Ampliación Sistemas Operativos

Práctica 2: Parking MPI

Alumno: Santiago Ramos Gómez

Titulación: Ingeniería de computadores

Universidad: Rey Juan Carlos

Campus: Móstoles

Curso: 2021-2022

Introducción

La segunda práctica consistía en trabajar en base al ecosistema MPI, debiendo de hacer una simulación de un Parking.

Mi práctica no contiene la ampliación de varias plantas aun sabiendo ahora que lo escribo que debe ser poco más que añadir un vector extra y anidar algún bucle.

En cualquier caso, la práctica exige generar un parking en el cual hay varias plazas y donde dos tipos de vehículos, coches y camiones intentan entrar de seguido. Los camiones ocupan dos plazas, mientras que los coches únicamente una.

La estructura y funcionamiento de la práctica será idéntica a la anterior, por lo que se continuará poniendo imágenes de código y se comentará sobre él. En esta segunda práctica se han puesto más de comentarios dentro de la práctica, pero aun así se comentará todo independientemente de la información que den.

Parking.c

El archivo Parking.c es el maestro en este trabajo, y los vehículos son los esclavos.

```c
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <mpi.h>
int main(int argc, char**argv){
    int plazas = atoi(argv[1]);
    int num_plazas = plazas;
    int parking[plazas];
    MPI_Status estado;
    int bloquear = 0;

    for(int i = 0; i<plazas;i++)parking[i]=0;

    printf("Plazas totales del PARKING: %d\n", plazas);
    printf("Parking:");
    for(int i=0; i<plazas; i++){
        printf(" [%d]", parking[i]);
    }
    printf("\n");

    //LO QUE DEBE RECIBIR ES: el id del proceso, si se pira o se queda, si es un coche o un camion

    int datos[3];


    MPI_Init(&argc, &argv);

    MPI_Barrier(MPI_COMM_WORLD); // se bloquean todos los procesos
    while(1){
        //int MPI_Recv(void *buf, int count, MPI_Datatype datatype,int source, int tag, MPI_Comm comm, MPI_Status *status)
        MPI_Recv(
            datos, //el vector que recibe los datos y los almacena
            3, //cuantos elementos se van a recibir
            MPI_INT, //Tipo de dato que se recibe
            MPI_ANY_SOURCE, //pid del proceso
            0, //una etaiqueta (tag), se puede dejar como MPI_ANY_TAG
            MPI_COMM_WORLD, //comunicador por el que se recibe, como una pipe
            &estado); //información sobre el estado

            // id proceso = datos[0];
            // accion = datos[1]; //0 salir, 1 entrar
            // vehiculo = datos[2]; //0 coche, 1 camión


            //printf("----------Soy el ID %d\n", datos[0]);
```

La primera parte del archivo parking incluye las correspondientes librerías e inicia la función main.

Se declaran unas cuantas variables esenciales:

- **Plazas:** guarda el número de plazas (lo pasan por parámetro)
- **Num_plazas:** un contador para tener un buen seguimiento de las disponibles.
- **Parking[plazas]:** Array de tamaño plazas el cual simula los sitios del parking.
- **MPI_Status estado:** necesario para las funciones de **MPI_Recv.**

- **Bloquear:** Esta es con diferencia la variable que más me ha desesperado y con mucha diferencia. Sin esta variable controlando el acceso a los próximos "if's", un proceso de una ID cualquiera accedía continuadamente a estos y ocupaba todas las plazas con su ID, por lo que me vi forzado a poner esta variable de control, ya que surgía algo tal que:

  Proceso 1 aparca en 0.

  Proceso 1 aparca en 1.

  Proceso 1 aparca en 2.

  Proceso 1 aparca en 3.

  Proceso 1 aparca en 4.

  Proceso 1 aparca en 5.

  Después se despertaba, liberando todo y a continuación lo mismo con el proceso de ID 2. Modificar las posiciones de los Barrier, cambiar estructuras lógicas o cualquier otra cosa me resultaba absurdo, pero aun así lo intenté. Ya pensando que tal vez no estaba bien configurado algo de mi procesador, inicie el archivo de prueba y funcionaba correctamente, por lo que como idea feliz surgió hacer esto, aun siendo ciertamente cutre me sirvió y ahora todos los procesos con distinto ID trabajan verdaderamente a la vez.

A continuación, se ponen todas las plazas del array a 0 y se imprime un mensaje visual sobre el estado del parking.

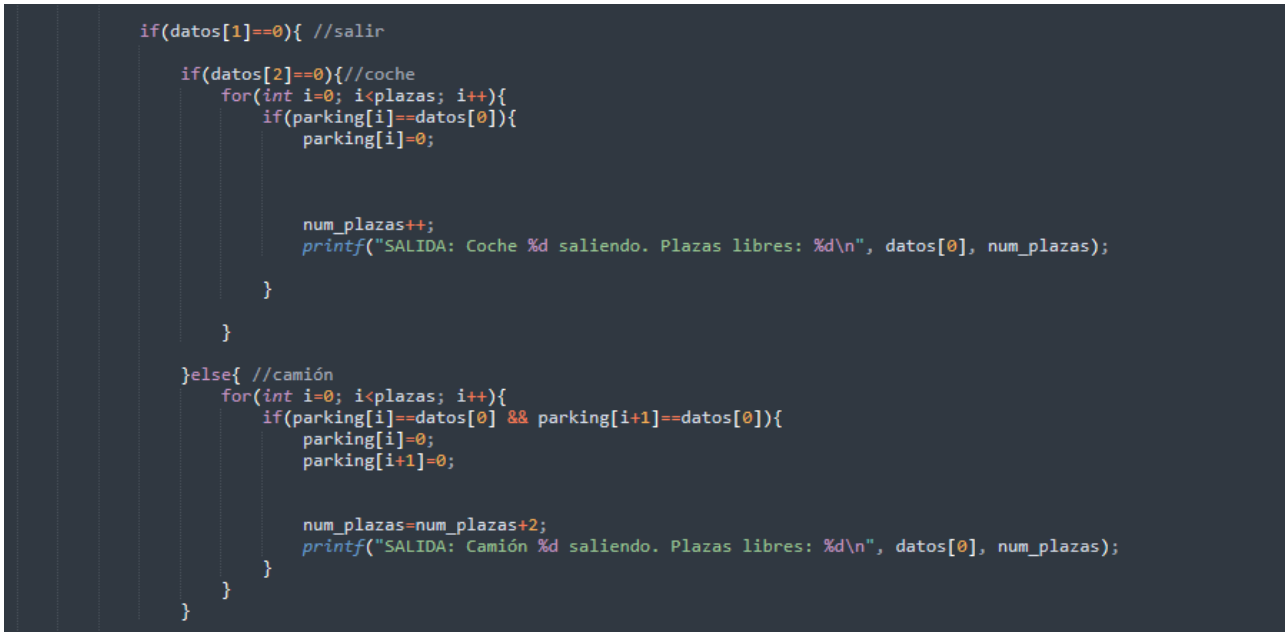
Se declara el array **datos [3],** esta variable es la más importante de todas, ya almacena:

- **Datos [0]:** Id del proceso con el que se trabaja.
- **Datos [1]:** La acción del proceso, si libera (0) u ocupa (1) una plaza.
- **Datos [2]:** El tipo de vehículo, coche (0) o camión (1).

Se inicializa el entorno MPI y se realiza **MPI_Barrier** para esperar a todos los procesos, parecido a un semáforo.

Ahora se genera un bucle infinito para que la ejecución del programa sea así y aparece una función previamente comentada, **MPI_Recv**, la cual necesita el vector que va a recibir los datos, la cantidad de datos que va a recibir, el tipo de dato que se recibe, el PID del proceso, una Tag que no se necesita, por lo que con poner 0 es suficiente, el comunicador por el cual se realizan los intercambios de mensaje, similar a una pipe y por último información sobre el estado.

Esta segunda parte explica lo que ocurre si un proceso quiere salir del parking.



```
if(datos[1]==0){ //salir

    if(datos[2]==0){//coche
        for(int i=0; i<plazas; i++){
            if(parking[i]==datos[0]){
                parking[i]=0;



                num_plazas++;
                printf("SALIDA: Coche %d saliendo. Plazas libres: %d\n", datos[0], num_plazas);

            }

        }

    }else{ //camión
        for(int i=0; i<plazas; i++){
            if(parking[i]==datos[0] && parking[i+1]==datos[0]){
                parking[i]=0;
                parking[i+1]=0;


                num_plazas=num_plazas+2;
                printf("SALIDA: Camión %d saliendo. Plazas libres: %d\n", datos[0], num_plazas);
            }
        }
    }
```

En primer lugar, se comprueba si se desea salir del parking a partir de **datos [1]**, que guarda esta acción. SI es así se debe comprobar el tipo de vehículo, si es un coche se comprueba la posición en la que está situado y se pone el valor a 0, aumentando el contador de variables disponibles y un mensaje informativo. En caso de ser un camión se deben de comprobar dos plazas contiguas, liberando ambas con un valor 0 y aumentando en este caso en 2 unidades el contador de plazas libres y añadiendo el mensaje informativo.

Tercera parte donde se explica lo que ocurre si un proceso quiere entrar al parking.

```
            }else{//entrar
                if(num_plazas==0){
                    printf("PARKING: No hay plazas disponibles\n");
                    int error = -1;
                    MPI_Send(&error, 1, MPI_INT, datos[0], 0, MPI_COMM_WORLD);

                }else{

                    if(datos[2]==0){//coche
                        for(int i=0; i<plazas; i++){
                            if(parking[i]==0 && bloquear==0){
                                parking[i]=datos[0];

                                bloquear = 1;

                                num_plazas--;
                                printf("ENTRADA: Coche %d aparca en  %d. Plazas libres: %d\n", datos[0], i, num_plazas);
                                MPI_Send(&i, //lo que se va a enviar
                                1, //cuantos elementos se van a enviar
                                MPI_INT, //tipo de dato que se envía
                                datos[0], //proceso destino
                                0, //etiqueta, sin mas
                                MPI_COMM_WORLD); // comunicador por el que se manda, como una pipe
                            }

                        }
                    }else{ //camión
                        for(int i=0; i<plazas; i++){
                            if(parking[i]==0 && parking[i+1]==0 &&bloquear==0){
                                parking[i]=datos[0];
                                parking[i+1]=datos[0];


                                bloquear = 1;

                                num_plazas=num_plazas-2;
                                printf("ENTRADA: Camión %d aparca en %d y %d. Plazas libres: %d\n", datos[0], i,i+1, num_plazas);
                                MPI_Send(&i, 1, MPI_INT, datos[0], 0, MPI_COMM_WORLD);

                            }

                        }
                    }
                }
            }
            bloquear = 0;
            sleep(1);
            printf("Parking:");
            for(int i=0; i<plazas; i++){
                printf(" [%d]", parking[i]);
            }
            printf("\n");
        }
        MPI_Finalize();
    }
```

Para comenzar se debe comprobar si quedan plazas libres, en caso de no ser así se muestra un mensaje informativo y se declara una variable con valor “-1” ya que la función **MPI_Send** debe recibir la dirección de memoria de lo que se quiere enviar, el número de elementos a enviar, tipo de dato a enviar, el ID del proceso destino, una etiqueta y el comunicador por el que se manda.

Por otra parte, si hay plazas libres se vuelve a comprobar si el tipo de vehículo.

Si es un coche se comprueba que plaza tiene un valor 0 y si la variable bloquear está 0 (pudiendo aparcar), si todo está correcto se guarda el id del proceso en la plaza y la variable bloquear pasa a ser 1 para que no pueda ocupar todas.

Se resta una unidad al contador de plazas y se imprime un mensaje informativo, además de mandar un mensaje a partir de **MPI_Send** en esta vez se manda el ID del proceso en vez de "-1", que representaba error y de esta manera indica que todo ha salido correctamente. En caso de ser un camión es casi todo idéntico, salvo que esta vez se debe comprobar si la plaza libre y la plaza contigua a esta valen ambas 0, el contador de decrementa en 2.

Una vez se salen de todas las anidaciones se resetea la variable bloquear para que la próxima vez pueda entrar al parking de nuevo, se duerme un segundo para que no se impriman mensajes tan seguidamente de manera que no se pueda seguir el flujo de información, por último, se manda un mensaje informativo sobre el estado del parking.

Coche.c y Camion.c

Archivos que actúan como esclavos en el entorno MPI.

Debido a que ambos archivos son IDÉNTICOS y SOLO varía un dato cuando se manda un mensaje en el cual se indica si es un coche o un camión, solo se va a explicar el archivo coche.c, indicando dónde varía en caso de ser un camión.

```c
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <mpi.h>
int main(int argc, char**argv){

int id_proceso;
int datos[3];
int plaza=-1;
MPI_Status estado;


MPI_Init(&argc, &argv);

MPI_Comm_rank(MPI_COMM_WORLD,&id_proceso);


//Los siguientes datos deben estar correctamente cohesionados con los definidos en parking.c

datos[0] = id_proceso; //id del proceso
datos[1] = 1; //se pone en 0' en un estado inicial porque en parking se ha definido que 0 es para salir y 1 para entrar
datos[2] = 0; // 0 porque es un coche, en el fichero del camion esto será un 1


MPI_Barrier(MPI_COMM_WORLD);

while(1){

    sleep(2);
    while(plaza == -1){
    MPI_Send(datos, 3, MPI_INT, 0, 0, MPI_COMM_WORLD);
    MPI_Recv(&plaza, 1, MPI_INT, MPI_ANY_SOURCE, 0, MPI_COMM_WORLD, &estado);

    }
    sleep(2);

    datos[1]=0; //ya que quiere salir

    MPI_Send(datos, 3, MPI_INT, 0, 0, MPI_COMM_WORLD);

    datos[1]=1; //para entrar
    plaza = -1; //entrar al bucle de nuevo
}

MPI_Barrier(MPI_COMM_WORLD);

MPI_Finalize();

}
```

Una vez más lo primero que se incluyen son las librerías necesarias y se iniciar la función main. A continuación, se declaran las siguientes variables:

- **Id_proceso:** guarda el id del proceso.
- **Datos [3]:** array para guardar datos (id, acción y tipo vehículo).
- **Plaza:** variable que controla si un proceso ha conseguido aparcar o continua en espera. (“-1” continua en espera, otro número el proceso ha conseguido aparcar).
- **MPI_Status estado:** guarda información del estado actual.

Una vez todo está declarado se inicializa el entorno MPI y se asigna un ID a cada proceso en función del número total de procesos dados para este archivo (en el parámetro “–np“ en consola). Una vez todos los procesos tienen su ID correspondiente, lo guardan el array datos:

- **Datos [0]**: guarda el id recién tomado.
- **Datos [1]:** debe ser 0, ya que en parking.c se definió que 0 indica aparcar y 1 liberar plaza, por lo que para comenzar el programa se debe aparcar primero.
- **Datos [2]:** Indica el tipo de vehículo, en archivo camion.c se pondría un 1 en vez de un 0.

A continuación, se realiza una barrera para esperar a todos los procesos y ponerlos en ejecución a la vez y se inicia un bucle infinito para estar en constante funcionamiento.

Al principio del bucle se duerme 2 segundos (lo indica el enunciado), y entra en otro bucle while, del cual se saldrá una vez se haya aparcado el vehículo y, por ende, haya cambiado el valor de la variable plaza.

MPI_Send manda un mensaje igual que antes con los datos, numero de datos, etc.

MPI_Recv obtiene la respuesta de parking.c sobre el estado de las plazas, mandando un "-1" en “&plaza" en caso de no haber o el número del proceso en otro caso.

Finalmente se vuelve a dormir 2 segundos si consigue aparcar (lo indica el enunciado), cambia el valor de datos [1] a 0, indicando que el vehículo quiere salir y se manda un mensaje a parking.c a través de **MPI_Send**. Una vez ha liberado la plaza, datos [1] vuelve a ser 1 para que continue el flujo del programa y plaza vuelve a ser "-1" por la misma razón, de manera que pueda volver a entrar al bucle.

Adicionalmente se ha creado un fichero "hostfile.config" en el cual hay definidos un numero de procesos totales con los que se quiere trabajar, ya que se deben ejecutar los 3 archivos a la vez.

Conclusión

Esta segunda práctica ha sido muchísimo más sencilla que la anterior y menos mal. Lo que más me costó fue establecer una estructura inicial, con los procesos MPI el init, los datos de SEND y RECV, etc. Por lo demás la lógica general del problema es realmente sencilla y no había que pensar mucho a excepción de si es un 0 o un 1.

En algún caso he pensado que la práctica quería simular un avión, porque cada vez que ejecutaba el programa y escuchaba a mi PC solo me faltaba que el comandante empezara a hablar sobre la hora de llegada del vuelo.

Adicionalmente, hay 2 links que me han ayudado muchísimo, sobre todo el primero, el cual es una muy buena guía de MPI en español y daban ya estructuras iniciales dadas y explicaban todas las funciones junto con que quiere decir cada parámetro, así como un ejemplo, muy recomendada.

https://lsi2.ugr.es/jmantas/ppr/ayuda/mpi_ayuda.php?ayuda=mpi_const_types

https://www.open-mpi.org/

Por último, quiero dejar presente que este código no ha sido cedido a NINGÚN alumno y no se ha trabajado con NINGÚN otro alumno, en caso de cualquier similitud o inclusión de mi nombre en otro trabajo se me debe hacer sabe para la correspondiente defensa.