COSC 3360-Operating System Fundamentals
Assignment #3: The Whittier Tunnel
➔ Due Monday, April 30 2018 at 11:59:59 pm ←

OBJECTIVE

This project will familiarize you with the use of pthreads, pthread mutexes and pthread condition variables.

THE PROBLEM

The city of Whittier, AK is linked to the outside world by a one-lane tunnel. It is 2.3 mile long, making it the second longest tunnel in the US and links Whittier with Bear Valley. The tunnel is operated according to a time division scheme giving to each direction 15 minutes of crossing time followed by 15 minutes of transition time during which no car can enter the tunnel. You are to simulate the operation of the tunnel using semaphores and a shared memory segment.

Your program should consist of

1. A ***main thread*** that will fork a tunnel process and the car processes according to the input specifications.
2. A ***tunnel thread*** that will start by allowing traffic going to Whittier for five seconds, will then disallow access to the tunnel for five seconds, then allow traffic from Whittier for five seconds, before disallowing again the traffic for five seconds and repeating the cycle.
3. One ***car thread*** per car wanting to cross the tunnel either from Whittier or from Bear Valley.

The input to your program consists of the maximum number of cars in the tunnel, the tunnel travel time and an ordered list of arriving cars as in:

```
10   // up to ten cars in the tunnel
1 WB 1  // car bound for Whittier arrives at t = 1s
// will take 1s to cross
2 BB 2  // car bound for Bear Valley arrives 2s after
// will take 2s to cross
1 WB 1  // car bound for Whittier arrives 1s after
// will take 1s to cross
```

Your program will terminate when all cars have crossed the tunnel.

YOUR OUTPUT

Your program should track both the current state of the tunnel and the cars crossing it. It should print one line of input each time the traffic direction changes and one line of output each time a car (a) arrives at the tunnel, (b) enters the tunnel and (c) leaves the tunnel. This line of output should identify each car by (a) its sequence number in the input file, starting with 1, and (b) whether the car is Whittier-bound or Bear Valley-bound:

```
The tunnel is now open to Whittier-bound traffic.
Car #1 going to Whittier arrives at the tunnel.
Car #1 going to Whittier enters the tunnel.
Car #1 going to Whittier exits the tunnel.
Car #2 going to Bear Valley arrives at the tunnel.
Car # 3 going to Whittier arrives at the tunnel.
Car # 3 going to Whittier enters the tunnel.
The tunnel is now closed to ALL traffic.
Car # 3 going to Whittier exits the tunnel.
The tunnel is now open to Bear Valley-bound traffic.
Car # 2 going to Bear Valley enters the tunnel.
Car # 2 going to Bear Valley exits the tunnel

1 car(s) going to Bear Valley arrived at the tunnel.
2 car(s) going to Whittier arrived at the tunnel
```

At the end of the simulation, your program should also print a summary with

1. The total number of Whittier-bound cars that arrived at the tunnel;
2. The total number of Bear Valley-bound cars that arrived at the tunnel;
3. The total number of cars that had to wait because the tunnel was full.

This summary could look like

```
2 car(s) going to Whittier arrived at the tunnel.
1 car(s) going to Bear Valley arrived at the tunnel.
0 car(s) had to wait because the tunnel was full.
```

PTHREADS

1. Don't forget the pthread include:

   ```
   #include <pthread.h>
   ```

2. All variables that will be shared by all threads must be declared `static` as in:

   ```
   static int whittierBound;
   ```

3. If you want to pass an integer value to your thread function, you should declare it **`void`** as in:

```
void *car(void *arg) {
    int seqNo;
    seqNo = (int) arg;
    …
} // car
```

Since most C++ compilers treat the cast of a **void** into an **int** as a fatal error, you must use the flag **-fpermissive**.

4. To start a thread that will execute the customer function and pass to it an integer value use:

```
pthread_t tid;
int i;
…
pthread_create(&tid, NULL,
       car, (void *) seqNo);
```

Had you wanted to pass more than one argument to the **car** function, you should have put them in a single array or a single structure.

5. To terminate a given thread from inside the thread function, use:

```
pthread_exit((void*) 0);
```

Otherwise, the thread will terminate with the function.

6. To terminate another thread function, use:

```
#include <signal.h>
pthread_kill(pthread_t tid, int sig);
```

Note that **pthread_kill(…)**is a dangerous system call because its default action is to immediately terminate the target thread even when it is in a critical section. The safest alternative to kill a thread that repeatedly executes a loop is through a shared variable that is periodically tested by the target thread.

7. To wait for the completion of a specific thread use:

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

Note that the pthread library has no way to let you wait for an unspecified thread and do the equivalent of:

```
for (i = 0; i < nchildren; i++)
    wait(0);
```

Your main thread will have to 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);
```

PTHREAD MUTEXES

1. To be accessible from all threads pthread mutexes must be declared `static`:

```
static pthread_mutex_t access;
```

2. To create a mutex use:

```
pthread_mutex_init(&access, NULL);
```

Your mutex will be automatically initialized to one.

3. To acquire the lock for a given resource, do:

```
pthread_mutex_lock(&access);
```

4. To release your lock on the resource, do:

```
pthread_mutex_unlock(&access);
```

PTHREAD CONDITION VARIABLES

1. The easiest way to create a condition variable is:

```
static pthread_cond_t ok =
    PTHREAD_COND_INITIALIZER;
```

2. Your condition waits must be preceded by a successful lock request on the mutex that will be passed to the wait:

```
pthread_mutex_lock(&access);
while (ncars > maxNCars)
    pthread_cond_wait(&ok, &access);
…
pthread_mutex_unlock(&access);
```

3. To avoid unpredictable scheduling behavior, the thread calling **pthread_cond_signal()** must ***own*** the mutex that the thread calling **pthread_cond_wait()** had specified in its call:

```
pthread_mutex_lock(&access);
…
pthread_cond_signal(&ok);
pthread_mutex_unlock(&access);
```

All programs passing arguments to a thread must be compiled with the **-fpermissive** ***flag. Without it, a cast from a*** **void** ***to anything else will be flagged as an error by some compilers.***

This document was updated last on Saturday, April 14, 2018.