



11 DE ENERO DE 2019

MEMORIA PRACTICA ASO 2
PROGRAMACION PARALELA MEDIANTE PASO DE MENSAJES

JORGE LÓPEZ NATAL



Índice

Introducción	2
includes/vehiculo.h e includes/vehiculo.c	2
./coche.c y ./camion.c	4
includes/parking.h y ./parking.c.....	4
Ejemplo de ejecución	6
Problemas encontrados durante la realización de la práctica	6

Introducción

La práctica está compuesta de varios ficheros que en conjunto logran simular múltiples vehículos entrando y saliendo indefinidamente de un parking. El resumen del funcionamiento es el siguiente:

- Los vehículos (coches y camiones) mandarán en bucle mensajes al parking (proceso maestro) solicitando una plaza donde aparcar.
- El parking procesará el mensaje y comprobará si existen plazas disponibles. Si fuera el caso, el vehículo procedería a almacenarse en el parking.
- En el caso contrario, se bloquearía al vehículo y a todos los siguientes, formando una cola. Los vehículos en la cola entrarán en el parking conforme los vehículos aparcados abandonen el parking.
- El vehículo sale del parking.

Antes de pasar a describir los ficheros clave de la práctica, vamos a destacar otros detalles de menor importancia:

- La práctica se puede encontrar en su entereza en https://github.com/Jormii/Practica_ASO_2.
- Los ficheros `includes/queue.h` e `includes/queue.c` implementan una cola muy sencilla. El código no es original, sino que fue tomado del siguiente enlace: <https://gist.github.com/ArnonEilat/4471278>. Se modificó la fuente ligeramente con tal de adecuarse a la práctica. Esta modificación afecta al struct `queue_node`, declarado en `includes/queue.h`.
- El ejecutable shell `./compilar` compila los archivos `./coche.c`, `./camion.c` y `./parking.c`.
- El ejecutable shell `./ejecutar` lanza la práctica. Aquí hay que hacer notar varias cosas:
 - El programa `./parking` es el primero que debe ser lanzado. Esto garantiza que el proceso parking tendrá el identificador 0. Si no fuera así, la práctica no funcionaría.
 - El programa `./parking` puede tomar ningún argumento o dos enteros, los cuales se corresponden, respectivamente, con el número de pisos del parking y el número de plazas en cada uno de ellos. Independientemente de si no se indican argumentos o alguno de los valores argumento es inválido, el parking tendrá unas dimensiones mínimas de 1x2.
- El fichero `./hostfile` puede ver su valor de slots modificado. Este valor indica el número máximo de procesos que puede lanzar mpi.

`includes/vehiculo.h` e `includes/vehiculo.c`

Estos dos sencillos ficheros tienen el objetivo de representar un vehículo y de ejecutar un bucle sin fin en el que el vehículo mandará mensajes al parking para entrar y salir de éste.

El vehículo es representado por medio de un struct llamado *vehiculo*. Este struct está formado por dos enteros sin signo: la matrícula y el tamaño del vehículo. A efectos de la práctica, las matrículas serán los identificadores de sus procesos, valor que se encontrará en el intervalo [1, slots-1], siendo slots el número especificado en `./hostfile`.

El bucle de envío de mensajes se realiza a través de la función **`void vehiculo_bucle_principal(unsigned int tamano_vehiculo, int *argc, char **argv[])`**, la cual agrupa la entera funcionalidad del vehículo. Esta función sigue el siguiente esquema:

- Se inicia MPI y se crea el struct del vehículo.
- Se configura la variable que agrupa los argumentos que se enviarán por medio de `MPI_Send()` al proceso parking para indicarle que quiere aparcar.
- Una vez envíe el mensaje, invocará `MPI_Recv()`. El vehículo se bloqueará en espera de un mensaje enviado por el parking. Esto tiene dos objetivos: primero, recibir confirmación de que el vehículo ha sido aparcado, y segundo, conocer en qué planta y plaza ha sido aparcado (valores de retorno).
- Dormirá un tiempo aleatorio, lo cual simula una permanencia dentro del parking.
- Una vez despierto, volverá a configurar la variable de argumentos para indicar que desea salir del parking.
- Volverá a dormir un tiempo aleatorio antes de volver al segundo paso.

En la descripción de la función se han mencionado una variable de argumentos y unos valores de retorno. Estas dos variables son arrays de enteros sin signo. El primero está formado por cuatro elementos. Cuando el vehículo quiera aparcar, el array habrá de contener: código de operación, tamaño y matrícula (identificador). Por otro lado, cuando el vehículo quiera salir del parking, los argumentos habrán de ser: código de operación, tamaño, piso en el que está aparcado, y plaza.

Con tal de unir todos los argumentos en única variable, se ha creado un array de cuatro elementos. En el primer caso, uno de los elementos estará inutilizado, mientras que en el segundo estará completo. El orden usado para los argumentos es el siguiente:

- Argumentos de aparque: Código de operación, tamaño, matrícula, *desuso*.
- Argumentos de salida: Código de operación, tamaño, piso, plaza.

En el caso del array de retorno, se trata de un array de dos posiciones que almacena respectivamente: piso y plaza ocupados.

Estos dos ficheros funcionan a modo de superclase para los posteriores `./coche.c` y `./camion.c`.

Versiones anteriores

Si se accede al GitHub, se puede comprobar que existe una rama del proyecto con el nombre de “Vehiculo.c-unificado”. Ésta es una versión previa a la final que poseía un funcionamiento distinto del bucle: cada vez que el vehículo mandaba el mensaje de aparque, el parking siempre respondía al vehículo indicando únicamente si había encontrado sitio para él. Si se encontraba, continuaba como se mencionó en el esquema anterior.

Pero en caso de que la respuesta fuese negativa, el vehículo dormía un tiempo de 1 segundo. Una vez transcurrido, volvía a intentar aparcar. De esta forma, los vehículos que no disponían de plaza se alternaban circularmente. Esto tenía dos problemas:

- No se simulaba el comportamiento de una cola real.
- No se garantizaba que algunos vehículos accediesen al parking.

./coche.c y ./camion.c

Estos dos programas sólo realizan una tarea y es llamar a la función `vehiculo_bucle_principal()` de `includes/vehiculo.c`, indicándole el tamaño del vehículo correspondiente: 1 para el coche, 2 para el camión.

Versiones anteriores

Tanto `./coche.c` como `./camion.c` poseían el código de la función `vehiculo_bucle_principal()`. Debido a que el código estaba (en enorme medida) duplicado, hacer un cambio suponía modificar ambos ficheros para un mismo cambio.

includes/parking.h y ./parking.c

El objetivo de estos dos ficheros es simular el comportamiento del gestor del parking. Para representar el parking se utilizan dos structs:

- *parking*: El cual almacena un array de structs *piso* y las dimensiones del parking.
- *piso*: Almacena el número de plazas disponibles en el piso y un array de enteros sin signo que almacena las matrículas de los vehículos aparcados en cada plaza. Un cero equivale a una plaza vacía.

También se utilizará una cola cuya función será almacenar en ella todos los vehículos que al intentar aparcar o bien no había sitio para ellos o bien la cola ya poseía algún elemento, es decir, había otros vehículos esperando para entrar.

El proceso parking se encontrará en funcionamiento sólo cuando reciba algún mensaje por parte de los vehículos. Es en este momento en el que el parking empezará a operar en función del código de operación que haya recibido: aparcar o salir.

- Caso aparcar:
 - Primero el parking comprobará si existe algún vehículo en la cola esperando para entrar al parking. Si es así, introducirá el vehículo en la cola.
 - En caso contrario, se busca en el parking algún sitio en el que aparcar el vehículo. Si se encuentra, se aparcará al vehículo y se le enviarán los valores de retorno mencionados anteriormente. Si no se encontrase, se introduciría el vehículo en la cola.
- Caso salir:
 - Se vacían las plazas que ocupaba el vehículo.
 - Se realiza peek sobre la cola y se comprueba si el vehículo que ha abandonado el parking ha dejado espacio libre para el vehículo que se encuentra en la cabeza de la cola. Si se encuentra plaza para el vehículo, se aparca, se le mandan por mensaje los valores de retorno y se elimina de la cola. Se volverá a realizar esta operación hasta que se vacíe la cola o no se encuentre sitio para el vehículo en su cabeza.

Se contempla el caso en el que se haya producido un error durante el envío de mensajes (se controla mediante el código de operación). En dicha situación, el parking ignora el mensaje recibido. Así el proceso remitente permanecerá bloqueado, puesto que éste está esperando un mensaje del parking.

Para llevar a cabo el funcionamiento mencionado, `./parking.c` recurre a las siguientes funciones:

- **`void crear_parking(parking *park, int pisos, int plazas)`**: Crea un parking de pisos por plazas dimensiones al que apuntará el puntero `park`.
- **`void destruir_parking(parking *park)`**: Destruye el parking argumento creado previamente por `crear_parking()`.
- **`char buscar_plaza(parking *park, unsigned int *returns_array, unsigned int matricula, unsigned int tamano)`**: Función que itera los pisos del parking y sus plazas en busca de sitio en el que aparcar el vehículo con matrícula y tamaño argumentos. Si se encuentra, aparcará el vehículo, escribirá en `returns_array` el piso y plaza en el que fue aparcado y devolverá 1. Si no lo encuentra, sencillamente devolverá 0.
- **`void vaciar_plaza(parking *park, unsigned int tamano, unsigned int piso, unsigned int primera_plaza)`**: Vacía las plazas ocupadas por el vehículo aparcado en el piso y plaza argumentos.
- **`void queue_vehiculo(Queue *cola, unsigned int tamano, unsigned int matricula)`**: Introduce un vehículo en la cola de vehículos en espera de una plaza en la que aparcarlo.
- **`void imprimir_parking(const parking *park)`**: Imprime una representación del parking.
- **`void imprimir_cola(const Queue *cola)`**: Imprime la cola de vehículos en espera.

Versiones anteriores

El fichero `./parking.c` apenas ha sufrido modificaciones notables. La más destacable es la que afecta a `vaciar_plaza()`. Antiguamente, la función recibía el tamaño del vehículo y su matrícula. Entonces iteraba el parking del mismo modo que `buscar_plaza()` y liberaba las plazas ocupadas.

Ejemplo de ejecución

```
El parking ha recibido un mensaje. Argumentos:
    Operacion: Aparcar
    Tamano del vehiculo: 1
    Matricula del vehiculo: 10
Se va a almacenar el vehiculo en la cola
Parking actualizado:
{Planta 0}      [5] [5] [1] [1] [6]
{Planta 1}      [2] [2] [3] [3] [7]
Cola: { 8 9 4 10 }
```

```
El parking ha recibido un mensaje. Argumentos:
    Operacion: Salir
    Tamano del vehiculo: 2
    Piso en el que el vehiculo esta aparcado: 1
    Primera plaza que ocupa: 2
Se va a comprobar si el vehiculo 8 de tamano 1 tiene sitio en el parking
{P0}: 0 plazas libres de 5
{P1}: 2 plazas libres de 5
Se encontro en el piso 1. Primera plaza: 2
Se va a comprobar si el vehiculo 9 de tamano 1 tiene sitio en el parking
{P0}: 0 plazas libres de 5
{P1}: 1 plazas libres de 5
Se encontro en el piso 1. Primera plaza: 3
Se va a comprobar si el vehiculo 4 de tamano 2 tiene sitio en el parking
{P0}: 0 plazas libres de 5
{P1}: 0 plazas libres de 5
No se encontro
Parking actualizado:
{Planta 0}      [5] [5] [1] [1] [6]
{Planta 1}      [2] [2] [8] [9] [7]
Cola: { 4 10 }
```

Problemas encontrados durante la realización de la práctica

- Deducir cómo convertir al proceso parking en el maestro. Cómo hacerlo se indica en la introducción de la práctica.
- Descubrir cómo lanzar más procesos que núcleos en el sistema. En un principio, se encontró el argumento de mpirun --map-by socket:OVERSUBSCRIBE. Sí permitía exceder el número de procesos permitidos, pero sólo llegaban a ejecutarse sobre la mitad de los procesos declarados y unos pocos más.
- Averiguar cómo usar MPI_Send() y MPI_Recv(). La documentación oficial no incluía mucha información acerca de los argumentos que requería la función. Por ejemplo, el argumento *int source* en MPI_Recv() debe recibir el identificador del proceso remitente del mensaje. En ningún lugar de la documentación (https://www.mpich.org/static/docs/latest/www3/MPI_Recv.html) se indica que el argumento puede tomar el valor especial MPI_ANY_SOURCE para que se puedan recibir mensajes de cualquier proceso.