TP 1

Unidad I a IV

Año: 2024
Materia: Sistemas Operativos
Profesores: Guillermo Cherencio & Juan Carlos Romero
Alumno: Francis Cirmi

Objetivo: Implementar una serie de ejercicios de programación ANSI C a modo de sintetizar toda la ejercitación propuesta en clase y en los prácticos no obligatorios de las Unidades I a IV del programa de estudios de la asignatura.

EJERCICIO 1

Implementación del programa Shell (en su versión más completa, que soporta procesos foreground y background; que haga uso de la señal SIGCHLD para verificar la finalización de procesos en background) que evite la existencia de procesos zombies. El programa termina cuando el usuario ingresa el comando “salir”. Si el usuario presiona CTRL+C el programa esperará a que terminen todos los procesos y luego finalizará sin dejar procesos zombies o huérfanos.

```c
// Librerías necesarias
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <signal.h>
#include <sys/types.h>
#include <sys/wait.h>

// Flag para indicar si se debe terminar el programa
volatile sig_atomic_t exit_requested = 0;

// Maneja la señal SIGCHLD para limpiar procesos hijos terminados
Codeium: Refactor | Explain | ×
void sigchld_handler(int signo) {
    while (waitpid(-1, NULL, WNOHANG) > 0); // Recolecta todos los procesos hijos zombies
}

// Maneja la señal SIGINT para permitir una salida controlada
Codeium: Refactor | Explain | ×
void sigint_handler(int signo) {
    printf("\nSe recibió CTRL+C. Esperando que terminen los procesos en background...\n");
    exit_requested = 1;
}
```

```c
int main() {
    char input[256];
    char *args[20];
    pid_t pid;
    int background;

    // Configurar manejadores de señales
    signal(SIGCHLD, sigchld_handler); // Maneja la finalización de procesos hijos
    signal(SIGINT, sigint_handler);  // Maneja CTRL+C

    while (1) {
        // Mostrar el prompt
        printf("shell> ");
        fflush(stdout);

        // Leer la entrada del usuario
        if (fgets(input, sizeof(input), stdin) == NULL) break;

        // Quitar el salto de línea
        input[strcspn(input, "\n")] = '\0';

        // Comprobar si el usuario quiere salir
        if (strcmp(input, "salir") == 0 || exit_requested) break;

        // Parsear la entrada en argumentos
        int i = 0;
        char *token = strtok(input, " ");
        while (token != NULL) {
            args[i++] = token;
            token = strtok(NULL, " ");
        }
        args[i] = NULL;

        // Comprobar si el proceso debe ejecutarse en background
        background = (i > 0 && strcmp(args[i - 1], "&") == 0);
        if (background) {
            args[i - 1] = NULL; // Eliminar el "&" de los argumentos
        }
```

```
        // Crear un proceso hijo
        pid = fork();
        if (pid < 0) {
            perror("Error al crear el proceso");
            continue;
        } else if (pid == 0) {
            // Proceso hijo
            if (execvp(args[0], args) == -1) {
                perror("Error al ejecutar el comando");
                exit(EXIT_FAILURE);
            }
        } else {
            // Proceso padre
            if (!background) {
                // Si no es en background, esperar al hijo
                waitpid(pid, NULL, 0);
            } else {
                printf("Proceso en background iniciado con PID %d\n", pid);
            }
        }
    }

    // Esperar a que terminen todos los procesos en background antes de salir
    printf("Saliendo de la shell. Esperando procesos en background...\n");
    while (waitpid(-1, NULL, 0) > 0);

    printf("Todos los procesos terminados. Adiós.\n");
    return 0;
}
```

RESUMEN

1. **Señales**:
    - La señal SIGCHLD es manejada por sigchld_handler para limpiar procesos hijos terminados y evitar zombies.
    - La señal SIGINT es manejada por sigint_handler para permitir una salida controlada al recibir CTRL+C.
2. **Main**:
    - Muestra un prompt (shell>), lee comandos del usuario y los parsea.
    - Si el comando incluye "&", se ejecuta en background.
    - Los comandos se ejecutan en un proceso hijo mediante fork() y execvp().
    - El programa espera que terminen los procesos foreground, mientras que los procesos background son manejados con SIGCHLD.
3. **Salida**:
    - Cuando el usuario escribe "salir" o presiona CTRL+C, el programa espera a que todos los procesos en background finalicen antes de salir.

TEST

```
gcc -o ejercicio1 ejercicio1.c
./ejercicio1
```

```
shell> ls
ejercicio1      ejercicio1.c    links.txt
shell> salir
Saliendo de la shell. Esperando procesos en background...
Todos los procesos terminados. Adiós.
```

```
shell> ^C
Se recibió CTRL+C. Esperando que terminen los procesos en background...

Saliendo de la shell. Esperando procesos en background...
Todos los procesos terminados. Adiós.
```

EJERCICIO 2

Implementar un proceso al cual se le indique por línea de comando la cantidad de procesos a crear, todos los procesos a crear serán hermanos; cada uno de ellos retornará un valor entero distinto al proceso padre. Los procesos hijos quedan en un loop eterno, en una espera no activa, cuando recibe la señal SIGUSR1 o SIGINT el proceso hijo termina retornando un valor entero distinto al de sus hermanos. El proceso padre reportará por pantalla la sumatoria de los retornos de los procesos hijos creados. No se permitirá que existan procesos huérfanos o zombies.

```c
// Librerias necesarias
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <signal.h>
#include <sys/types.h>
#include <sys/wait.h>

pid_t *child_pids; // Arreglo para almacenar los PIDs de los procesos hijos
int child_index;   // Índice único para cada hijo

Codeium: Refactor | Explain | Generate Function Comment | ×
void signal_handler(int signo) {
    for (int i = 0; child_pids[i] != 0; i++)
        kill(child_pids[i], SIGUSR1); // Enviar SIGUSR1 a los hijos
}

Codeium: Refactor | Explain | Generate Function Comment | ×
void child_signal_handler(int signo) {
    exit(child_index); // Cada hijo retorna su índice único
}
```

```c
int main(int argc, char *argv[]) {
    if (argc != 2) {
        fprintf(stderr, "Uso: %s <cantidad_de_procesos>\n", argv[0]);
        return EXIT_FAILURE;
    }

    int num_children = atoi(argv[1]);
    if (num_children <= 0) {
        fprintf(stderr, "La cantidad de procesos debe ser un número positivo.\n");
        return EXIT_FAILURE;
    }

    child_pids = (pid_t *)calloc(num_children + 1, sizeof(pid_t));
    if (!child_pids) {
        perror("Error al reservar memoria");
        return EXIT_FAILURE;
    }

    signal(SIGINT, signal_handler); // Configurar manejador para SIGINT

    for (int i = 0; i < num_children; i++) {
        pid_t pid = fork();
        if (pid < 0) {
            perror("Error al crear el proceso");
            free(child_pids);
            return EXIT_FAILURE;
        } else if (pid == 0) {
            // Proceso hijo
            signal(SIGUSR1, child_signal_handler); // Manejador para SIGUSR1
            child_index = i + 1; // Índice único del hijo
            printf("Hijo %d creado con PID %d.\n", child_index, getpid());
            while (1) pause(); // Espera no activa
        } else {
            // Proceso padre
            child_pids[i] = pid; // Guardar PID del hijo
        }
    }
```

```c
    // Proceso padre: Esperar a los hijos y calcular la sumatoria de retornos
    int status, sum = 0;
    for (int i = 0; i < num_children; i++) {
        pid_t child_pid = waitpid(child_pids[i], &status, 0);
        if (WIFEXITED(status)) {
            int exit_code = WEXITSTATUS(status);
            printf("Hijo con PID %d terminó con código de salida %d.\n", child_pid, exit_code);
            sum += exit_code;
        }
    }

    printf("La sumatoria de los retornos de los hijos es: %d\n", sum);
    free(child_pids);
    printf("Proceso padre finalizado.\n");
    return EXIT_SUCCESS;
}
```

RESUMEN

1. **Procesos**:
    - El programa recibe como argumento el número de procesos a crear.
    - Crea n procesos hijos (hermanos) usando fork().
    - Cada hijo entra en un bucle infinito en espera no activa (pause()), esperando la señal SIGUSR1 para terminar.
2. **Señales**:
    - Padre: Maneja SIGINT (Ctrl+C) para enviar SIGUSR1 a todos los hijos, indicándoles que deben terminar.
    - Hijos: Manejan SIGUSR1 para salir de manera controlada retornando su código de salida.
3. **Evitar zombies**:
    - El proceso padre usa waitpid() para esperar a que terminen todos los hijos y recolecta sus códigos de salida.
4. **Salida**:
    - El padre calcula y muestra la sumatoria de los códigos de salida de los hijos.

TEST

```
gcc -o ejercicio2 ejercicio2.c
./ejercicio2 5
```

```
Hijo 1 creado con PID 14141.
Hijo 2 creado con PID 14142.
Hijo 3 creado con PID 14143.
Hijo 4 creado con PID 14144.
Hijo 5 creado con PID 14145.
^CHijo con PID 14141 terminó con código de salida 1.
Hijo con PID 14142 terminó con código de salida 2.
Hijo con PID 14143 terminó con código de salida 3.
Hijo con PID 14144 terminó con código de salida 4.
Hijo con PID 14145 terminó con código de salida 5.
La sumatoria de los retornos de los hijos es: 15
Proceso padre finalizado.
```

EJERCICIO 3

Implementación de una sincronización de los hilos A, B y C de forma tal, que la secuencia de ejecución y acceso a su sección crítica sea la siguiente: ABAC...
Detener el proceso luego de N iteraciones completas (el número N se ingresa por línea de comandos). Resolver la sincronización con variables Mutex (librería pthread).

```c
// Librerias necesarias
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>

// Mutex y variables de condición
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t cond_A = PTHREAD_COND_INITIALIZER;
pthread_cond_t cond_B = PTHREAD_COND_INITIALIZER;
pthread_cond_t cond_C = PTHREAD_COND_INITIALIZER;

int turn = 0; // Variable para controlar el turno (0: A1, 1: B, 2: A2, 3: C)
int iterations = 0; // Contador de iteraciones completas
int N; // Número de iteraciones

Codeium: Refactor | Explain | Generate Function Comment | ×
void *thread_A(void *arg) {
    while (1) {
        pthread_mutex_lock(&mutex);

        // Espera según su turno
        while (turn != 0 && turn != 2)
            pthread_cond_wait(&cond_A, &mutex);

        printf("Hilo A\n");

        if (turn == 0) {
            turn = 1; // A1 sigue con B
            pthread_cond_signal(&cond_B);
        } else if (turn == 2) {
            turn = 3; // A2 sigue con C
            pthread_cond_signal(&cond_C);
        }

        pthread_mutex_unlock(&mutex);
    }
    return NULL;
}
```

```c
void *thread_B(void *arg) {
    while (1) {
        pthread_mutex_lock(&mutex);

        // Espera según su turno
        while (turn != 1)
            pthread_cond_wait(&cond_B, &mutex);

        printf("Hilo B\n");
        turn = 2; // B sigue con A2
        pthread_cond_signal(&cond_A);

        pthread_mutex_unlock(&mutex);
    }
    return NULL;
}
```

Codeium: Refactor | Explain | Generate Function Comment | ×

```c
void *thread_C(void *arg) {
    while (1) {
        pthread_mutex_lock(&mutex);

        // Espera según su turno
        while (turn != 3)
            pthread_cond_wait(&cond_C, &mutex);

        printf("Hilo C\n");

        // Incrementa el contador de iteraciones al finalizar una secuencia completa
        iterations++;
        if (iterations >= N) {
            pthread_mutex_unlock(&mutex);
            break; // Salir del bucle
        }

        turn = 0; // C sigue con A1
        pthread_cond_signal(&cond_A);

        pthread_mutex_unlock(&mutex);
    }
    return NULL;
}
```

```c
int main(int argc, char *argv[]) {
    if (argc != 2 || (N = atoi(argv[1])) <= 0) {
        fprintf(stderr, "Uso: %s <número_de_iteraciones>\n", argv[0]);
        return EXIT_FAILURE;
    }

    pthread_t threadA, threadB, threadC;

    // Crear hilos
    pthread_create(&threadA, NULL, thread_A, NULL);
    pthread_create(&threadB, NULL, thread_B, NULL);
    pthread_create(&threadC, NULL, thread_C, NULL);

    // Esperar al hilo C (controlador principal de las iteraciones)
    pthread_join(threadC, NULL);

    // Cancelar los hilos restantes
    pthread_cancel(threadA);
    pthread_cancel(threadB);

    // Destruir mutex y variables condicionales
    pthread_mutex_destroy(&mutex);
    pthread_cond_destroy(&cond_A);
    pthread_cond_destroy(&cond_B);
    pthread_cond_destroy(&cond_C);

    printf("Ejecución completada con %d iteraciones.\n", N);
    return EXIT_SUCCESS;
}
```

RESUMEN

1. **Variables Mutex y Condicionales**:
    - **pthread_mutex_t mutex**: Garantiza que solo un hilo acceda a la variable compartida turn a la vez.
    - **pthread_cond_t cond_A, cond_B, cond_C**: Despiertan a los hilos correspondientes según el turno.
2. **Turnos**:
    - Se utiliza una variable turn para determinar qué hilo debe ejecutarse:
        - 0: Turno del hilo **A** (primera aparición).
        - 1: Turno del hilo **B**.
        - 2: Turno del hilo **A** (segunda aparición).
        - 3: Turno del hilo **C**.
3. **Finalización**:
    - Un contador iterations se incrementa en el hilo **C** al finalizar cada secuencia completa.
    - El hilo **C** decide cuándo se cumple el número de iteraciones.
    - El programa usa pthread_cancel para detener los hilos **A** y **B** al terminar el ciclo.

TEST

```
gcc -o ejercicio3 ejercicio3.c
./ejercicio3 3
```

```
Hilo A
Hilo B
Hilo A
Hilo C
Hilo A
Hilo B
Hilo A
Hilo C
Hilo A
Hilo B
Hilo A
Hilo C
Ejecución completada con 3 iteraciones.
```

EJERCICIO 4

Implementación de una sincronización con procesos independientes A, B y C de forma tal, que la secuencia de ejecución y acceso a su sección crítica sea la siguiente: ABAC... Detener el proceso luego de N iteraciones completas (el número N se ingresa por línea de comandos). Resolver la sincronización con: semáforos SVR4.

```c
// Librerias necesarias
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>
#include <sys/wait.h>

// Operaciones con semáforos
struct sembuf wait_op = {0, -1, 0};  // Operación WAIT
struct sembuf signal_op = {0, 1, 0}; // Operación SIGNAL

Codeium: Refactor | Explain | Generate Function Comment | ×
int main(int argc, char *argv[]) {
    if (argc != 2) {
        fprintf(stderr, "Uso: %s <número de iteraciones>\n", argv[0]);
        exit(EXIT_FAILURE);
    }

    int n = atoi(argv[1]); // Número de iteraciones
    if (n <= 0) {
        fprintf(stderr, "El número de iteraciones debe ser mayor a 0.\n");
        exit(EXIT_FAILURE);
    }

    // Crear semáforos (3 semáforos: A, B y C)
    int sem_id = semget(IPC_PRIVATE, 3, IPC_CREAT | 0666);
    if (sem_id == -1) {
        perror("Error al crear los semáforos");
        exit(EXIT_FAILURE);
    }

    // Inicializar los semáforos (A=1, B=0, C=0)
    semctl(sem_id, 0, SETVAL, 1); // Semáforo A
    semctl(sem_id, 1, SETVAL, 0); // Semáforo B
    semctl(sem_id, 2, SETVAL, 0); // Semáforo C
```

```c
// Crear procesos A, B y C
pid_t pidA = fork();
if (pidA == 0) {
    // Proceso A
    for (int i = 0; i < 2 * n; i++) {
        wait_op.sem_num = 0;    // Espera en el semáforo A
        semop(sem_id, &wait_op, 1);
        printf("Proceso A\n");
        signal_op.sem_num = (i % 2 == 0) ? 1 : 2; // Señal a B o C
        semop(sem_id, &signal_op, 1);
    }
    exit(EXIT_SUCCESS);
}

pid_t pidB = fork();
if (pidB == 0) {
    // Proceso B
    for (int i = 0; i < n; i++) {
        wait_op.sem_num = 1;    // Espera en el semáforo B
        semop(sem_id, &wait_op, 1);
        printf("Proceso B\n");
        signal_op.sem_num = 0; // Señal a A
        semop(sem_id, &signal_op, 1);
    }
    exit(EXIT_SUCCESS);
}

pid_t pidC = fork();
if (pidC == 0) {
    // Proceso C
    for (int i = 0; i < n; i++) {
        wait_op.sem_num = 2;    // Espera en el semáforo C
        semop(sem_id, &wait_op, 1);
        printf("Proceso C\n");
        signal_op.sem_num = 0; // Señal a A
        semop(sem_id, &signal_op, 1);
    }
    exit(EXIT_SUCCESS);
}
```

```c
    // Liberar recursos de los semáforos
    semctl(sem_id, 0, IPC_RMID, 0);
    printf("Ejecución completada con %d iteraciones.\n", n);

    return EXIT_SUCCESS;
}
```

RESUMEN

1. **Semáforos SVR4**:
    - Se utiliza un conjunto de tres semáforos para coordinar los procesos:
        - Semáforo **A (0)**: Controla el acceso del proceso **A**.
        - Semáforo **B (1)**: Controla el acceso del proceso **B**.
        - Semáforo **C (2)**: Controla el acceso del proceso **C**.
    - Los semáforos se inicializan de forma que **A** tiene acceso al inicio (valor inicial: 1), mientras que **B** y **C** están bloqueados (valor inicial: 0).
2. **Procesos independientes**:
    - Se crean tres procesos hijos (**A**, **B** y **C**) con fork(). Cada proceso:
        - Espera su turno usando el semáforo correspondiente.
        - Ejecuta su sección crítica (imprime su nombre).
        - Libera el semáforo del siguiente proceso en la secuencia.
3. **Secuencia de sincronización**:
    - El proceso **A**:
        - Se ejecuta primero.
        - Alterna entre liberar el semáforo de **B** y luego el de **C**.
    - Los procesos **B** y **C**:
        - Se ejecutan en su turno.
        - Siempre liberan el semáforo de **A** al finalizar.
4. **Finalización**:
    - El proceso padre espera la finalización de todos los hijos con wait().
    - Los recursos asociados a los semáforos se liberan mediante semctl(IPC_RMID).

TEST

```
gcc -o ejercicio4 ejercicio4.c
./ejercicio4 3
```

```
Proceso A
Proceso B
Proceso A
Proceso C
Proceso A
Proceso B
Proceso A
Proceso C
Proceso A
Proceso B
Proceso A
Proceso C
Ejecución completada con 3 iteraciones.
```

EJERCICIO 5

Implementación de una sincronización con procesos emparentados PadreA, HijoB y HijoC de forma tal, que la secuencia de ejecución y acceso a su sección crítica sea la siguiente: PadreA HijoB PadreA HijoC...
Detener el proceso luego de N iteraciones completas (el número N se ingresa por línea de comandos). HijoB e HijoC son hermanos. Resolver la sincronización de la forma que a Ud. le parezca más apropiada (no usar señales).

```c
// Librerias necesarias
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/wait.h>

Codeium: Refactor | Explain | Generate Function Comment | ×
int main(int argc, char *argv[]) {
    if (argc != 2) {
        fprintf(stderr, "Uso: %s <N_iteraciones>\n", argv[0]);
        return EXIT_FAILURE;
    }

    int N = atoi(argv[1]);
    if (N <= 0) {
        fprintf(stderr, "El número de iteraciones debe ser un entero positivo.\n");
        return EXIT_FAILURE;
    }

    // Pipes para sincronización
    int pipeAB[2], pipeBA[2], pipeAC[2], pipeCA[2];
    if (pipe(pipeAB) == -1 || pipe(pipeBA) == -1 || pipe(pipeAC) == -1 || pipe(pipeCA) == -1) {
        perror("Error al crear pipes");
        return EXIT_FAILURE;
    }
```

```c
pid_t pidB = fork(); // Crear HijoB
if (pidB == 0) {
    // Proceso HijoB
    close(pipeAB[1]); // Cerrar escritura de pipeAB
    close(pipeBA[0]); // Cerrar lectura de pipeBA
    close(pipeAC[0]); // No usado por HijoB
    close(pipeAC[1]); // No usado por HijoB
    close(pipeCA[0]); // No usado por HijoB
    close(pipeCA[1]); // No usado por HijoB

    for (int i = 0; i < N; i++) {
        char signal;
        read(pipeAB[0], &signal, 1); // Espera señal del PadreA
        printf("HijoB\n");
        write(pipeBA[1], "B", 1); // Envía señal de vuelta al PadreA
    }

    close(pipeAB[0]);
    close(pipeBA[1]);
    exit(EXIT_SUCCESS);
}

pid_t pidC = fork(); // Crear HijoC
if (pidC == 0) {
    // Proceso HijoC
    close(pipeAC[1]); // Cerrar escritura de pipeAC
    close(pipeCA[0]); // Cerrar lectura de pipeCA
    close(pipeAB[0]); // No usado por HijoC
    close(pipeAB[1]); // No usado por HijoC
    close(pipeBA[0]); // No usado por HijoC
    close(pipeBA[1]); // No usado por HijoC

    for (int i = 0; i < N; i++) {
        char signal;
        read(pipeAC[0], &signal, 1); // Espera señal del PadreA
        printf("HijoC\n");
        write(pipeCA[1], "C", 1); // Envía señal de vuelta al PadreA
    }

    close(pipeAC[0]);
    close(pipeCA[1]);
    exit(EXIT_SUCCESS);
}
```

```c
    // Proceso PadreA
    close(pipeAB[0]); // Cerrar lectura de pipeAB
    close(pipeBA[1]); // Cerrar escritura de pipeBA
    close(pipeAC[0]); // Cerrar lectura de pipeAC
    close(pipeCA[1]); // Cerrar escritura de pipeCA

    for (int i = 0; i < N; i++) {
        printf("PadreA\n");
        write(pipeAB[1], "A", 1); // Señala a HijoB
        char signal;
        read(pipeBA[0], &signal, 1); // Espera señal de HijoB

        printf("PadreA\n");
        write(pipeAC[1], "A", 1); // Señala a HijoC
        read(pipeCA[0], &signal, 1); // Espera señal de HijoC
    }

    close(pipeAB[1]);
    close(pipeBA[0]);
    close(pipeAC[1]);
    close(pipeCA[0]);

    // Esperar a que terminen los hijos
    waitpid(pidB, NULL, 0);
    waitpid(pidC, NULL, 0);

    printf("Ejecución completada con %d iteraciones.\n", N);
    return EXIT_SUCCESS;
}
```

RESUMEN

1. **Estructura:**
    - PadreA: Es el proceso principal que coordina la ejecución.
    - HijoB: Segundo en la secuencia, sincroniza con el PadreA a través del pipe pipeAB y devuelve señal usando pipeBA.
    - HijoC: Último en la secuencia, sincroniza con el PadreA a través del pipe pipeAC y devuelve señal usando pipeCA.
2. **Comunicación entre procesos:**
    - Se utilizan pipes unidireccionales para intercambiar señales entre procesos:
        - pipeAB: Señala al HijoB que es su turno.
        - pipeBA: HijoB devuelve la señal al PadreA.
        - pipeAC: Señala al HijoC que es su turno.
        - pipeCA: HijoC devuelve la señal al PadreA.
3. **Secuencia de sincronización:**
    - El PadreA imprime su mensaje y envía una señal a HijoB a través de pipeAB.
    - HijoB responde con su mensaje y devuelve una señal al PadreA mediante pipeBA.
    - El PadreA imprime nuevamente su mensaje y envía una señal a HijoC a través de pipeAC.
    - HijoC responde con su mensaje y devuelve una señal al PadreA mediante pipeCA.
    - Este ciclo se repite hasta completar las N iteraciones.
4. **Finalización:**
    - El PadreA espera a que ambos hijos terminen (waitpid) y cierra los pipes antes de finalizar.
    - Los hijos terminan de forma controlada tras completar su participación en todas las iteraciones.

TEST

```
gcc -o ejercicio5 ejercicio5.c
./ejercicio5 3
```

```
PadreA
HijoB
PadreA
HijoC
PadreA
HijoB
PadreA
HijoC
PadreA
HijoB
PadreA
HijoC
Ejecución completada con 3 iteraciones.
```

EJERCICIO 6

Realizar un programa que reciba por línea de comandos un comando a ejecutar y sus argumentos. El programa va crear un proceso hijo con dicho comando y el proceso padre leerá la salida del comando. Para lograr esto, cree un pipe entre proceso padre e hijo y utilice la función dup() o dup2() para duplicar la salida en el pipe y luego leer la salida usando el pipe. No está permitido usar la función popen() que resume o simplifica la técnica propuesta. No generar procesos huérfanos ni zombies.

```c
// Librerias necesarias
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>

Codeium: Refactor | Explain | Generate Function Comment | ×
int main(int argc, char *argv[]) {
    if (argc < 2) {
        fprintf(stderr, "Uso: %s <comando> [argumentos]\n", argv[0]);
        return EXIT_FAILURE;
    }

    int pipefd[2]; // pipefd[0]: lectura, pipefd[1]: escritura

    // Crear el pipe
    if (pipe(pipefd) == -1) {
        perror("Error al crear el pipe");
        return EXIT_FAILURE;
    }

    pid_t pid = fork();

    if (pid < 0) {
        perror("Error al hacer fork");
        return EXIT_FAILURE;
    }

    if (pid == 0) {
        // Proceso hijo
        close(pipefd[0]); // Cerrar el extremo de lectura del pipe

        // Redirigir la salida estándar (stdout) al pipe
        if (dup2(pipefd[1], STDOUT_FILENO) == -1) {
            perror("Error al redirigir la salida estándar");
            exit(EXIT_FAILURE);
        }

        // Cerrar el extremo de escritura original del pipe
        close(pipefd[1]);
```

```c
        // Ejecutar el comando con sus argumentos
        execvp(argv[1], &argv[1]);
        perror("Error al ejecutar el comando");
        exit(EXIT_FAILURE);
    } else {
        // Proceso padre
        close(pipefd[1]); // Cerrar el extremo de escritura del pipe

        char buffer[1024];
        ssize_t bytes_read;

        // Leer la salida del pipe y escribirla en la salida estándar del padre
        while ((bytes_read = read(pipefd[0], buffer, sizeof(buffer) - 1)) > 0) {
            buffer[bytes_read] = '\0'; // Null-terminar la cadena
            printf("%s", buffer);
        }

        close(pipefd[0]); // Cerrar el extremo de lectura del pipe

        // Esperar a que termine el proceso hijo
        int status;
        waitpid(pid, &status, 0);

        if (WIFEXITED(status)) {
            printf("\nEl comando terminó con código de salida %d.\n", WEXITSTATUS(status));
        } else {
            printf("\nEl comando no terminó correctamente.\n");
        }
    }

    return EXIT_SUCCESS;
}
```

RESUMEN

1. **Entrada de Comando:**
    - El programa espera recibir al menos un argumento: el nombre del comando a ejecutar, seguido de sus argumentos opcionales.
2. **Creación del Pipe:**
    - Se utiliza pipe() para crear un pipe que permite la comunicación entre el proceso padre y el hijo. Este pipe consta de dos extremos: uno para la escritura y otro para la lectura.
3. **Creación del Proceso Hijo:**
    - Se crea un proceso hijo mediante fork(). Si el fork() es exitoso, el proceso hijo ejecutará el comando proporcionado.
4. **Redirección de la Salida:**
    - En el proceso hijo, se redirige la salida estándar (stdout) al extremo de escritura del pipe utilizando dup2(). Esto asegura que cualquier salida generada por el comando se envíe al pipe en lugar de la consola.
5. **Ejecución del Comando:**
    - El proceso hijo ejecuta el comando usando execvp(). Si la ejecución del comando es exitosa, la salida del comando se escribe en el pipe.
6. **Lectura de la Salida en el Proceso Padre:**
    - En el proceso padre, se cierra el extremo de escritura del pipe y se leen los datos desde el extremo de lectura. La salida del comando se imprime en la consola.
7. **Finalización:**
    - El proceso padre espera a que el proceso hijo termine utilizando waitpid(). Esto asegura que no se creen procesos huérfanos ni zombies.
    - Al final, el programa imprime el código de salida del proceso hijo para indicar si el comando se ejecutó con éxito.

TEST

```
gcc -o ejercicio6 ejercicio6.c
./ejercicio6 ls
```

```
ejercicio1
ejercicio1.c
ejercicio2
ejercicio2.c
ejercicio3
ejercicio3.c
ejercicio4
ejercicio4.c
ejercicio5
ejercicio5.c
ejercicio6
ejercicio6.c
links.txt

El comando terminó con código de salida 0.
```

EJERCICIO 7

Implementar un servidor (basado en hilos) usando sockets tcp/ip. El cliente (use telnet) envía un comando Linux al servidor, el servidor lo ejecuta y devuelve la salida del comando al cliente. La interacción entre cliente y servidor termina cuando el cliente envía el comando “salir”. Reusar el código del ejercicio anterior.

```c
// Librerias necesarias
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <pthread.h>
#include <arpa/inet.h>
#include <sys/types.h>
#include <sys/wait.h>

#define PORT 8080          // Puerto en el que escucha el servidor
#define BUFFER_SIZE 1024   // Tamaño del buffer para la comunicación

// Estructura para pasar parámetros a los hilos
typedef struct {
    int client_socket;
} thread_arg;

// Función que ejecuta un comando y envía la salida al cliente
Codeium: Refactor | Explain | X
void* handle_client(void* arg) {
    thread_arg* t_arg = (thread_arg*)arg; // Cast de argumento
    int client_socket = t_arg->client_socket;
    free(t_arg); // Liberar memoria del argumento

    char buffer[BUFFER_SIZE];
    ssize_t read_size;

    while (1) {
        // Leer el comando del cliente
        memset(buffer, 0, BUFFER_SIZE);
        read_size = read(client_socket, buffer, BUFFER_SIZE - 1);
        if (read_size <= 0) {
            perror("Error al leer del socket");
            break;
        }

        // Eliminar el salto de línea al final del comando
        buffer[strcspn(buffer, "\n")] = 0; // Elimina el salto de línea
        buffer[strcspn(buffer, "\r")] = 0; // Elimina retorno de carro (si está presente)
```

```c
// Verificar si el cliente desea salir
if (strcmp(buffer, "salir") == 0) {
    printf("Cliente desconectado.\n");
    break;
}

// Crear un pipe para ejecutar el comando
int fd[2];
if (pipe(fd) == -1) {
    perror("Error al crear pipe");
    break;
}

pid_t pid = fork();
if (pid < 0) {
    perror("Error al crear el proceso hijo");
    break;
}

if (pid == 0) {
    // Proceso hijo
    close(fd[0]); // Cerrar el extremo de lectura del pipe
    dup2(fd[1], STDOUT_FILENO); // Redirigir stdout al pipe
    dup2(fd[1], STDERR_FILENO); // Redirigir stderr al pipe
    close(fd[1]); // Cerrar el extremo de escritura del pipe

    // Ejecutar el comando
    char* args[] = {"/bin/sh", "-c", buffer, NULL}; // Usar /bin/sh para ejecutar comandos
    execv(args[0], args);
    perror("Error al ejecutar el comando"); // Mensaje de error
    exit(EXIT_FAILURE);
} else {
    // Proceso padre
    close(fd[1]); // Cerrar el extremo de escritura del pipe
    wait(NULL); // Esperar a que el hijo termine
```

```
            // Leer la salida del comando desde el pipe
            memset(buffer, 0, BUFFER_SIZE);
            ssize_t bytes_read = read(fd[0], buffer, BUFFER_SIZE - 1);
            if (bytes_read > 0) {
                write(client_socket, buffer, bytes_read); // Enviar la salida al cliente
            } else {
                write(client_socket, "No output from command.\n", 24); // Mensaje si no hay salida
            }

            close(fd[0]); // Cerrar el extremo de lectura del pipe
        }
    }

    close(client_socket); // Cerrar el socket del cliente
    return NULL;
}
```

Codeium: Refactor | Explain | Generate Function Comment | ×

```
int main() {
    int server_socket, client_socket;
    struct sockaddr_in server_addr, client_addr;
    socklen_t addr_size = sizeof(client_addr);

    // Crear el socket del servidor
    server_socket = socket(AF_INET, SOCK_STREAM, 0);
    if (server_socket < 0) {
        perror("Error al crear socket");
        exit(EXIT_FAILURE);
    }

    // Configurar la dirección del servidor
    server_addr.sin_family = AF_INET;
    server_addr.sin_addr.s_addr = INADDR_ANY;
    server_addr.sin_port = htons(PORT);

    // Enlazar el socket
    if (bind(server_socket, (struct sockaddr*)&server_addr, sizeof(server_addr)) < 0) {
        perror("Error al enlazar el socket");
        close(server_socket);
        exit(EXIT_FAILURE);
    }
```

```c
    // Escuchar conexiones entrantes
    if (listen(server_socket, 5) < 0) {
        perror("Error al escuchar");
        close(server_socket);
        exit(EXIT_FAILURE);
    }

    printf("Servidor escuchando en el puerto %d...\n", PORT);

    while (1) {
        // Aceptar una nueva conexión
        client_socket = accept(server_socket, (struct sockaddr*)&client_addr, &addr_size);
        if (client_socket < 0) {
            perror("Error al aceptar conexión");
            continue;
        }

        printf("Cliente conectado.\n");

        // Crear un hilo para manejar al cliente
        pthread_t thread_id;
        thread_arg* t_arg = malloc(sizeof(thread_arg)); // Reservar memoria para los argumentos
        t_arg->client_socket = client_socket; // Asignar el socket del cliente

        if (pthread_create(&thread_id, NULL, handle_client, t_arg) != 0) {
            perror("Error al crear hilo");
            close(client_socket);
            free(t_arg);
        }
        pthread_detach(thread_id); // Desvincular el hilo
    }

    close(server_socket); // Cerrar el socket del servidor
    return 0;
}
```

RESUMEN

1. **Estructura:**
    - Socket del Servidor: El programa crea un socket TCP que escucha en un puerto específico (8080 en este caso) para aceptar conexiones entrantes.
    - Manejo de Conexiones: Cuando un cliente se conecta, se crea un nuevo hilo para manejar la comunicación con ese cliente, lo que permite que el servidor acepte múltiples conexiones simultáneamente.
2. **Proceso de Ejecución:**
    - Recepción de Comandos: El servidor lee el comando enviado por el cliente a través de un socket.
    - Ejecución del Comando: Utiliza fork() para crear un proceso hijo que ejecuta el comando usando execvp(). La salida del comando se redirige al cliente a través de un pipe.
    - Devolución de Resultados: El servidor envía la salida del comando de vuelta al cliente. Si el comando es "salir", el servidor cierra la conexión.
3. **Finalización:**
    - La interacción entre el cliente y el servidor termina cuando el cliente envía el comando "salir", momento en el cual el servidor cierra la conexión y termina el hilo asociado.

TEST

```
gcc -o ejercicio7 ejercicio7.c -lpthread
./ejercicio7
```

TERMINAL SERVIDOR

```
Servidor escuchando en el puerto 8080...
Cliente conectado.
Cliente desconectado.
```

TERMINAL CLIENTE (telnet localhost 8080)

```
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^]'.
ls
ejercicio1
ejercicio1.c
ejercicio2
ejercicio2.c
ejercicio3
ejercicio3.c
ejercicio4
ejercicio4.c
ejercicio5
ejercicio5.c
ejercicio6
ejercicio6.c
ejercicio7
ejercicio7.c
links.txt
salir
Connection closed by foreign host.
```