Código con Comentarios Explicados

```
#include <sys/types.h>     // Incluye definiciones de tipos de datos utilizados en operaciones de sistema

#include <sys/ipc.h>       // Incluye las funciones para manejar IPC (Inter-Process Communication)

#include <sys/shm.h>       // Incluye las funciones para manejar memoria compartida

#include <stdio.h>         // Proporciona las funciones estándar de entrada/salida, como printf()

#include <stdlib.h>        // Proporciona funciones útiles como malloc(), exit(), etc.

#include <unistd.h>        // Incluye funciones del sistema como getpid(), sleep(), fork(), etc.


#define SHMKEY 75          // Define una constante para la clave de la memoria compartida

#define NUM_PROCESOS 2     // Define el número de procesos que alternarán turnos

#define TRUE 1             // Define el valor TRUE como 1 (para bucles infinitos)


// Prototipos de funciones que serán definidas más adelante

int region_critica(int pid);

int region_no_critica(int pid);

int esperando_turno(int pid);


int main() {

    int shmid, pid, i;     // Declaración de variables enteras: shmid para el ID de la memoria compartida, pid para el ID del

proceso, i para el bucle

    int *turno;            // Declaramos un puntero entero para manejar el turno en memoria compartida


    // Crear el segmento de memoria compartida

    shmid = shmget(SHMKEY, sizeof(int), 0777 | IPC_CREAT);

     // 'shmget' obtiene o crea un segmento de memoria compartida de tamaño 'sizeof(int)' (4 bytes), con permisos 0777
```

```
(lectura/escritura para todos)
    // Si shmget devuelve -1, hubo un error
    if (shmid == -1) {
        perror("Error al crear la memoria compartida");  // Imprime un mensaje de error si no se puede crear la memoria compartida
        exit(1);  // Termina el programa con un código de error
    }

    // Asociar la memoria compartida al proceso
    turno = (int *)shmat(shmid, 0, 0);
     // 'shmat' asocia el segmento de memoria compartida con la dirección de memoria del proceso, devolviendo la dirección donde empieza el segmento
    // Si 'shmat' devuelve (int *) -1, significa que hubo un error
    if (turno == (int *)-1) {
        perror("Error al asociar la memoria compartida");  // Imprime un mensaje de error si no se puede asociar
        exit(1);  // Termina el programa con un código de error
    }

    *turno = 0;  // Inicializa el turno en 0 (le da el turno al primer proceso)

    // Crear múltiples procesos
    for (i = 0; i < NUM_PROCESOS; i++) {
        pid = fork();  // Crea un nuevo proceso hijo usando 'fork()'
        // Si 'fork()' devuelve 0, significa que estamos en el proceso hijo
        if (pid == 0) {
            pid = getpid();  // Obtiene el ID del proceso actual y lo guarda en 'pid'
```

```
        while (TRUE) {  // Bucle infinito para simular que los procesos siguen alternando sus turnos
            // Mientras no sea el turno de este proceso, esperamos
            while (*turno != i) {  // Si el valor en 'turno' no es igual al índice del proceso (i), el proceso sigue esperando
                esperando_turno(pid);  // Llama a la función que imprime que el proceso está esperando
            }

            // Si es su turno, entra en la región crítica
            region_critica(pid);  // El proceso accede a la región crítica

            // Después de la región crítica, ejecuta la región no crítica
            region_no_critica(pid);  // El proceso realiza una tarea no crítica

            // Cambia el turno al siguiente proceso
            *turno = (*turno + 1) % NUM_PROCESOS;  // Incrementa el turno y lo alterna entre 0 y NUM_PROCESOS-1
        }
        exit(0);  // El proceso hijo sale del bucle infinito y termina
    }
}

// El proceso padre espera a que los hijos terminen (aunque los hijos no terminan en este ejemplo)
for (i = 0; i < NUM_PROCESOS; i++) {
    wait(NULL);  // 'wait()' espera a que un proceso hijo termine
}

// Desasociar la memoria compartida
```

```
    shmdt(turno);  // 'shmdt()' desasocia la memoria compartida del proceso
    shmctl(shmid, IPC_RMID, 0);  // 'shmctl()' elimina el segmento de memoria compartida, liberando los recursos

    return 0;  // El programa principal termina correctamente
}

// Función que imprime un mensaje mientras el proceso está esperando su turno
int esperando_turno(int pid) {
    printf("
 [I] Proceso %d está esperando su turno", pid);  // Imprime el ID del proceso que está esperando
    sleep(1);  // Pausa la ejecución por 1 segundo
    return 0;
}

// Función que simula la ejecución de una región crítica
int region_critica(int pid) {
    printf("

 [0] Proceso %d está en la región crítica", pid);  // Imprime que el proceso ha entrado en la región crítica
    for (int i = 0; i < 5; i++) {  // Bucle que simula 5 operaciones en la región crítica
        printf("
 [O] Proceso %d: Turno %d", pid, i + 1);  // Imprime el turno actual en la región crítica
        sleep(1);  // Pausa de 1 segundo entre cada operación
    }
    return 0;  // La función retorna 0 (indicando éxito)
}
```

```
// Función que simula la ejecución de una región no crítica

int region_no_critica(int pid) {

    printf("

 [O] Proceso %d ha salido de la región crítica

", pid);  // Imprime que el proceso ha salido de la región crítica

    sleep(1);  // Pausa por 1 segundo antes de continuar

    return 0;  // La función retorna 0 (indicando éxito)

}
```