



PRÀCTICA 2

FONAMENTS DE SISTEMES OPERATIUS



1 DE ABRIL DE 2024

ALFONSO SANCHEZ FERRER
EDUARD VERICAT BATALLA

ÍNDICE

1.	FASE 1.....	3
1.1.	Estudio de las funciones y variables más importantes del programa.	3
1.2.	Elaboración de los hilos de ejecución.	3
1.3.	Explicación código pthreads.	6
1.3.1.	Especificación de los hilos de ejecución.	6
1.3.2.	Creación de los hilos de ejecución basicos.	6
1.3.3.	Espera de los hilos de ejecución.....	7
1.4.	Esquema de trabajo realizado (base).	7
1.5.	Hilos de ejecución múltiples.	8
1.5.1.	Reestructuración del código.	8
1.5.2.	Creación de los hilos por cada paleta del ordenador.	10
1.6.	Temporizador y movimientos hechos por las paletas.	11
1.7.	Juegos de pruebas.	12
2.	FASE 2.....	14
2.1.	Sincronización de los procesos mediante semáforos.	14
2.1.1.	Modificaciones de la función pelota.	15
2.1.2.	Modificaciones paletas del usuario.	16
2.1.3.	Modificaciones paletas del ordenador.	17
2.1.4.	Modificaciones función mostrar información.....	17
2.1.5.	Pruebas de funcionamiento.	18
2.2.	Contador de movimientos de las paletas del ordenador.....	18
2.3.	Pausa de los elementos del juego.	19
2.4.	Muestra de información de juego.....	20
2.4.1.	Movimientos hechos.	21
2.4.2.	Movimientos restantes.	21
2.4.3.	Tiempo.	21
2.5.	Retardo por defecto de 100ms.	21
2.6.	Movimientos infinitos.	22
2.7.	Juegos de pruebas.	23
3.	FASE 3.....	24
3.0.	¿Qué debemos conseguir?	24
3.1.	Modificación y creación de zonas de memoria de memoria compartida.	24
3.2.	Asignación de las zonas de memoria compartida.	25
3.3.	Creación de los procesos de las paletas del ordenador.	25

3.3.1. Paso de la función de las paletas del ordenador al archivo pal_ord3.	26
3.3.2. Creación de los procesos de las paletas del ordenador.	26
3.4. Actualización de la pantalla de juego.....	27
3.5. Eliminación de los procesos.	27
3.6. Prueba de ejecución.	28
3.7. Problemas encontrados.	28
4. FASE 4: Sincronización de procesos, threads y comunicación entre procesos.	29
4.1. Elaboracion de los buzones de comunicación.	29
4.1.1. Creación de los buzones.	29
4.1.2. Uso de los buzones tanto para el envío como para recibir los mensajes.	30
4.1.3. Eliminación de los buzones.	31
4.2. Transmisión de movimientos entre las paletas usando buzones.....	31
4.2.1. Modificación de la función pelota y envío de información a los buzones de las paletas.	31
4.2.2. Trato de los datos recibidos.	32
4.2.3. Función a partir de los datos.	33
4.3. Juegos de pruebas.	34
5. Conclusiones.....	38

1. FASE 1

En esta fase lo que deberemos es entender primero el código para poder reestructurarlo y que de esta forma funcione mediante hilos de ejecución de hilos no sincronizados. La sincronización la haremos en la siguiente fase. Para ello deberemos entender las funciones esenciales del programa y sus variables.

1.1. Estudio de las funciones y variables más importantes del programa.

Estudio de las funciones esenciales del programa:

- **Moure_Pilota:** Esta función genera el movimiento de la pelota y devuelve un valor que puede ser alguno de los siguientes:
 - -1: Significa que la pelota no ha entrado en ninguna portería.
 - 0: Significa que la pelota ha entrado en la portería del usuario.
 - 1: Significa que la pelota ha entrado en la portería del ordenador.
- **Moure_Paleta_Usuari:** Esta función genera el movimiento de la paleta del usuario a partir de las teclas que pulse. Si pulsa espai, s'atura els elements, si pulsa return, s'acaba el joc.
- **Moure_Paleta_Ordinador:** Esta función al igual que la del usuario, genera el movimiento de la paleta del ordenador, generando un movimiento en bucle, es decir, hacia arriba hasta llegar al límite indicado y hacia abajo hasta el límite indicado.

Estudio de las variables más importantes del programa:

- **Retard:** Esta función nos permite asignar un retardo. Principalmente se usa mediante la función de win_retard. Que genera un retardo para que los elementos se muevan cada n ms.
- **Moviments:** Es el número máximo de movimientos de paletas para terminar el juego. Cuando este número se llega a -1 significa que se ha acabado el juego porque hemos hecho todos los movimientos indicados.
- **Tec:** Esta variable indica la tecla que pulsa el usuario a la hora de mover la paleta, aunque también nos sirve para poder leer teclas como el espacio que crea una pausa del juego o el return que termina el juego.
- **Cont:** Esta variable indica si la pelota se ha salido o no del juego, básicamente es el estado de la pelota. Si es -1 es que no se ha salido del juego, si es 0 es que la pelota ha entrado en la portería del usuario y si es 1 es que ha entrado en la pelota del ordenador. Dando como resultado quien ha ganado el juego.

1.2. Elaboración de los hilos de ejecución.

Una vez hemos hecho un análisis de todas las variables y funciones que intervienen en el programa hemos empezado a realizar los cambios en el programa para que funcionen en múltiples hilos de ejecución.

Iniciamos entendiendo el bucle principal del programa.

```
if (inicialitza_joc() != 0) /* intenta crear el taulell de joc */
    exit(4); /* aborta si hi ha algun problema amb taulell */

//do          /****** bucle principal del joc *****/
/*
{ tec = win_gettec();
if (tec != 0) mou_paleta_usuari(tec);
mou_paleta_ordinador();
cont = moure_pilota();
win_retard(retard);
} while ((tec != TEC_RETURN) && (cont == -1) && ((moviments > 0) || moviments == -1));*/
```

Este bucle registra las teclas, mueve la paleta del usuario, la paleta del ordenador, la pelota, y añade un retardo. Además, se hará siempre y cuando queden movimientos o que la pelota haya salido del tablero o que el usuario presione la tecla return.

A partir de esta información podemos elaborar lo siguiente:

Primero indicaremos las variables que queremos que sean globales, de esta forma los hilos podrán ir modificando estas variables.

```
1  int retard; /* valor del retard de moviment, en mil.lisegons */
2  int moviments; /* numero max de moviments paletes per acabar el joc */
3
4  int tec; // Tecla que pulsa el usuari
5  int cont = -1; // Contador actual.
6
```

Una vez hecho esto, procederemos con hacer las funciones y sus modificaciones para que puedan actuar como hilos independientes del main.

Nota: Hemos decidido simplificar lo que debería tener el código y las partes más importantes porque la documentación podría ser muy extensa.

Función moure_pilota:

```
void * moure_pilota(void * cap) {
    while ((tec != TEC_RETURN) && (cont == -1) && ((moviments > 0) || moviments == -1))
    {
        win_retard(retard);

        // CODI PILOTA

        cont = result
    }
}
```

En este código haremos el mismo bucle que teníamos en el main. Lo que tenemos ahora es que todos los hilos se acaben cuando uno de los eventos que esperamos ocurran. Para esto simplemente copiamos el bucle while. Añadimos el retardo especificado y copiamos el código de la función correspondiente. Quedando la estructura general como la imagen superior.

Función moure_paleta_usuari:

```
12 void * mou_paleta_usuari(void * cap) {
13     while ((tec != TEC_RETURN) && (cont == -1) && ((moviments > 0) || moviments == -1))
14     {
15         tec = win_gettec();
16         if(tec != 0) {
17             // CODI PALETA_USUARI
18         }
19         win_retard(retard);
20     }
21 }
```

En este caso tenemos una estructura bastante similar, con un bucle while que romperá el hilo una vez se alcance alguna condición. Como notación a añadir tenemos que añadir que solo se ejecute el código de paleta si la tecla es diferente de 0. Es decir, que tenemos alguna tecla pulsada. Por último, a final del bucle añadimos un retardo.

Codi mou_paleta_ordinador:

```
void *mou_paleta_ordinador(void *index) {
    while ((tec != TEC_RETURN) && (cont == -1) && ((moviments > 0) || moviments == -1))
    {
        // CODI PALETA_ORDINADOR
        win_retard(retard);
    }
}
```

En este caso tenemos un código muy similar a la paleta de usuari. Solo que ahora solamente deberemos ejecutar el código de la paleta del ordenador de forma constante con el retardo especificado.

Una vez hemos elaborado las nuevas funciones y las variables globales procederemos a implementar los hilos de ejecución. Para ello añadimos la librería pthreads a nuestro archivo.c.

```
#include <unistd.h> // include de unistd
#include <pthread.h> // libreria pthread.
```

Una vez hecho esto, podemos modificar el main para que genere los hilos de ejecución. Recordemos ahora cuando se verifica que el juego se ha podido crear correctamente no se hace un bucle, si no, creamos los hilos, y esperamos a que terminen su ejecución, para ello añadimos el código que explicamos en el punto 1.3.

1.3. Explicación código pthreads.

En este punto abordaremos la explicación de la elaboración de la lógica de creación y eliminación de los hilos de ejecución. Aquí primero nos centraremos en elaborar los hilos base, es decir:

- Paleta del ordenador.
- Paleta del usuario.
- Pelota.

Luego programaremos para que la paleta del ordenador se generen varios hilos por paletas del ordenador se nos introduzcan en el archivo que pueden ser hasta 9.

Visión general:

Una vez hemos hecho y modificado todas las funciones para que trabajen de forma independiente mediante la librería pthreads creamos los hilos de ejecución de las 3 funciones.

```
pthread_t thread_pilota, thread_paleta_usuari, thread_paleta_ordinador;

// Crear els fils
pthread_create(&thread_pilota, NULL, moure_pilota, NULL);
pthread_create(&thread_paleta_usuari, NULL, mou_paleta_usuari, NULL);
pthread_create(&thread_paleta_ordinador, NULL, mou_paleta_ordinador, NULL);

// Esperar que els fils acabin
pthread_join(thread_paleta_usuari, NULL);
pthread_join(thread_paleta_ordinador, NULL);
pthread_join(thread_pilota, NULL);
win_fi();
```

El código de la imagen tiene 3 partes bien definidas:

- Especificación de los pthread a ejecutar (nombramiento de los hilos).
- Creación de los hilos de ejecución.
- Espera al fin de ejecución de los hilos.

Ahora profundizaremos en estas 3 partes del código:

1.3.1. Especificación de los hilos de ejecución.

```
pthread_t thread_pilota, thread_paleta_usuari, thread_paleta_ordinador;
```

Primero deberemos especificar las variables de tipo pthread_t. Este tipo de variables son un tipo que pueden contener referencias a hilos de la biblioteca de hilos POSIX. Esto es necesario ya que luego para la creación de estos hilos necesitaremos tener un tipo de variable que permita tener referencias a otros hilos.

1.3.2. Creación de los hilos de ejecución basicos.

```
pthread_create(&thread_pilota, NULL, moure_pilota, NULL);
pthread_create(&thread_paleta_usuari, NULL, mou_paleta_usuari, NULL);
pthread_create(&thread_paleta_ordinador, NULL, mou_paleta_ordinador, NULL);
```

En este fragmento de código es donde específicamente hacemos la creación de los hilos de ejecución. Para ello hacemos uso de una función de la librería pthread que es `pthread_create`. Esta función espera los siguientes parámetros:

- 1) Un puntero a una variable de tipo `pthread_t`. Esta variable se usará para guardar el identificador del hilo creado.
- 2) Un puntero a una estructura de tipo `pthread_attr_t` que especifica los atributos del hilo. Actualmente lo hemos puesto a `NULL` veremos más adelante si nos es necesario modificar esta variable en las fases posteriores.
- 3) Un puntero a la función que se ejecutará en el nuevo hilo.
- 4) Un puntero a un argumento que se pasará a la función del hilo.

Ahora si vemos el código propuesto para esta fase de la práctica tenemos que:

- Tenemos en todas las funciones la variable `pthread_t` que hemos creado anteriormente.
- Tenemos en todas las funciones el puntero a la función que hemos creado anteriormente.

Veremos en fases siguientes si debemos modificar este tipo de creación de hilos o si tenemos que usar las variables 2 y 4 de la función `pthread_create`.

1.3.3. Espera de los hilos de ejecución.

```
// Esperar que els fils acabin
```

```
pthread_join(thread_paleta_usuario, NULL);  
pthread_join(thread_paleta_ordinador, NULL);  
pthread_join(thread_pilota, NULL);
```

Una vez hemos creado los hilos de ejecución procederemos ahora a esperar a que estos terminen. Para eso hemos hecho uso de la función `pthread_join`. Esta función lo que hace es bloquear el hilo de llamada hasta que el hilo especificado termine su ejecución. Además, toma 2 argumentos:

- 1) Un identificador de hilo de tipo `pthread_t`.
- 2) Un puntero a una variable de tipo `void *` donde se almacenará el valor de retorno del hilo.

Es por ello que en la primera variable siempre indicamos el `thread_t` que queremos esperar. Como ahora no nos interesa el valor de retorno lo dejamos en `null`.

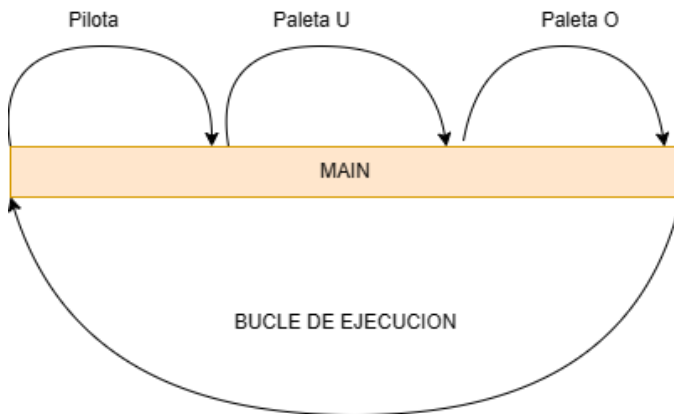
1.4. Esquema de trabajo realizado (base).

Hemos decidido añadir un esquema de ejecución del programa para poder observar que ocurre con los hilos y procesos. Este esquema lo hemos hecho, basándonos en lo observado con los procesos en Linux.

Mediante el siguiente comando podemos ver el proceso principal que es el 5859 y sus subprocesos 5860, 5861, 5862.

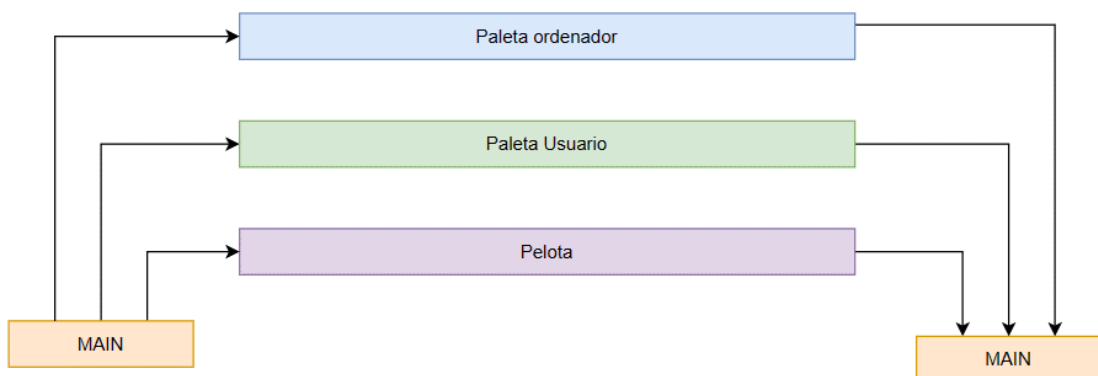
[illegible]

Esquema original:



En el programa original proporcionado podemos observar que teníamos un bucle de ejecución el cual se repetía hasta obtener una condición deseada e iba llamando a varias funciones que ejecutaban distintas características del juego.

Esquema hilos básicos:



En este esquema vemos todo lo que hemos hecho hasta este punto del documento. Donde tenemos 3 hilos que se ejecutan de forma paralela. Cuando se alcanza la condición se terminan todos los hilos.

1.5. Hilos de ejecución múltiples.

Una vez tenemos el esquema base podemos empezar a elaborar el código que controlará las múltiples paletas del ordenador. Es decir, se especifica que el juego en varios mapas tendrá múltiples paletas. Siendo un número entre 0 y 9. Debemos crear un hilo por cada paleta.

1.5.1. Restructuración del código.

Para ello primero reestructuramos las variables del código añadiendo la siguiente estructura y eliminando las variables que aparecen en ella.

```
typedef struct {
    int ipo_pf, ipo_pc; /* posicio del la paleta de l'ordinador */
    float v_pal; /* velocitat de la paleta del programa */
    float pal_ret; /* percentatge de retard de la paleta */
    float po_pf; /* pos. vertical de la paleta de l'ordinador, en valor real */
} Fila;

Fila matrizPaletas[NUMMAXPALETAS];
```

Además, hemos añadido una variable global llamada `n_pal` que es el numero de paletas del ordenador que se han leído.

Como podemos observar, hemos creado una matriz de Fila que es el struct que será entre 0 y 9. Hemos creado también una constante que es `NUMMAXPALETAS` que definirá el número máximo de paletas que puede tener el juego. Esto es así para poder modificarlo y dar flexibilidad a futuras adaptaciones.

Una vez tenemos la estructura debemos modificar la carga de datos. Actualmente solo lee el primer valor de la paleta del ordenador. Por lo que deberemos modificar para que lea siempre hasta el final del archivo, ya que tenemos en cuenta que las paletas del ordenador siempre se introducen al final del archivo.

Como hemos eliminado las variables globales, deberemos añadirlas a la función de forma local, para así que la función sea más fácil de implementar.

```
int ipo_pf, ipo_pc;          /* posicio del la paleta de l'ordinador */
float v_pal;                /* velocitat de la paleta del programa */
float pal_ret;              /* percentatge de retard de la paleta */
```

Ahora para que lea siempre hasta el final del archivo, hemos añadido el siguiente bucle.

```
while(!feof(fit)) {
    fscanf(fit, "%d %d %f %f\n", &ipo_pf, &ipo_pc, &v_pal, &pal_ret);
    if ((ipo_pf < 1) || (ipo_pf + 1 * pal_ret > n_pal - 2)) ||

exit(5);
}
matrizPaletas[n_pal].ipo_pf = ipo_pf;
matrizPaletas[n_pal].ipo_pc = ipo_pc;
matrizPaletas[n_pal].v_pal = v_pal;
matrizPaletas[n_pal].pal_ret = pal_ret;
n_pal++;
```

Como se puede observar, usamos la variable `n_pal` para ir añadiendo los elementos a la matriz. De esta forma no creamos ninguna variable nueva, así tenemos también el numero de paletas correspondiente.

Tras esto hemos modificado la función de la paleta del ordenador, para que use las variables correspondientes en su índice pasado por parámetro. Un extracto del código sería el siguiente:

```
void *mou_paleta_ordinador(void *index) {
    int f_h;
    /* char rh,rv,rd; */
    while ((tec != TEC_RETURN) && (cont == -1) && ((moviments > 0) || moviments == -1)) {
        win_retard(retard);
        f_h = matrizPaletas[(int*)index].po_pf + matrizPaletas[(int*)index].v_pal; /* posic
        if (f_h != matrizPaletas[(int*)index].ipo_pf) /* si pos. hipotetica no coincideix am
        {
            if (matrizPaletas[(int*)index].v_pal > 0.0) /* verificar moviment cap avall */
```

Donde obtenemos el valor del puntero de memoria index para poder acceder a la fila correspondiente de la matriz. Una vez hecho esto, indicamos la variable que queremos usar. Hemos hecho esto con todas las variables que maneja la matriz. Así cada hilo, tiene unas variables distintas según la paleta que le corresponda.

1.5.2. Creación de los hilos por cada paleta del ordenador.

Para crear los hilos de forma independiente. Debemos primero crear una matriz de variables de tipo pthread_t. Esto es necesario para poder almacenar los identificadores de los hilos.

Por lo tanto, añadimos el siguiente código que es el que nos permite crear esta matriz.

```
pthread_t threads_pal_ordinador[n_pal];  
//pthread_create(&threads_pal_ordinador[index
```

Tras esto mediante el siguiente bucle podemos ir creando los hilos:

```
for (int i = 0; i < n_pal; i++)  
{  
    int *arg = malloc(sizeof(*arg)); // Creem un espai de memoria per allotjar el index.  
    if (arg == NULL) { // Si no es genera, donem errada, i tanquem el joc.  
        fprintf(stderr, "No se ha pogut crear els fils de la paleta del ordinador, error!");  
        exit(EXIT_FAILURE);  
    }  
    *arg = i; // Asignem el index a la memoria creada.  
    pthread_create(&threads_pal_ordinador[i], NULL, mou_paleta_ordinador, arg); // Creem  
}
```

En este bucle seguimos estos pasos:

- 1) Asignamos una posición de memoria del tamaño de un int ya que cogemos la variable arg que definimos al otro lado de la asignación. De esta forma hacemos la reserva de memoria justa.
- 2) Comprobamos de que esta asignación de memoria haya sido exitosa. Si no ha sido así deberemos terminar el programa. La mejor forma en este caso es dar un mensaje de error y salir mediante un exit indicando un código de error.
- 3) Si se ha podido generar, asignamos en esta posición reservada de memoria el índice del for.
- 4) Creamos el thread(hilo) de la paleta correspondiente. Le pasamos su índice que es el valor correspondiente del índice del bucle for.

Importancia de reservar memoria:

Lo que conseguimos es que cada hilo tenga un espacio de memoria reservada para su índice. Si no se hiciera esto, tendríamos colisiones de acceso a memoria ya que todos intentarían acceder al mismo índice. Se generaría el error conocido como “segmentation fault” que ocurre cuando aparecen estas colisiones o cuando se intenta acceder a un puntero de memoria inexistente.

Luego al final de cada hilo, importante, liberamos la memoria de su *index que es el arg que le pasamos al crear cada hilo.

```

    else matrizPaletas[* (int*)index].po_pf += matrizPaletas[* (int*)index].v
}
free(index); // Liberem la memoria de la variable index.
return NULL;
}

```

Luego deberemos esperar a que los hilos terminen, esto podemos hacerlo mediante una iteración que espere primero por el hilo 0 de la paleta del ordenador, luego por el 1, y sucesivamente. No daría problemas por que todos los hilos tienen la misma condición de finalización, por lo que si termina el primer hilo, terminan todos.

```

for (int i = 0; i < n_pal; i++)
{
    pthread_join(threads_pal_ordinador[i], NULL); // Esperem que acaben to
}

```

1.6. Temporizador y movimientos hechos por las paletas.

Una vez hemos abordado todos los elementos del juego podemos añadir nuevos elementos como un temporizador que indica cuando tiempo lleva el juego y cuantos movimientos han hecho las paletas.

Hemos analizado el código y hemos pensado que la forma más interesante de hacer esto, es crear un hilo que se encargue de esta ejecución y que este en uso permanente hasta que se le mande una señal de apagado. Luego en la fase 2, entrará en sincronización con el resto de los hilos. En esta fase, solo haremos el hilo de ejecución.

Creamos la siguiente función:

```

void *time_moviments() {
    time_t start_time = time(NULL);
    while ((tec != TEC_RETURN) && (cont == -1) && ((moviments > 0) || moviments == -1)) {
        printf("%d\n", moviments_inicials - moviments);
        time_t current_time = time(NULL);
        time_t elapsed_time = current_time - start_time;
        int minuts = elapsed_time / 60;
        int segons = elapsed_time % 60;
        printf("Time: %d min %d sec\n", minuts, segons);
        win_retard(1000);
    }
    return NULL;
}

```

Donde calculamos el tiempo y mostramos los movimientos hechos siguiendo este esquema de ejecución de la función:

- 1) Obtenemos el tiempo actual.
- 2) Realizamos un bucle de ejecución el cual se termina cuando se cumple alguna de las condiciones finales de juego. Los siguientes bucles ocurren siempre dentro del bucle.
- 3) Mostramos los movimientos hechos que son los iniciales (una variable que guarda los movimientos máximos introducidos por parámetro) menos los movimientos

(que se restan hasta llegar a -1). Por lo tanto, el resultado de esta resta siempre será el número de movimientos que ha hecho la paleta.

- 4) Luego simplemente calculamos el tiempo que es tiempo actual – tiempo inicial.
- 5) Calculamos los segundos y minutos.
- 6) Mostramos el tiempo en pantalla.
- 7) Añadimos un retardo de 1000ms para que se ejecute todo el bucle cada 1s.

Luego simplemente creamos el hilo:

```
pthread_create(&thread_time_moviments, NULL, time_moviments, NULL);
```

Y esperamos a la finalización de su ejecución que será al final del juego:

```
pthread_join(thread_time_moviments, NULL);
```

1.7. Juegos de pruebas.

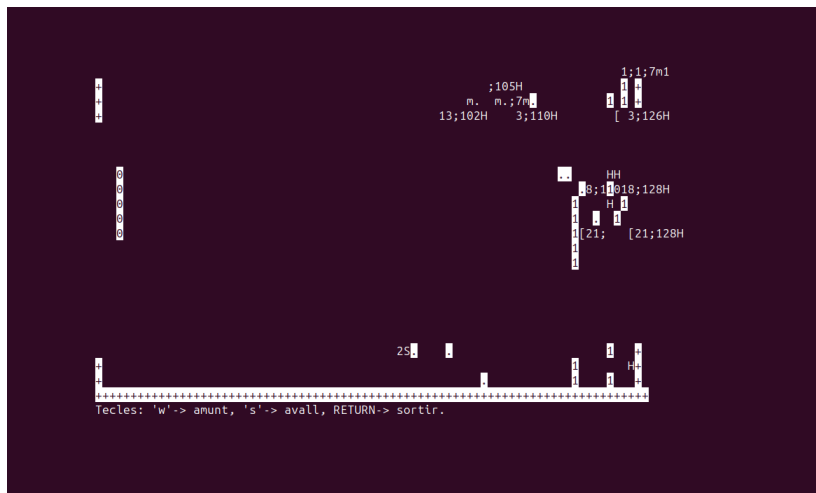
En este caso los juegos de pruebas no pueden ser muy extensos ya que en este punto el juego no es especialmente jugable. Lo que si podemos es intentar observar, que el juego cree distintas paletas del ordenador, que cree un temporizador correctamente y que muestre el numero de movimientos.

También podemos analizar el proceso y todos sus hilos para observar que se crean correctamente.

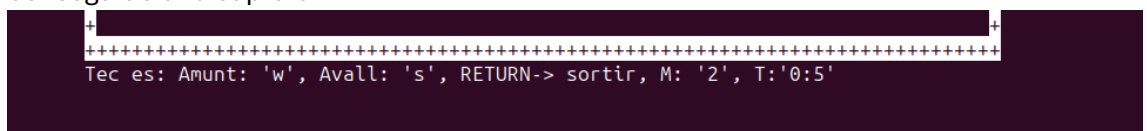
Análisis del programa:

1) Observación visual:

En este caso observamos que la pantalla no aparece bien, pues todos los procesos están intentando escribir en pantalla, pero si que observamos una pelota en movimiento, la paleta del usuario y 3 paletas de diferentes formatos en movimiento.



Si queremos también ver el tiempo es un poco mas complicado ya que al no estar sincronizado, se muestran caracteres aleatorios en pantalla. Igualmente, hemos conseguido una captura.



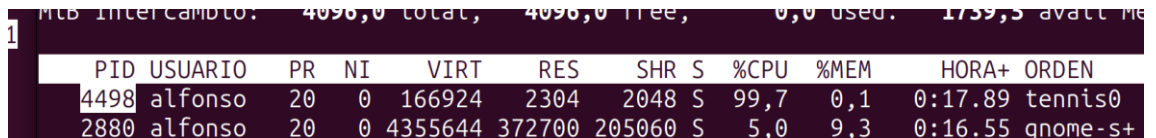
Donde M es el numero de movimientos hechos por las paletas y T es el tiempo que lleva el programa en ejecución.

2) Observación por procesos:

Para este apartado, podemos ejecutar el juego con un retardo alto, para que se quede en ejecución y que tarde en terminar por que la pelota entre en alguna portería.

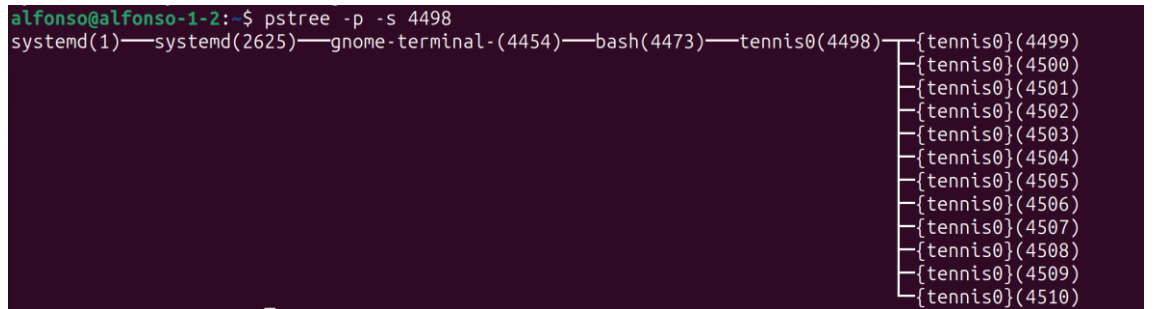
Primero buscamos mediante el comando \$top el proceso main de tennis. Que es el siguiente:

4498.



1												
MTB Intercambio: 4096,0 total, 4096,0 free, 0,0 used. 1739,5 avail mem												
	PID	USUARIO	PR	NI	VIRT	RES	SHR	S	%CPU	%MEM	HORA+	ORDEN
	4498	alfonso	20	0	166924	2304	2048	S	99,7	0,1	0:17.89	tennis0
	2880	alfonso	20	0	4355644	372700	205060	S	5,0	9,3	0:16.55	gnome-s+

Una vez identificado mediante el siguiente comando podemos ver todos los subprocesos.



```
alfonso@alfonso-1-2:~$ pstree -p -s 4498
systemd(1)─systemd(2625)─gnome-terminal-(4454)─bash(4473)─tennis0(4498)─{tennis0}(4499)
                                                    {tennis0}(4500)
                                                    {tennis0}(4501)
                                                    {tennis0}(4502)
                                                    {tennis0}(4503)
                                                    {tennis0}(4504)
                                                    {tennis0}(4505)
                                                    {tennis0}(4506)
                                                    {tennis0}(4507)
                                                    {tennis0}(4508)
                                                    {tennis0}(4509)
                                                    {tennis0}(4510)
```

Donde tenemos en este caso 12 subprocesos, que son los siguientes:

- 1) Paleta usuario.
- 2) Pelota.
- 3) 9 paletas del ordenador.
- 4) Contador de tiempo y movimientos.

3) Observaciones generales:

Si nos ceñimos al resultado actual, no podemos hacer en gran parte las pruebas oportunas debido a que no podemos determinar si todo funciona correctamente por el acceso no sincronizado de los hilos de ejecución y de el problema de caracteres aleatorios escritos en pantalla que esto supone.

2. FASE 2.

En esta fase se nos piden los siguientes requisitos a cumplir:

1. Añadir sincronización de los hilos de ejecución en las secciones críticas que son variables globales compartidas y entornos de dibujo.
2. Añadir un contador total de los movimientos de las paletas del ordenador y de las paletas del usuario.
3. Mostrar el numero de movimientos hechos por la paleta y el ordenador y un contador de tiempo de juego en pantalla.
4. Realizar un pause de todos los elementos del juego si se presiona el espacio.

A partir de estos puntos elaborados por el enunciado de la práctica podemos empezar a trabajar.

2.1. Sincronización de los procesos mediante semáforos.

Para este punto primero debemos aprender que son los semáforos y cómo funcionan para establecer una sincronización entre los distintos hilos de ejecución.

Los semáforos son unos “puntos” que nos permiten bloquear totalmente la ejecución de todos los threads cuando se va a modificar o realizar alguna acción crítica. A acción crítica podemos llamarlo:

1. Acceso a pantalla para lectura.
2. Acceso a pantalla para escritura.
3. Modificación de variables compartidas.

Estas acciones deben protegerse, en los casos 1 y 2 provocan lecturas erróneas y aparición de caracteres no deseados en la pantalla. En el caso 3 aparecen operaciones o modificaciones no controladas, provocadas por la modificación al mismo tiempo de la misma variable por dos hilos distintos. Además, si estas variables determinan el final de un bucle, puede provocar la finalización del bucle de forma no controlada.

Además de estas acciones, todos los hilos deben acceder a las variables y pantalla al tiempo correspondiente. Es decir, no puede un hilo acceder cada 10ms y otro cada 30ms, pues no estarían sincronizados. Por lo que todos los hilos deben tener el `win_retard`(retardo) en la misma posición. Por decisión de diseño hemos añadido esto siempre al inicio del bucle de finalización de juego.

Una vez conocemos podemos ir modificando el código para tener el control de los hilos de ejecución.

Creamos primero el “semáforo” que nos permitirá tener este control:

```
pthread_mutex_t sem = PTHREAD_MUTEX_INITIALIZER;
```

Una vez creado el semáforo, antes de crear los hilos deberemos inicializarlo.

```
pthread_t thread_pilota, thread_paleta_usuari, thread_time_moviments;
pthread_mutex_init(&sem, NULL); // Inicialitzem el semàfor;
// Crear els fils
pthread_create(&thread_pilota, NULL, moure_pilota, NULL);
pthread_create(&thread_paleta_usuari, NULL, mou_paleta_usuari, NULL);
pthread_create(&thread_time_moviments, NULL, time_moviments, NULL);
```

El segundo parámetro está a NULL para que obtenga los valores por defecto de la constante PTHREAD_MUTEX_INITIALIZER.

Ahora, simplemente deberemos ir modificando las funciones:

- Pelota.
- Paleta Ordenador.
- Paleta Usuario.
- Tiempo y movimientos.

Esas funciones son las que generan situaciones críticas debido a que realizan alguna acción crítica descrita anteriormente.

Recordemos que siempre al final del juego deberemos terminar el semáforo mediante la siguiente línea de código:

```
// Destruim el semàfor
pthread_mutex_destroy(&sem); // Destruim el semàfor.
```

2.1.1. Modificaciones de la función pelota.

Analizando la función que ejecuta el movimiento de la pelota obtenemos que debemos añadir los siguientes semáforos:

Verificación de los caracteres para obstáculos:

```
pthread_mutex_lock(&sem);
rv = win_quincar(f_h, ipil_pc); /* veure si hi ha algun obstacle */
pthread_mutex_unlock(&sem);

pthread_mutex_lock(&sem);
rh = win_quincar(ipil_pf, c_h); /* veure si hi ha algun obstacle */
pthread_mutex_unlock(&sem);

pthread_mutex_lock(&sem);
rd = win_quincar(f_h, c_h);
pthread_mutex_unlock(&sem);
```

Escritura de la pelota en pantalla además de algunas verificaciones de los caracteres de la pantalla:


```

pthread_mutex_lock(&sem);
if (win_quincar(f_h, c_h) == ' ') /* verificar posicio definitiva */
{
    pthread_mutex_unlock(&sem); /* si no hi ha obstacle */
    pthread_mutex_lock(&sem);
    win_escricar(ipil_pf, ipil_pc, ' ', NO_INV); /* esborra pilota */
    pthread_mutex_unlock(&sem);
    pil_pf += pil_vf; pil_pc += pil_vc;
    ipil_pf = f_h; ipil_pc = c_h; /* actualitza posicio actual */
    if ((ipil_pc > 0) && (ipil_pc <= n_col)){
        /* si no surt */
        pthread_mutex_lock(&sem);
        win_escricar(ipil_pf, ipil_pc, '.', INVERS); /* imprimeix pilota */
        pthread_mutex_unlock(&sem);
    }
    else
    {
        result = ipil_pc; /* codi de finalitzacio de partida */
    }
}
pthread_mutex_unlock(&sem);

```

Modificaci3n de variable global:

```

pthread_mutex_lock(&sem);
cont = result; // Actualitzem la informaci3n de la pilota.
pthread_mutex_unlock(&sem);

```

2.1.2. Modificaciones paletas del usuario.

Analizando la funci3n que realiza los movimientos de la paleta del usuario a~adimos los siguientes sem~foros:

Modificaci3n de variable global:

```

pthread_mutex_lock(&sem);
tec = win_gettec();
pthread_mutex_unlock(&sem);

```

Modificaci3n de caracteres en pantalla y lectura de pantalla:

```

pthread_mutex_lock(&sem);
if ((tec == TEC_AVALL) && (win_quincar(ipu_pf+1_pal,ipu_pc) == ' '))
{
    pthread_mutex_unlock(&sem);
    pthread_mutex_lock(&sem);
    win_escricar(ipu_pf,ipu_pc,' ',NO_INV); /* esborra primer bloc */
    pthread_mutex_unlock(&sem);
    ipu_pf++; /* actualitza posicio */
    pthread_mutex_lock(&sem);
    win_escricar(ipu_pf+1_pal-1,ipu_pc,'0',INVERS); /* impri. ultim bloc */
    pthread_mutex_unlock(&sem);
    if (moviments > 0) moviments--; /* he fet un moviment de la paleta */
}
pthread_mutex_unlock(&sem);
pthread_mutex_lock(&sem);
if ((tec == TEC_AMUNT) && (win_quincar(ipu_pf-1,ipu_pc) == ' '))
{
    pthread_mutex_unlock(&sem);
    pthread_mutex_lock(&sem);
    win_escricar(ipu_pf+1_pal-1,ipu_pc,' ',NO_INV); /* esborra ultim bloc */
    pthread_mutex_unlock(&sem);
    ipu_pf--; /* actualitza posicio */
    pthread_mutex_lock(&sem);
    win_escricar(ipu_pf,ipu_pc,'0',INVERS); /* imprimeix primer bloc */
    if (moviments > 0) moviments--; /* he fet un moviment de la paleta */
    pthread_mutex_unlock(&sem);
}
pthread_mutex_unlock(&sem);

```

2.1.3. Modificaciones paletas del ordenador.

Para la función de la paleta del ordenador deberemos también hacer el mismo control que las anteriores. Por ello realizamos lo siguiente:

Modificación de caracteres en pantalla y lectura en pantalla:

```
pthread_mutex_lock(&sem);
if (win_quincar(f_h+1_pal-1,matrizPaletas[(int*)index].ipo_pc) == ' ') /* si no hi ha obstacle */
{
    pthread_mutex_unlock(&sem);
    pthread_mutex_lock(&sem);
    win_escribir(matrizPaletas[(int*)index].ipo_pf,matrizPaletas[(int*)index].ipo_pc,' ',NO_INV); /* esborra primer bloc */
    pthread_mutex_unlock(&sem);
    matrizPaletas[(int*)index].po_pf += matrizPaletas[(int*)index].v_pal; matrizPaletas[(int*)index].ipo_pf = matrizPaletas[(int*)index].po_pf;
    pthread_mutex_lock(&sem);
    win_escribir(matrizPaletas[(int*)index].ipo_pf+1_pal-1,matrizPaletas[(int*)index].ipo_pc,'1'+(int*)index,INVERS); /* impr. ultim bloc */
    if (moviments > 0){
        moviments--; /* he fet un moviment de la paleta */
        matrizMovimentsPaletas[(int*)index]++; // Actualitzem el numero de moviments de la paleta.
    }
    pthread_mutex_unlock(&sem);
} else {
    /* si hi ha obstacle, canvia el sentit del moviment */
    pthread_mutex_unlock(&sem);
    matrizPaletas[(int*)index].v_pal = -matrizPaletas[(int*)index].v_pal;
}

else /* verificar moviment cap amunt */
{
    pthread_mutex_lock(&sem);
    if (win_quincar(f_h,matrizPaletas[(int*)index].ipo_pc) == ' ') /* si no hi ha obstacle */
    {
        pthread_mutex_unlock(&sem);
        pthread_mutex_lock(&sem);
        win_escribir(matrizPaletas[(int*)index].ipo_pf+1_pal-1,matrizPaletas[(int*)index].ipo_pc,' ',NO_INV); /* esbo. ultim bloc */
        pthread_mutex_unlock(&sem);
        matrizPaletas[(int*)index].po_pf += matrizPaletas[(int*)index].v_pal; matrizPaletas[(int*)index].ipo_pf = matrizPaletas[(int*)index].po_pf; /* actualitza posicio */
        pthread_mutex_lock(&sem);
        win_escribir(matrizPaletas[(int*)index].ipo_pf,matrizPaletas[(int*)index].ipo_pc,'1'+(int*)index,INVERS); /* impr. primer bloc */
        if (moviments > 0){
            moviments--; /* he fet un moviment de la paleta */
            matrizMovimentsPaletas[(int*)index]++; // Actualitzem el numero de moviments de la paleta.
        }
        pthread_mutex_unlock(&sem);
    }
} else /* si hi ha obstacle, canvia el sentit del moviment */
{
    pthread_mutex_unlock(&sem);
    matrizPaletas[(int*)index].v_pal = -matrizPaletas[(int*)index].v_pal;
}
```

En este caso, las variables que modifica son variables propias, en un índice indicado por parámetro, por lo que cada hilo tiene sus variables “locales”.

2.1.4. Modificaciones función mostrar información.

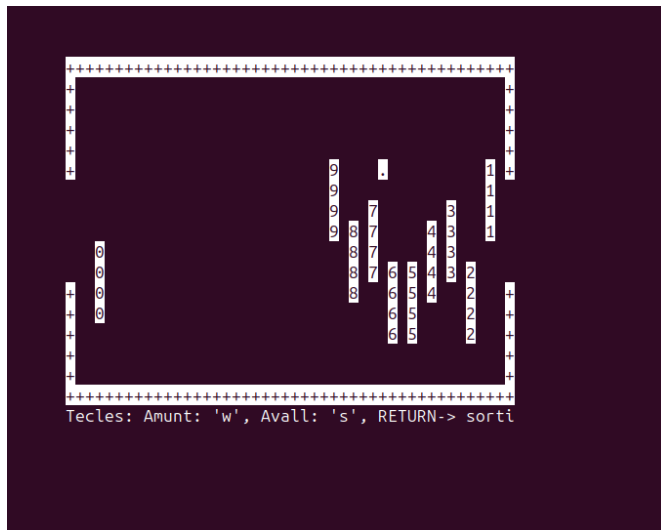
En esta fase hemos realizado primero un renombramiento de la función, luego observaremos el por qué, los cambios realizados han sido los siguientes:

Modificación en la pantalla:

```
if(moviments_infinits == 0) {
    sprintf(strin,"Tecles: Amunt: '%c\\', Avall: '%c\\', RETURN-> sortir, MF: '%d\\', MR: '%d\\' T:'%d:%d\\'";
    pthread_mutex_lock(&sem); // Bloquejem el semafor perque anem a escriure en pantalla.
    win_escrstr(strin);
    pthread_mutex_unlock(&sem); // Desbloquejem el semafor.
} else {
    sprintf(strin,"Tecles: Amunt: '%c\\', Avall: '%c\\', RETURN-> sortir, MF: '%d\\', MR: INF T:'%d:%d\\'",T);
    pthread_mutex_lock(&sem); // Bloquejem el semafor perque anem a escriure en pantalla.
    win_escrstr(strin);
    pthread_mutex_unlock(&sem); // Desbloquejem el semafor.
}
```

2.1.5. Pruebas de funcionamiento.

Como pruebas de funcionamiento, ahora deberíamos poder visualizar el juego sin ningún tipo de problema.



Además, observamos que no se generan en la mayoría de los casos caracteres aleatorios o incorrectos por desincronizaciones. El único mapa que genera algún problema es el 4. Creemos que es por tener demasiados elementos en movimiento que ponen al límite la sincronización. Seguramente quede algún error que desconocemos por pulir.

2.2. Contador de movimientos de las paletas del ordenador.

Una vez hemos realizado la sincronización de los elementos de las paletas del ordenador podemos añadir otras nuevas funcionalidades. Una de ellas es el contador de movimientos de las paletas del ordenador. Entendemos que este contador es único por cada paleta. Es decir que cada hilo tendrá su variable donde escribir este dato.

Teniendo en cuenta esto hemos añadido una matriz estática de NUMMAXPALETAS donde se guardan en cada posición un contador.

```
int matrizMovimientosPaletas[NUMMAXPALETAS]; // Matriz donde se guarda el numero de movimientos hechos.

for (size_t i = 0; i < n_pal; i++)
{
    matrizMovimientosPaletas[i] = 0; // INICIALIZAMOS TODOS LOS MOVIMIENTOS A 0.
}
```

Una vez creada la matriz y inicializada, solo modificamos la función de la paleta del ordenador para que cada hilo use una posición determinada para usarla como contador e ir incrementándolo.

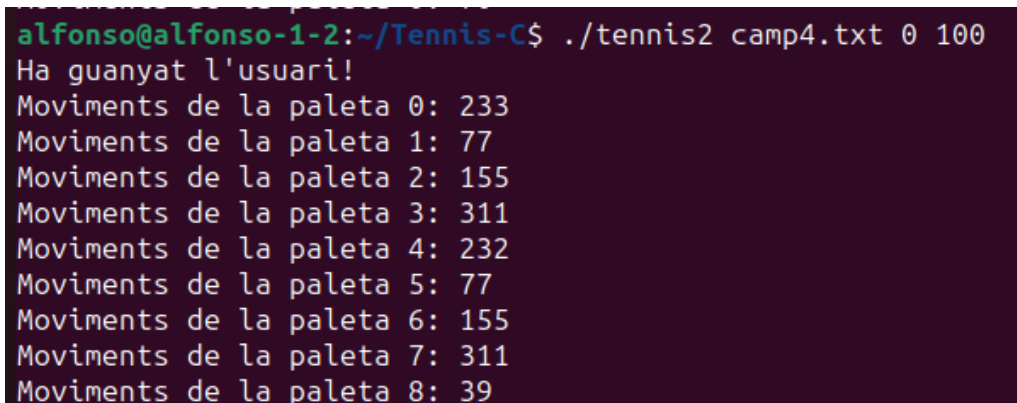
```
if (moviments > 0){
    moviments--; /* he fet un moviment de la paleta */
    matrizMovimientosPaletas[(int*)index]++; // Actualitzem el numero de moviments de la paleta.
}
```

Esta línea de código se añade tanto en el if de subida de la paleta como de la bajada de la paleta. Por lo que, de esta manera, podemos registrar correctamente todos los movimientos de las paletas.

Una vez termina el juego simplemente mostramos un mensaje con todos los contadores indicando a que paleta corresponde.

```
for (int i = 0; i < n_pal; i++) {  
    printf("Moviments de la paleta %d: %d\n", i, matrizMovimientosPaletas[i]);  
}
```

Por lo que el resultado final obtenemos lo siguiente:



```
alfonso@alfonso-1-2:~/Tennis-C$ ./tennis2 camp4.txt 0 100  
Ha guanyat l'usuari!  
Moviments de la paleta 0: 233  
Moviments de la paleta 1: 77  
Moviments de la paleta 2: 155  
Moviments de la paleta 3: 311  
Moviments de la paleta 4: 232  
Moviments de la paleta 5: 77  
Moviments de la paleta 6: 155  
Moviments de la paleta 7: 311  
Moviments de la paleta 8: 39
```

2.3. Pausa de los elementos del juego.

En esta fase, también se nos pide añadir una pausa de los elementos del juego. Para ello podríamos pensar en una variable global y usarla como bloqueador de los hilos. Esto es muy ineficiente, pues la CPU se queda esperando a que esta variable cambie. Podemos hacer uso del semáforo, podemos bloquearlo, para que se bloqueen todos los hilos. Quedando un uso de CPU mucho más eficiente.

Para ello, en el hilo de mostrar la información hemos decidido hacer una implementación de la pausa del juego.

La solución alcanzada por nosotros ha sido la siguiente:

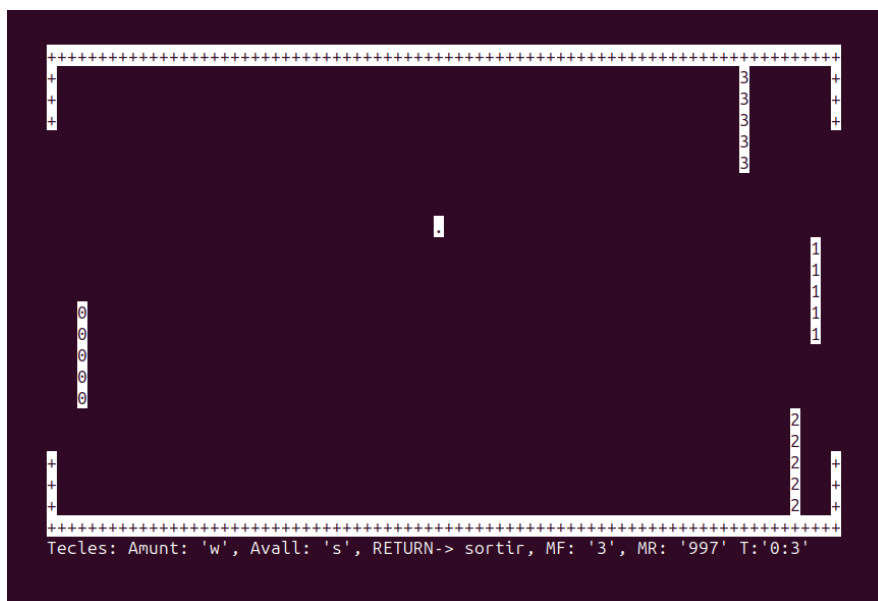
```
if (tec == TEC_ESPAI) {  
    pthread_mutex_lock(&sem);  
    tec = 0;  
    while (tec != TEC_ESPAI) {  
        win_retard(retard);  
        time_t current_time = time(NULL);  
        time_t elapsed_time = current_time - start_time;  
        int minuts = elapsed_time / 60;  
        int segons = elapsed_time % 60;  
        tec = win_gettec();  
        if (moviments_inifinit == 0) {  
            sprintf(strin, "Tecles: Amunt: '%c', Avall: '%c', RETURN-> sortir, MF: '%d\\', MR: '%d\\' T: '%d: %d\\'", TEC_AMUNT, TEC_AVALL, moviments_ini);  
            win_escrstr(strin);  
        } else {  
            sprintf(strin, "Tecles: Amunt: '%c', Avall: '%c', RETURN-> sortir, MF: '%d\\', MR: INF T: '%d: %d\\'", TEC_AMUNT, TEC_AVALL, moviments_inicia);  
            win_escrstr(strin);  
        }  
    }  
    pthread_mutex_unlock(&sem);  
}
```

Donde leemos si la tecla es el espacio. Si es así, bloqueamos todos los hilos. Y escribimos solamente la información de pantalla, calculando igualmente el tiempo. Cambiamos la tecla a 0. Y nos quedamos en un bucle “infinito” hasta que se vuelva a pulsar espacio. Lo que nos lleva a pensar en la eficiencia. Pero no es lo mismo, un

De esta forma cuando pulsamos espacio, todos los elementos del juego quedan pausados excepto el de información el cual sigue mostrando toda la información y contando el tiempo:



Un ejemplo:



Además, **esto es una reescritura mejorada del código hecho en el punto 1.6 de este documento.**

En la parte inferior siempre mostramos:

- Teclas
- Movimientos hechos.
- Movimientos restantes.
- Tiempo.

Ahora, explicaremos como añadir estos 3 últimos elementos:

2.4.1. Movimientos hechos.

Para los movimientos hechos podemos aplicar la siguiente formula:

$\text{Movimientos_hechos} = \text{movimientos_iniciales} - \text{movimientos_restantes}.$

Mediante esta fórmula podemos expresar y mostrar todos los movimientos que se han hecho. Por eso primero, guardamos en una variable cuantos movimientos tenemos al inicio del programa.

```
moviments_inicials = moviments;
```

Y para mostrarlos, aplicamos esta resta.

```
sprintf(strin, "Teclcs: Amunt: '%c'\', Avall: '%c'\', RETURN-> sortir, HF: '%d'\', MR: '%d' T: '%d:%d'\'", TEC_AMUNT, TEC_AVALL, moviments_inicials - moviments, moviments, minuts, segons);
```

2.4.2. Movimientos restantes.

Simplemente debemos mostrar la variable `moviments`, pues siempre se añade esta variable al string para poderlo mostrar:

```
sprintf(strin, "Teclcs: Amunt: '%c'\', Avall: '%c'\', RETURN-> sortir, HF: '%d'\', MR: '%d' T: '%d:%d'\'", TEC_AMUNT, TEC_AVALL, moviments_inicials - moviments, moviments, minuts, segons);
```

2.4.3. Tiempo.

Para el tiempo dentro del bucle de juego de la función `información`, podemos calcular el tiempo de la siguiente manera:

$\text{Tiempo_de_juego} = \text{tiempo_actual} - \text{tiempo_inicial}.$

De esta forma obtenemos el tiempo que lleva el juego ejecutándose.

```
time_t start_time = time(NULL);
while ((tec != TEC_RETURN) && (cont == -1) && ((moviments > 0) || moviments == -1 || moviments_infinits == 1)) {
    win_retard(retard);
    time_t current_time = time(NULL);
    time_t elapsed_time = current_time - start_time;
    int minuts = elapsed_time / 60;
    int segons = elapsed_time % 60;
```

2.5. Retardo por defecto de 100ms.

Para esto simplemente podemos verificar si está el retardo en los argumentos. Si no está, añadimos nosotros el retardo de 100ms.

```
if (n_args == 4) retard = atoi(ll_args[3]);
else retard = 100; // Si no esta el retard, posem el retard a 100.
```

2.6. Movimientos infinitos.

Para esta parte nos piden que el juego tenga movimientos infinitos si se indica que los movimientos máximos son 0. Para esto primero deberemos añadir una variable de tipo “booleana” que controlará si los movimientos son infinitos o no.

```
int moviments_infinits; /* Es una variable booleana, on indica si els moviments son infinits o no*/
```

Una vez añadida esta variable, verificamos si el argumento de entrada es 0. Si es 0, añadimos un 1 en la variable moviments_infinits la cual genera una serie de cambios que ahora explicaremos. Si es diferente de 0 procedemos a añadir un 0 a la variable moviments_infinits y se realiza la ejecución normal del juego.

Como vemos, para simular movimientos infinitos, añadimos una cantidad de movimientos muy elevada, concretamente 2^{32} movimientos. Eso sí, el cálculo de movimientos hechos se tiene que poder hacer. Por eso esta asignación.

```
moviments=atoi(ll_args[2]);
if (moviments == 0) {
    moviments = __INT_MAX__;
    moviments_inicials = __INT_MAX__;
    moviments_infinits = 1;
} else {
    moviments_inicials = moviments;
    moviments_infinits = 0;
}
```

Ahora, modificamos 2 cosas importantes:

1. **Bucle de finalización de juego.**
2. **Mensaje de información en pantalla.**

En el **bucle de finalización del juego** modificamos para que nunca termine si moviments_infinits es igual a 1.

```
while ((tec != TEC_RETURN) && (cont == -1) && ((moviments > 0) || moviments == -1 || moviments_infinits == 1)) {
```

En el mensaje de información en pantalla ahora, los movimientos restantes se muestran como INF.

```
if(moviments_infinits == 0) {
    sprintf(strin,"Tecles: Amunt: '%c', Avall: '%c', RETURN-> sortir, MF: '%d', MR: '%d' T:'%d:%d'",TE
    pthread_mutex_lock(&sem); // Bloquejem el semafor perque anem a escriure en pantalla.
    win_escristr(strin);
    pthread_mutex_unlock(&sem); // Desbloquejem el semafor.
} else {
    sprintf(strin,"Tecles: Amunt: '%c', Avall: '%c', RETURN-> sortir, MF: '%d', MR: INF T:'%d:%d'",TEC_A
    pthread_mutex_lock(&sem); // Bloquejem el semafor perque anem a escriure en pantalla.
    win_escristr(strin);
    pthread_mutex_unlock(&sem); // Desbloquejem el semafor.
}
```

Quedando el siguiente resultado:

3. FASE 3.

3.0. ¿Qué debemos conseguir?

En la F3 de la práctica se nos solicita poder convertir los hilos de las paletas del ordenador en procesos. Para ello deberemos hacer unos cambios en la estructura del programa.

1. Crear una zona de memoria compartida para las variables críticas.
2. Establecer una zona de memoria compartida para la escritura y lectura de la pantalla (zona

3.1. Modificación y creación de zonas de memoria de memoria compartida.

Procederemos a decidir que variables van a ser usadas para las zonas de memoria compartida.

```
typedef struct {
    int tec;
    int cont;
    int moviments;
    int moviments_inicials;
    int moviments_infinits;
    int retard;
    int l_pal;      /* longitud de les paletes */
    int n_fil;
    int n_col;
} DadesCompartides;

DadesCompartides *dades; // Les dades essencials del joc
```

Como se puede observar tenemos aquí todas las variables globales compartidas entre las paletas del ordenador y el proceso main que es el que maneja el control del juego. Además, se observa como ahora desde es un puntero y no una variable.

También hemos añadido varios ids que serán las direcciones de memoria donde se guardarán estas estructuras.

Observar también que ahora la estructura de las paletas del ordenador, y la matriz que cuenta los movimientos de las paletas, son punteros de memoria.

```
int id_shm; // id de direcció de la memoria

typedef struct {
    int ipo_pf, ipo_pc;      /* posicio del la paleta de l'ordinador */
    float v_pal;            /* velocitat de la paleta del programa */
    float pal_ret;          /* percentatge de retard de la paleta */
    float po_pf;            /* pos. vertical de la paleta de l'ordinador, en valor real */
} Fila;

int id_shm_matrizPaletas; // Identificador de memoria la matriz de paletas.
Fila* matrizPaletas;

int id_shm_matrizMovimientosP;
int* matrizMovimientosPaletas;

int id_shm_retwin;
void* shared_mem_retwin;
```

Si observamos en la imagen también tenemos un puntero final junto a su variable con el id de la pantalla del juego, ya que esto deberá asignarse a una zona de memoria compartida.

Además, luego en el main, inicializamos estas zonas de memoria compartida.

```
id_shm = ini_mem(sizeof(DadesCompartides)); // Inicialitzem la memoria compartida
dades = map_mem(id_shm); // Mapejem la memoria compartida
dades->cont = -1; // Inicializamos el contador a -1.

id_shm_matrizPaletas = ini_mem(sizeof(Fila) * NUMMAXPALETAS); // Inicialitzem la memoria compartida
matrizPaletas = map_mem(id_shm_matrizPaletas); // Mapejem la memoria compartida

id_shm_matrizMovimientosP = ini_mem(sizeof(int) * NUMMAXPALETAS); // Inicialitzem la memoria compartida
matrizMovimientosPaletas = map_mem(id_shm_matrizMovimientosP); // Mapejem la memoria compartida
```

3.2. Asignación de las zonas de memoria compartida.

Una vez hemos definido las variables y están inicializadas deberemos ir cambiando en el main para que las variables antiguas pasen a ser todas las del struct.

Por ejemplo:

```
while(!feof(fit)) {
    fscanf(fit, "%d %d %f %f\n", &ipo_pf, &ipo_pc, &v_pal, &pal_ret);
    if ((ipo_pf < 1) || (ipo_pf+dades->l_pal > dades->n_fil-2) ||
        (ipo_pc < 5) || (ipo_pc > dades->n_col-2) ||
        (v_pal < MIN_VEL) || (v_pal > MAX_VEL) ||
        (pal_ret < MIN_RET) || (pal_ret > MAX_RET))
    {
        fprintf(stderr, "Error: parametres paleta ordinador incorrectes:\n");
        fprintf(stderr, "\t1 =< ipo_pf (%d) =< n_fil-1_pal-3 (%d)\n", ipo_pf, (dades->n_fil-dades->l_pal-3));
        fprintf(stderr, "\t5 =< ipo_pc (%d) =< n_col-2 (%d)\n", ipo_pc, (dades->n_col-2));
        fprintf(stderr, "\t%.1f =< v_pal (%.1f) =< %.1f\n", MIN_VEL, v_pal, MAX_VEL);
        fprintf(stderr, "\t%.1f =< pal_ret (%.1f) =< %.1f\n", MIN_RET, pal_ret, MAX_RET);
        fclose(fit);
        exit(5);
    }
    matrizPaletas[n_pal].ipo_pf = ipo_pf;
    matrizPaletas[n_pal].ipo_pc = ipo_pc;
    matrizPaletas[n_pal].v_pal = v_pal;
    matrizPaletas[n_pal].pal_ret = pal_ret;
    n_pal++;
}
```

Observar que ahora en varias variables se hace uso de `dades->variable`. La `matrizPaletas`, se mantiene intacta, al ser un puntero a una matriz donde se accede por índice. Solo que ahora actúa como un puntero de memoria para la zona de memoria compartida.

Zona de memoria compartida de la pantalla:

```
retwin = win_ini(&dades->n_fil, &dades->n_col, '+', INVERS); /* intenta crear taulell */

id_shm_retwin = ini_mem(retwin); // Inicialitzem la memoria compartida
shared_mem_retwin = map_mem(id_shm_retwin); // Mapejem la memoria compartida
win_set(shared_mem_retwin, dades->n_fil, dades->n_col); // Guardem la finestra en la memoria compartida
```

Primero obtenemos el tamaño de memoria que ocupa el tablero. Luego inicializamos la memoria a partir de este tamaño y guardamos la dirección. A partir de la dirección guardada, el número de filas y el número de columnas creamos el acceso a esta ventana creada. Creando así la zona de memoria compartida.

Además, siempre guardamos este identificador a la zona de memoria, por lo que así los procesos también podrán hacer uso de la pantalla para realizar los movimientos de las paletas del ordenador.

3.3. Creación de los procesos de las paletas del ordenador.

En esta parte explicaremos como hemos elaborado el cambio para que las paletas del ordenador pasen a ser procesos hijos del proceso padre:

1. Pasar la función de las paletas del ordenador al archivo pal_ord3.
2. Crear los procesos que usen esta función.

3.3.1. Paso de la función de las paletas del ordenador al archivo pal_ord3.

Simplemente deberemos eliminar la función de el archivo padre, y lo pegaremos en el archivo pal_ord3.c.

```
al Help
M Makefile C pal_ord3.c X C tennis3.c C winsupport2.h
C pal_ord3.c > main(int, char *[])
14 int main (int n_args, char *ll_args[]) {
57
58 while ((dades->tec != TEC_RETURN) && (dades->cont == -1) && ((dades->moviments > 0) || dades->moviments == -1 || dades->moviments_infinits == 1)) {
59 win_retard(dades->retard * matrizPaletas[index].pal_ret);
60 f_h = matrizPaletas[index].po_pf + matrizPaletas[index].v_pal; /* posicio hipotetica de la paleta */
61 if (f_h != matrizPaletas[index].ipo_pf) /* si pos. hipotetica no coincideix amb pos. actual */
62 {
63     if (matrizPaletas[index].v_pal > 0.0) /* verificar moviment cap avall */
64     {
65         if (win_quincar(f_h+dades->l_pal-1,matrizPaletas[index].ipo_pc) == ' ') /* si no hi ha obstacle */
66         {
67             win_escriure(matrizPaletas[index].ipo_pf,matrizPaletas[index].ipo_pc,' ',NO_INV); /* esborra primer bloc */
68             matrizPaletas[index].po_pf += matrizPaletas[index].v_pal; matrizPaletas[index].ipo_pf = matrizPaletas[index].po_pf; /* actualitza posicio */
69             win_escriure(matrizPaletas[index].ipo_pf+dades->l_pal-1,matrizPaletas[index].ipo_pc,'1'+index,INVERS); /* impr. ultim bloc */
70             if (dades->moviments > 0){
71                 dades->moviments--; /* he fet un moviment de la paleta */
72                 matrizMovimentsPaletas[index]++; // Actualitzem el numero de moviments de la paleta.
73             }
74         }
75     }
76 }
```

Ahora podemos crear los procesos, e ir realizando las modificaciones pertinentes.

3.3.2. Creación de los procesos de las paletas del ordenador.

Primero deberemos convertir las direcciones de memoria a char. Esto es así porque la función `execvp` necesita que los argumentos sean strings. Por ende, primero lo convertiremos en string y luego se lo indicaremos como argumento. Como además hay N paletas, realizamos un for de N paletas, donde guardaremos el ID del nuevo proceso en la tabla `tpid` usando la función `fork` que crea un proceso hijo del proceso padre.

```
// Conversio de IDs a String.
char id_shm_str[32];
char id_shm_matrizPaletas_str[32];
char id_shm_matrizMovimentsP_str[32];
char id_shm_retwin_str[32];
sprintf(id_shm_str, "%d", id_shm);
sprintf(id_shm_matrizPaletas_str, "%d", id_shm_matrizPaletas);
sprintf(id_shm_matrizMovimentsP_str, "%d", id_shm_matrizMovimentsP);
sprintf(id_shm_retwin_str, "%d", id_shm_retwin);
char index_str[12];
for (int i = 0; i < n_pal; i++)
{
    tpid[i] = fork(); // Creem el proces fill.
    if(tpid[i] == (pid_t) 0) {
        sprintf(index_str, "%d", i); // Convertim l'index a string.
        execvp("./pal_ord3", "./pal_ord3", id_shm_str, id_shm_matrizMovimentsP_str, id_shm_matrizPaletas_str, index_str, id_shm_retwin_str, (char*)0); // Creem el proc
        fprintf(stderr, "No se ha pogut crear els fils de la paleta del ordinador, error!");
        exit(0);
    }
}
```

Si se genera algún error, se genera un error en pantalla.

Ahora, bien, el proceso padre, deberá acceder a estas zonas de memoria. Para ello primero copiamos los structs que usara la paleta del ordenador. Ojo, esto se podría hacer un archivo .h y así evitar duplicados. Pero esto lo hacemos de esta forma, porque el

```

}

typedef struct {
    int tec;
    int cont;
    int moviments;
    int moviments_inicials;
    int moviments_infinits;
    int retard;
    int l_pal;
    int n_fil;
    int n_col;
} DadesCompartides;

typedef struct {
    int ipo_pf, ipo_pc; /* posicio del la paleta de l'ordinador */
    float v_pal; /* velocitat de la paleta del programa */
    float pal_ret; /* percentatge de retard de la paleta */
    float po_pf; /* pos. vertical de la paleta de l'ordinador, en valor real */
} Fila;

```

Ahora, podemos a traves de las direcciones de memoria, obtener los punteros de memoria y ya tener los datos correspondientes cargados.

```

int id_shm = atoi(ll_args[1]);
int id_shm_matrizMovimientosP = atoi(ll_args[2]);
int id_shm_matrizPaletas = atoi(ll_args[3]);
int index = atoi(ll_args[4]); // Index de la paleta.
int pWin = atoi(ll_args[5]);
void *pWinP = map_mem(pWin);
DadesCompartides* dades = map_mem(id_shm);
int* matrizMovimientosPaletas = map_mem(id_shm_matrizMovimientosP);
Fila* matrizPaletas = map_mem(id_shm_matrizPaletas);
int f_h;

```

Donde se obtiene el string que se le pasa por argumento, en formato int ya que realmente el string siempre será un int para nuestro caso. Y se carga luego mediante map_mem el puntero a memoria. Cargando así los datos correspondientes a esa posición de memoria.

Una carga importante es la de la variable pWin. Esta variable es la dirección de la pantalla de juego (con su puntero pWinP). Es la que usará la paleta, para modificar en pantalla.

```

win_set(pWinP, dades->n_fil, dades->n_col); // Posem la escritura de pantalla també per a les paletes del ordinador.

```

Mediante win_set asignamos ya así el entorno de modificación de pantalla de la paleta del ordenador.

Con esto ya tendríamos los procesos funcionando. Ahora, deberemos actualizar la pantalla del juego.

3.4. Actualización de la pantalla de juego.

Una vez tenemos los procesos, deberemos ir actualizando la para ir observando los cambios de los distintos elementos.

Para ello hemos elaborado un thread que mediante win_update va actualizando la pantalla.

```

void *actualitzar_pantalla(void *cap) {
    while ((dades->tec != TEC_RETURN) && (dades->cont == -1) && ((dades->moviments > 0) || dades->moviments == -1 || dades->moviments_infinits == 1)) {
        win_retard(dades->retard);
        win_update();
    }
    return NULL;
}

```

3.5. Eliminación de los procesos.

Una vez creados los procesos deberemos eliminarlos cuando termine el juego, para ello podemos hacerlo mediante el siguiente código.

```

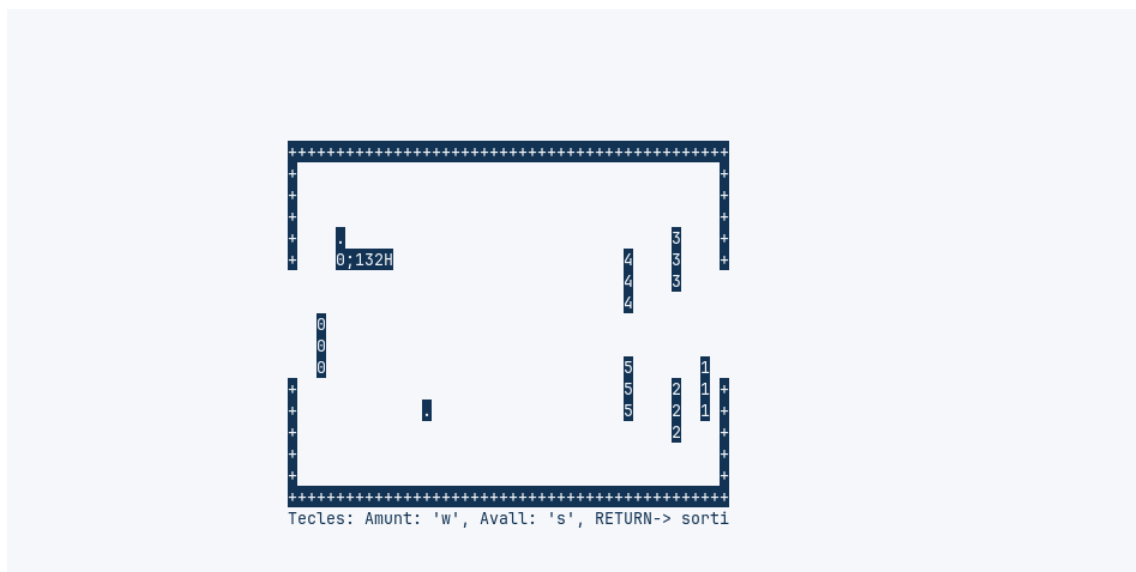
for (int i = 0; i < n_pal; i++)
{
    waitpid(tpid[i], NULL, 0); // Esperem que acabi el proces fill.
}

```

Este código simplemente realiza un for que recorre toda la tabla de pid llamando a la función waitpid para confirmar que todos los procesos se han acabado antes de finalizar el proceso padre.

3.6. Prueba de ejecución.

Vemos que ahora, se ejecuta todo de forma normal. Aparecen caracteres inesperados debido a la no sincronización de acceso a las zonas de memoria compartida ya que hemos eliminados los semáforos ya que en la F4 es cuando arreglaremos esta sincronización.



3.7. Problemas encontrados.

En estas pruebas de ejecución encontramos los siguientes problemas:

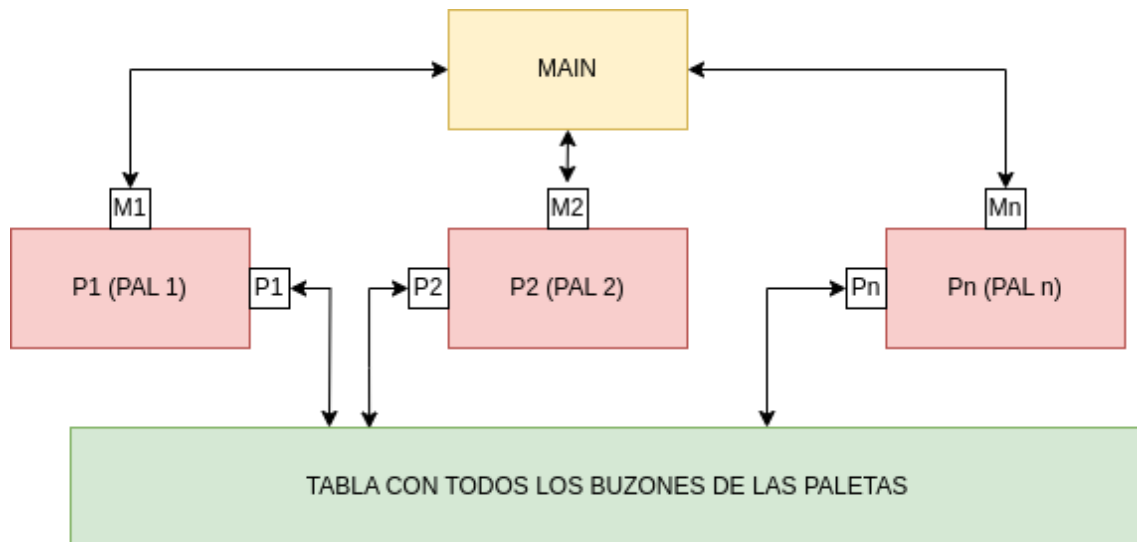
1. Se generan caracteres inesperados en la pantalla por el acceso inesperado de los procesos y threads a las variables globales y zonas de memoria compartida.
2. No funciona la pausa del juego. Esto es así porque hemos eliminado todos los semáforos, para replantear su uso durante la F4 y la pausa se realiza mediante semáforos.

4. FASE 4: Sincronización de procesos, threads y comunicación entre procesos.

En esta fase deberemos implementar la sincronización al acceso de los recursos que se encuentran en las zonas de memoria para evitar accesos no controlados provocando por ejemplo los caracteres no controlados en pantalla.

4.1. Elaboración de los buzones de comunicación.

Primero deberemos establecer los buzones de comunicación de los procesos. Para ello hemos establecido el siguiente esquema:



De esta forma encontramos que las paletas tendrán un buzón que va con el main. Esto se ha hecho a modo de poder en un futuro escalarlo. No tiene utilidad actualmente. Lo que sí tiene utilidad es la tabla con todos los buzones de las paletas. Esta tabla simplemente es una zona de memoria compartida donde todas las paletas pueden acceder a ella.

Una vez tenemos el esquema planteado de lo que queremos conseguir, procedemos a preparar todo el código correspondiente.

4.1.1. Creación de los buzones.

Creamos las variables:

1. Matriz donde residirán todos los buzones de las paletas.
2. Buzón del proceso main (padre).

```
139 int id_sem_rebot;
140 int mem_busties_pal;
141 int *p_mem_busties_pal;
142 int bustia_main;
143
```

Ahora podemos crear los buzones en sí, esto se hace mediante `ini_mis()` el cual crea un buzón reservando un conjunto de mensaje del sistema de forma privada para el proceso que lo llama.

```

// Crear bustia main
bustia_main = ini_mis();

// Crear busties paletes.
mem_busties_pal = ini_mem(sizeof(int[NUMMAXPALETAS]));
p_mem_busties_pal = map_mem(mem_busties_pal);
// Creem les busties de les paletes
for(int i = 0; i < n_pal; i++) {
    p_mem_busties_pal[i] = ini_mis();
}

```

Una vez creado, deberemos pasarles la información a los procesos.

```

r, bustia_main_str, mem_busties_str, (char*)0);

```

Imagen: Recorte de los parámetros que se le pasa a los procesos.

4.1.2. Uso de los buzones tanto para el envío como para recibir los mensajes.

En la paleta del ordenador, ahora tenemos un proceso que se encarga de recibir los mensajes, haciendo uso de receiveM.

```

*/
void *comprobarMiss() {
    int missatge;
    while(1) {
        missatge = receiveM(p_mem_busties[numeroPaleta], mis);
        if (missatge != 0) {
            missatgeRebut = true;
            waitS(id_sem);
            if (mis[0] == '1') {
                dir = 1; // derecha
                op = 1; // Direccion del movimiento es decir col + op
            }
            if (mis[0] == '2') {
                dir = 2; // izquierda
                op = -1; // Direccion del movimiento es decir col + op
            }
            signalS(id_sem);
        }
    }
    return 0;
}

```

Respecto al mensaje recibido, y su clasificación hablaremos más adelante.

Para poder enviar los mensajes lo hacemos de la siguiente forma (ejemplo):

```
    sprintf(mis, "%c", c);  
    }  
    sendM(p_mem_busties_pal[(int) rd - 49], mis, 2); // rh - 49 ja que convertim el codi ascii a integer y, les paletes van de 0 a 8.  
    }
```

Donde mandamos el id del mensaje, el mensaje en sí, y luego el número de bytes a mandar.

4.1.3. Eliminación de los buzones.

Simplemente deberemos hacer uso de la función `elim_mis(id_bustia)`. Donde se eliminará el IPC privado generado al crear el buzón.

```
    elim_mis(bustia_main);  
    for (int i = 0; i < n_pal; i++)  
    {  
        elim_mis(p_mem_busties_pal[i]);  
    }  
  
    elim_mem(mem_busties_pal);
```

4.2. Transmisión de movimientos entre las paletas usando buzones.

Una vez hemos aprendido y elaborado el esquema correspondiente. Procedemos a elaborar una mejora al juego, añadiendo las situaciones descritas en la práctica, cumpliendo lo siguiente:

- Si la pelota rebota en una paleta, donde no tiene a nadie en ningún lateral, la paleta se desplaza en la dirección de dónde venía la pelota.
- Si la pelota rebota en una paleta, y tiene a otra paleta pegada a ella, se le transmite este rebote a la otra paleta.
- Si alguna paleta está en un borde y se produce rebote, se eliminará la paleta.

Con esto en mente, procedemos a modificar el código para añadir estas funcionalidades.

4.2.1. Modificación de la función pelota y envío de información a los buzones de las paletas.

Procederemos a calcular y enviar los mensajes correspondientes a las paletas mediante el siguiente código:

```
    if(rh != '+' && rh != '0') {  
        if(c_h > ipil_pc) {  
            sprintf(mis, "%c", '1');  
        } else if (c_h < ipil_pc) {  
            sprintf(mis, "%c", '2');  
        }  
        sendM(p_mem_busties_pal[(int) rh - 49], mis, 2); // rh - 49 ja que convertim el codi ascii a integer y, les paletes van de 0 a 8.  
    }  
    c_h = pil_pc + pil_vc; /* actualitza posicio hipotetica */  
}
```



```

rd = win_quincar(f_h, c_h);

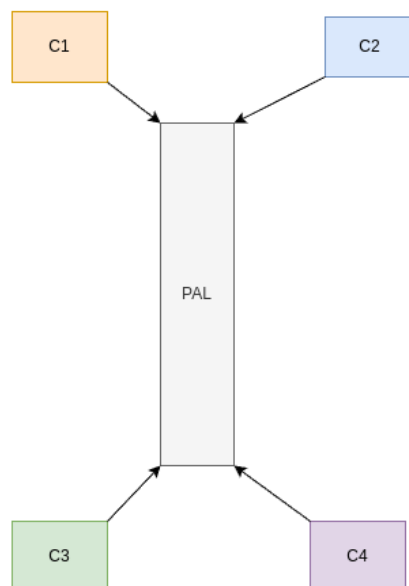
if (rd != ' ')          /* si no hi ha obstacle */
{
    pil_vf = -pil_vf; pil_vc = -pil_vc;    /* canvia velocitats */

    if(rd != '+' && rd != '0') {
        if((c_h > ipil_pc && f_h > ipil_pf)|| (c_h > ipil_pc && f_h < ipil_pf)){
            sprintf(mis, "%c", '1');
        }else if((c_h < ipil_pc && f_h < ipil_pf)|| (c_h < ipil_pc && f_h > ipil_pf)){
            sprintf(mis, "%c", '2');
        }
        sendM(p_mem_busties_pal[(int) rd - 49], mis, 2); // rh - 49 ja que convertim el c
    }
    c_h = pil_pc + pil_vc;    /* actualitza posicio entera */
    f_h = pil_pf + pil_vf;
}

```

Lo que hacemos es calcular tanto de forma horizontal como diagonal de donde proviene la pelota. Y indicar una dirección (1 = derecha, 2 = izquierda). Luego mandamos este mensaje a la paleta correspondiente. Importante remarcar que hacemos $rd - 49$ por queremos el integer entre 0-9. Ya que rd/rh está en ASCII.

Si analizamos detalladamente vemos que consideramos 4 casos:



4.2.2. Trato de los datos recibidos.

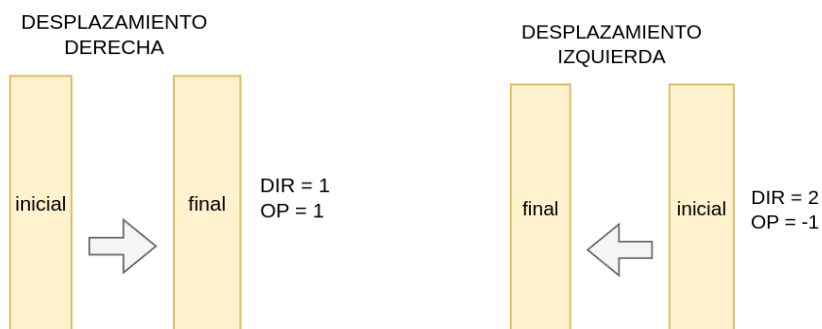
Como se ha comentado anteriormente tenemos un thread por cada proceso que se encargará de ir comprobando si llega algún mensaje.

```

*/
void *comprobarMiss() {
    int missatge;
    while(1) {
        missatge = receiveM(p_mem_busties[numeroPaleta], mis);
        if (missatge != 0) {
            missatgeRebut = true;
            waitS(id_sem);
            if (mis[0] == '1') {
                dir = 1; // derecha
                op = 1; // Direccion del movimiento es decir col + op
            }
            if (mis[0] == '2') {
                dir = 2; // izquierda
                op = -1; // Direccion del movimiento es decir col + op
            }
            signalS(id_sem);
        }
    }
    return 0;
}

```

Cuando llega un mensaje, esperamos al semáforo. Si el mensaje es 1, indicamos que la dir es 1, y el operando es 1. Si es 2, indicamos que el dir es 2, y que el operando es -1. Esto es así porque queremos saber a qué dirección tenemos que desplazar. Además, el operando funciona de la siguiente manera.



Lo que tenemos es que cuando desplazamos a la paleta, hacia la derecha estamos sumando 1 columna. Si es a la izquierda estamos restando una columna.

4.2.3. Función a partir de los datos.

Una vez tratados los datos, procedemos a elaborar la función que hará este desplazamiento de las paletas.

1. Verificamos si hemos recibido un mensaje. Si se ha recibido. Analizamos al lateral donde vamos a movernos, si tenemos otra paleta. J será la otra paleta, donde pasaremos la información.

```

if (missatgeRebut) { // Verifiquem si hem rebut un missatge.
    waitS(id_sem); // Esperem al semafor.
    if (dir == 1 || dir == 2) { // Si la direccio es 1(dreta) o 2(esquerra)
        bool enviar = false;
        int fila = matrizPaletas[numeroPaleta].ipo_pf;
        int columna = matrizPaletas[numeroPaleta].ipo_pc;
        int j = 0;

        for(int i = 0; i < dades->l_pal && !enviar; i++) {
            int rh = win_quincar(fila+i, (columna + op));
            if (rh != ' ' && rh != '+') {
                enviar = true;
                j = ((int)rh - 49);
            }
        }
    }
}

```

2. Si no tenemos otra paleta. Verificamos si estamos en alguno de los bordes del mapa. Hemos establecido que el borde final será $n_col - 1$ y el borde inicial será 4.

Esto es así para evitar problemas con la paleta del usuario y que se pueda jugar bien. Si es así eliminamos la paleta, y finalizamos su proceso.

```

if (!enviar) { // Si no la trobem
if((dir == 1 && ((columna + 1) == (dades->n_col-1))) || (dir == 2 && ((columna - 1) == 4))) { // Si estem al final del tauler-1 o al inici del tauler+4
for (int i = 0; i < dades->l_pal; i++) {
win_escricar(fila, columna, ' ', NO_INV); // Eliminem la paleta
fila++;
}
pthread_cancel(missatge); // Tanquem tant els threads com el proces de la paleta.
pthread_join(missatge, NULL);
signals(id_sem);
exit(0);
}

```

3. Si no tenemos otra paleta, y podemos desplazarnos ya que no vamos a sobrepasar los bordes del mapa, nos desplazamos.

```

} else {
for (int i = 0; i < dades->l_pal; i++) // Si no estem al final del tauler
{
win_escricar(fila, columna, ' ', NO_INV); // Eliminem la paleta de la pantalla
win_escricar(fila, (columna + op), '1' + numeroPaleta, INVERS); // La escribim en la nova posicio
fila++;
}
matrizPaletas[numeroPaleta].ipo_pc = columna + op; // Augmentem la posicio horizontal (columna)
}
}

```

4. Finalmente, si tenemos otra paleta. Enviamos un mensaje. Al final de todo el proceso, restablecemos el valor de missatge_rebut a false y enviamos señal al semáforo para la correcta sincronización.

```

} else { // Si es te que enviar, enviem un missatge a la altra paleta per fer el desplaçament.
char miss[2];
sprintf(miss, "%c", mis[0]); // Preparem el missatge, en resum miss_enviar = mis rebut.
sendM(p_mem_busties[j], miss, 2); // Enviem el missatge.
}
signals(id_sem); // Enviem señal al semafor
missatgeRebut = false; // Tornem a dir que no hem rebut cap missatge.
}
}

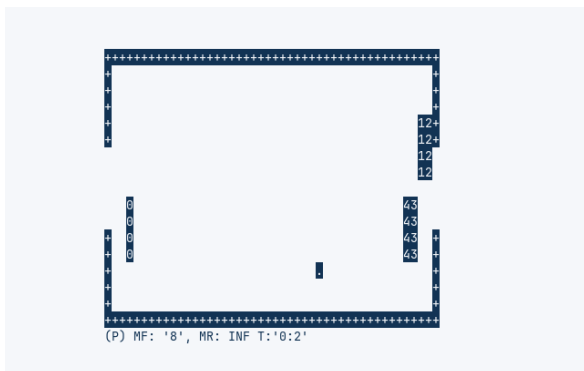
```

4.3. Juegos de pruebas.

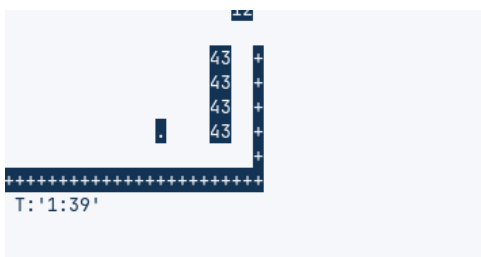
Ahora podemos elaborar varios juegos de pruebas para poder comprobar su correcto funcionamiento:

Prueba 1 (mapa 6.txt):

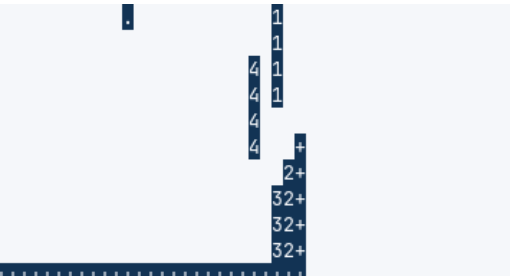
En esta prueba lo que trataremos es de verificar si se transmite el movimiento entre 2 paletas. Además de que puede generar borrado sobre todo por la paleta de atrás (2).



Antes de impacto:



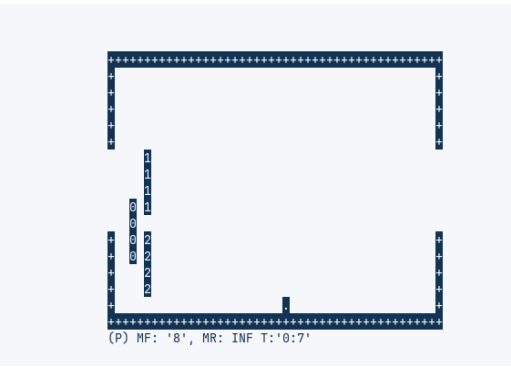
Después de impacto:



Vemos que se ha desplazado la paleta 3 que estaba junto a la 4. Movimiento correcto. Se ha perdido un carácter de la paleta 3 que luego vuelve a aparecer.

Prueba 2 (mapa 7.txt)

En esta prueba verificaremos que si la paleta se acerca al borde inicial del mapa se elimina del mapa.

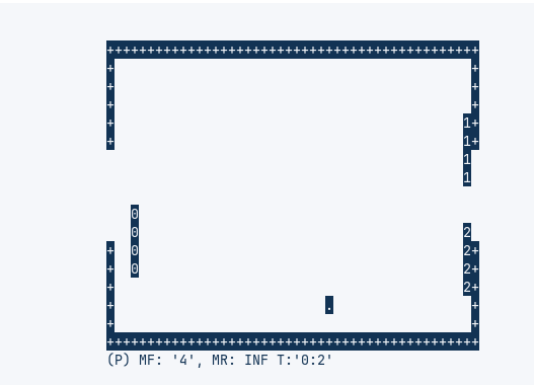


Antes del impacto: Después del impacto:



Prueba 3 (mapa 5.txt):

En esta prueba verificaremos si la paleta al estar al final del mapa se elimina.



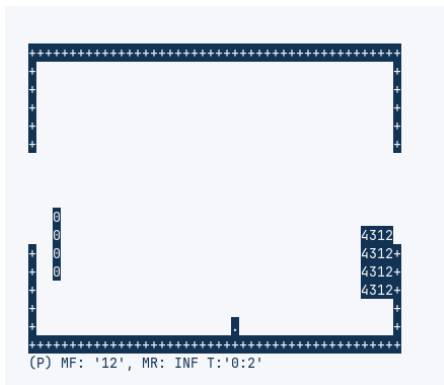
Antes del impacto: Después del impacto:



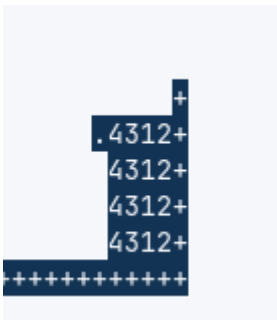
Se elimina la paleta tal y como se espera.

Prueba 4 (mapa 8.txt):

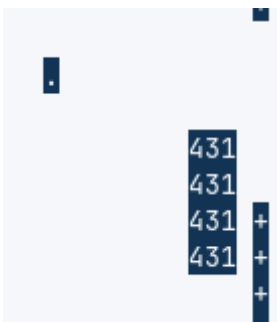
En esta prueba lo que haremos será adjuntar 2 pruebas, la 1 y la 3. De esta forma verificamos si se transmite los mensajes a lo largo de todas las paletas, y además se cumplen los límites.



Antes del impacto:



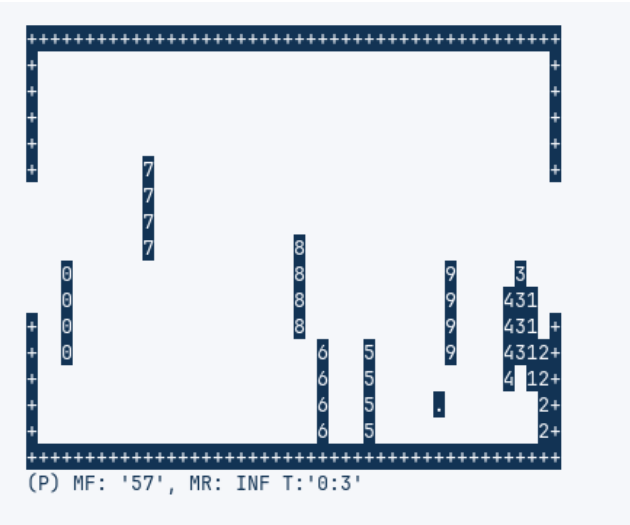
Después del impacto:



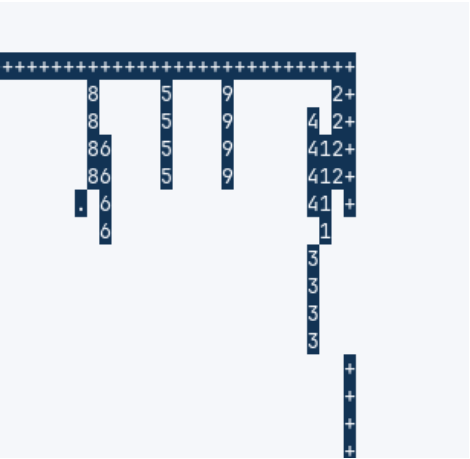
Se ha eliminado la paleta 2 correctamente, tal y como esperábamos.

Prueba 5 (mapa 9.txt):

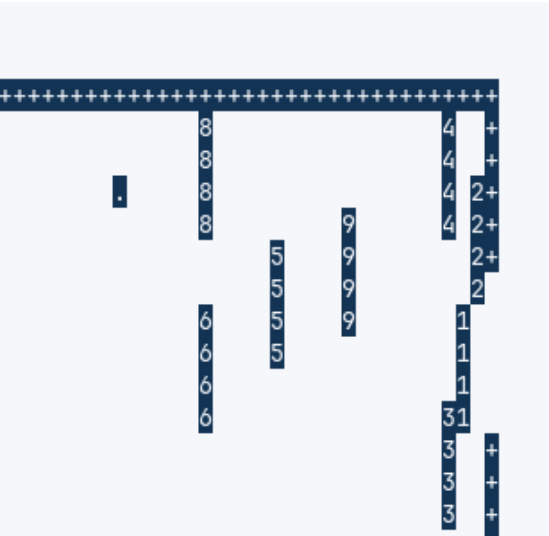
En esta prueba lo que haremos será probar los desplazamientos laterales en ambas direcciones.



Antes del impacto:

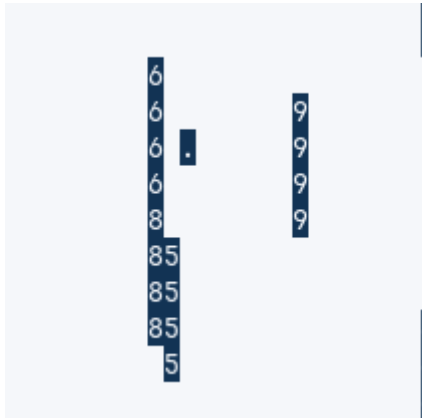


Después del impacto:

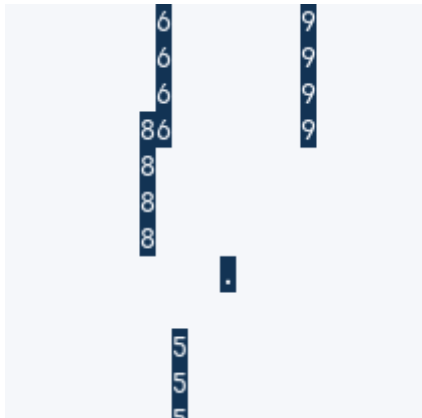


Al desplazarse rápido, ha impactado con el 8. Generando impacto diagonal desde la izquierda. Movimiento de la paleta hacia la derecha, misma altura que la paleta 6.

Antes del impacto:



Después del impacto:



Al igual que antes, el desplazamiento de las paletas es muy rápido, y genera impacto con la paleta 8. Impacto desde la derecha, movimiento de la paleta hacia la izquierda.

5. Conclusiones.

La práctica realizada ha sido una herramienta valiosa para comprender y manejar el uso de threads, procesos y la comunicación entre ellos. Mediante la implementación de mapas de memoria y buzones de mensajes, se han podido explorar los conceptos teóricos y aplicarlos a un caso práctico.

La fase 4, en particular, ha presentado un desafío interesante al abordar la lógica de los movimientos de las paletas. Esta dificultad ha permitido profundizar en la comprensión de la sincronización y la comunicación entre procesos.