作業系統導論 LAB 3

系所：通訊三　　學號 :409430030　　姓名:翁佳煌

<實驗工具>

1. Keil MDK。
2. Nu-Link_Keil_Driver。
3. NUC100_Series_BSP_CMSIS。
4. FreeRTOS_project
5. NUC140 開發板一塊。
6. USB cable

<實驗過程與方法>

Part A:

首先看到下圖 1，50~53 行分別先宣告任務函數，分別哲學家 A~D。
56~57 行的 xSemaphoreHandle Mutex 這是一個互斥信號量的聲明，用於保護對按鍵的訪問。xSemaphoreHandle Chopstick[4]這是一個計數信號量的聲明，表示四根筷子的可用數量。
再來 60~69 行為自定義的函示宣告，下方將會詳細介紹函式的功能。

```
46  /* hardware initail function declaration */
47  void GPIO_Init(void);
48
49  /* task function declaration */
50  static void philosopher_A( void *p );
51  static void philosopher_B( void *p );
52  static void philosopher_C( void *p );
53  static void philosopher_D( void *p );
54
55  /* Semaphore function declaration*/
56  xSemaphoreHandle Mutex;  //for scankey
57  xSemaphoreHandle Chopstick[4];
58
59
60  void hungry(int);
61  void take_a_break(void);
62  int ScanKey_protected(void);
63
64
65  //by myself
66  void pick_chopstick(int philosopher_index);
67  void release_chopstick(int philosopher_index);
68  void eating_rice(int philosopher_index);
69  void thinking(int philosopher_index);
70  //by myself
```

▲圖 1

下圖 2，77~83 行，主要是在設定 GPIO 的模式，將對應的腳位為輸出，並呼叫 GPIO_Init() 函數，進行進一步的 GPIO 初始化。

84 行，Mutex = xSemaphoreCreateMutex()負責創建一個互斥信號量，並將其分配給 Mutex 變量。

91~120 行，則是創建任務，並啟動 FreeRTOS 任務調度器。

```
74  int main(void)
75  {
76        /* Unlock protected registers */
77      SYS_UnlockReg();
78      /* Lock protected registers */
79      SYS_LockReg();
80      GPIO_SetMode(PA,BIT12,GPIO_PMD_OUTPUT);
81      GPIO_SetMode(PA,BIT13,GPIO_PMD_OUTPUT);
82      GPIO_SetMode(PA,BIT14,GPIO_PMD_OUTPUT);
83      GPIO_Init();
84      Mutex = xSemaphoreCreateMutex();
85
86      /* create the non-sharable resource's semaphore */
87      for(i = 0; i < 4; i++)
88        Chopstick[i] = xSemaphoreCreateCounting(1,1);
89
90
91      xTaskCreate( philosopher_A,         /* The function
92              NULL,                 /* The text name assigned
93              configMINIMAL_STACK_SIZE,   /* The size of
94              NULL,               /* The parameter passed t
95              1,                /* The priority assigned to
96              NULL );
97
98      xTaskCreate( philosopher_B,
99              NULL,
100             configMINIMAL_STACK_SIZE,
101             NULL,
102             1,
103             NULL );
104
105     xTaskCreate( philosopher_C,
106             NULL,
107             configMINIMAL_STACK_SIZE,
108             NULL,
109             1,
110             NULL );
111
112     xTaskCreate( philosopher_D,
113             NULL,
114             configMINIMAL_STACK_SIZE,
115             NULL,
116             1,
117             NULL );
118
119     vTaskStartScheduler();
120     while(1);
121 }
```

▲圖 2

下圖 3，當按下對應的 Scankey 時候，我們舉哲學家 A 的 123~132 行為例子，也就是當按下 NUC140 鍵盤的 1 位置時，代表哲學家 A 想要吃飯了，也就會進到 hungry 這個函式，其他哲學家 B~D 以此類推。

```
123  static void philosopher_A( void *p )
124  {
125    while(1)
126    {
127      /* Constantly check if BUTTON-1 is pressed */
128      if(ScanKey_protected() == 1)
129        hungry(0);
130    }
131  }
132
133  static void philosopher_B( void *p )
134  {
135    while(1)
136    {
137      if(ScanKey_protected() == 2)
138        hungry(1);
139    }
140  }
141
142  static void philosopher_C( void *p )
143  {
144    while(1)
145    {
146      if(ScanKey_protected() == 3)
147        hungry(2);
148    }
149  }
150
151  static void philosopher_D( void *p )
152  {
153    while(1)
154    {
155      if(ScanKey_protected() == 4)
156        hungry(3);
157    }
158  }
159
```

▲圖 3

當按下鍵盤後，便會進入下圖 4 的 hungry 函式，代表目前該哲學家肚子餓了，也就會進入第 216 行的 pick_chopstick 函數。

```
213  void hungry(int philo)
214  {
215    printf("philo %d is hungry\n",philo);
216    pick_chopstick(philo);
217  }
```

▲圖 4

下圖 5，首先第 172~173 行，會根據傳入的 philosopher_index 計算出左手筷子和右手筷子的索引。
176 行使用 xSemaphoreTake() 函數等待左手筷子的可用性，並將其取走，如果左手筷子不可用，這個函數會一直等待，直到筷子可用。
在成功取走左手筷子後，會執行 take_a_break()進行 500 毫秒的休息。

休息完後到 184 行，使用 xSemaphoreTake() 函數等待右手筷子的可用性，並將其取走，如果右手筷子不可用，這個函數會一直等待，直到筷子可用。

在成功取走右手筷子後，會進入 188 行的 eating_rice() 函數，進行進食。該函數負責控制哲學家進食一段時間和 LED 開關。

```
170 void pick_chopstick(int philosopher_index)
171 {
172     int left_chopstick_index = philosopher_index;
173     int right_chopstick_index = (philosopher_index + 1) % 4;
174
175     /* Pick up left chopstick */
176     xSemaphoreTake(Chopstick[left_chopstick_index], portMAX_DELAY);
177     printf("Success pick up left chopstick\n");
178
179     printf("Take a break for 500ms");
180     /*Take a break*/
181     take_a_break();
182
183     /* Pick up right chopstick */
184     xSemaphoreTake(Chopstick[right_chopstick_index], portMAX_DELAY);
185     printf("Success pick up right chopstick\n");
186
187     /*start eating*/
188     eating_rice(philosopher_index);
189 }
```

```
220 void take_a_break()
221 {
222     vTaskDelay(500);
223 }
224
```

▲圖 5

下圖 6，進到 eating_rice 後，表示哲學家開始進食，因此通過設置 GPIO 達到控制紅色 LED 的目的，將相應的 LED 打開，也就是會根據傳入的 philo 參數，計算出對應的 GPIO 編號，並將其設置為低電平，從而打開對應的 LED。

並在 208 行執行 vTaskDelay(1000)函數，使該任務延遲 1000 毫秒。這個延遲的時間模擬了哲學家進食的時間。

通過再次設置 GPIO 達到控制紅色 LED 的目的，將相應的 LED 關閉，根據傳入的 philo 參數，計算出對應的 GPIO 編號，並將其設置為高電平，從而關閉對應的 LED。

最後在第 210 行調用 release_chopstick() 函數，釋放哲學家手中的筷子，這將使其他哲學家能夠繼續取用這些筷子。

```
203 void eating_rice(int philo){
204   printf("start eating rice for 1000ms\n");
205
206   /*Eating rice*/
207   GPIO_PIN_DATA(2, 12 + philo) = 0; // turn the corresponding LED on
208   vTaskDelay(1000);
209   GPIO_PIN_DATA(2, 12 + philo) = 1; // turn the corresponding LED off
210   release_chopstick(philo);         // release chopsticks
211 }
212
```

▲圖 6

下圖 7，191~201 為釋放筷子的函式，首先透過 193~194 行計算出左手和右手筷子的索引，接下來，通過 197 和 200 行的 xSemaphoreGive 函數釋放左手和右手的筷子。

```
191 void release_chopstick(int philosopher_index)
192 {
193     int left_chopstick_index = philosopher_index;
194     int right_chopstick_index = (philosopher_index + 1) % 4;
195
196     /* Pick up left chopstick */
197     xSemaphoreGive(Chopstick[left_chopstick_index]);
198
199     /* Pick up right chopstick */
200     xSemaphoreGive(Chopstick[right_chopstick_index]);
201 }
```

▲圖 7

Part B:

ParB 的要求為請在已完成的 Part A 程式上實作，讓其中一個 deadlock 條件永遠不會發生，以實現 Deadlock Prevention。

因此我針對下圖 8，pick_chopstick 的函數，去拿掉 no preemtion 這個 dedalock 的條件。

這函式被修改以處理一個特殊情況。當一位哲學家試圖拿起右手的筷子時，如果該筷子已被其他哲學家佔用，並且等待的時間超過 300 個 tick，則將放下已拿起的左手筷子並進行等待後重新嘗試。

這樣就可成功避免 deadlock 的問題。

```
238  void pick_chopstick(int philosopher_index)
239  {
240      int left_chopstick_index = philosopher_index;
241      int right_chopstick_index = (philosopher_index + 1) % 4;
242
243      a:
244      /* Pick up left chopstick */
245      xSemaphoreTake(Chopstick[left_chopstick_index], portMAX_DELAY);
246      printf("Success pick up left chopstick\n");
247
248      printf("Take a break for 500ms");
249      /*Take a break*/
250      take_a_break();
251
252      /* Pick up right chopstick */
253      if( xSemaphoreTake(Chopstick[right_chopstick_index], 300)==pdTRUE ){
254          printf("Success pick up right chopstick\n");
255            /*start eating*/
256          eating_rice(philosopher_index);
257      }
258      else{                                                    //no preemtion :
259          xSemaphoreGive(Chopstick[left_chopstick_index]);
260          vTaskDelay(1000);
261          goto a;
262      }
263  }
```

▲圖 8

<問題與討論>

1. 在 Part A 的程式，何種情況會發生 Deadlock?

我發現可能會發生死結（Deadlock）的情況是在哲學家就餐的過程中。以下是可能導致死結的情況：

1.位哲學家都同時拿起自己左邊的筷子。假設所有哲學家都同時執行到 xSemaphoreTake(Chopstick[left_chopstick_index], portMAX_DELAY) 這行程式碼，並且所有筷子都被其他哲學家佔用。在這種情況下，每位哲學家都等待左邊的筷子釋放，但其他哲學家都在等待其他筷子釋放，導致死結。

2.每位哲學家都同時拿起自己右邊的筷子。假設所有哲學家都同時執行到 xSemaphoreTake(Chopstick[right_chopstick_index], portMAX_DELAY) 這行程式碼，並且所有筷子都被其他哲學家佔用。在這種情況下，每位哲學家都等待右邊的筷子釋放，但其他哲學家都在等待其他筷子釋放，導致死結。

2. Deadlock 的發生與按下 keypad 的順序有關嗎?

是的，Deadlock 的發生與按下 keypad 的順序有關。在這段程式碼中，每位哲學家的任務（task）都在一個無窮迴圈中等待按下特定的 keypad 按鈕才會進行相應的操作。

如果按下按鈕的順序不當，例如，同時或快速地按下按鍵 1~4，就可能導致 Deadlock 的發生。Deadlock 通常發生在多個執行緒或任務（task）需要同時佔用多個共享資源，而這些資源彼此相互等待，導致所有執行緒或任務都無法繼續執行的情況。

3. 在 part B 你以哪個方法實作 Deadlock Prevention? 為什麼這樣就能避免 Deadlock?

我在 pick_chopstick 這個函數中，拿掉 no preemtion 這個 dedalock 的條件。
這函式被修改為當一位哲學家試圖拿起右手的筷子時，如果該筷子已被其他哲學家佔用，並且等待的時間超過一定時間，則放下目前拿起的左手筷子並進行等待後重新嘗試。

<心得與收穫>

這是最後一次的 LAB，它讓我更加了解 Deadlock，Deadlock 是多個進程或執行緒之間的一種阻塞狀態，每個進程都在等待其他進程所擁有的資源，從而無法繼續執行。在這個問題中，哲學家需要兩支筷子才能就餐，但如果每個哲學家都先拿起左邊的筷子，那麼就會發生 Deadlock。

以及學習解決 Deadlock 的策略，要解決這問題，我們可以採用多種策略，如避免、預防、檢測和恢復。在哲學家就餐問題中，我是使用「no preemption」策略，即一旦哲學家無法獲得兩支筷子，就放下已經拿起的筷子，等待一段時間後重新嘗試。這樣可以避免一位哲學家佔用兩支筷子，使其他哲學家無法繼續執行。

總而言之，通過解決哲學家吃飯的問題並避免 Deadlock，我深入了解了作業系統的概念和解決策略，並學習了在程式碼設計中應用同步機制和適當的資源管理的重要性。

<參考資料>

https://ithelp.ithome.com.tw/articles/10206625

https://hackmd.io/@ExcitedMail/Hy6Ot7s7Y

http://wiki.csie.ncku.edu.tw/embedded/freertos

https://www.keil.com/demo/eval/arm.htm