COSC 3360
THIRD ASSIGNMENT

Spring 2020

The problem

More

- A poorly ventilated tunnel
- Decided to restrict
  - Number of northbound cars
  - Number of southbound cars
  - Total number of cars

  In the tunnel at any time

Your tasks

- Simulate the tunnel operation in real-time using POSIX threads (pthreads)

- Evaluate the number of cars affected by this limitation

Your program

- Main program will
  - For each input line describing a car arrival:
    - Read a time delay, a direction, and a travel time
    - Sleep for time delay
    - Create a child thread
  - Wait until all car threads have terminated
  - Print the simulation summary

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 rules of the game

- A northbound car can enter the tunnel when there are
  - Less than **`maxNCars`** cars ***and***
  - Less than **`maxNNBCars`** northbound inside

- A southbound car can enter the tunnel when there are
  - Less than **`maxNCars`** cars ***and***
  - Less than **`maxNSBCars`** southbound inside

Implementation

- Quite easy with one mutex, shared counters and one condition variable

Using shared variables

- At least seven shared variables
    - Maximum number of cars in the tunnel
        - Northbound, southbound and total
    - Current numbers of cars in the tunnel
        - Northbound, southbound and total
    - Number of cars that had to wait
- Must be accessed in mutual exclusion
    - Use a mutex

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

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)

- Decided to have different thread functions for northbound and southbound cars
  - □ Even though they are nearly identical
  - □ Makes code much simpler
  - □ Should not brag about it
- They share
  - Single **`car_lock`** mutex
  - Single **`wake_up`** condition variable

The car threads revisited (II)

- Your car threads will
  - Request **`car_lock`** mutex
  - Print a message
  - Check tunnel admission conditions
  - If needed, wait for a signal from a ***car leaving***
  - Update counters
  - Print a message
  - Release **`car_lock`** mutex

The car threads revisited (III)

- …
  - Sleep for the duration of their crossing time
  - Request **`traffic_lock`** mutex
  - Update counters
  - Print a message
  - Broadcast a change
  - Release **`traffic_lock`** mutex
  - Terminate

The condition variable

- A car may have to wait for the departure of
  - A car going in its direction
  - A car going in the opposite direction
- Use the same condition variable for all cars
- Receiving a signal does not guarantee that the car will be able to enter the tunnel
  - ***Must*** use a **`while`**

```
while (cannot_enter) {
   pthread_cond_wait(&wake_up, &car_lock);
```

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 less than 200 lines