COSC 3360-Operating System Fundamentals
Assignment #3: The Poorly Ventilated Tunnel
➔ Due Monday, April 27 2020 at 11:59:59 pm ←

OBJECTIVE

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

*Your program will earn you **no credit** if it uses **semaphores** instead of mutexes and condition variables.*

THE PROBLEM

A car tunnel is so poorly ventilated that it has become necessary to restrict:

1. The number of northbound cars in the tunnel,
2. The number of southbound cars in the tunnel,
3. The total number of cars in the tunnel.

Your assignment is to write a C/C++ simulating the enforcement of these restrictions using Pthread mutexes and condition variables. Assignments using semaphores will

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. One ***car thread*** per car wanting to cross the tunnel.

The input to your program consists of the maximum number of northbound cars in the tunnel, the maximum number of southbound cars , and the maximum total number of cars in the tunnel followed by an ordered list of arriving cars as in:

```
3   // up to three cars at a time
2  //up to two NB cars at a time
2  //up to two SB cars at a time
1 S 3  // SB car arrives at t = 1s
// will take 3s to go through the tunnel
2 N 4  // northbound car arrives 2s after
// will take 4s to go through the tunnel
1 S 4  // southbound car arrives 1s after
// will take 4s to go through the tunnel
```

Your program will terminate when all cars have gone through the tunnel.

YOUR OUTPUT

Your program should start by echoing the first here lines of input as in:

```
Maximum number of cars in the tunnel: 2
Maximum number of northbound cars: 1
Maximum number of southbound cars: 1
```

It should print 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 its northbound or southbound sequence number as in

```
Southbound car # 1 arrives at the tunnel.
Southbound car # 1 enters the tunnel.
Northbound car # 1 arrives at the tunnel.
Northbound car # 1 enters the tunnel.
Southbound car # 2 arrives at the tunnel.
Northbound car # 2 arrives at the tunnel.
Southbound car # 1 exits the tunnel.
Southbound car # 2 enters the tunnel.
```

At the end of the simulation, your program should also print a summary with the total number of northbound and southbound cars that went through the tunnel as well as the total number of cars that had to wait because of the restrictions.

This summary could look like:

```
2 northbound car(s) crossed the tunnel.
2 southbound car(s) crossed the tunnel.
2 car(s) had to wait.
```

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 MaxNCarsInTunnel;
   ```

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. If you have to terminate another thread function, you many 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.***