CS422: Object-Oriented Design and Programming

Program Assignment 1
Asynchronous Timer Queues
Due Saturday, September 6, 1997

A Timer Queue is an *Abstract Data Type* (ADT) that allows applications to manage a set of timers and their associated handlers. Common operations on a Timer Queue include *schedule*, *cancel*, and *expire*. Your first programming assignment focuses on building a Timer Queue that can run correctly when used in an asynchronous application (*i.e.*, one that can perform timer queue operations in the context of signal handlers).

Timer Queue Interface and Implementation

The interface for the `Timer_Queue` must be defined as follows:

```
class Timer_Queue
  // = TITLE
  //      Provides an interface for timer operations.
{
public:
  // = Initialization and termination methods.
  Timer_Queue (void);
  // Default constructor.

  ~Timer_Queue (void);
  // Destructor - make virtual for proper destruction of inherited
  // classes.

  int is_empty (void) const = 0;
  // True if queue is empty, else false.

  const Time_Value &earliest_time (void) const = 0;
  // Returns the time of the earliest node in the Timer_Queue.

  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
  // 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.

  int expire (void);
  // Run the <handle_timeout> method for all Timers whose values are
  // <= <Time_Value::gettimeofday>.  Returns the number of Timers
  // expired.
};
```

As shown above, the following are a number of classes associated with the `Timer_Queue`:

```
class Time_Value
  // = TITLE
  //     Operations on "timeval" structures.
{
public:
  // = Useful constants.
  static const Time_Value zero;
  // Constant "0".

  // = Initialization methods.

  Time_Value (long sec = 0, long usec = 0);
  // Constructor.

  Time_Value (const Time_Value &tv);
  // Copy constructor.

  // = The following are accessor/mutator methods.

  long sec (void) const;
  // Get seconds.

  void sec (long sec);
  // Set seconds.

  long usec (void) const;
  // Get microseconds.

  void usec (long usec);
  // Set microseconds.

  // = Helper method
  static Time_Value gettimeofday (void);

  // = The following are arithmetic methods for operating on
  // Time_Values.

  void operator += (const Time_Value &tv);
  // Add <tv> to this.

  void operator -= (const Time_Value &tv);
  // Subtract <tv> to this.

  friend Time_Value operator + (const Time_Value &tv1, const Time_Value &tv2);
  // Adds two Time_Value objects together, returns the sum.

  friend Time_Value operator - (const Time_Value &tv1, const Time_Value &tv2);
  // Subtracts two Time_Value objects, returns the difference.

  friend int operator < (const Time_Value &tv1, const Time_Value &tv2);
  // True if tv1 < tv2.

  friend int operator > (const Time_Value &tv1, const Time_Value &tv2);
  // True if tv1 > tv2.

  friend int operator <= (const Time_Value &tv1, const Time_Value &tv2);
  // True if tv1 <= tv2.

  friend int operator >= (const Time_Value &tv1, const Time_Value &tv2);
  // True if tv1 >= tv2.

  friend int operator == (const Time_Value &tv1, const Time_Value &tv2);
  // True if tv1 == tv2.

  friend int operator != (const Time_Value &tv1, const Time_Value &tv2);
  // True if tv1 != tv2.

private:
  void normalize (void);
  // Put the timevalue into a canonical form.
```

```
  long tv_sec_;
  // Seconds.

  long tv_usec_;
  // Microseconds.
};

class Event_Handler
  // = TITLE
  //      Provides an interface to an Event Handler, whose
  //      <handle_timeout> method is invoked when a
  //      previously registered timer expires.
{
public:
  virtual int handle_timeout (const Time_Value &,
                              const void *arg) = 0;
};
```

For this assignment, you will implement the `Timer_Queue` class using an *ordered linked list* of *absolute* times. Therefore, in the average- and worst-case, scheduling and canceling `Event_Handler` timers is $O(N)$ (where N is the total number of timers) and expiring timers is $O(K)$ (where K is the total number of timers that are $<$ the current time of day).

Asynchronous Test Driver

Implementing the Timer Queue is the easy part (in fact, I've provided you with a simple working version). The hard part is getting the timer queues to work correctly when used with asynchronous UNIX signals.

To test your timer queue, you will need to write a test program with a main event loop that repeatedly prompts the user for the following types of input and :

```
****
1) schedule_timer <usecs>
2) cancel_timer <timer_id>
^C) list_timers
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, schedule this with your `Timer_Queue`, and inform the OS kernel (via `ualarm(3C)`) when to generate a SIGALRM signal.

Your program must register a function to handle the SIGALRM signal (via `sigaction(2)`). When this function is called, your `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.[1]

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

[1] 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.

```
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 types ^C, this will generate a SIGINT signal, which your program also must register to catch (via `sigaction(2)`). When the SIGINT signal occurs, you must iterate through the `Timer_Queue` and print out the values of all the nodes in the linked list.

It is of utmost importance to make sure that methods on your `Timer_Queue` block out the SIGINT and SIGALRM signals (via `sigprocmask(2)`) when updating internal data structures. Otherwise, the linked list of the `Timer_Queue` can be corrupted. However, the SIGQUIT signal should never be blocked. When it occurs, your program should exit.

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-1/`. Here's a script that shows you how to set everything up and get these files:

```
% cd ~/cs422
% mkdir assignment-1
% cd assignment-1
% cp -r /project/adaptive/cs422/assignment-1/* .
% make
```

The `Makefile` and various header files are written for you. All you need to do is edit the `*.C` files to add the code to implement the signal-safe Timer Queue.