

Memoria Ejercicio feedback2

Autor: Martina García González NP:141478

1. Índice de contenidos

1.	ÍNDICE DE CONTENIDOS	2
2.	SOLUCIÓN EJERCICIO 1	3
2.1.	APARTADO1	3
2.2.	APARTADO2	4
2.3.	APARTADO3	5
2.4.	APARTADO4	6
3.	SOLUCIÓN EJERCICIO 2	7
3.1.	APARTADO 1	7
3.2.	APARTADO 2	7
3.3.	APARTADO 3	7
4.	SOLUCIÓN EJERCICIO 3	9
4.1.	APARTADO1	9
4.2.	APARTADO2	9
5.	ANEXO 1: CÓDIGO DE LA SOLUCIÓN	11
•	Github: https://github.com/martinagg7/so_feedback.git	11

2. Solución Ejercicio 1

2.1. Apartado1



```
#include <stdio.h>
#include <pthread.h>
#include <semaphore.h>

#define N 10
#define NHP 35

typedef struct {
    int id;
    int val;
} hebval_t;

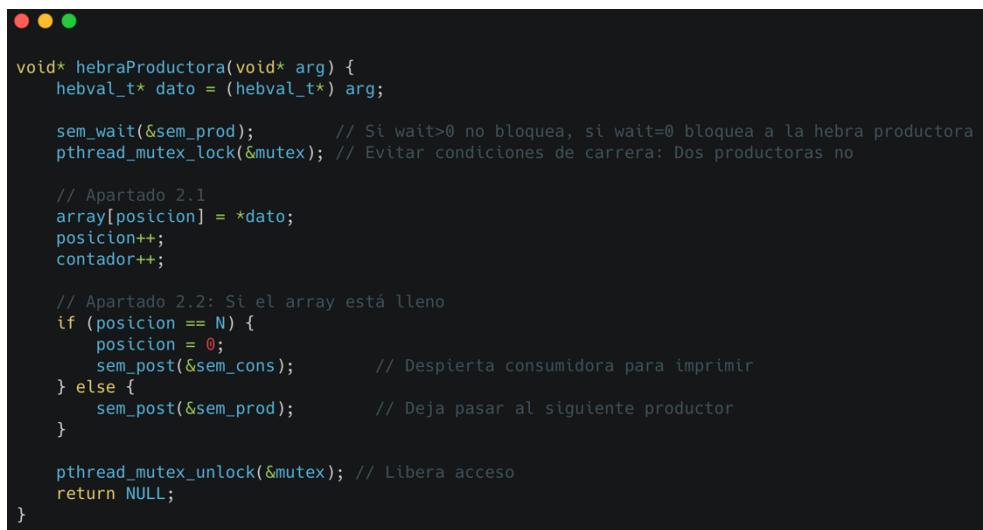
hebval_t array[N];
int posicion = 0; // Puntero para insertar en el array
int contador = 0; // Numero total de hebras terminado

sem_t sem_prod, sem_cons;
pthread_mutex_t mutex;
}
```

- **Variables N y NHP:** Se inicializan estas constantes para representar el tamaño del array compartido y el número total de hebras productoras .
- **Estructura de las hebras (hebval_t):** Esta estructura contiene
 1. **ID:** Identificador único para cada hebra.
 2. **Valor:** El dato que se guardará en el array.
- **Array compartido:** Se inicializa un array **hebval_t** para guardar las estructuras generadas por las productoras hasta un máximo de elementos.
- **Variable posición:** Esta variable sirve para mirar e indicar en qué índice exacto del array se debe realizar la próxima inserción (del 0 al 9).
- **Variable contador:** Se utiliza para saber el número de hebras que han terminado su ejecución, permitiendo controlar el final del programa.
- **Semáforos generales (sem_prod y sem_cons):** Sirven para que las hebras productoras y la consumidora no se puedan ejecutar a la vez, gestionando los turnos de llenado e imprimir.

- **Mutex:** Se utiliza para evitar condiciones de carrera cuando las hebras productoras tratan de escribir sus valores en el array y actualizar la variable `posicion` al mismo tiempo.

2.2. Apartado2



```

void* hebraProductora(void* arg) {
    hebval_t* dato = (hebval_t*) arg;

    sem_wait(&sem_prod);           // Si wait>0 no bloquea, si wait=0 bloquea a la hebra productora
    pthread_mutex_lock(&mutex);   // Evitar condiciones de carrera: Dos productoras no

    // Apartado 2.1
    array[posicion] = *dato;
    posicion++;
    contador++;

    // Apartado 2.2: Si el array está lleno
    if (posicion == N) {
        posicion = 0;
        sem_post(&sem_cons);      // Despierta consumidora para imprimir
    } else {
        sem_post(&sem_prod);      // Deja pasar al siguiente productor
    }

    pthread_mutex_unlock(&mutex); // Libera acceso
    return NULL;
}

```

- La hebra recibe mediante un puntero el dato de tipo `hebval_t` que debe insertar.
- **Sincronización `sem_wait(&sem_prod)`:** Si llega una hebra producto y el semáforo está a 0, la hebra se bloquea para no producir cuando la consumidora está trabajando o el array está lleno.
- **Exclusión Mutua (`mutex`):** Se bloquea la sección crítica antes de tocar las variables globales para evitar **condiciones de carrera**, asegurando que ninguna otra hebra modifique la `posicion` simultáneamente.
- **Inserción y actualización:** Se realiza la copia del dato en el índice indicado por `posicion` y se incrementan tanto el puntero de inserción como el `contador` general de hebras finalizadas.
- **Control de buffer lleno (`if (posicion == N)`):**
 1. Si el array llega a su capacidad máxima , se resetea la `posicion` a **0**.
 2. Se utiliza `sem_post (&sem_cons)` para despertar a la hebra consumidora.
 3. No se hace post al semáforo productor ya que debe esperar que la hebra consumidora termine de imprimir los valores.

2.3. Apartado3

```
● ● ●

void* hebraConsumidora(void* arg) {
    while (contador < NHP) {
        sem_wait(&sem_cons); // Espera a que la productura llene el array

        // Imprimir id,valor
        printf("Contenido del array: [ ");
        for (int i = 0; i < N; i++) {
            printf("[id %d, val %d] ", array[i].id, array[i].val);
        }
        printf("]\n\n");

        // Volver avisar para continuar produciendo
        if (contador < NHP) {
            sem_post(&sem_prod);
        }
    }
    return NULL;
}
```

- **Bucle de control (`while`):** La hebra permanece en ejecución mientras el `contador` global sea inferior a 35 .
- **Sincronización `sem_wait(&sem_cons)`:** Esta instrucción mantiene a la consumidora bloqueada (`contador =0`, cuando hace wait `contador =-1`) hasta que una hebra productora detecta que el array está lleno y envía la señal mediante un `post` (`contador=0`).
- **Salida por pantalla:** Una vez activada, recorre el array completo imprimiendo los campos `id` y `val` .
- **Reactivación del flujo (`sem_post`):**
 1. Tras mostrar los datos, la hebra consumidora comprueba si todavía quedan hebras por producir.
 2. Si es así, ejecuta `sem_post (&sem_prod)`, lo cual desbloquea de nuevo a las productoras para que rellenen el array desde la posición 0.

2.4. Apartado4

```

int main() {
    // Apartado 4.1
    pthread_t h_productores[NHP], h_consumidora;
    hebval_t datos[NHP];

    // Inicialización de semáforos y mutex
    sem_init(&sem_prod, 0, 1);
    sem_init(&sem_cons, 0, 0);
    pthread_mutex_init(&mutex, NULL);

    //A apartado 4.2: Inicializar el array con -1
    for (int i = 0; i < N; i++) {
        array[i].id = -1;
        array[i].val = -1;
    }

    // Apartado 4.3 : lanzar hebras productoras
    for (int i = 0; i < NHP; i++) {
        datos[i].id = i;
        datos[i].val = i;
        pthread_create(&h_productores[i], NULL, hebraProductora, &datos[i]);
    }

    // Apartado 4.4: lanzar hebra consumidora
    pthread_create(&h_consumidora, NULL, hebraConsumidora, NULL);

    // Apartado 4.5: evitar hebras huérfanas .
    for (int i = 0; i < NHP; i++) {
        pthread_join(h_productores[i], NULL);
    }
    pthread_join(h_consumidora, NULL);

    sem_destroy(&sem_prod);
    sem_destroy(&sem_cons);
    pthread_mutex_destroy(&mutex);

    return 0;
}

```

- **Configuración de Sincronización:** Se inicializan los semáforos estableciendo el estado inicial del sistema: `sem_prod` se inicia en **1** (permite que la primera productora comience) y `sem_cons` en **0** (fuerza a la consumidora a esperar).
- **Estado inicial del Array:** Se realiza el array con todo valor **-1** en sus posiciones
- **Creación de Hebras:** Se generan las **35** hebras productoras, asignándoles a cada una su estructura de datos con un ID y valor secuencial.
- **Gestión de Terminación (`pthread_join`):** Se utiliza esta función para que el proceso principal espere a que todas las hebras terminen su ejecución. Esto evita que el programa finalice antes de tiempo y que queden hebras huérfanas en el sistema.

3. Solución Ejercicio 2

3.1. Apartado 1

1) Número total de memoria en posiciones

- El tamaño de página es igual al tamaño de los marcos de página en la RAM.
- Partiendo de que necesitamos 7 bits para direccionar una palabra dentro de una imagen, las imágenes virtuales de los procesos tienen un tamaño de $2^7 = 128$ palabras/posiciones.
- Como tenemos 2^6 (64) marcos de página en RAM, estos deben tener el mismo tamaño que las páginas.
- **Tamaño memoria:** $2^7 \times 2^6 = 2^{13}$ palabras/posiciones.

2) Tamaño palabras marcos página

- Los marcos de página (RAM) deben tener el mismo tamaño que las páginas virtuales.
- **Tamaño página:** $2^7 = 128$ palabras.

3) Número máximo de páginas

- Como vimos, el número total de marcos en la RAM es 64. Suponiendo que en la RAM no hay ningún otro proceso, el número máximo de páginas que podría almacenar en RAM sería de **64 páginas**.
- Esta afirmación solo sería correcta si suponemos que el espacio lógico = físico, sino el proceso podría contar con más páginas en RAM.

3.2. Apartado 2

1) Obtener la tabla de páginas de los procesos A, B y C

La tabla de páginas establecerá una correspondencia entre las páginas virtuales de los procesos (A, B, C) y los correspondientes marcos físicos que ocupan:

Proceso A		Proceso B		Proceso C	
Pág. Virt	Marco	Pág. Virt	Marco	Pág. Virt	Marco
0	0	0	2	0	5
1	1	1	3	1	6
2	8	2	4	2	7
3	9	3	10		
		4	11		

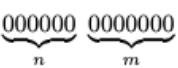
3.3. Apartado 3

1) Dirección virtual y física de la última dirección lógica del proceso A

- El proceso A tiene 512 posiciones. Como el tamaño de página es de $2^7 = 128$, el proceso tiene $512/128 = 2^2 \Rightarrow 4$ páginas totales.

- Sus direcciones van desde 0 hasta 511.

- Consideramos que la memoria tiene una capacidad de 2^{13} posiciones con ($n = 6$) y ($m = 7$).

▪ Dirección lógica 0: 

- La página 0 corresponde con el marco de página 0.

▪ Dirección física: 

▪ Dirección lógica 511: 

- La página 3 corresponde con el marco de página 9.

▪ Dirección física 511: 

2) Dirección física correspondiente a la dirección virtual 130 del proceso A

▪ Dirección lógica 130: 

- El marco de RAM correspondiente con la página 1 es el 1.

▪ Dirección física 130: 

3) Primera y última dirección del proceso C

- El proceso C tiene un tamaño de 384 posiciones. Como el tamaño de página es de $2^7 = 128$, el proceso tiene $384/128 = 3$ páginas.

▪ Dirección lógica 0: 

- Como en el proceso C la página 0 se corresponde con el marco de página 5:

▪ Dirección física 0: 

▪ Dirección lógica 383: 

- El marco de página correspondiente con la pág 2 es 7.

▪ Dirección física 383: 

4. Solución Ejercicio 3

4.1. Apartado1

```

● ● ●

void concatenar_archivos(const char *dir, const char *f1, const char *f2, const char *f_out) {
    char r1[512], r2[512], r_out[512], buffer[BUFFER_SIZE];
    int fd1, fd2, fd_out;
    ssize_t n;

    // Se construyen las rutas: directorio/archivo
    sprintf(r1, "%s/%s", dir, f1);
    sprintf(r2, "%s/%s", dir, f2);
    sprintf(r_out, "%s/%s", dir, f_out);

    // Se abren los archivos de origen y se crea el de salida
    fd1 = open(r1, O_RDONLY);
    fd2 = open(r2, O_RDONLY);
    fd_out = open(r_out, O_WRONLY | O_CREAT | O_TRUNC, 0644);

    // Se lee del primer archivo y se escribe en el resultado
    while ((n = read(fd1, buffer, BUFFER_SIZE)) > 0) write(fd_out, buffer, n);
    // Se lee del segundo archivo y se escribe a continuación (concatenación)
    while ((n = read(fd2, buffer, BUFFER_SIZE)) > 0) write(fd_out, buffer, n);

    close(fd1); close(fd2); close(fd_out);
}

```

- **Gestión de Rutas:** Se utiliza `sprintf` para construir las rutas de acceso uniendo el nombre del directorio con el de cada archivo.
- **Apertura con Flags:** Se abren los originales en **solo lectura** y el de salida con `O_CREAT` (crear si no existe) y `O_TRUNC` (sobrescribir si ya existe), con permisos 0644.
- **Proceso de copia:** Se utiliza un **buffer** para mover los datos. El programa lee bloques del primer archivo y los escribe en el nuevo; después repite lo mismo con el segundo. Al cerrar los descriptores al final, aseguramos que los datos se guarden bien.

4.2. Apartado2

```

● ● ●

void listar_archivos(const char *directorio) {
    DIR *d = opendir(directorio);
    struct dirent *ent;
    struct stat info;
    char ruta[512];

    if (d == NULL) {
        perror("Error al abrir el directorio");
        return;
    }

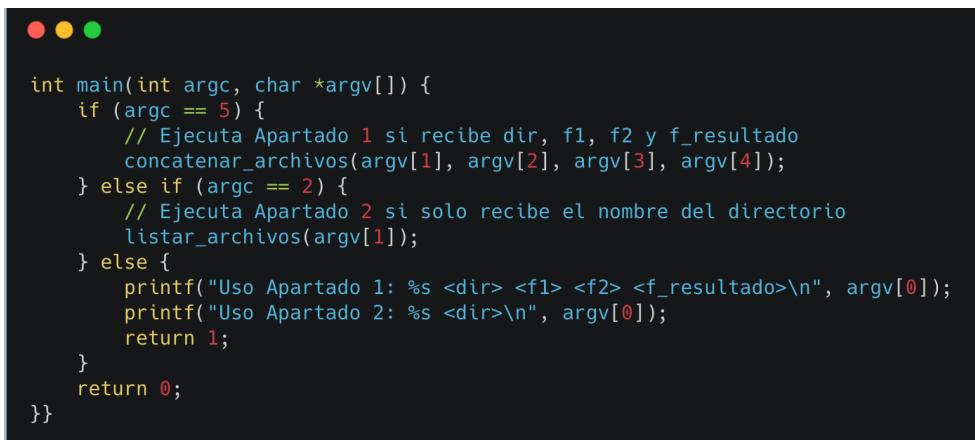
    printf("Listado de archivos en: %s\n", directorio);
    while ((ent = readdir(d)) != NULL) {
        if (ent->d_name[0] == '.') continue; // Ignorar archivos ocultos

        sprintf(ruta, "%s/%s", directorio, ent->d_name);

        // Con stat obtenemos el inodo y número de enlaces duros
        if (stat(ruta, &info) == 0) {
            printf("Nombre: %s | Inodo: %lu | Enlaces Duros: %lu\n",
                   ent->d_name, info.st_ino, info.st_nlink);
        }
    }
    closedir(d);
}

```

- **Exploración de Directorio:** Se abre el directorio con `opendir` y se recorre con `readdir`. Se filtran las entradas que empiezan por punto (.) para no mostrar archivos ocultos o de sistema.
- **Llamada a `stat`:** Para cada archivo, llamamos a `stat` usando su ruta completa para obtener sus metadatos.
- **Metadatos extraídos:**
 1. **Inodo:** El número de identidad único que el sistema operativo asigna al archivo en el disco.
 2. **Enlaces Duros:** El contador de cuántas referencias existen hacia ese mismo inodo (contenido físico)



```

int main(int argc, char *argv[]) {
    if (argc == 5) {
        // Ejecuta Apartado 1 si recibe dir, f1, f2 y f_resultado
        concatenar_archivos(argv[1], argv[2], argv[3], argv[4]);
    } else if (argc == 2) {
        // Ejecuta Apartado 2 si solo recibe el nombre del directorio
        listar_archivos(argv[1]);
    } else {
        printf("Uso Apartado 1: %s <dir> <f1> <f2> <f_resultado>\n", argv[0]);
        printf("Uso Apartado 2: %s <dir>\n", argv[0]);
        return 1;
    }
    return 0;
}

```

- **Control de argumentos (`argc`):** El programa decide qué hacer basándose en cuántos parámetros escribas en la consola.
- **Flexibilidad:** Si pasas 4 parámetros (directorio y 3 nombres de archivos), entiende que quieres unir ficheros. Si solo pasas 1 (un directorio), entiende que quieres listar su contenido técnico.

5. Anexo 1: Código de la solución

- Github: https://github.com/martinagg7/so_feedback.git

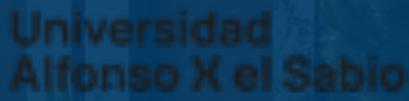
[



WELCOME
TO
UAX



UAX



Universidad
Alfonso X el Sabio



GRACIAS