Project 2 : Concurrency
15-412 Operating mazetems
Due: Wed Feb 19 23:59:59 EST 2003

Making a $100 Video Game Console out of $1000 Hardware

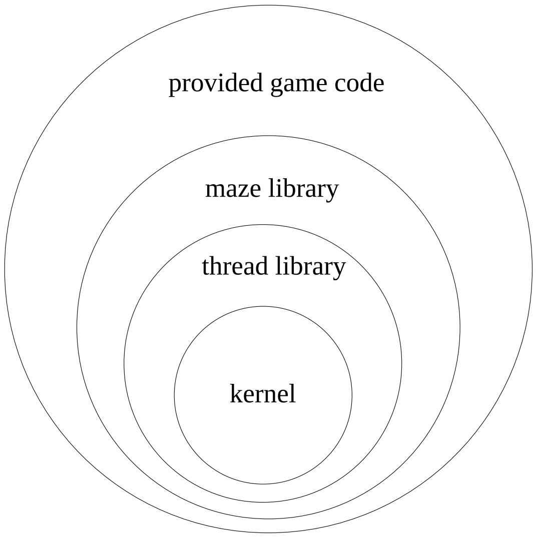

Figure 1: 412maze structure

UPDATED

- We have removed the pointer arguments from `maze_init()` and `maze_cleanup()`. They both now take no parameters.
- We have removed `maze_abort_request()` from the API. You no longer have to write this function.
- Clarification is required involving the connection between threads and monsters. Threads will partition the monster space, meaning that no monster will be manipulated by more than one thread (over the course of its entire game life). The thread that calls `maze_new_monster()` is the only thread that will ever use the `mon_id` returned. Because of this, you can assume that when a monster is involved in a `maze_request_move()`, no other maze_* functions will be called with its `mon_id`.
- You may also assume that a monster will appear in a single `move_combo_t` array at most once.
- Once a thread has called `maze_request_move()` on a set of monsters and received `SUCCESS`, evey monster in that set will call `maze_move_monster()` in the specified direction before the thread calls `maze_request_move()` again.
- We wrote the function `maze_authors()` which you must call from your `maze_init()`. This function will allow us to display a splash screen describing who wrote the game!

1 Introduction to 412maze

Now that a thread library has been created, it is a good idea to put it to work. We have created a game, called 412maze, which will test all aspects of your thread library. 412maze requires the implementation of the thread library and the maze_* functions listed below in order to run properly.

The game is composed of monsters, which are controlled by threads. In most cases, one thread will handle only one monster, however on occasion (as is the case with abstract entities like walls which are internally represented by groups of monsters) a single thread **will** control the movement of multiple monsters. Regardless of how many monsters are controlled by a given thread, a thread will never use a `mon_id` of a monster that it did not create.

Each monster will behave according to its own predefined set of rules, so you should assume it to be unpredictable (meaning proper error detection and return is necessary). 412maze exposes a wealth of concurrency issues because multiple agents will be simultaneously vying for critical resources (i.e., space on the board) by making requests to some authority. Deadlocking is also a problem that can occur if these requests are not handled properly. This project will require management of the board such that appropriate invariants are upheld (like no two pieces occupying the same space at the same time), and under all circumstances deadlock must be avoided. We provide code for handling game logic and the graphics.

2 How The Game Works

Before the game actually begins, a series of maze_* functions are called to set up the board. After they have completed, threads handling game objects (walls, monsters, the user) are spawned and the game begins.

412maze is from the class of games where the player runs around a level, collects objects, and avoids monsters (and other evil objects). The hero is manipulated using keyboard keystrokes with the goal of collecting all the little dots (the gold) on a level. The keys for movement are the following: a or h (left), d or l (right), s or j (down), and w or k(up). If a monster runs into the player, the game machine will declare the game over and call the board library routines to close down the application. If the player collects all of the gold at a given level, all existing threads end (through the use of the board library routines), the next level's board is set up, and the game continues.

Each level has a different configuration, each of which introduces new problems for you to solve. We have included some levels that are conducive to deadlocks, which will have to be handled.

3 The Game's Data Structure

As was stated earlier, you are going to be writing the functions that are going to remember the state of the board. The board is a `BOARD_WIDTH` x `BOARD_HEIGHT` structure (these constants are defined in `common.h`) of cells. Cell (0,0) is located at the top left corner of the screen with `BOARD_WIDTH` specifying the number of cells to the right and `BOARD_HEIGHT` specifying the number of cells down. In order to draw the board to the screen, you are going to have to call the function `game_draw_board_tile()` which takes as parameters a cell's coordinates, and a tile identifier (which specifies what should be drawn at that cell). The different values of a tile are supplied in `common.h` (examples of these values are `PLAYER`, `GOLD`, and `WALL`). In order for a change in the board state to take effect on the screen, each modified board tile must be redrawn.

4 Maze Game Functions

For the game section of the project you will provide this API, which will be called by 412maze. This API is as follows:

4.1 Maze Game Functions

- `int maze_init(void)` - This function will do any initialization required. 412maze will call this function when there is only one thread, and `thread_init()` has already been called. You should initialize each cell to contain an `EMPTY` tile and draw the whole board to the screen. It returns `SUCCESS` on success, and `FAILURE` on error.

- `int maze_cleanup(void)` - You can use this function to clean up any data you allocated. Returns `SUCCESS` or `FAILURE` appropriately.

- `tile_t maze_get_board_contents(int x, int y)` - Returns the tile at the cell with coord (`x`, `y`). If the coordinates (x, y) are outside the range of the board, then the value `FAILURE` is returned.

- `int maze_set_board_contents(int x, int y, tile_t tile)` - Sets the contents of the cell at (x,y) to the value `tile`. 412maze will never use this to modify a cell that a monster is occupying. This function will only be used when initially setting up the board (`WALL`, `GOLD`, and `CHERRY` placement) Returns `FAILURE` if the coordinates are invalid, `SUCCESS` otherwise.

- `monid_t maze_new_monster(tile_t tile, int x, int y)` - Creates a new monster [1] at the cell (x,y) that will be represented by the image `tile` (defined in `common.h`). `maze_new_monster()` returns a monster identifier which will be used for the remainder of the monster's game-life. The other maze_* functions request only a monster's id, so you're going to have to store any data that you will need to know about the monster on your own. `maze_new_monster()` must also draw to the screen using `game_draw_board_tile()` before returning the `monit_t`. If the coordinates are invalid or the monster is unable to enter the cell (determined by using `game_can_monster_enter_cell()` in the same way as `maze_move_monster()`), return `FAILURE`.

- `int maze_destroy_monster(monid_t mon)` - This function destroys the monster's board representation, erases its tile from the screen (redrawing the appropriate tile in the cell), and returns `SUCCESS`. If the monster is currently requesting or has requested another cell, he should still be destroyed instantly and allow another monster to request that cell. If the monster `mon` does not exist, return `FAILURE`.

[1] Note that this function only creates a board representation of the monster. Its actual behavior is controlled by a handler that lives in our code. Also note that certain game objects (eg: walls) are modeled as a collection of distinct monsters (albeit running in a single thread.)

- `tile_t maze_move_monster(monid_t mon, direction_t dir)` - This will attempt to move the monster identified by the parameter `mon` one cell in the direction `dir`. The directions are defined in `common.h`. This function must observe the following rules:
  - Every time a monster is moved, the tile that occupied the cell before the monster entered (`CHERRY`, `FOOD`, or `EMPTY`) must be restored when the monster leaves.
  - There are many instances where you will be unable to move the specified monster in the specified direction. If a monster is trying to move out of the board's range, return `FAILURE`. If a monster is trying to move onto a cell that they are not allowed to move into, return `FAILURE`. To help you determine whether or not the move is allowable (based on cell occupancy), we've provided the function `game_can_monster_enter_cell(tile_t mon, tile_t dst)`. This function takes two tiles and returns `true` if the monster represented by `mon` can move into a cell containing `dst`, or `false` otherwise. If the move is not allowed, `maze_move_monster()` should return `FAILURE`
  - If this function is successful it returns the tile value of the previous contents of the cell.

- `int maze_consume(monid_t mon)` - If a monster calls this function, they value of the tile that they are standing over becomes `EMPTY`. For example, this function will be called to make the player pick up the gold that he is standing on. Return `FAILURE` if `mon` does not represent a valid monster.

- `int maze_request_move(move_combo_t *ml, int n)` - This is a blocking function. `ml` is an *array* of n (`monid_t`, `direction_t`) pairs (which are defined in `maze.h`) representing a group of monsters [2] intending to move. `maze_request_move()` will return only when each monster is guaranteed that the next time it calls `maze_move_monster()` it will receive a `SUCCESS` (assuming it actually moves in the direction requested). The same rules apply for movement as for `maze_move_monster()`.

  If multiple monsters (of separate groups) attempt to "request" the same cell (in different requests), one group must be allowed to advance while the others must block. As soon as movement for any one of these blocked groups becomes possible, they should be unblocked and the function should return SUCCESS. A `mon_id` will never appear more than once in a particular `move_combo_t` array.

  Note that this function does not actually move anything – that is the job of `maze_move_monster()`.

  There are few cases where you are allowed to return with an error. If the cell that a monster requests contains a `WALL` tile or would take the monster past the board's boundary, return `FAILURE`. If this function is interrupted by `maze_abort()`, return `FAILURE`. If there is a deadlock present, you should return `ERROR_LOCK` (there is more on deadlock in section 5). Otherwise return `SUCCESS`.

  When this function returns `SUCCESS`, you can assume that the calling thread will not call `maze_request_move()` again until after it has called `maze_move_monster()` on each monster in the `move_combo_t` array (with the appropriate direction).

[2] Possibly members of an entity like a wall.

4.2 Maze Setup Functions

- `void maze_set_game_status(int s)` - This sets the value of `game_status`, an integer with no meaning to you, but which must be kept thread safe.
- `int maze_get_game_status(void)` - This returns that value of `game_status`.
- `void maze_abort(void)` - This will wake up all threads currently blocked in maze functions, including `maze_wait()` and `maze_request_block()`.
- `void maze_wait(void)` - This is a blocking function. It will block the caller thread until `maze_abort()` is called.

5 Deadlock Detection

In order to properly implement maze_request_move(), you will need to correctly identify a deadlock when one occurs. As stated above, a deadlock should not exist for longer than 3 second. How you determine this is up to you.

Our grading process will depend on the approach you take and how clearly you document it. You must document how your design works and why you chose it over other methods. Document it in the `DESIGN` file explained in the first half of the handout.

6 Functions We Provide

- `int game_draw_board_tile(int x, int y, int tile)` - Draws a board tile on the screen at logical coordinates (x, y). The value of `tile` specifies which image to draw.
- `int game_can_monster_enter_cell(tile_t mon,`
  `tile_t dst)` - Should a monster represented by tile `mon` be allowed to move into a cell containing `tile`? Returns `true` if the answer to this question is yes, `false` otherwise.
- `void maze_authors(char ** author_usernames, char ** author_realnames,`
  `int n)` - `author_usernames` and `author_realnames` should contain your usernames and andrewIDs. `n` specifies how many members are in your group. This must be called during `maze_init()`.

7 Suggested Plan of Attack

1. Get the thread library finished and tested.
2. Write `maze_wait()` and `maze_abort()`.
3. Determine how to represent the board and monsters.
4. Implement all of the movement functions except `maze_request_move()`.
5. Implement `maze_request_move()` so that it works without handling deadlock.

6. Create a program in UNIX that does deadlock detection on mock data structures that are similar to the ones in the game.
7. Integrate deadlock detection into 412maze.
8. Celebrate!!!
9. Rest up for the kernel project...