

Actividades P4

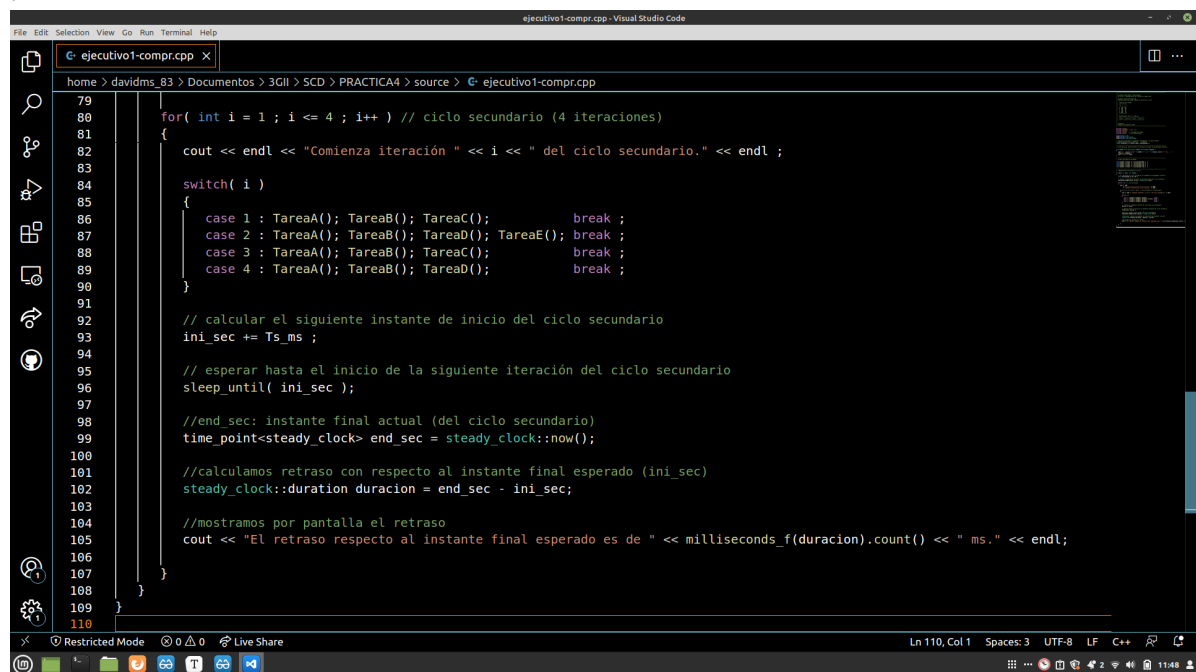
Actividad 1

En esta actividad tenemos que añadir una nueva funcionalidad al código **ejecutivo1.cpp**.

El programa debe mostrar (cada vez que acaba un ciclo secundario) el retraso del instante final actual respecto al instante final esperado. La comprobación se tiene que hacer al final del bucle, inmediatamente después de **sleep_until**.

Para hacer todo esto, añadimos dentro del bucle (justo después de **sleep_until**), un **time_point** llamado **end_sec** (siempre **steady_clock**), y usamos la función **now()**. Así obtenemos el instante final actual.

Para calcular el retraso declaramos una variable **duration** y calculamos **end_sec - ini_sec** (que es el instante final esperado, el parámetro de **sleep_until**). Esta variable la mostramos por pantalla de la forma que se indica en el código.



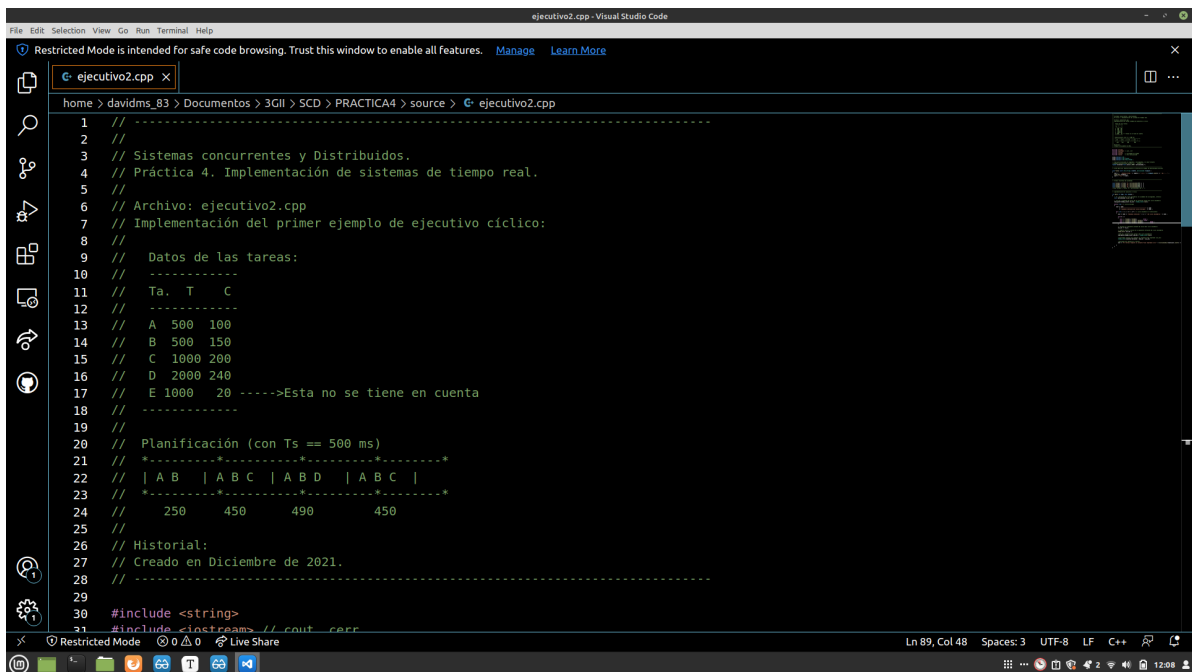
```
79
80 for( int i = 1 ; i <= 4 ; i++ ) // ciclo secundario (4 iteraciones)
81 {
82     cout << endl << "Comienza iteración " << i << " del ciclo secundario." << endl ;
83
84     switch( i )
85     {
86         case 1 : TareaA(); TareaB(); TareaC();           break ;
87         case 2 : TareaA(); TareaB(); TareaD(); TareaE(); break ;
88         case 3 : TareaA(); TareaB(); TareaC();           break ;
89         case 4 : TareaA(); TareaB(); TareaD();           break ;
90     }
91
92     // calcular el siguiente instante de inicio del ciclo secundario
93     ini_sec += Ts_ms ;
94
95     // esperar hasta el inicio de la siguiente iteración del ciclo secundario
96     sleep_until( ini_sec );
97
98     //end_sec: instante final actual (del ciclo secundario)
99     time_point<steady_clock> end_sec = steady_clock::now();
100
101     //calculamos retraso con respecto al instante final esperado (ini_sec)
102     steady_clock::duration duracion = end_sec - ini_sec;
103
104     //mostramos por pantalla el retraso
105     cout << "El retraso respecto al instante final esperado es de " << milliseconds_f(duracion).count() << " ms." << endl;
106
107 }
108
109
110 }
```

Actividad 2

En este ejercicio debemos diseñar una planificación para las tareas y restricciones que se nos muestra en la última diapositiva de la Práctica 4.

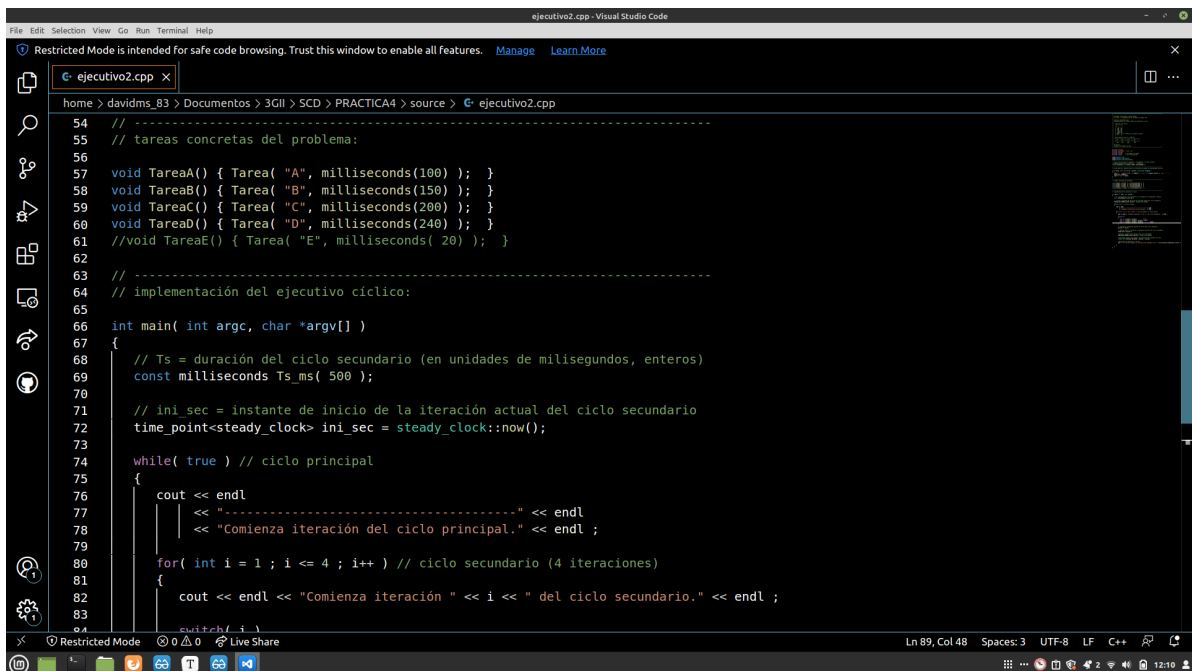
A partir de la tabla podemos obtener que el **ciclo principal** dura **2000ms** ya que es **mcm(500,500,1000,2000)**.

En esta captura se da una posible planificación del sistema con una **duración del ciclo secundario de 500 ms** (realizamos 4 iteraciones, ahí tendríamos los 2000ms anteriores).



```
1 // -----
2 //
3 // Sistemas concurrentes y Distribuidos.
4 // Práctica 4. Implementación de sistemas de tiempo real.
5 //
6 // Archivo: ejecutivo2.cpp
7 // Implementación del primer ejemplo de ejecutivo cíclico:
8 //
9 // Datos de las tareas:
10 // -----
11 // Ta. T C
12 // -----
13 // A 500 100
14 // B 500 150
15 // C 1000 200
16 // D 2000 240
17 // E 1000 20 ----->Esta no se tiene en cuenta
18 // -----
19 //
20 // Planificación (con Ts == 500 ms)
21 // *-----*-----*-----*
22 // | A B | A B C | A B D | A B C |
23 // *-----*-----*-----*
24 // 250 450 490 450
25 //
26 // Historial:
27 // Creado en Diciembre de 2021.
28 // -----
29
30 #include <string>
31 #include <iostream> // cout, cerr
```

En cuanto al código, solo hay que cambiar la duración del ciclo secundario y el número de tareas que hay así como el tiempo que duermen. También cambiamos la planificación según nuestro esquema.



```
54 // -----
55 // tareas concretas del problema:
56
57 void TareaA() { Tarea( "A", milliseconds(100) ); }
58 void TareaB() { Tarea( "B", milliseconds(150) ); }
59 void TareaC() { Tarea( "C", milliseconds(200) ); }
60 void TareaD() { Tarea( "D", milliseconds(240) ); }
61 //void TareaE() { Tarea( "E", milliseconds( 20) ); }
62
63 // -----
64 // implementación del ejecutivo cíclico:
65
66 int main( int argc, char *argv[] )
67 {
68     // Ts = duración del ciclo secundario (en unidades de milisegundos, enteros)
69     const milliseconds Ts_ms( 500 );
70
71     // ini_sec = instante de inicio de la iteración actual del ciclo secundario
72     time_point<steady_clock> ini_sec = steady_clock::now();
73
74     while( true ) // ciclo principal
75     {
76         cout << endl
77              << "-----" << endl
78              << "Comienza iteración del ciclo principal." << endl ;
79
80         for( int i = 1 ; i <= 4 ; i++ ) // ciclo secundario (4 iteraciones)
81         {
82             cout << endl << "Comienza iteración " << i << " del ciclo secundario." << endl ;
83
84             switch( i )
```

```
72 time_point<steady_clock> ini_sec = steady_clock::now();
73
74 while( true ) // ciclo principal
75 {
76     cout << endl
77     << "-----" << endl
78     << "Comienza iteración del ciclo principal." << endl ;
79
80     for( int i = 1 ; i <= 4 ; i++ ) // ciclo secundario (4 iteraciones)
81     {
82         cout << endl << "Comienza iteración " << i << " del ciclo secundario." << endl ;
83
84         switch( i )
85         {
86             case 1 : TareaA(); TareaB(); break ;
87             case 2 : TareaA(); TareaB(); TareaC(); break ;
88             case 3 : TareaA(); TareaB(); TareaD(); break ;
89             case 4 : TareaA(); TareaB(); TareaC(); break ;
90         }
91
92         // calcular el siguiente instante de inicio del ciclo secundario
93         ini_sec += Ts_ms ;
94
95         // esperar hasta el inicio de la siguiente iteración del ciclo secundario
96         sleep_until( ini_sec );
97
98         //end sec: instante final actual (del ciclo secundario)
99         time_point<steady_clock> end_sec = steady_clock::now();
100
101         //calculamos retraso con respecto al instante final esperado (ini_sec)
```

En cuanto a las cuestiones a responder:

1. El mínimo tiempo de espera que queda al final de las iteraciones del ciclo secundario sería el del tercer ciclo, puesto que son **500-490=10 ms**, frente a los demás, que son todos mayores.
2. Si la tarea D tuviese un tiempo de cómputo de 250 ms seguiría siendo planificable con nuestra solución. Lo único que cambia es que la iteración 3 en vez de durar 490 ms dura **500 ms**.