COSC 3360/6310 THIRD ASSIGNMENT

Spring 2018

Updated
April 11, 2018

The problem

More

- Long one-lane tunnel between Bear Valley, AK and Whittier, AK
- As it is run now
  - Loops through
    - 15 minutes for Whittier-bound cars
    - 15 minutes dead time
    - 15 minutes for Bear Valley-bound cars
    - 15 minutes dead time

  during tunnel opening hours

Your task

- Simulate its operation in compressed real-time using POSIX threads (pthreads)
- Our assumptions
  - Tunnel will be always open
  - One-hour cycle time compressed in 20s
    - 5s for Whittier-bound cars
    - 5s dead time
    - 5s for Bear Valley-bound cars
    - 5s dead time

Additional assumption

- We want to
  - Be able to control the number of cars that can be in the tunnel at the same time
  - Estimate the number of cars affected by this limitation

Your program

- Main program will
  - Create a thread updating the tunnel status every 5s
  - For each input line describing a car arrival:
    - Read a time delay, a direction, and a crossing time
    - Sleep for time delay
    - Create a child thread
  - Wait until all car threads have terminated
  - Terminate the tunnel thread

The car threads

- Your car threads will
  - Print a message
  - Wait until they can enter the tunnel
  - Print a message
  - Sleep for the duration of its crossing time
  - Print a message
  - Exit the tunnel and terminate

The tunnel thread

- Your tunnel thread will
  - Change the status of the tunnel every 5s
    - Whittier-bound only
    - No traffic
    - Bear Valley-bound only
    - No traffic
  - Be killed by the main thread once the simulation is over

The rules of the game

- A car can enter the tunnel
  - When the tunnel is open for cars moving in its directions
  - When there are less than **maxNCars** cars in the tunnel

Implementation

- Quite easy with two mutexes, shared counters and three condition variables

Using shared variables (I)

- At least six shared variables
    - Status of tunnel
        - Only updated by the tunnel thread
    - Maximum number of cars in the tunnel
    - Current number of cars in the tunnel
        - Updated by all car threads
        - Must be accessed in mutual exclusion
            - Use a mutex

Using shared variables (II)

- …
  - Number of Bear Valley-bound cars that crossed the tunnel
  - Number of Withier-bound cars that crossed the tunnel
  - Number of cars that had to wait because there were too many car in the tunnel

Creating pthreads (I)

- Declare first a child function:

  ```
  void *car(void *arg) {
          int i;
          // must cast the argument
          carNo = (int) arg;
          …
  } // car
  ```

- Thread ends with the function

Creating pthreads (II)

- Declare a thread ID
  - `pthread_t tid;`
- Start the thread:
  - `pthread_create(&tid, NULL, car, (void *) carNo);`
- Do not lose or overwrite the thread ID
  - You will need it again

Waiting for a specific thread

- Use pthread_join()
  - `pthread_join(tid, NULL);`

The problem

- The pthread library has no way to
  - □ Let you wait for an unspecified thread
  - □ Do the equivalent of:

    -
      ```
      for (i = 0; i < totalNCars; i++)
            wait(0);
      ```

The solution

- Must keep track of the thread id’s of all the threads of all the threads it has created:

  ```
  pthread_t cartid[maxcars];
  …
  …
  for (i = 0; i < totalNCars; i++)
      pthread_join(cartid[i], NULL);
  ```

Killing a thread

- You can use pthread_kill(…)
  - □
    ```
    #include <signal.h>
    pthread_kill(pthread_t tid, int sig);
    ```
- But
  - □ May terminate a thread that is inside a critical region
    - Mutex will be frozen in ***locked state***
  - □ Not a problem for this assignment

Safest alternative

- Use a shared variable

```
while (done == 0) {
    pthread_mutex_lock(&traffic_lock);
        traffic = "WB";
        pthread_cond_broadcast(&clear);
    pthread_mutex_unlock(&traffic_lock);
    . . .
```

Passing arguments to a thread

- **pthread_create()** allows a single **void** * argument to be passed to the new thread

```
pthread_create(&tid,NULL,
      car(void *) carNo);
```

- If you want to pass more than one argument, you must store them
  - □ In an array
  - □ In a structure

Pthread locks

- To create a pthread lock, use:

  -
    ```
    static pthread_mutex_t mylock;
    // must be declared static
    …
    pthread_mutex_init(&mylock, NULL);
    ```
- To request the lock, use:

  -
    ```
    pthread_mutex_lock(&mylock);
    ```
- To release the lock, use:

  -
    ```
    pthread_mutex_unlock(&mylock);
    ```

Pthread condition variables (I)

- The easiest way to create a condition variable is:

  -
    ```
    pthread_cond_t clear =
        PTHREAD_COND_INITIALIZER;
    ```

Pthread condition variables (II)

- To wait on a condition:

  ```
  pthread_mutex_lock(&mymutex);
     while (…)
        pthread_cond_wait(&clear,
                                &mymutex);
  …
  pthread_mutex_unlock(&mymutex);
  ```

A reminder

- Signals that are not caught by a waiting process are lost
  - □ Before setting up a **`pthread_cond_wait()`,** you must be sure that the resource you are waiting for is ***actually unavailable*** and the thread that holds it will do a **`pthread_cond_signal()`** when it releases it.
  - □ A thread holding a resource or changing the status of the tunnel should always send a **`pthread_cond_signal()`**

Pthread condition variables (III)

- To signal a condition:

```
pthread_mutex_lock(&mymutex);
   …
   pthread_cond_signal(&clear);
pthread_mutex_unlock(&mymutex);
```

- Critical section ***must*** use the same mutex as the one used around the corresponding `pthread_cond_wait( )`

Pthread condition variables (IV)

- To wake up ***everyon***e:

```
pthread_mutex_lock(&mymutex);
   …
   pthread_cond_broadcast(&clear);
pthread_mutex_unlock(&mymutex);
```

- Critical section ***must*** use the same mutex as the one used around the corresponding **`pthread_cond_wait( )`**

The car threads revisited (I)

- Will have different thread functions for Bear Valley-bound and Whittier-bound cars
  - Makes code much simpler
- Have separate mutexes for accessing
  - Current traffic directions
    - **`direction_lock`**
  - Current number of cars in the tunnel
    - **`car_lock`**

The car threads revisited (II)

- Your car threads will
  - Request **`direction_lock`** mutex
  - Print a message
  - Check current traffic direction
  - If needed, wait for broadcast from tunnel
  - Release **`direction_lock`** mutex

The car threads revisited (III)

- …
  - Request **`car_lock`** mutex
  - Check current number of cars in the tunnel
  - If needed, wait for signal from exiting car
  - Increment number of cars in tunnel
  - Print a message
  - Release **`car_lock`** mutex
  - Sleep for the duration of their crossing time

The car threads revisited (IV)

- …
  - Request **`car_lock`** mutex
  - Decrement number of cars in the tunnel
  - Send signal to a waiting car
  - Print a message
  - Update counter
  - Release **`car_lock`** mutex
  - Terminate

The tunnel thread

- While its work is not done, our tunnel thread will
  - Change the status of the tunnel every 5s
    - Whittier-bound only
      - Broadcast new condition
    - No traffic
    - Bear Valley-bound only
      - Broadcast new condition
    - No traffic

Mutexes and condition variables

- Two mutexes
  - **direction_lock**
  - **car_lock**

- Three condition variables
  - **`bb_can`**
  - **`wb_can`**
  - **`not_full`**

A last word

- This assignment is about learning to use pthread calls and condition variables
- Two mild challenges are
  - Learning to pass multiple arguments to pthreads
  - Accessing condition variables from within the correct critical sections
- Your code should be short and straightforward