**嵌入式作業系統實作**

**Embedded OS Implementation**

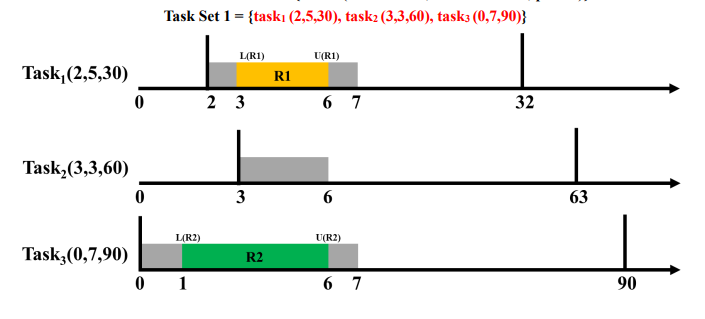
**PA\_3**

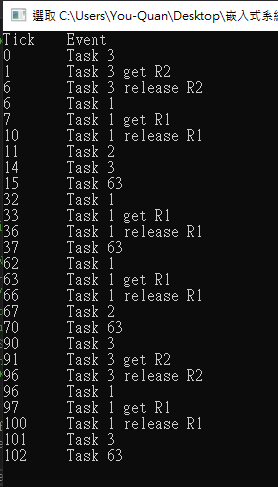
指導教授: 陳雅淑 教授

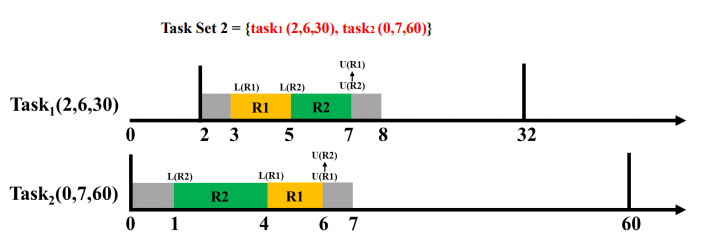
課程學生: M10907314張祐銓

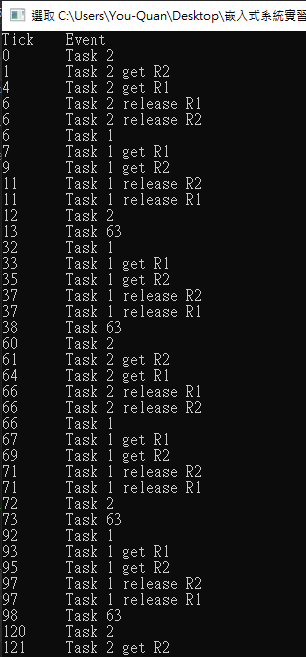
**[ PART I] NPCS Implementation [50%]**

* The screenshot result (with the given format) of the two task sets. (Time tick 0- 100) (10%)

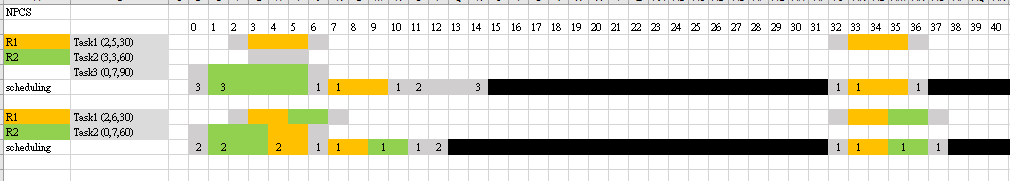








* A report that describes your implementation, including scheduling results of two task sets, modified functions, data structure, etc. (please ATTACH the screenshot of the code and MARK the modified part). (40%)



上次RMS的實現條列說明:

1. 在OS\_TCB中新增變數

INT32U begin\_ready\_time; //紀錄task變ready的時間點

INT32U response; //反應時間

INT32U arrival; //到達時間

INT32U execution; //執行時間

INT32U period; //週期

INT32U job\_id; //工作次數

1. 在OSTimeTick中去察看目前的ReadyTable有哪些Task正在Ready狀態，把所有正在Ready且不是目前正在執行的task的所有task的response都加一，因為task正在Ready卻沒有被執行表示被延後了一個tick，response一開始初始值是週期內的執行時間，所以task工作執行只要沒有執行滿response的時間就會一直卡在while迴圈內。
2. Task執行完會執行OS\_Sched()去切換給下個Task，所以在OS\_Sched()中顯示完成的字串。
3. 當TimeTick中斷產生之後會進OSIntExit()把低優先權的task中斷給高優先權的task，所以在這裡顯示中斷狀態的字串。
4. 當有task已經快要完成時有可能會被其他優先權高的task給搶占，所以會在OSIntExit()增加判斷程式迴避掉那次的context switch，讓快要做完的task先完成它的工作。
5. 當有一個task完成了一個周期內的工作後要做下個週期的工作時因為不會進入OS\_Sched()做context switch，所以在OSTimeDly(0)時表示task的反應時間等於週期，在OSTimeDly內去print完成的字串。
6. 判斷是否有task已經miss deadline，只要在OSTimeTick中去判斷reponse是否大於period，表示說反應時間已經大於週期就是miss deadline。

NPCS的實現條列說明:

1. 在OS\_TCB中新增變數

INT32U deadline; //EDF的最後期限

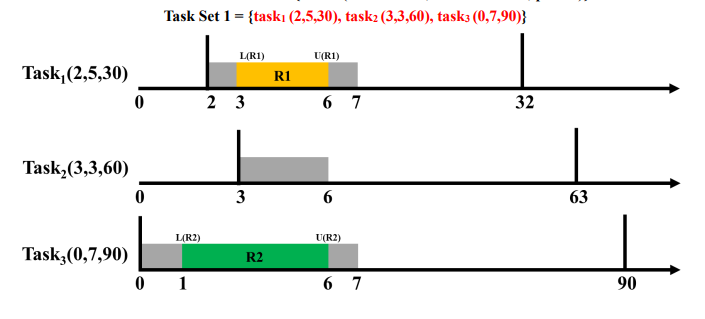
INT32U runtime; //執行了多久

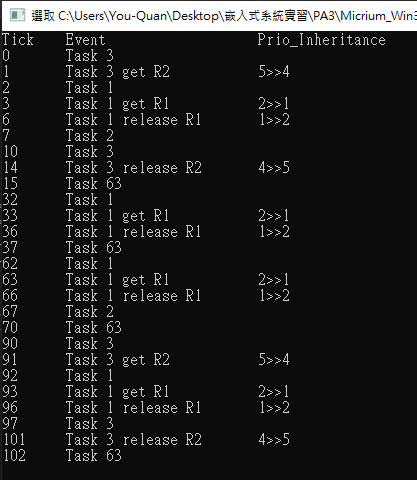
1. 在ucos\_ii.h中新增OS\_EXT OS\_EVENT \*Jack\_event\_Tbl[OS\_LOWEST\_PRIO + 1u] ，記錄所有mutex event。
2. 在OSTimeTick()中當前工作的task的runtime就加1。
3. 在OSIntExit()和OS\_Sched()中會做context switch，在做之前先做判斷，判斷Jack\_event\_Tbl中當前task是否有占用任何資源，若有占用到資源就迴避不做context switch。

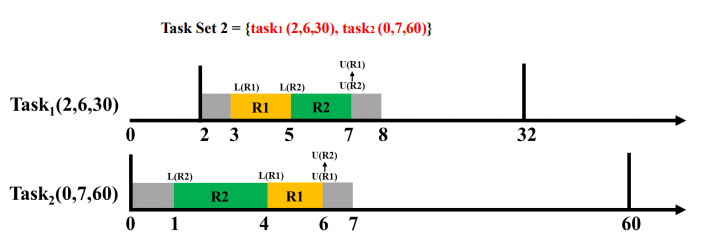
* 程式截圖部分跟CPP的程式截圖整合在一起

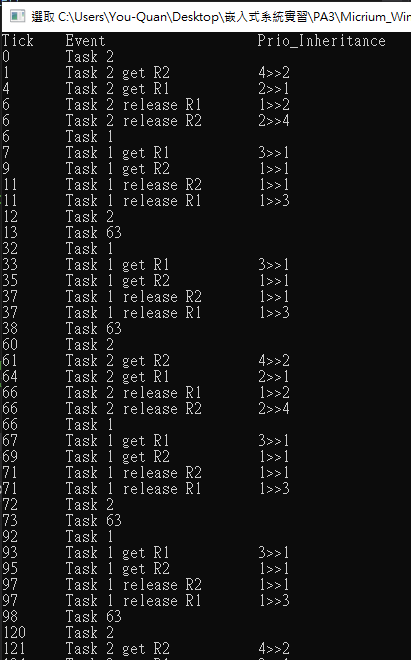
**[ PART II] CPP Implementation [40%]**

* The screenshot result (with the given format) of the two task sets. (Time tick 0- 100) (10%)

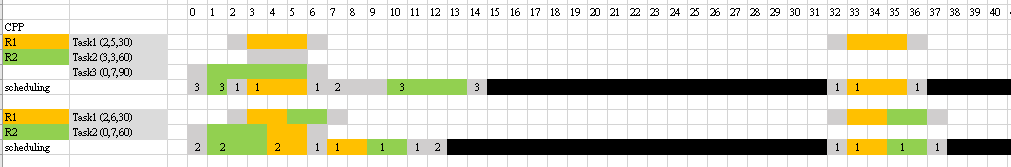








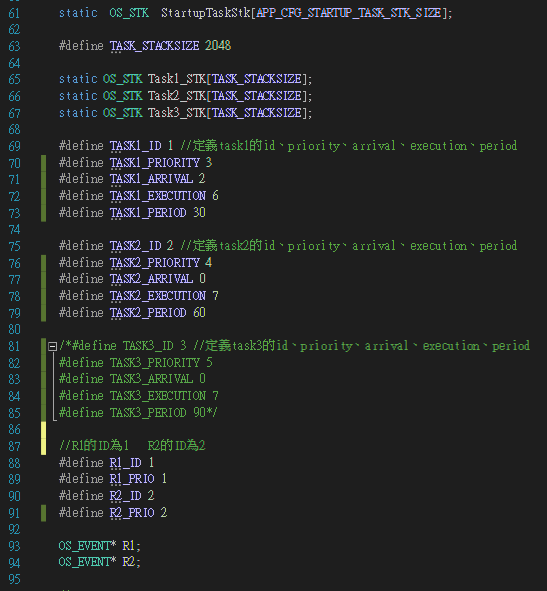
* A report that describes your implementation, including scheduling results of two task sets, modified functions, data structure, etc. (please ATTACH the screenshot of the code and MARK the modified part). (30%)

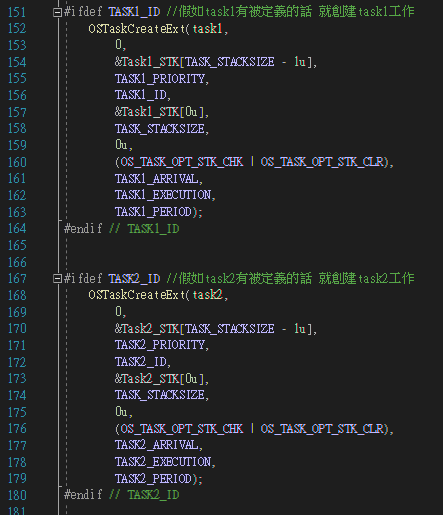


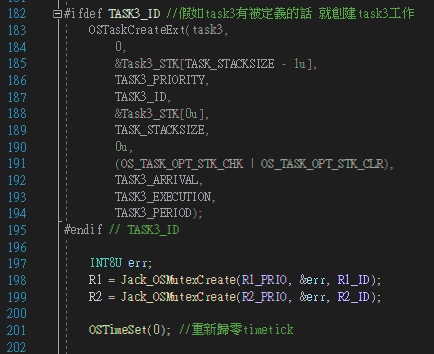
CPP實現方式條列說明:

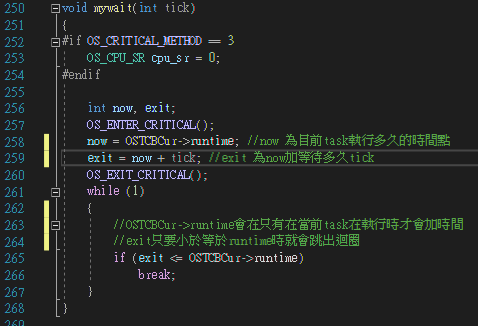
1. 根據NPCS再進行後續的追加實現CPP。
2. 在OSIntExit()和OS\_Sched()中會做context switch，在做之前先做判斷，判斷Jack\_event\_Tbl中當前task占用的資源中，若有占用的資源的優先權高於下個要切換的task的話，就迴避不做這次的context switch。

Main.c

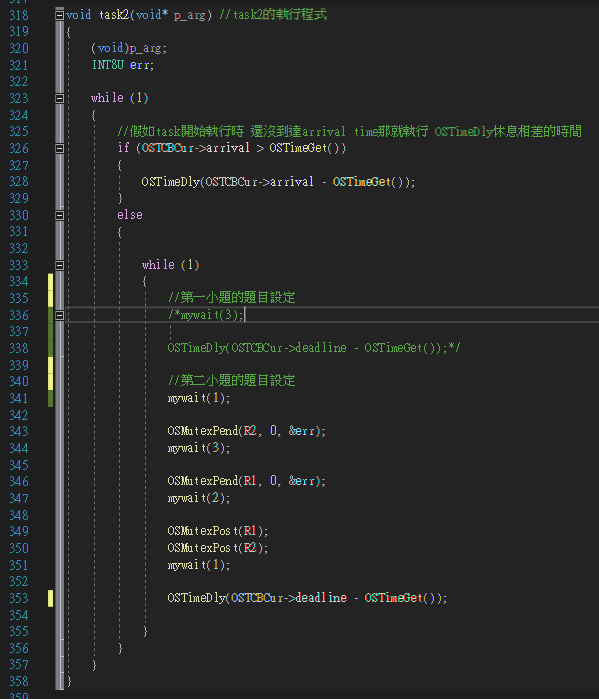


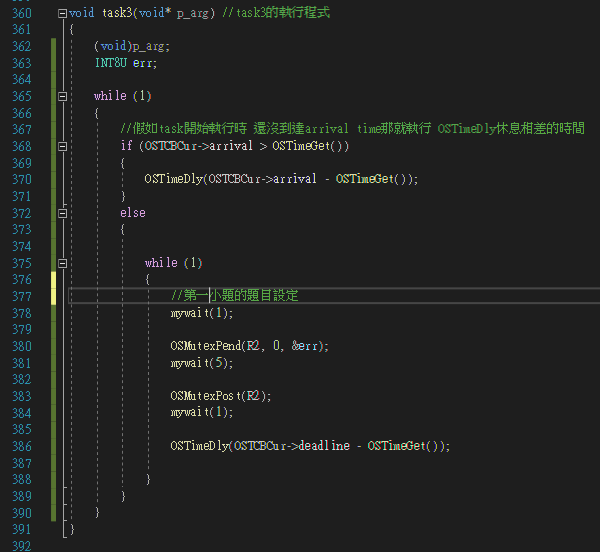




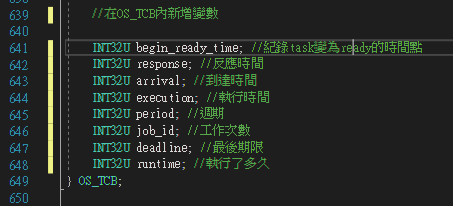


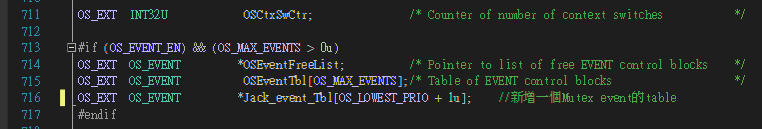


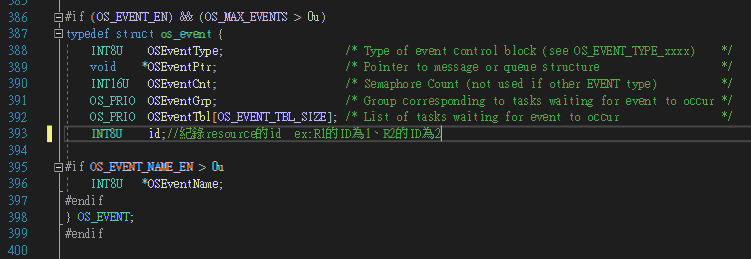




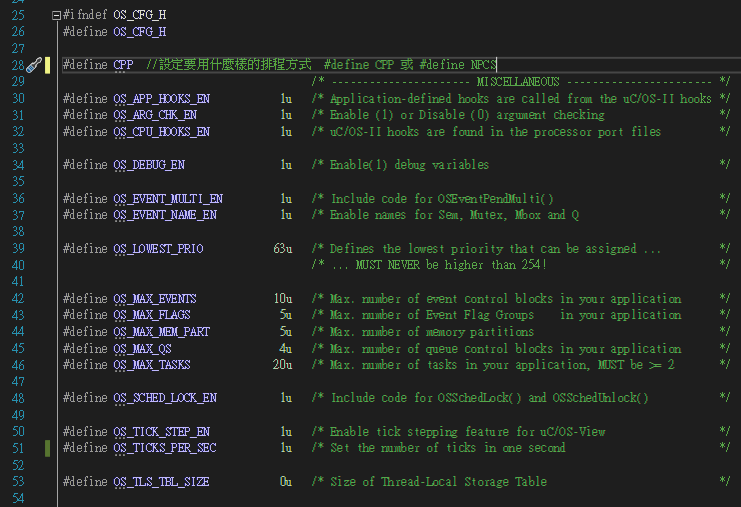
ucos\_ii.h



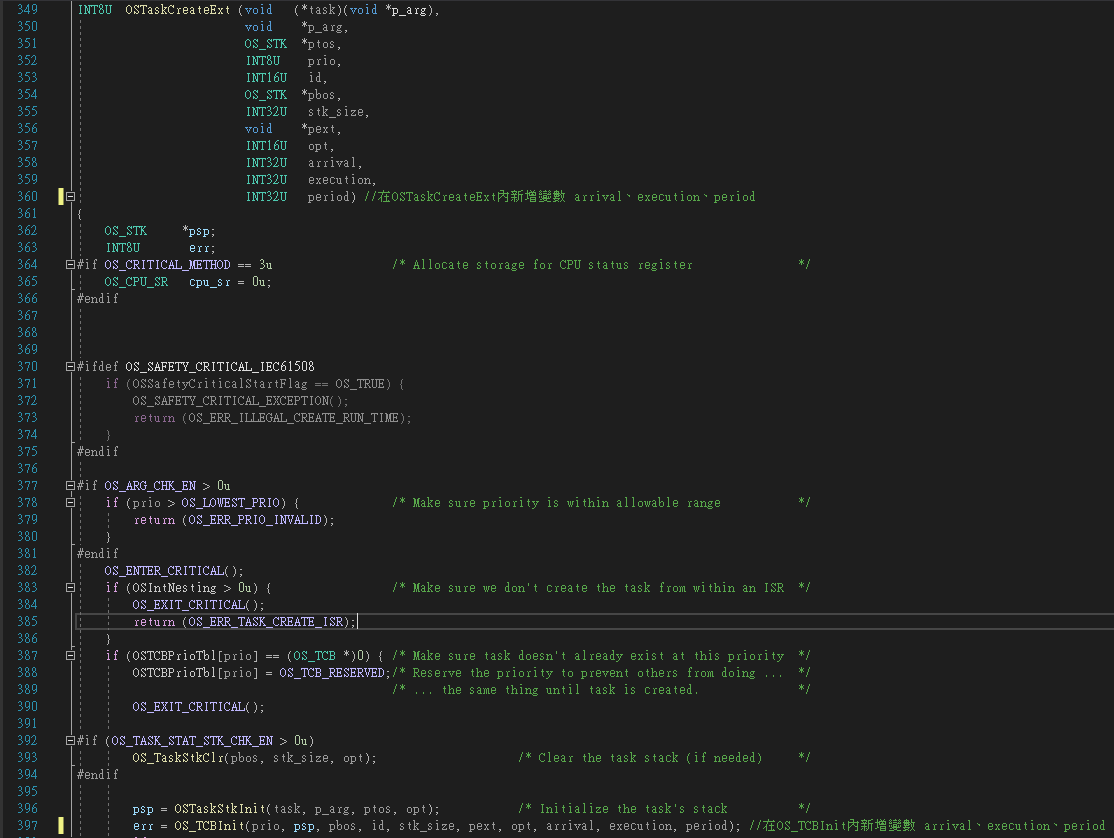




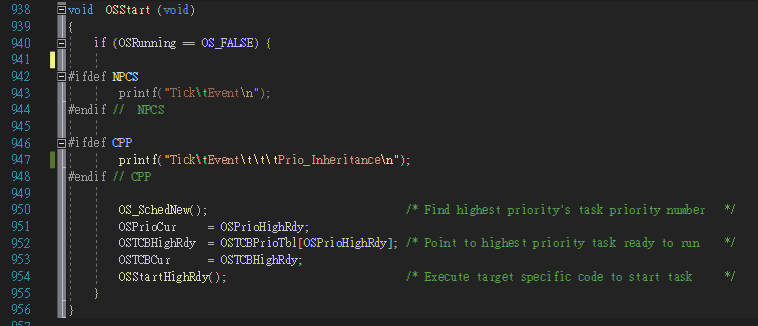
os\_cfg.h

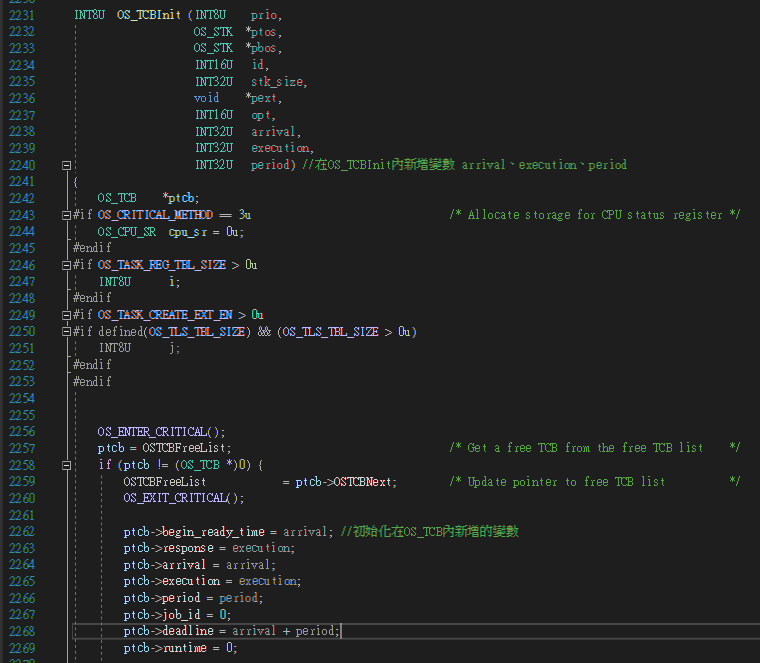


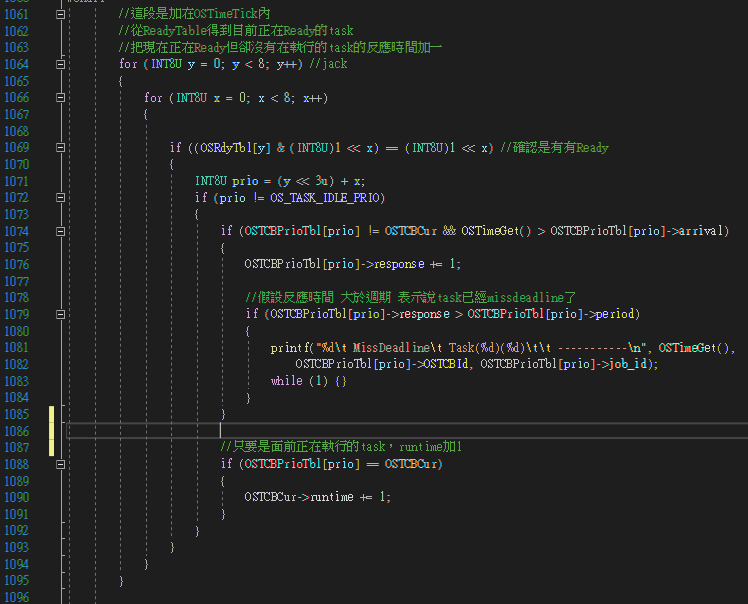
os\_task.c



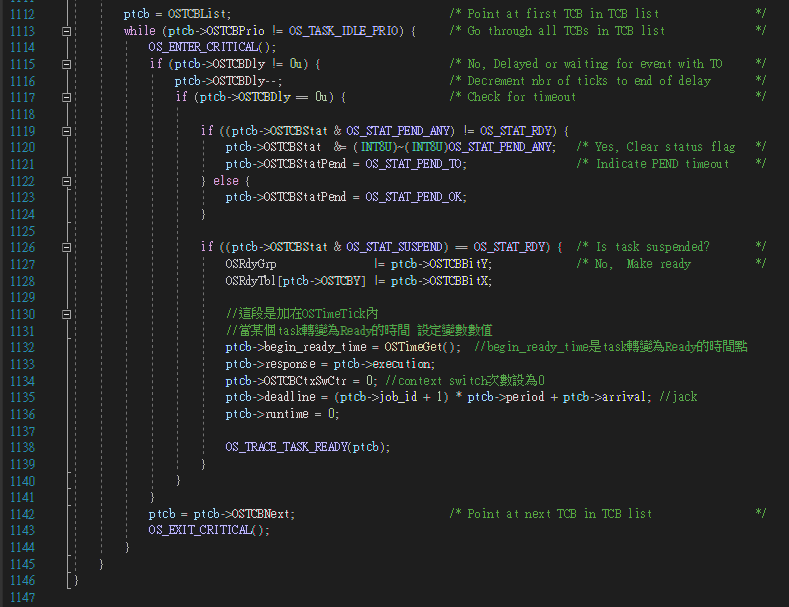
os\_core.c



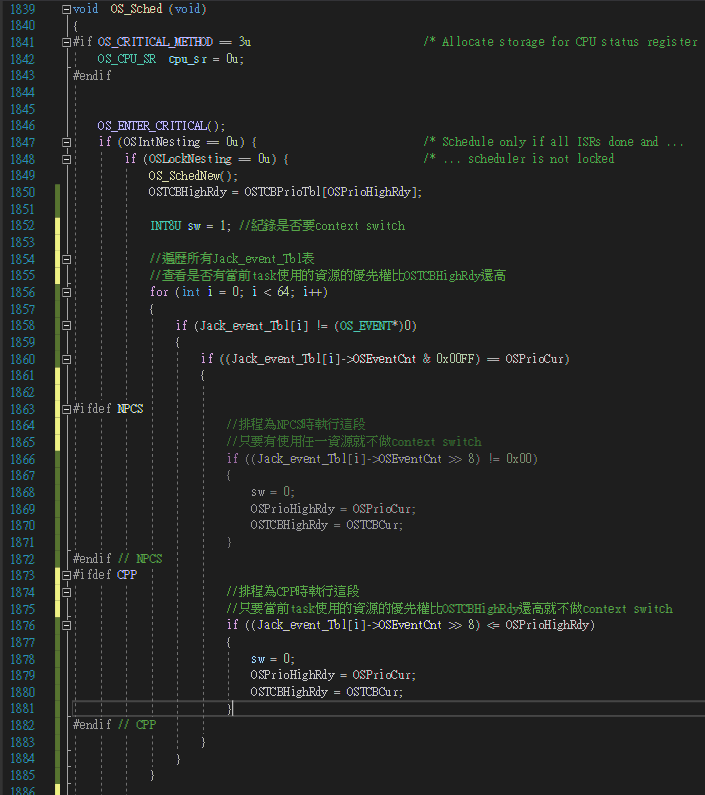


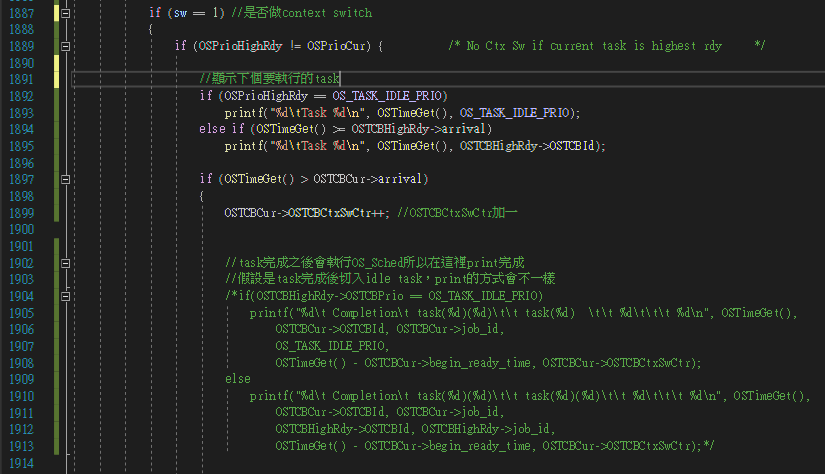


OSTimeTick()

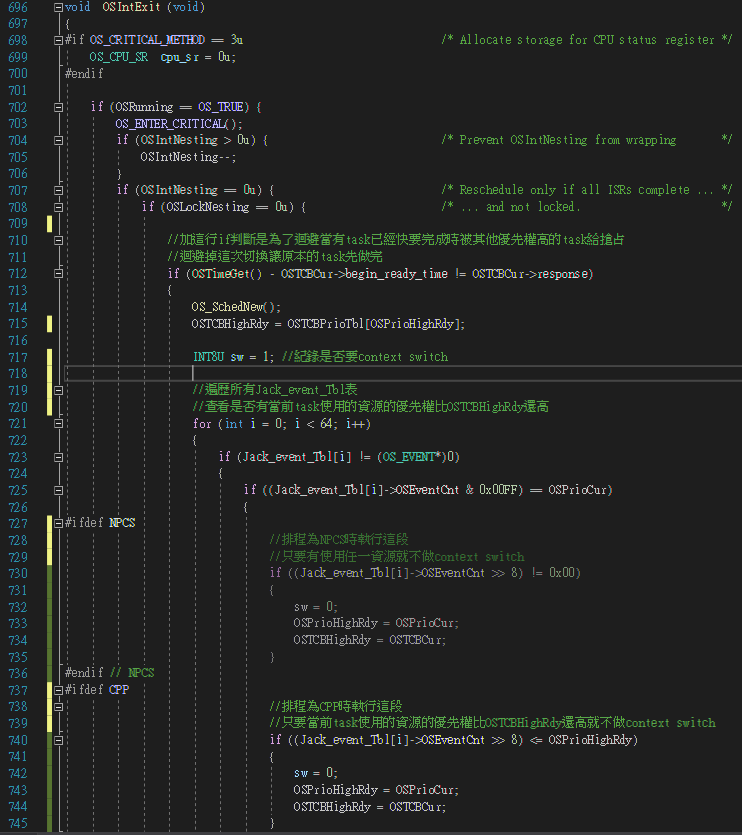


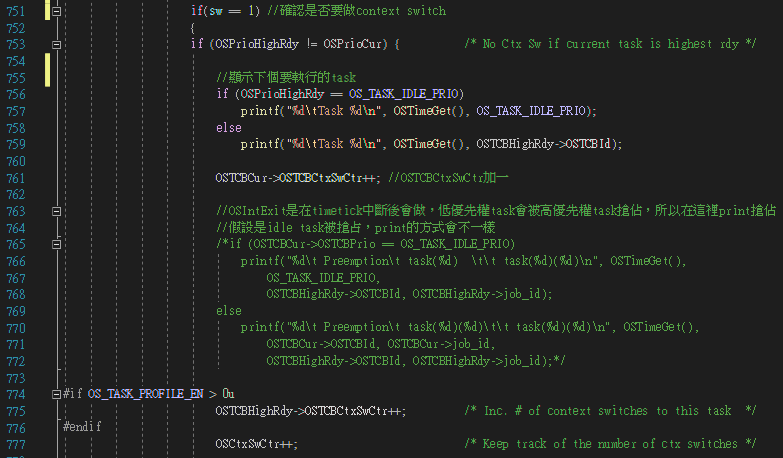
OSTimeTick()



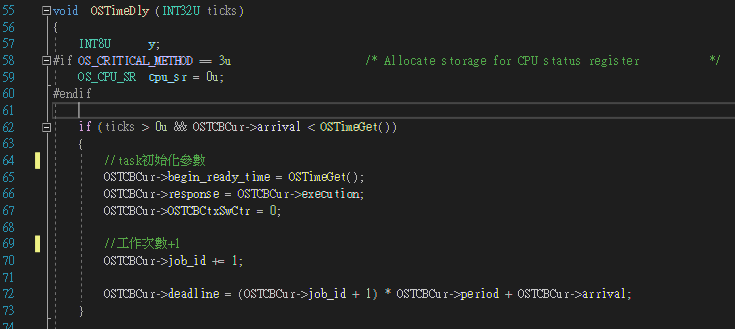


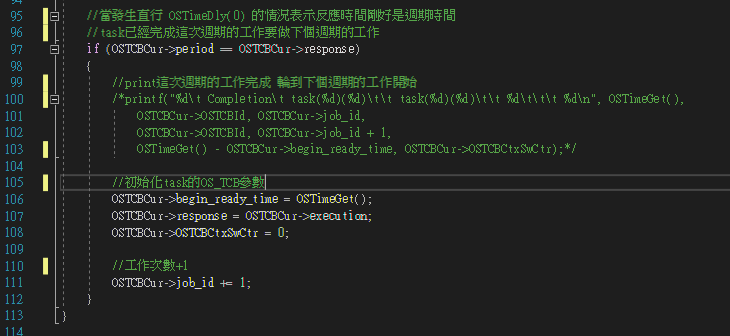
OS\_Sched()





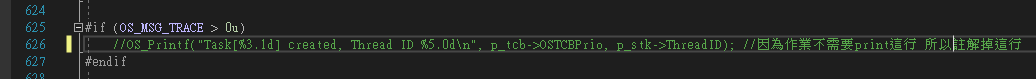
os\_time.c



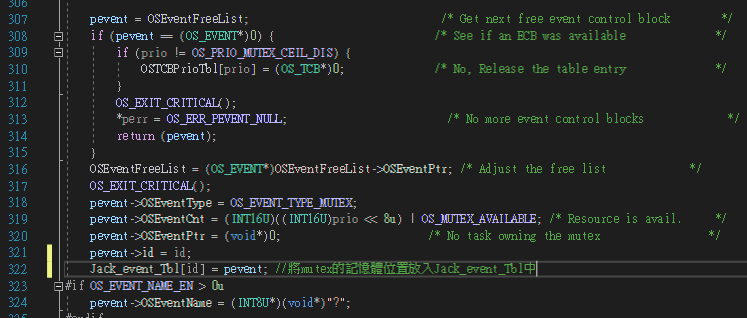


OSTimeDly()

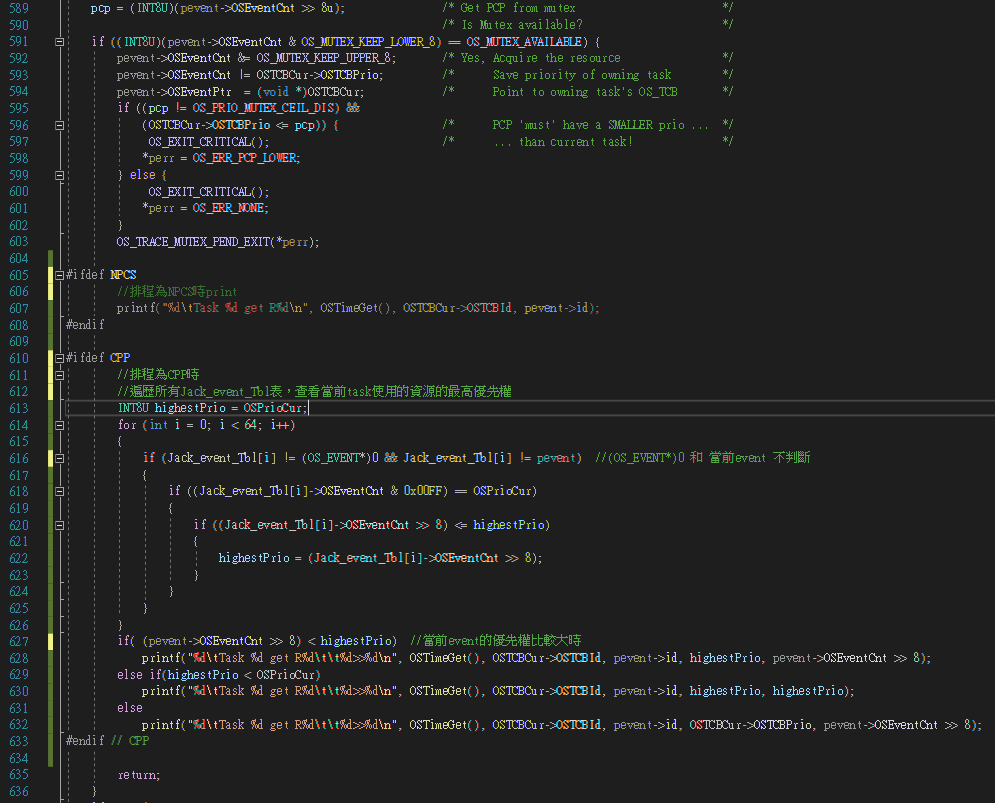
os\_cpu\_c.c



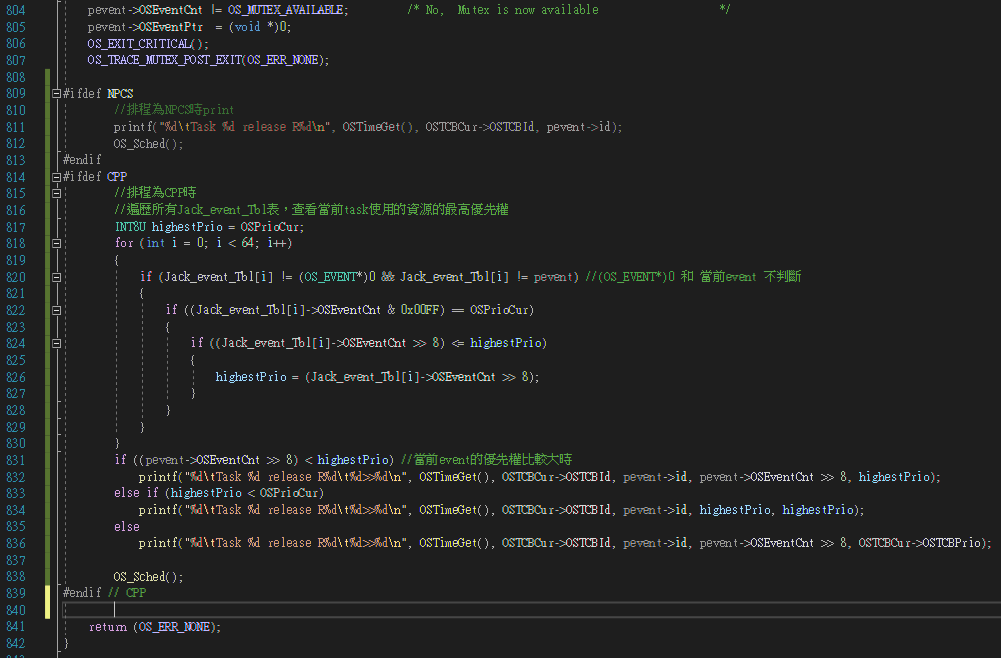
os\_mutex.c



Jack\_OSMutexCreate()



OSMutexPend()



OSMutexPost()

**[ PART III] Performance Analysis [50%]**

* Compare the scheduling behaviors between NPCS and CPP with the results of PART I and PART II. (5%)

[PART1]

NPCS跟CPP都是根據RM的方式，CPP因為資源都只有自己的task在用所以排程的結果就是單純以RM排程的結果，而NPCS不同的是當task使用到資源時直到釋放掉才可以切換到其他的task。

[PART2]

比較看結果NPCS與CPP的結果相同，因為CPP兩個task都有用到兩個資源所以當有task在使用資源時就不會被其他task中斷，結果就如同NPCS一樣只要在使用資源時直到釋放掉資源才可以切換到其他task。

* Explain how NPCS and CPP avoid the deadlock problem. (5%)

NPCS因為當有task在使用到資源時就會等到他做完才能切換到其他task，所以不會有其他task能夠同時想使用資源就不會發生deadlock。

CPP因為當task在使用資源時會繼承會使用到這個資源的最高task高一點點的優先權，只有比這個資源優先權還高的task才能中斷，但這些中斷的task不會用到同個資源，而會使用到同個資源的task因為無法中斷所以不會搶到同個資源，所以不會發生deadlock。