CS422: Object-Oriented Design and Programming

Program Assignment 5
Threaded Timer Queues
Due Saturday, October 17, 1997

Earlier programming assignments focused on asynchronous timer queues using UNIX signals and synchronous timer queues using `select`. This assignment will provide you with an opportunity to integrate your timer queue with another synchronous mechanism based on *threads*.

A thread is a single sequence of execution steps performed in the context of a process. In addition to an instruction pointer, a thread consists of other resources such as a run-time stack of function activation records, a set of general-purpose registers, and thread-specific data. The remainder of this document explains the thread and synchronization operations you'll need to use for this assignment. In addition, portions of the test drive code are shown.

Thread Operations

The thread operations you'll be using for your assignment include the following:

- *Thread creation* – Thread creation is handled via the `thr_create` function, which has the following signature:

```
int thr_create (void *stack_base,
                size_t stack_size,
                void *(*start_routine)(void *),
                void *arg,
                long flags,
                thread_t *new_thread);
```

  The `thr_create` function creates and starts a new thread using the `start_routine` pointer to function specified in the call. It returns 0 on success and non-0 on failure. The identify of the thread is returned to the caller via the `new_thread` parameter. The caller may supply a stack – if a `NULL` is used the library allocates a default stack.

- *Thread exit* – The `thr_exit` function terminates the invoking thread and sets the exit status to the specified value. The signature for `thr_exit` is:

```
void thr_exit (void *status);
```

  If the thread was *not* created detached, its identifier and status are retained until `thr_join` is called via another thread. If there are no remaining application threads, the process is exited with a 0 exit status.

- *Thread joining* – The `thr_join` function blocks until the specified thread exits. It's signature is:

```
int thr_join (thread_t wait_for,
              thread_t *departed,
              void **status);
```

If `wait_for` is 0, the functions waits for any undetached thread in the process to terminate. Otherwise, it waits for the `wait_for` thread id to terminate. If `departed` is non-`NULL` it points to location storing the ID of the terminated thread. If `status` is non-`NULL` it points to a location storing the exits status of the terminated thread.

- *Thread cancellation* – The `pthread_cancel` function enables a thread to terminate the execution of any thread in the process. The signature for `pthread_cancel` is:

  ```
  int pthread_cancel (pthread_t target_thread);
  ```

  `pthread_cancel` requests that `target_thread` be canceled. By default, cancellation is deferred until the target thread reaches a cancellation point. Cancellation cleanup handlers for `target_thread` are called when the cancellation is acted on.

See the manual pages for more information on threads.

Mutual Exclusion (Mutex) Operations

The simplest type of synchronization variable is the *mutex* (mutual exclusion) lock. Only one thread at a time may *own* a mutex lock. Mutexes are implemented to be highly efficient, but limited in functionality. For instance, mutex lock/unlock operations must be "fully-bracketed." The mutex operations you'll use in this program include the following

- *Initialization and termination functions* – These function initialize and destroy the state of a mutex.

  ```
  int mutex_init (mutex_t *mp, int type, void *arg);
  int mutex_destroy (mutex_t *mp);
  ```

  For this program, `type` should be `USYNC_THREAD` and `arg` should be `NULL`.

- *Mutex acquisition* – The `mutex_lock` function acquires ownership of the lock (wait on priority queue if necessary).

  ```
  int mutex_lock (mutex_t *mp);
  ```

- *Mutex release* – The `mutex_unlock` function releases the lock and unblocks the thread at head of priority queue.

  ```
  int mutex_unlock (mutex_t *mp);
  ```

  Only the owner of a mutex may unlock it.

See the manual pages for more information on mutexes.

Condition Variable Operations

Condition variables are a more powerful type of synchronization variable. Condition variables are used to wait on a particular *condition expression* until the condition is signaled. Condition expressions can be arbitrary C++ code, which typically involves data shared among multiple threads. Compared with mutexes, condition variables allow more complex scheduling decisions. The condition variable operations you'll use in this program include the following:

- *Initialization and termination functions* – These function initialize and destroy the state of a condition variable.

  ```
  int cond_init (cond_t *mp, int type, void *arg);
  int cond_destroy (cond_t *mp);
  ```

  For this program, `type` should be `USYNC_THREAD` and `arg` should be `NULL`.

- *Condition timed wait* – The `cond_timedwait` function Block on a condition or until absolute time-of-day has passed.

  ```
  int cond_timedwait (cond_t *cvp, mutex_t *mp, timestruc_t *abstime);
  ```

- *Condition signal* – The `cond_signal` function signals one thread that is blocked in `cond_timedwait`. If no thread is waiting, the signal is ignored. You must call `cond_timedwait` with an initialized, locked mutex.

  ```
  int cond_signal (cond_t *cvp);
  ```

See the manual pages for more information on condition variables.

Threaded Test Driver

The main goal of the assignment is getting the timer queue to work correctly when run concurrently with standard input from the terminal. The key to making this work correctly is to use mutexes to ensure that method calls on the `Timer_Queue` are accessed serially. Moreover, you'll need to use condition variables to synchronize the timer queue thread and the input thread. In addition, you'll need to use the `pthread_cancel` operation to allow the input handler thread to cancel the timer queue thread.

To test your `Reactor`, you will complete a test program that is structured as follows:

```
const int STDIN = 0;

struct Args
{
  thread_t tid_;
  Timer_Queue *tq_;
  cond_t *tq_cond_;
};

static void *
stdin_handler (void *arg)
{
  Args *args = (Args *) arg;

  Timer_Queue *tq = args->tq_;

  for (;;)
    {
      // Read user-input from stdin and perform commands.
      // ...
    }

  return 0;
}

static void *
timer_handler (void *arg)
```

```
{
  Args *args = (Args *) arg;
  Timer_Queue *tq = args->tq_;

  for (;;)
    {
      // Process timers.
      // ...
    }

  return 0;
}

int
main (void)
{
  thread_t tid;
  Timer_Queue tq;
  cond_t tq_cond;

  if (cond_init (&tq_cond, USYNC_THREAD, 0) != 0)
    return -1;

  Args args;
  args.tq_ = &tq;
  args.tq_cond_ = &tq_cond;

  if (thr_create (0, 0,
                  timer_handler,
                  (void *) &args,
                  THR_NEW_LWP,
                  &tid) != 0)
    perror ("thr_create"), exit (1);

  args.tid_ = tid;

  if (thr_create (0, 0,
                  stdin_handler,
                  (void *) &args,
                  THR_NEW_LWP,
                  &tid) != 0)
    perror ("thr_create"), exit (1);

  thread_t departed;
  void *status;

  while (thr_join (0, &departed, &status) == 0)
    fprintf (stderr, "thread %d exited with status %d\n",
             departed, status);

  thr_exit (0);
  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` passed as a parameter to the `stdin_handler` function. When the timer elapses, the `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.*:

```
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 the `Timer_Queue` and print out the values of all the nodes in the linked list.

Finally, if the user selects operation 4 you'll need to shutdown the program. This requires the input handler thread to cancel (`pthread_cancel(3T)`) the timer queue thread. Make sure that your timer queue handler thread correctly protects itself against thread cancellation by using the `pthread_cleanup_push` and `pthread_cleanup_pop` macros to protect the condition variable it uses to synchronize with the input handler thread.

Getting Started

You can get the thread-timer-test.C for the program from your account on cec. This file is stored in `/project/adaptive/cs422/assignment-5/`. Here's a script that shows you how to set everything up and get these files:

```
% cd ~/cs422
% mkdir assignment-5
% cd assignment-5
% cp -r /project/adaptive/cs422/assignment-5/thread-timer-test.C .
% make
```

The `Makefile` and various header files can be reused from your last assignment. You'll need to edit the `*.C` files and add the code to implement the test driver.

[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.