CS422: Object-Oriented Design and Programming

Program Assignment 3
Asynchronous Timer Queues
Due Tuesday, September 30th, 1997

Your first programming assignment focused on asynchronous timer queues using UNIX signals. Although asynchrony is powerful, signals are error-prone and non-portable. Moreover, signals don't interact well with other sources of I/O, such as sockets or standard input.

This assignment will provide you with an opportunity to integrate your timer queue with a synchronous I/O event demultiplexing model known as the *Reactor* pattern.[1] The behavior of the `Reactor` is described below.

Reactor

The `Reactor` is an OO interface for synchronously demultiplexing events and dispatching event handlers to process the events. For this assignment, events will correspond to I/O operations and timers. The following interface will be used to define a `Reactor` Singleton:

```
typedef int HANDLE;

class Reactor
  // = TITLE
  //      Provides an OO interface for synchronous
  //      event demultiplexing and dispatching.
{
public:
  static Reactor *instance (void);
  // Reactor is a Singleton.

  // = I/O operations.

  int register_handler (HANDLE handle,
                        Event_Handler *eh);
  // Register <eh> to process events that occur on
  // <handle>.  When events occur, the <Reactor> will
  // dispatch the <Event_Handler::handle_input> hook
  // method.  Returns -1 on failure, 0 on success.

  int remove_handler (HANDLE handle,
                      Event_Handler **eh = 0);
  // Remove <handle> from the <Reactor>.  If <eh> is
  // non-0, pass back the pointer to the <Event_Handler>
  // so that the caller can decide how to release the
  // memory.  Returns -1 on failure, 0 on success.

  // = Timer operations.

  int schedule (Event_Handler *event_handler,
                const void *arg,
                const Time_Value &delay) = 0;
  // Schedule an <Event_Handler> that will expire after <delay> amount
  // of time.  If it expires then <arg> is passed in as the value to
  // the <event_handler>'s <handle_timeout> callback method.  This method
  // returns a <timer_id> that uniquely identifies the <event_handler>
  // in an internal list.  This <timer_id> can be used to cancel an
  // <event_handler> before it expires.  The cancellation ensures that
  // <timer_ids> are unique up to values of greater than 2 billion
  // timers.  As long as timers don't stay around longer than this
```

[1] See `www.cs.wustl.edu/~schmidt/PLoP-94.ps.gz` for more information on the Reactor pattern.

```
  // there should be no problems with accidentally deleting the wrong
  // timer.  Returns -1 on failure (which is guaranteed never to be a
  // valid <timer_id>).

  int cancel (int timer_id, const void **arg = 0) = 0;
  // Cancel the single <Event_Handler> that matches the <timer_id>
  // value (which was returned from the <schedule> method).  If arg is
  // non-NULL then it will be set to point to the ``magic cookie''
  // argument passed in when the <Event_Handler> was registered.  This
  // makes it possible to free up the memory and avoid memory leaks.
  // Returns 1 if cancellation succeeded and 0 if the <timer_id>
  // wasn't found.

  // = Event loop operation.

  virtual int handle_events (void);
  // Run the <Reactor>'s event loop, which demuxes and
  // dispatches <Event_Handlers> based on I/O and timer
  // expiration.  Returns the number of <Event_Handlers>
  // dispatched per invocation and -1 on error.

protected:
  // = Initialization and termination methods.
  Reactor (void);
  ~Reactor (void);

private:
  static Reactor *instance_;
  // Static instance for the Reactor.

  // You fill in here...
};
```

The secret to the `Reactor` is a system call known as `select(3c)`. The `select` call examines the I/O file descriptor sets whose addresses are passed in readfds, writefds, and exceptfds to see if any of their file descriptors are ready for reading, are ready for writing, or have an exceptional condition pending, respectively. In addition, `select` can also be used to specify a maximum interval to wait for the selection to complete, which can be used in conjunction with your Timer Queue mechanism to implement a *synchronous* timer mechanism.

The `Event_Handler` class must be extended to handle input events, as follows:

```
class Event_Handler
  // = TITLE
  //      Provides an interface to an <Event_Handler> that
  //      the <Reactor> uses as the target of its callbacks.
{
public:
  virtual int handle_timeout (const Time_Value &,
                              const void *arg);
  // Invoked by the <Reactor> when a previously registered timer
  // expires.

  virtual int handle_input (HANDLE handle);
  // Invoked by the <Reactor> when it is possible to perform an input
  // operation without blocking.
};
```

Synchronous Test Driver

The main goal of the assignment is getting the timer queue to work correctly when used in conjunction with other sources of I/O, *i.e.*, standard input from the terminal.

To test your `Reactor`, you will write a test program with a main event loop that is structured as follows:

```
const int STDIN = 0;

class STDIN_Handler : public Event_Handler
{
public:
  STDIN_Handler (void);
  // Constructor registers <this> with Reactor
  // Singleton to demux user input from STDIN.

  virtual int handle_input (HANDLE handle);
  // Processes user input from the terminal.

  void display_prompt (void);
  // Display the user prompt string.

  int done (void) const;
  // Terminate the main program event loop.

private:
  int done_;
  // Keeps track of whether the program is finished.
};

STDIN_Handler::STDIN_Handler (void)
  : done_ (0)
{
  if (Reactor::instance ()->register_handler
        (STDIN, this) == -1)

  // You fill in here...
}

int
STDIN_Handler::display_prompt (void)
{
  printf ("****"
     "1) schedule timer <usecs>\n"
  "2) cancel timer <timer_id>\n"
  "3) list timers\n"
  "4) shutdown program\n"
  "please enter your choice: ");
}

int
STDIN_Handler::handle_input (HANDLE handle)
{
  // You fill in here...

  this->display_prompt ();
}

int
STDIN_Handler::done (void) const
{
  return this->done_;
}

int
main (void)
{
  STDIN_Handler stdin_handler;

  stdin_handler.display_prompt ();

  // Keep looping until the user shuts us down.
  while (stdin_handler.done () == 0)
    Reactor::instance ()->handle_events ();

  return 0;
}
```

As shown above, this program repeatedly prompts the user for the following types of input:

```
****
1) schedule timer <usecs>
2) cancel timer <timer_id>
3) list timers
4) shutdown program
please enter your choice:
```

If the user selects operation 1 they must indicate the number of microseconds in the future when they want an Event_Handler to expire. You must then create a new Event_Handler subclass object (*e.g.,* a Timer_Handler) and schedule this handler with the Timer_Queue in your Reactor. When the timer elapses, your Reactor's Timer_Queue must expire all timers that have earlier deadlines than the current time of day (see gettimeofday(3C)).

When a user schedules a timeout, your program should print out the timer id of the newly registered Event_Handler, *i.e.*,

```
please enter your choice: 1 10000000
scheduling timer 264680
```

will cause an Event_Handler::handle_timeout to be invoked in 10 seconds by the Timer_Queue mechanism.[2]

If the user selects operation 2 they must indicate the timer id of the previously registered Event_Handler that must be cancelled, *i.e.*:

```
please enter your choice: 2 264680
```

will cancel the Event_Handler with timer id 264680. Users are responsible for giving valid timer ids.

If the user selects operation 3 you must iterate through your Reactor's Timer_Queue and print out the values of all the nodes in the linked list.

Finally, if the user selects operation 4 you'll set the done_ flag, which informs the main event loop to close down the program.

Getting Started

You can get the shells and Makefile for part one of the program from your account on cec. These files are stored in /project/adaptive/cs422/assignment-3/. Here's a script that shows you how to set everything up and get these files:

```
% cd ~/cs422
% mkdir assignment-3
% cd assignment-3
% cp -r /project/adaptive/cs422/assignment-3/* .
% make
```

The Makefile and various header files are written for you. You'll need to do is edit the *.C files to add the code to implement the Reactor and test driver.

[2]If the timeout occurs before the Event_Handler is canceled, handle_timeout method should print the current time and delete this to avoid memory leaks.