2023

Memoria

PRÁCTICA 2

PAULA CASTILLEJO BRAVO

1. Compilación:

La fórmula para compilar un archivo en lenguaje C, conociendo de antemano que se trabaja en una terminal en el sistema operativo linux, consiste en el comando gcc -I include/ -o manager src/manager.c, en donde la opción -I indica los directorios donde han de buscarse bibliotecas mientras –o compila un programa en C y genera un archivo ejecutable.

2. Ejecución:

Una vez poseyendo el ejecutable del programa, la ejecución de este es un paso más simple desde terminal, ubicándonos en el directorio donde está el ejecutable y escribiendo, en el siguiente orden, ./manager 3 10 . Estás dos cifras son reemplazables por otras, indicando el número de teléfonos y el número de líneas, respectivamente.

3. Discusión:

Una vez se domina los procedimientos de compilación y ejecución del programa, se proporcionará constancia acerca de las pequeñas conclusiones[1] durante todo el proyecto abarcado en las siguientes tres semanas.

3.1. MakeFile

En primera instancia, un archivo makefile es un conjunto de herramientas make para construir y crear ejecutables y objetos en base de los archivos dependientes .c y .h, ubicados en los directorios /src e /include respectivamente. Los nombres lógicos principales que lo componen serían:

```
1    DIROBJ := obj/
2    DIREXE := exec/
3    DIRHEA := include/
4    DIRSRC := src/

5    CFLAGS := -I$(DIRHEA) -c -Wall -ggdb
6    LDFLAGS := -lpthread -lrt
7    CC := gcc
```

Desde la línea 1 hasta la 4 se instancian los directorios en su correspondiente variable, en las líneas 5 y 6 indicamos las banderas Wall, la cual reportará errores y advertencias, y lpthread, dónde se pretende cargar las bibliotecas (existentes o inventadas), y en la línea 7 indicamos el compilador. Ahora trataremos con las reglas implícitas en el documento:

```
8    all : dirs manager telefono linea

9    dirs:
10       mkdir -p $(DIROBJ) $(DIREXE)

11   manager: $(DIROBJ)manager.o $(DIROBJ)semaforoI.o $(DIROBJ)memoriaI.o
```

[1] Al tratarse este una práctica realizada de forma individual, no se ha generado ninguna discusión o debate como tal, pero sí existe la ayuda por parte de otros compañeros de clase y del profesorado de la asignatura en cuestión.

```
12          $(CC) -o $(DIREXE)$@ $^ $(LDFLAGS)

13      telefono: $(DIROBJ)telefono.o $(DIROBJ)semaforoI.o $(DIROBJ)memoriaI.o
14          $(CC) -o $(DIREXE)$@ $^ $(LDFLAGS)

15      linea: $(DIROBJ)linea.o $(DIROBJ)semaforoI.o $(DIROBJ)memoriaI.o
16          $(CC) -o $(DIREXE)$@ $^ $(LDFLAGS)

17      $(DIROBJ)%.o: $(DIRSRC)%.c
18          $(CC) $(CFLAGS) $^ -o $@

20      clean :
21          rm -rf *~ core $(DIROBJ) $(DIREXE) $(DIRHEA)*~ $(DIRSRC)*~
```

En la línea 8, all se usa para enumerar todos los subobjetivos necesarios para construir el proyecto. En las líneas 11, 13 y 15 se forman los ejecutables, siendo manager el del proceso principal. En la línea 17 se procede a establecer los ficheros objetos. En la línea 20, la función clean sirve para limpiar todas las reglas anteriores.

3.2. Manager.c

Redactar los diez métodos del programa no es moco de pavo, pero nos centraremos en las consideraciones de los más cruciales y complicados. En este caso, iniciar_tabla_procesos() y esperar_procesos().

A la hora de iniciar las tablas de los procesos, al inicio se realizaba la suma de los procesos de líneas y teléfonos cuando estos en realidad se emplean por separado. Por tanto, intuímos que habrá dos bucles en donde los pid de cada una de las tablas de los procesos se inicializará a 0.

```
for (i = 0; i < g_lineas_Processes; i++)
{
    g_process_lineas_table[i].pid = 0;
}

for (i = 0; i < g_telefonos_Processes; i++)
{
    g_process_telefonos_table[i].pid = 0;
}
```

En contraste, cuando esperamos por los procesos, únicamente terminaremos exclusivamente el proceso línea (ya que el proceso teléfono morirá una vez que lo hayan hecho todas las líneas primero, pero esta funcionalidad se especifica en el módulo terminar_procesos()).

```
while (n_num_processes > 0)
{
```

```
                pid = wait(NULL);
                for (i = 0; i < g_lineas_Processes; i++)
                {
                        if (pid == g_process_lineas_table[i].pid)
                        {
                                printf("\t[MANAGER] Proceso %s terminado [%d]...\n",
        g_process_lineas_table[i].clase, g_process_lineas_table[i].pid);
                                g_process_lineas_table[i].pid = 0;
                                n_num_processes--;
                                break;
                        }
                }
        }
```

3.3. Telefono.c

Reducido al empleo de un solo método, el archivo telefono.c se ocupa de la espera, resepción y finalización de una llamada. Sin embargo, se detallarán los pasos vinculados con la inclusión de semaforos y memoria compartida ya que surgen varios conflictos en la colocación de los primeros.

Hay que tener en cuenta, sin lugar a dudas, la paralización de las líneas una vez el teléfono detecta la primera (tengamos en cuenta que se ocupará un teléfono por línea). Después, se ingresará a la sección de funcionalidad del módulo e impediremos el paso de otros archivos con la pausa del semáforo mutex; el cual se reactivará una vez el programa halla modificado el número de llamadas en espera (como el teléfono ha adquirido una línea, hay que reducir la cifra de las que esperan). Si no cerramos el flujo, podría corromperse el valor de las llamadas en espera ya que es una variable de memoria compartida. Esto se exhibe en el siguiente fragmento de código:

```
signal_sem(telefonos);
wait_sem(lineas);


wait_sem(mutex);
consultar_var(num_llamadas_espera, &valorEspera);
valorEspera--;
modificar_var(num_llamadas_espera, valorEspera);
signal_sem(mutex);
```

3.4. Líneas.c

Para concluir, se hablará del archivo linea.c. En este proceso se inicia una llamada, la cual espera hasta ser recibida por un teléfono. El método main es el único que añade valor en este archivo, mas no se genera demasiados cambios con referencia al archivo telefonos.c . La variable de memoria compartida vuelve a poder hacer estragos si no está bien controlada, por tanto, mencionaremos nuevamente la parada y reanudación del semáforo mutex en cuyo “interior” se optimizará el cambio de información del número de llamadas en espera (aumentando según estas van surgiendo y no son recibidas por nada). A continuación, durante la espera, se cerrará el semáforo de los teléfonos para señalar que dicha llamada ha sido recibida; y una vez esta es desviada, activamos la posibilidad de una nueva línea (la cual estaba detenida desde el archivo telefonos.c).

```
//Aumenta las llamadas en espera
wait_sem(mutex);
consultar_var(num_llamadas_espera, &valorEspera);
valorEspera++;
modificar_var(num_llamadas_espera, valorEspera);
signal_sem(mutex);


// Espera telefono libre
printf("Linea [%d] esperando telefono libre...Nº Llamadas en espera: %d\n", pid, valorEspera);
wait_sem(telefonos);


// Lanza la llamada
printf("Linea [%d] desviando llamada a un telefono...\n", pid);
signal_sem(lineas);
```