Settings.conf**" file.

5 Common.o and semaphoreSO.o

These files are very self explanatory, they're linked with all of the processors.

5.1 Common.o

The **common.h** and **common.c** contain included libraries, macros, defines and functions which are used by everyone or at least more than 2 processors. I also include some "Logging" code for debugging.

5.2 SemaphoreSO.o

This object contains everything related to the semaphores, including creation, initialization, locking/unlocking and wait functions.

6 The settings file

The settings file is called "**Settings.Conf**" and it's where you go when you want to modify something regarding to the game. Though, the file is infact "fragile", deleting a line might be catastrophical, being cautious is highly suggested. The function to read the values is located in "**common.c**", it is capable of doing very basic parsing and error signaling, for example, if I don't indicate a number for a specific variable, it'll kill the program and end in a termination.