Universidad Carlos III De Madrid

Sistemas Operativos

Práctica 3: Programación multi-hilo

Cristhian Alcántara López
NIA: 100356418 Grupo: 81
100356418@alumnos.uc3m.es

Iván Estévez Albuja
NIA: 100346193 Grupo: 81
100346193@alumnos.uc3m.es

Alexander González
NIA: 100389461 Grupo: 81
1003894613@alumnos.uc3m.es

Grado en Ingeniería Informática

3 de mayo de 2018

Índice

1. Descripción del diseño del código

Lo que se busca es simular el funcionamiento de un aeropuerto contando con 3 roles: Jefe de pista, Radar aéreo y Torre de control, los cuales deben de operar de forma concurrente. Para esto, usaremos 3 mutex que describimos a continuación:

- `mutex_id_creacion`: Controla el acceso sobre las variables `id` y `ultimo` por parte de los hilos productores.
- `mutex_insercion`: Controla el acceso por parte de los productores (jefe de pista y radar) a la variable `siguiente` y al buffer.
- `mutex_buffer`: Controla el acceso a la cola circular por parte de los productores y el consumidor (la torre de control).

Además, hacemos uso de cuatro variables condición:

- `no_lleno` y `no_vacio`: Asociadas a `mutex_buffer`.
- `insertado_jefe` e `insertado_radar`: Asociadas a `mutex_insercion`.

1.1. Hilo del jefe de pista

En primer lugar, el jefe de pista debe crear estructuras tipo plane. Mediante "`pthread_mutex_lock(&mutex_id_creacion)`" solicitamos acceso al mutex de creación para que el hilo se boquee hasta su obtención. Iremos aumentando el `id` (variable global) para cada nuevo avión creado. Luego le daremos valores a todos los atributos de `despegues` (array de punteros a aviones que despegan, esta variable es global, pero solamente el jefe de pista accederá a ella).

Una vez se crea el avión se procederá a mostrar en pantalla un mensaje "[TRACKBOSS] Plane with id (id del avión a despegar) checked" y se libera el mutex de creación. Una vez se llegue al último avión se actualiza la variable gobal `ultimo` a 0, indicando que el jefe de pista ha creado todos los aviones correspondientes.

Seguidamente se realiza la inserción del avión creado en el buffer circular. Primero se bloquea mediante "`pthread_mutex_lock(&mutex_insercion)`", luego evaluaremos si el avión que se insertará está en el orden correcto, esto se hace con la variable condición `insertado_radar`, que indica que el radar aéreo insertó en la posición correcta el avión creado por él, de no ser este el caso, se liberará el mutex de inserción y el hilo queda en espera.

En caso de que le toque al jefe de pista insertar, se debe acceder al buffer. Este se bloquea mediante "`pthread_mutex_lock(&mutex_buffer)`" luego evaluaremos si el buffer esta lleno, de ser este el caso, el mutex se libera y el hilo permanece bloqueado hasta que la variable condición `no_lleno` pase a ser verdadera.

Almacenamos el avión en el hangar, luego, la variable condición `no_vacio` pasa a ser verdadera ya que el buffer tiene ahora al menos un elemento. Pasaremos a mostrar por pantalla "[TRACKBOSS] Plane with id (id del avión a despegar) ready to take off" y liberamos el mutex del buffer.

Para finalizar, asignamos a `siguiente` el id del siguiente avión a insertarse en la cola, la variable condición `insertado_jefe` pasa a ser cierta ya que el jefe de pista ha insertado en el buffer en el orden correcto, desbloqueamos el mutex de inserción. Tras esto, el hilo termina.

1.2. Hilo del radar aéreo

El código para el radar aéreo es similar al del jefe de pista. La estructura es la misma.

Primero se bloquea el mutex de creación, se altera la variable `id` y se asignan los atributos de los aviones que aterrizan. Estos aviones se almacenan en el array `aterrizajes`, variable global a la que solamente accede este hilo.

Tras la creación del último avión a aterrizar, la variable gobal `ultimo` se actualiza a 1, indicando que el radar aéreo ha creado todos los aviones correspondientes. Si durante la creación

de este último avión la variable `ultimo` era -1 significa que no es el último avión del día, pero si es 0 significa que el jefe de pista ya ha terminado y este avión es el último del día.

Para la inserción en el buffer, se bloquea el mutex de inserción. Se comprueba que el avión a insertar esté en orden, en caso contrario el mutex se libera y el hilo se queda esperando con la variable condición `insertado_jefe`.

Para el acceso al buffer se bloquea el mutex de buffer. El mutex se libera si el buffer estaba lleno, hasta que la variable condición `no_lleno` pase a ser verdadera. El avión se inserta y se libera el mutex del buffer.

Tras esto, asignamos a `siguiente` el id del siguiente avión a insertarse en la cola, la variable condición `insertado_radar` pasa a ser cierta ya que el jefe de pista ha insertado en el buffer en el orden correcto, Por último, desbloqueamos el mutex de inserción y el hilo termina.

1.3. Hilo de la torre de control

Este hilo recorre todos los aviones que se han creado (`n_land` + `n_take_off`). Primero se intenta acceder al buffer, bloqueando el mutex mediente "`pthread_mutex_lock(&mutex_buffer)`". Si el buffer está vacío el hilo libera el mutex y se queda bloqueado hasta que la variable condición `no_vacio` sea verdadera.

Una vez se tiene acceso al buffer, se llama a la función `queue_get` para obtener el avión que lleve más tiempo en la cola. Tras esto se liberan los threads bloqueados por la variable condición `no_lleno`, ya que ahora, la cola tiene al menos, un espacio libre. Se libera el mutex del buffer.

Según el contenido del campo `action` de la estructura plane obtenida, se procesará el avión para despegue o aterrizaje. En cada uno de estos casos el hilo duerme el tiempo indicado en el campo `time_action` e imprime el mensaje correspondiente por pantalla. Además, se tiene en cuenta si el avión es el último avión del día mediante el atributo `last_flight`.

Por último el hilo crea un fichero (se trunca si ya existía), en el que escribe el contenido de las variables: processed, landed y taken_off.

1.4. Main

La función *main* se encarga de asignar los argumentos del programa a las variables que utilizarán los hilos. En caso de que los argumentos sean incorrectos, mostrará mensajes de error por pantalla.

Una vez conocidos el número de aterrizajes y despegues se reserva memoria dinámica para las estructuras donde irán almacenados. Luego, se inicializa el buffer.

Tras esto se inicializan los tres mutex: el de creación, el de inserción y el del buffer. También se inicializan las variables condición: `no_lleno`, `no_vacio`, `insertado_jefe` e `insertado_radar`.

Se lanzan los hilos mediante `pthread_create` y, el main espera a su finalización mediante `pthread_join`. Por último se liberan todos los recursos utilizados y se imprime el banner de despedida.

1.5. *queue.c*

La cola circular donde se almacenan estructuras tipo `plane` esta implementada en este fichero.

En primer lugar se define una estructura tipo Queue, con campos que definirán sus parámétros: `aviones` array de punteros a estructuras tipo plane, `capacity` número máximo de elementos que se pueden almacenar a la vez, `size` número de elementos encolados en un momento determinado, `front` y `rear` que controlan la posición del elemento a expulsar y la posición donde se insertará un nuevo elemento respectivamente.

Seguidamente se crea una variable tipo cola mediante "Queue *Q" que servirá como variable global para las funciones descritas a continuación.

1.5.1. *queue_init()*

Primero reservamos espacio para la cola, luego vemos que si hay algún problema con la asignación de espacio y de caso de haber error mostramos un mensaje. Luego declararemos los

parámetros de “Q”, el parámetro que se le introduce a este método será el número de aviones máximos que almacenara, de igual forma evaluamos si hubo algún problema en la asignación de espacio. Para finalizar la capacidad de “Q” será igual al parámetro (size) introducido y sus demás valores serán 0, ya que se está inicializando, al acabar de asignar valores se mostrará un mensaje “[QUEUE] Buffer initialized”.

1.5.2. *queue_put()*

Primero comprobaremos que la cola no este llena, en caso de estarlo mostraremos un mensaje: “ERROR queue_put: la cola está llena.” De no ser el caso, aumentamos el tamaño actual por el parámetro introducido en este proceso, desplazamos el punto de cola y actualizamos la cola. Para finalizar mostramos el mensaje: “[QUEUE] Storing plane with id (id del avión del parámetro introducido”.

1.5.3. *queue_empty()*

Esta función se encarga de consultar el estado de la cola, devuelve 1 si la cola esta vacía y 0 si no lo está.

1.5.4. *queue_full()*

Esta función se encarga de consultar el estado de la cola, devuelve 1 si la cola esta llena y 0 si no lo está.

1.5.5. *queue_destroy()*

Esta función elimina la cola y libera todos los recursos asignados.

1.5.6. *queue_get()*

Esta función extrae elementos de la cola si ésta no está vacía. Primero evaluamos que la cola no esté vacía, en caso de estarlo se muestra un mensaje de error. En caso de no serlo, creamos el elemento que se va a retornar (struct plane*) y reservamos su espacio. Luego evaluamos que la reserva de memoria haya sido exitosa, en caso de error se mostrará un mensaje. Almacenamos en la variable el primer elemento de la cola, decrementamos el tamaño actual y desplazamos el puntero del primer elemento. Finalizamos mostrando un mensaje en pantalla: “[QUEUE] Getting plane with id (id del avión)”.

2. Batería de Pruebas

Hemos realizado las pruebas en el servidor de la universidad `guernika.lab.inf.uc3m.es`. Para las pruebas suponemos que estamos trabajando en el directorio ∼`/Documentos/Practica3_SSOO/` donde se encuentran tanto el código realizado como el ejecutable *arcport*. La siguiente prueba de integración es usada para probar la funcionalidad de los diferentes componentes del programa. Para esta se usa el modo de ejecución con parámetros `<n_planes_takeoff> <time_takeoff> <n_planes_land> <time_landing> <size>`. El resultado en consola se ha enumerado enumerado para facilitar la explicación del funcionamiento en la memoria. Esta numeración no aparece en la ejecución.

```
a0389461@guernika-2018:∼/Documentos/Practica3_SSOO$  ./arcport 4 2 2 1 6
1.  ****************************************
2.  Welcome to ARCPORT - The ARCOS AIRPORT.
3.  ****************************************
4.  [QUEUE] Buffer initialized
5.  [CONTROL] Waiting for planes in empty queue
6.  [TRACKBOSS] Plane with id 0 checked
7.  [QUEUE] Storing plane with id 0
8.  [TRACKBOSS] Plane with id 0 ready to take off
9.  [TRACKBOSS] Plane with id 1 checked
10. [RADAR] Plane with id 2 detected!
11. [QUEUE] Getting plane with id 0
12. [CONTROL] Putting plane with id 0 in track
13. [QUEUE] Storing plane with id 1
14. [TRACKBOSS] Plane with id 1 ready to take off
15. [TRACKBOSS] Plane with id 3 checked
16. [QUEUE] Storing plane with id 2
17. [RADAR] Plane with id 2 ready to land
18. [RADAR] Plane with id 4 detected!
19. [QUEUE] Storing plane with id 3
20. [TRACKBOSS] Plane with id 3 ready to take off
21. [TRACKBOSS] Plane with id 5 checked
22. [QUEUE] Storing plane with id 4
23. [RADAR] Plane with id 4 ready to land
24. [QUEUE] Storing plane with id 5
25. [TRACKBOSS] Plane with id 5 ready to take off
26. [CONTROL] Plane 0 took off after 2 seconds
27. [QUEUE] Getting plane with id 1
28. [CONTROL] Putting plane with id 1 in track
29. [CONTROL] Plane 1 took off after 2 seconds
30. [QUEUE] Getting plane with id 2
31. [CONTROL] Track is free for plane with id 2
32. [CONTROL] Plane 2 landed in 1 seconds
33. [QUEUE] Getting plane with id 3
34. [CONTROL] Putting plane with id 3 in track
35. [CONTROL] Plane 3 took off after 2 seconds
36. [QUEUE] Getting plane with id 4
37. [CONTROL] Track is free for plane with id 4
38. [CONTROL] Plane 4 landed in 1 seconds
39. [QUEUE] Getting plane with id 5
40. [CONTROL] Putting plane with id 5 in track
41. [CONTROL] After plane with id 5 the airport will be closed
42. [CONTROL] Plane 5 took off after 2 seconds
43. Airport Closed!
```

```
44. ****************************************
45. ---> Thanks for your trust in us <---
46. ****************************************
a0389461@guernika-2018:~/Documentos/Practica3_SSOO$  more resume.air
        Total number of planes processed: 6
        Number of planes landed: 2
        Number of planes taken off: 4
```

2.1. Jefe de Pista

Dado el ejemplo anterior, se espera que 4 aviones despeguen, por la tanto se espera la siguiente secuencia 4 veces:

```
[TRACKBOSS] Plane with id <id> checked
    ⋮
[TRACKBOSS] Plane with id <id> ready to take off
```

En la línea 6 se muestra un mensaje indicando como el TRACKBOSS crea el primer avión y consecuentemente en la línea 8 después de poner el avión en la cola compartida, se retorna mensaje indicando que el avión con id 0 esta listo para despegar. La línea 12 nos demuestra que el CONTROL exitosamente fue notificado por el TRACKBOSS de que había un avión listo para despegar.

El TRACKBOSS repite este ciclo por cada uno de los 4 aviones supuestos a despegar. Se puede observar una cantidad correcta de aviones recibidos y alistados para despegar por el TRACKBOSS en líneas 6 y 8, 9 y 14, 15 y 20, 21 y 25. Se observan dos mensajes por cada uno de los 4 aviones como es esperado.

Es importante notar la secuencia numérica en el id de los aviones procesados es la esperada, y encaja con el procesamiento concurrente de los aviones que intentan aterrizar.

Otros casos importantes y casos extremos

- Dado [./arcport] como es esperado el numero de aviones por defecto a despegar es 4 y se muestra 8 mensajes indicados por el TRACKBOSS con una secuencia correcta de el id de los aviones
- Dado [./arcport -1 2 2 1 6] el mensaje “ERROR los argumentos no pueden ser negativos” es retornado como esperado
- Dado [./arcport 0 2 2 1 6] como esperado el TRACKBOSS no realiza ninguna acción
- Dado [./arcport 20 2 2 1 6] como esperado el TRACKBOSS procesa los aviones a despegar 20 veces y retorna los dos mensajes correspondientes a cada uno

2.2. Radar aéreo

Dado el ejemplo anterior se esperan que 2 aviones aterricen. Por lo tanto, se espera la siguiente secuencia dos veces:

```
[RADAR] Plane with id <id> detected!
    ⋮
[RADAR] Plane with id <id> ready to land
```

En la línea 10 se observa como el RADAR detecta el primer avión a aterrizar. A este le asigna un id correspondiente con la secuencia existente en cola y añade este avión a la cola. Luego en la línea 17 podemos observar como el RADAR notifica que existe un avión listo para aterrizar.

Podemos observar el mensaje y también podemos observar como el CONTROL eventualmente recibe la señal y aterriza este avión en la línea 32.

El RADAR repite este ciclo por cada uno de los 2 aviones supuestos a aterrizar. Se puede observar una cantidad correcta de aviones recibidos y alistados para aterrizar por el RADAR en líneas 10 y 17, 18 y 23. Se observan dos mensajes por cada uno de los 2 aviones como es esperado.

De forma similar al TRACKBOSS, es importante notar la secuencia numérica en el id de los aviones procesados es la esperada, y encaja con el procesamiento concurrente de los aviones que intentan despegar.

Otros casos importantes y casos extremos

- Dado [./arcport] como es esperado el número de aviones por defecto a aterrizar es 3 y se muestra 6 mensajes indicados por el RADAR con una secuencia correcta del id de los aviones
- Dado [./arcport 4 2 -1 1 6] el mensaje "ERROR los argumentos no pueden ser negativos" es retornado como esperado
- Dado [./arcport 4 2 0 1 6] como esperado el RADAR no realiza ninguna acción
- Dado [./arcport 4 2 20 1 6] como esperado el RADAR procesa los aviones a aterrizar 20 veces y retorna los dos mensajes correspondientes a cada uno

2.3. Torre de control

Mientras no haya aviones en la cola, el CONTROL debe retornar el mensaje:

```
[CONTROL] Waiting for planes in empty queue
```

Lo cual podemos apreciar en la línea 4 ya que en este punto, ni el TRACKBOSS o el RADAR han insertado aviones en la cola.

Adicionalmente, el CONTROL debe mostrar un mensaje indicando que un avión ha sido extraído de la cola circular para ejecutar un despegue. De ser así el siguiente mensaje debe ser retornado:

```
[CONTROL] Putting plane with id <id> in track
```

En el ejemplo anterior, dado que 4 vuelos están supuestos a despegar, esperamos encontrar este mensaje 4 veces por cada vez que el CONTROL reciba señal del TRACKBOSS de hay un avión listo para despegar. En el ejemplo observamos el mensaje en las líneas 12, 28, 34 y 40.

De forma similar el siguiente mensaje es esperado por cada avión extraído de la cola para aterrizar:

```
[CONTROL] Track is free for plane with id <id>
```

Al haber 2 aviones para aterrizar, esperamos que el CONTROL retorne el mensaje dos veces lo cual podemos comprobar en las líneas 31 y 37.

Consecuentemente, se debe retornar un mensaje indicando

```
[CONTROL] After plane with id <id> the airport will be closed
```

Para indicar que el ultimo avión va a ser ejecutado. Una vez este ha sido ejecutado se espera el mensaje

```
Airport Closed!
```

Los dos mensajes anteriores los podemos encontrar en líneas 41 y 43 respectivamente. Finalmente, se crea o se trunca un archivo resume.air que en este ejemplo obtenemos como esperado: Total number of planes processed: 6, Number of planes landed: 2 y Number of planes taken off: 4.

Otros casos importantes y casos extremos

- Dado [./arcport] como es esperado el tiempo por defecto para los aviones a despegar es 2 s y el tiempo para aterrizar es 3 s
- Dado [./arcport 4 2 -2 1 6] el mensaje "ERROR los argumentos no pueden ser negativos" es retornado como esperado
- Dado [./arcport 4 2 2 -1 6] el mensaje "ERROR los argumentos no pueden ser negativos" es retornado como esperado
- Dado [./arcport 4 2 2 1 -6] el mensaje "ERROR los argumentos no pueden ser negativos" es retornado como esperado
- Dado [./arcport 4 10 2 10 6] como es esperado las operaciones de aterrizaje y despegue tardan el tiempo especificado para cada una, lo que se pueda apreciar ya que en esta caso el programa tarda mucho mas en terminar de lo que terminaba antes
- Dado [./arcport 4 0 2 0 6] es fácil apreciar que el programa termina más rápido asumiendo que las operaciones de despeje y aterrizaje son instantáneas
- Dado [./arcport 4 2 2 1 0] se obtiene el mensaje "ERROR el tamano del buffer debe ser mayor que 0."
- Dado [./arcport 4 2 2 1 1] el programa funciona correctamente sin ningún problema
- Después de correr [./arcport 4 2 2 1 0] y luego correr [./arcport] se observa que el contenido de resume.air has sido sobrescrito.

2.4. Cola sobre un buffer circular

El funcionamiento correcto de todos los componentes en la prueba usando [./arcport 4 2 2 1 6] es en sí prueba del funcionamiento correcto de la cola sobre el buffer circular. Siendo mas especifico, se espera obtener el siguiente mensaje siempre que queue_put() se invoca:

```
[QUEUE] Storing plane with id <id>
```

Lo cual podemos apreciar en líneas 7, 13, 16, 19, 22, y 24 con un total de las 6 ocurrencias esperadas, 4 para despegar y 2 para aterrizar.

Adicionalmente se muestra el mensaje:

```
[QUEUE] Getting plane with id <id>
```

Cada vez que se invoque la función queue_get() lo cual podemos comprobar funcionando exitosamente ya que encontramos las 6 ocurrencias esperadas en las líneas 11, 27, 30, 33, 36, 39.

2.5. Modos de ejecución

Ya hemos comprobado la ejecución sin parámetros y la ejecución con los 5 parámetros esperados funciona exitosamente. Adicionalmente es importante notar los siguientes casos:

- Dado [./arcport 4 2 2 1 6 7] es decir dado más de 5 argumentos se retorna el siguiente error:

  ```
  usage 1: ./arcport
  usage 2: ./arcport <n_planes_takeoff> <time_to_takeoff> <n_planes_to_arrive>
  <time_to_arrive> <size_of_buffer>
  ```

- Dado [./arcport 4 2 2] es decir dado menos de 5 argumentos se retorna el siguiente error:

```
usage 1: ./arcport
usage 2: ./arcport <n_planes_takeoff> <time_to_takeoff> <n_planes_to_arrive>
<time_to_arrive> <size_of_buffer>
```

- Dado [./arcport 3 1 2 1 2] es decir un buffer menor que el numero de aviones a procesar, el programa funciona exitosamente y procesa todos los aviones en el orden esperado.

2.6. Integración y ejemplos de ejecución

El ejemplo al principio de la batería de pruebas nos sirve de muestra del comportamiento exitoso de la integración de los componentes del programa.

3. Conclusiones

Los principales problemas surgieron por la concurrencia. En un principio se usaban solo dos mutex, el de creación para proteger el id y el de buffer, pero esto causaba que no se insertaran en el buffer ordenados por id. Por este motivo tuvimos que recurrir a un tercer mutex para la inserción, la variable global `siguiente` y las dos nuevas variables condición.

También encontramos problemas en la creación de aviones. Si almacenábamos sus punteros en variables locales de los productores, estos "no sobrevivían" ya que los hilos productores terminan antes que el hilo de la torre de control. Por lo que decidimos almacenarlos en variables globales.

Consideramos que la práctica aporta mucho aprendizaje sobre programación multi-hilo y nos ha parecido bastante útil para entender cómo funcionan los hilos, así como los mecanismos para que no se den condiciones de carrera y se puedan acceder a sus recursos ordenadamente y de manera concurrente.