Reversi

- you can download and build the source code using the command line

-
  ```
  $ git clone https://github.com/gaiusm/reversi
  $ mkdir build-reversi
  $ cd build-reversi
  $ ../reversi/configure
  $ make
  ```

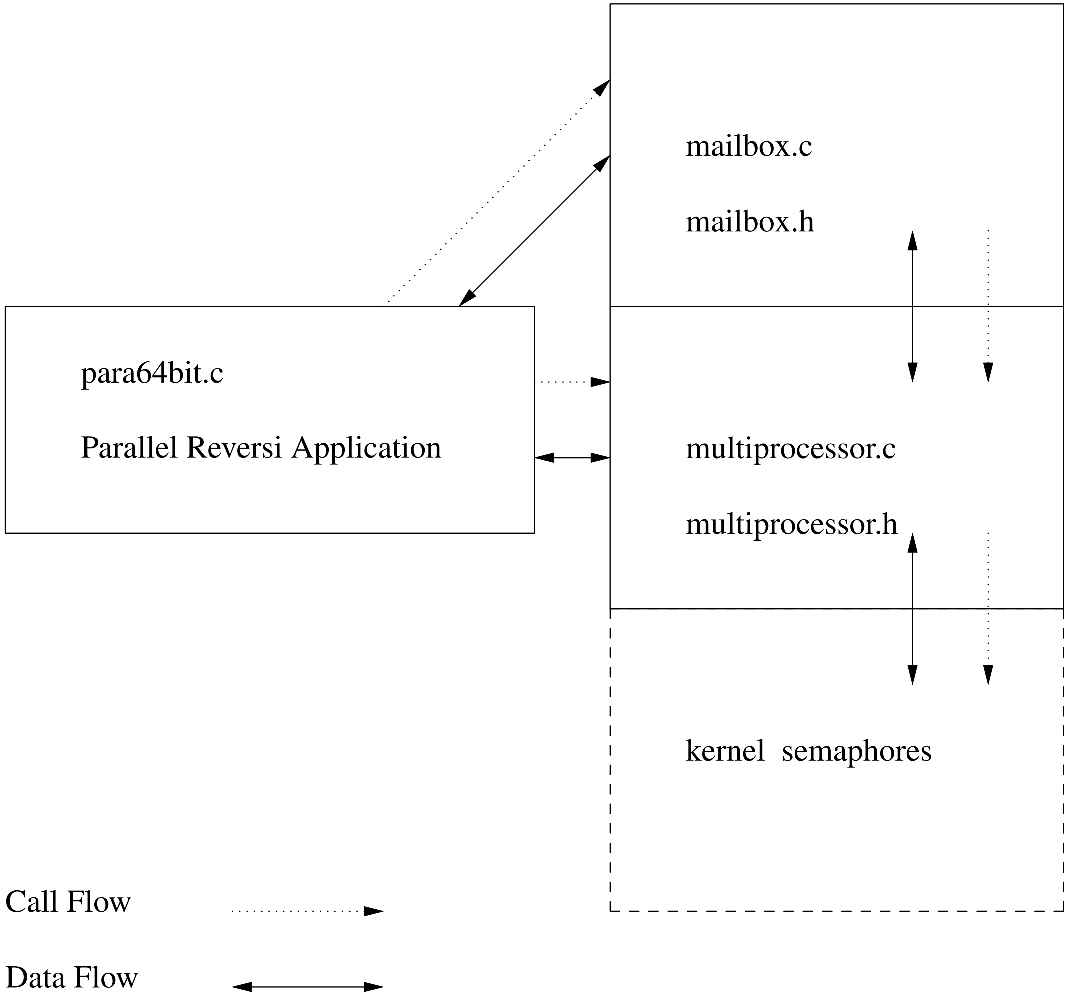
- the program source is incomplete and will not work out of the git repro
  - you need to complete the source

- in particular you need to implement:
  - `mailbox.c:mailbox_send`
  - `mailbox.c:mailbox_rec`
  - `paro64bit.c:parallelSearch`

Parallel Reversi overview

- the overall structure of the source code and modules is shown:

Parallel Reversi overview



para64bit.c

Parallel Reversi Application

mailbox.c

mailbox.h

multiprocessor.c

multiprocessor.h

kernel semaphores

Call Flow

Data Flow

Parallel Reversi overview

- there are 5 source files of which you need to be aware:
  - `multiprocessor.h` defines the interface for the library `multiprocessor.c`
  - `multiprocessor.c` provides a simple interface to semaphores and shared memory
  - `mailbox.h` defines the interface for the library `mailbox.c`
  - `mailbox.c` provides a simple bounded buffer implementation to allow children to return their game search results to the parent process.
  - `paro64bit.c` the reversi game implemented using parallel primitives. It will spawn a number of children to search the game tree in parallel.

Parallel Reversi overview

- the layered approach attempts to divide and conquer the problem of implementing a parallel reversi

- the reversi game implementation uses an alphabeta search strategy to explore a move
  - it evaluates all legal moves which can be played by the opponent and then all counter moves which you might make etc (down to a certain depth)
  - this is called a game tree and exploring game trees can be computationally expensive

- background reading about game trees ⟨`../games/14.html`⟩.

mailbox pseudo code

- mailbox implements a bounded buffer
    - `paro64bit.c` only creates a single mailbox object
    - the parent reads from the buffer (mailbox) and the children place the result of their game search results

mailbox pseudo code

-
  ```
  /*
   *  send - send (result, move_no, positions_explored) to the mailbox mbox.
   */

  void mailbox_send (mailbox *mbox, int result, int move_no, int positions_explored)
  ```

- `mailbox_send` is a producer to the shared buffer and it needs to store the: `result`, `move_no` and `positions_explored` into the shared buffer.

mailbox pseudo code

-
  ```
  /*
   *  rec - receive (result, move_no, positions_explored) from the
   *        mailbox mbox.
   */

  void mailbox_rec (mailbox *mbox,
            int *result, int *move_no, int *positions_explored)
  ```

  mailbox_rec is a consumer with the shared buffer and it needs to retrieve the: result, move_no and positions_explored from the shared buffer.

Exercise

- read and understand the interface files
  - `multiprocessor.h`
  - `mailbox.h`

Parallel Reversi High Level Algorithm

- it is heavily based on the sequential algorithm

C and Pseudo code

```
int parallelSearch (int *totalExplored, int *move,
                    int best, int *l, int noOfMoves,
                    BITSET64 c, BITSET64 u, int noPlies,
                    int o, int minscore, int maxscore)
{
   /* here we create a source and sink process, the source continually forks children
      one for every move, providing a processor is available.  The sink collects the
      results and ultimately returns the best move.  */
  int pid = fork ();
  if (pid == 0)
    {
      /* child is the source which spawns each move on a separate core.  */
      for i in noOfMoves do
          wait for a processor to become available;
          if (fork () == 0)
             /* child must search move i.  */
             use alphaBeta to search move i
             pass move_score, i, positions_explored back via mailbox
             exit (0);
          end
      end
      exit (0);
    }
```

C code

```
  else
    {
      /* parent is the sink, which waits for any move to be returned and
         remembers the best move score.  */
      int i, move_score, move_index, positions_explored;

      for (i=0; i < noOfMoves; i++)
        {
          printf ("parent waiting for a result\n");
          mailbox_rec (barrier, &move_score, &move_index, &positions_explored);
          printf (" ... parent has received a result: move %d has a score of %d after e
                  move_index, move_score, positions_explored);
          *totalExplored += positions_explored;  /* add count to the running total.  */
          if (move_score > best)
            {
              best = move_score;
              *move = l[move_index];
            }
        }
    }
  return best;
}
```