µC/OS-II

Introduction

Micro C/OS II 的特性

- 一種嵌入式即時作業系統，C語言寫成的
- Preemptible priority-driven real-time scheduling.
- 非常小、原始碼 5500 行，編譯後占記憶體 20KB
- 非開放原始碼，亦非免費，但可取得程式碼
- 64 priority levels (max 64 tasks)
- 8 reserved for *u*C/OS-II
- Each task is an infinite loop.
- Deterministic execution times for most
- *u*C/OS-II functions and services.
- Nested interrupts could go up to 256 levels.

特性

- Supports of various 8-bit to 64-bit platforms: x86, 68x, MIPS, 8051, Pic, AVR, etc
- Easy for development: Borland C++compiler and DOS (optional).
- However, *u*C/OS-II still lacks of the following features:
  - Resource synchronization protocols.
  - Sporadic task support.
  - Soft-real-time support.

About µC/OS-II

- Soft Real-time – tasks are performed as fast as possible
- Portable – runs on architectures ranging from 8-bit to 64 bit
- Scalable – features are configurable at compile time
- Multitasking – support 64 tasks simultaneously; including 8 reserved tasks
- Preemptive – preemptive multi-tasking with priority scheduling
- Kernel Services – provides task, time, memory management API; interprocess communication API; task synchronization API
- Nested Interrupt – up to 255 levels of nested interrupt
- Priority Inversion Problem – does not support priority inheritance
- Not using MMU – no well protected memory space like Unix or Win

Interprocess Communication (IPC)

- Mechanism for processes to communicate and to synchronize their actions
- Message system – processes communicate with each other without resorting to shared variables
- IPC facility provides two operations:
  - **send**(*message*) – message size fixed or variable
  - **receive**(*message*)
- If *P* and *Q* wish to communicate, they need to:
  - establish a *communication link* between them
  - exchange messages via send/receive
- Implementation of communication link
  - physical (e.g., shared memory, hardware bus)
  - logical (e.g., logical properties)

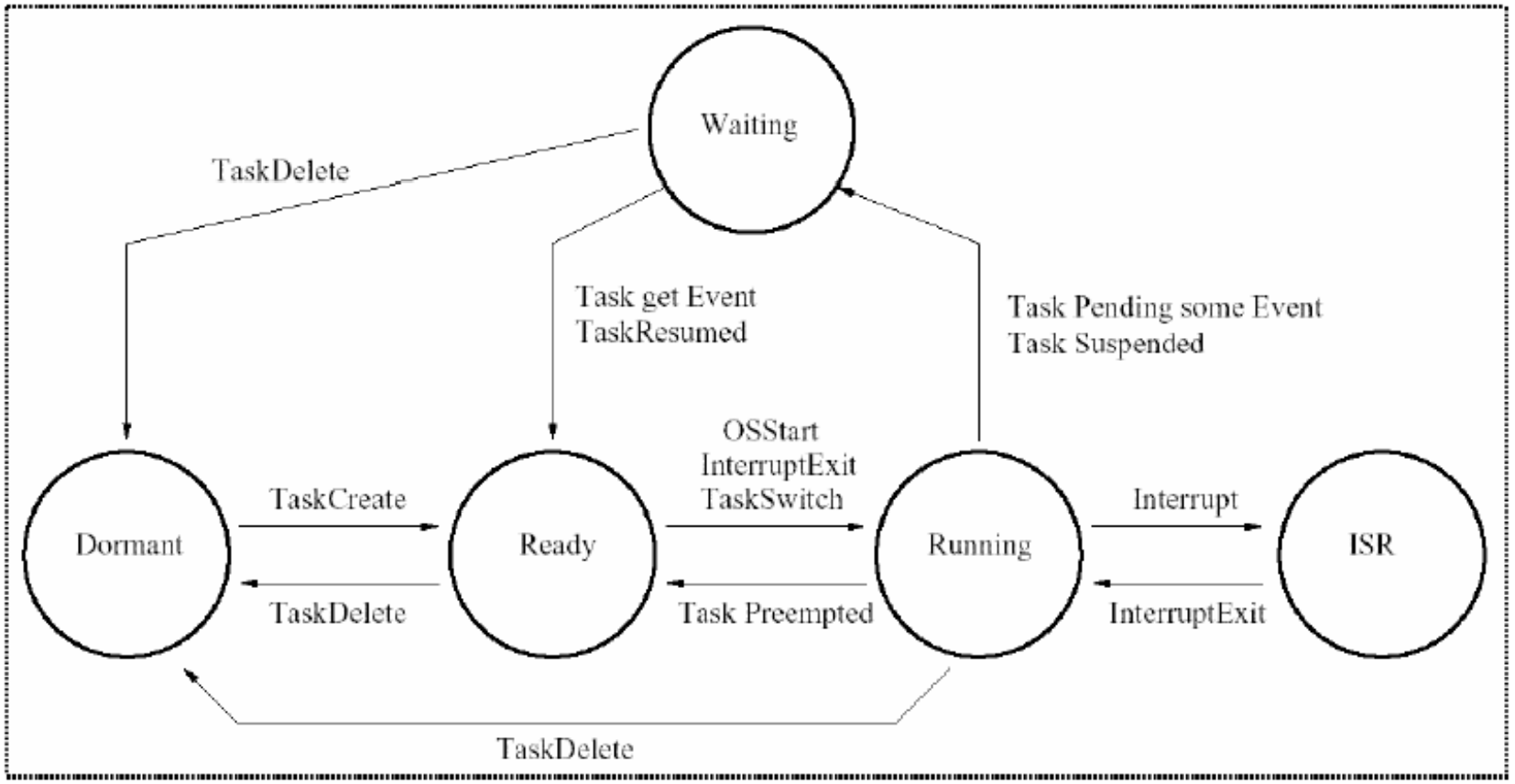
Interprocess Communication (IPC)

- Mechanism for processes to communicate and to synchronize their actions
- Message system – processes communicate with each other without resorting to shared variables
- IPC facility provides two operations:
  - **send**(*message*) – message size fixed or variable
  - **receive**(*message*)
- If *P* and *Q* wish to communicate, they need to:
  - establish a *communication link* between them
  - exchange messages via send/receive
- Implementation of communication link
  - physical (e.g., shared memory, hardware bus)
  - logical (e.g., logical properties)

Task in µC/OS-II

- A Task is a single instance of program
- Task thinks it has all CPU control itself
- Task has its own stack and own set of CPU registers backup in its stack
- Task is assigned a unique priority (highest 0 ~ lowest 63)
- Task is an infinite loop and never returns
- Task has states (see Figure next page)
- µC/OS-II saves task records in Task Control Block(TCB)

Task states in uC/OS-II



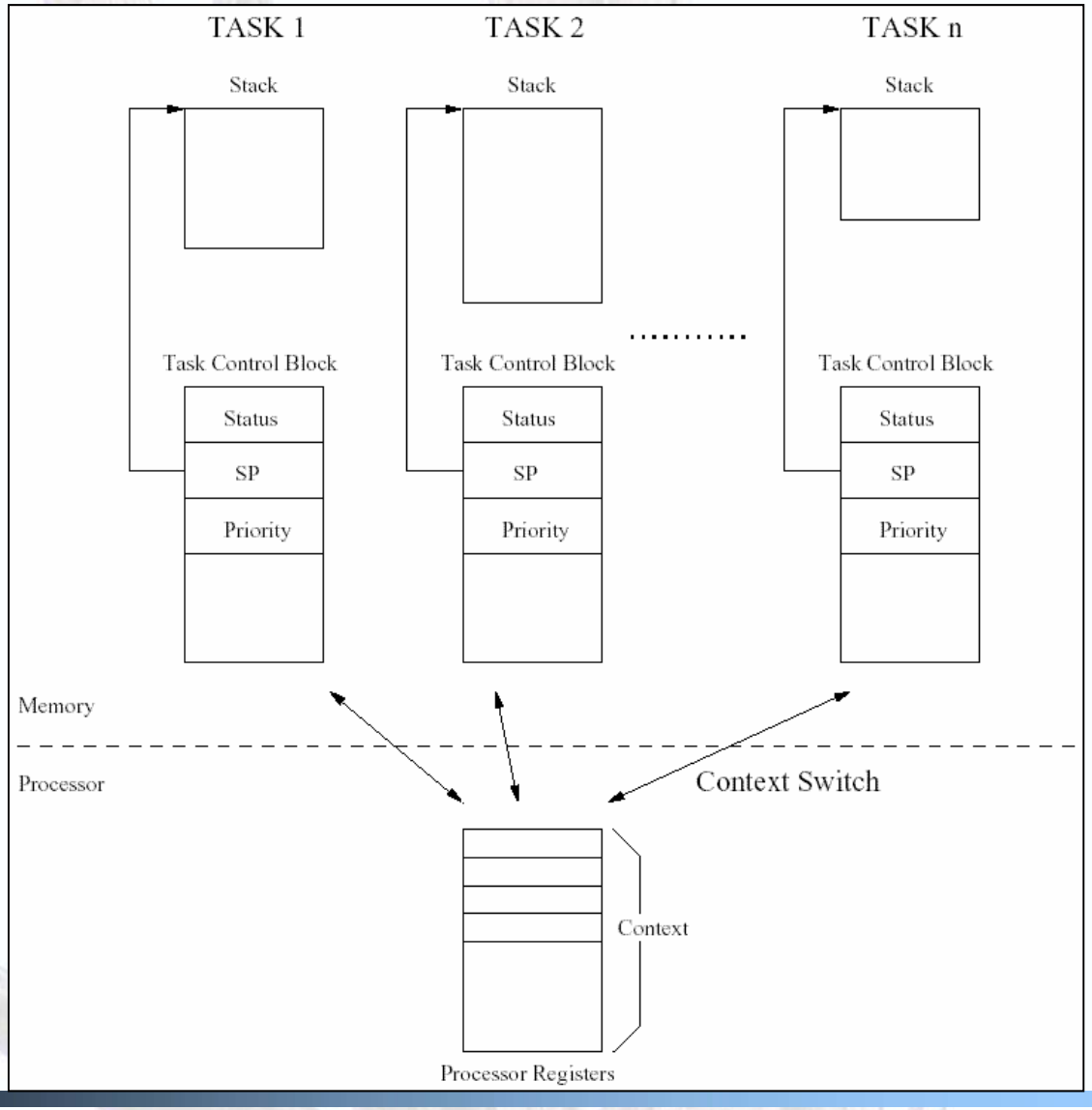
Task states in uC/OS-II

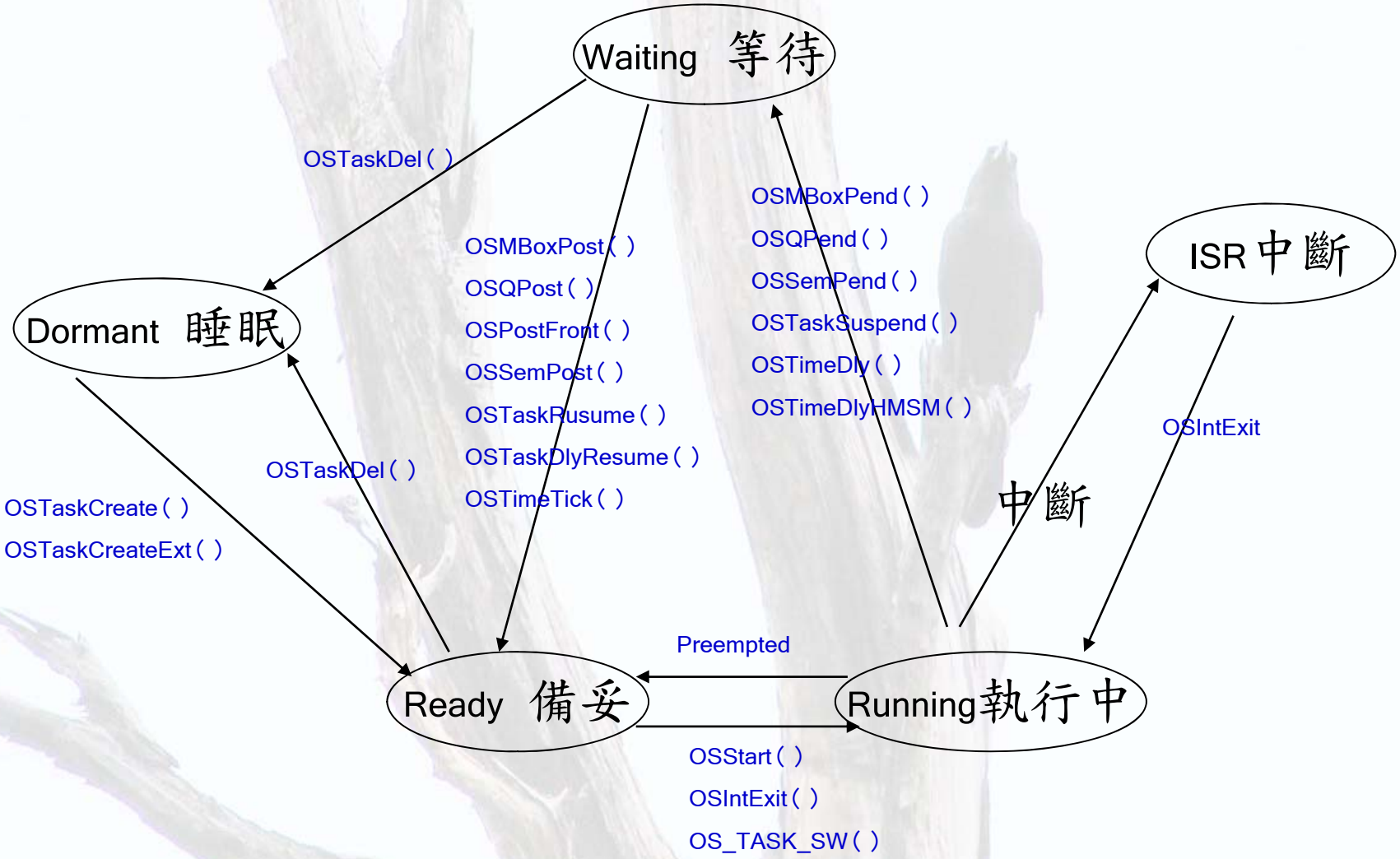
- Running – task has control of the processor and executing its job
- Ready – task is ready to execute but its priority is less than the running task
- Waiting – task requires the occurrence of an event to continue
- ISR – task is paused because the processor is handling an interrupt
- Dormant – task resides in memory, but not seen by the scheduler

Task Scheduling & Context Switch

- In µC/OS-II, task scheduling if performed on following conditions:
  - A task is created/deleted
  - A task changes state
  - On interrupt exit
  - On post signal
  - On pending event
  - On task suspension
- If the scheduler chooses a new task to run, context switch occurs. First,
- the context (processor registers) of current running task is saved in its stack.
- Next, the context of the new task is loaded into the processor. Finally, the processor continues execution. (see Figure)

Context switch in uC/OS-II

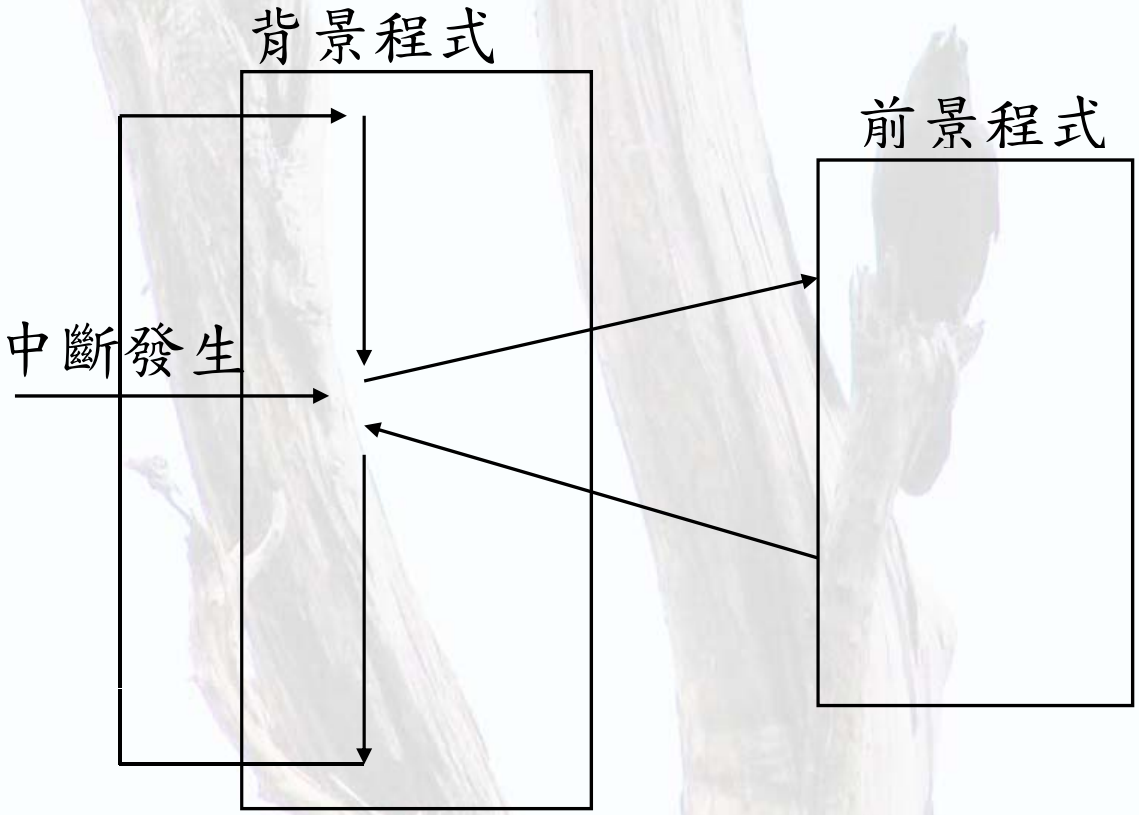



Waiting 等待
OSTaskDel ( )
OSMBoxPost ( )
OSQPost ( )
OSPostFront ( )
OSSemPost ( )
OSTaskRusume ( )
OSTaskDlyResume ( )
OSTimeTick ( )
OSMBoxPend ( )
OSQPend ( )
OSSemPend ( )
OSTaskSuspend ( )
OSTimeDly ( )
OSTimeDlyHMSM ( )
ISR 中斷
OSIntExit
Dormant 睡眠
OSTaskDel ( )
OSTaskCreate ( )
OSTaskCreateExt ( )
中斷
Preempted
Ready 備妥
Running執行中
OSStart ( )
OSIntExit ( )
OS_TASK_SW ( )

Critical Section

- A *critical section* is a portion of code that is not safe from race conditions because of the use of shared resources.
  - They can be protected by interrupt disabling/enabling interrupts or semaphores.
  - The use of semaphores often imposes a more significant amount of overheads.
  - A RTOS often use interrupts disabling/enabling to protect critical sections.
  - Once interrupts are disabled, neither context switches nor any other ISR's can occur.
  - Interrupt latency is vital to an RTOS!
  - Interrupts should be disabled as short as possible to improve the responsiveness.
  - It must be accounted as a blocking time in the schedulability analysis.
  - Interrupt disabling must be used carefully:
    - E.g., if OSTimeDly() is called with interrupt disabled, the machine might hang!

Real Time System



- Synchronization

Ring Buffer

```
#define BUFFER_SIZE 10
typedef struct {
. . .
} item;

item buffer[BUFFER_SIZE];
int in = 0;
int out = 0;
int counter = 0;
```

Ring-Buffer

- **Producer process**

```
item nextProduced;

while (1) {
    while (counter == BUFFER_SIZE)
        ; /* do nothing */
    buffer[in] = nextProduced;
    in = (in + 1) % BUFFER_SIZE;
    counter++;
}
```

Ring-Buffer

- **Consumer process**

```
item nextConsumed;

while (1) {
        while (counter == 0)
                ; /* do nothing */
        nextConsumed = buffer[out];
        out = (out + 1) % BUFFER_SIZE;
        counter--;
}
```

Ring-Buffer

- Assume **counter** is initially 5. One interleaving of statements is:
  - producer: **register1 = counter** (*register1 = 5*)
  - producer: **register1 = register1 + 1** (*register1 = 6*)
  - consumer: **register2 = counter** (*register2 = 5*)
  - consumer: **register2 = register2 – 1** (*register2 = 4*)
  - producer: **counter = register1** (*counter = 6*)
  - consumer: **counter = register2** (*counter = 4*)

- The value of **counter** may be either 4 or 6, where the correct result should be 5.

Ring-Buffer

- The statements

  counter++;

  counter--;

  must be performed *atomically*.

- Atomic operation means an operation that completes in its entirety without interruption.

Race Condition

- **Race condition**: The situation where several processes access – and manipulate shared data concurrently. The final value of the shared data depends upon which process finishes last.
- 指多個處理元同時並行（Concurrent）存取共用資源，系統依排程次序執行，而造成資源內的資料不正確的問題發生。
- To prevent race conditions, concurrent processes must be **synchronized**.

The Critical-Section(臨界區)Problem

- *n* processes all competing to use some shared data.
- Each process has a code segment, called *critical section*, in which the shared data is accessed.
- Problem – ensure that when one process is executing in its critical section, no other process is allowed to execute in its critical section.

進入臨界區(Critical-Section)間的條件

1. 互斥(Mutual Exclusion)

   當某個處理元在臨界區間內存取某些資源，則其他處理元不能進入臨界區間去存取相同資源。

2. 行進(Progress)

   當臨界區間內沒有處理元在執行，若有多個處理元要求進入臨界區間執行，則只有那些不在執行剩餘程式碼（Remainder Code）的處理元，才有資格被挑選為下一個進入臨界區間的處理元，而且這個挑選工作不能無限期的延遲下去。

3. 有限等待(Bounded Waiting)

   當某個處理元要求進入臨界區間，一直到它獲得進入臨界區間這段時間，允許其他處理元進入臨界區間的次數有限制。

Synchronization Hardware

- Many systems provide hardware support for critical section code
- Uniprocessors – could disable interrupts
  - Currently running code would execute without preemption
  - Generally too inefficient on multiprocessor systems
    - Operating systems using this not broadly scalable
- Modern machines provide special atomic hardware instructions
  - Atomic = non-interruptable
  - Either test memory word and set value
  - Or swap contents of two memory words

TestAndSet Instruction

Ref. only

- Definition:

```
boolean TestAndSet (boolean *target)
{
    boolean rv = *target;
    *target = TRUE;
    return rv:
}
```

Solution using TestAndSet

- Shared boolean variable lock., initialized to false.
- Solution:

```
do {
  while ( TestAndSet (&lock ))
        ;   /* do nothing

     //    critical section

  lock = FALSE;

     //      remainder section

} while ( TRUE);
```

Ref. only

Swap Instruction

- Definition:

Ref. only

```
void Swap (boolean *a, boolean *b)
{
    boolean temp = *a;
    *a = *b;
    *b = temp:
}
```

Semaphore Implementation with no Busy waiting

- With each semaphore there is an associated waiting queue. Each entry in a waiting queue has two data items:
  - value (of type integer)
  - pointer to next record in the list
- Two operations:
  - block – place the process invoking the operation on the appropriate waiting queue.
  - wakeup – remove one of processes in the waiting queue and place it in the ready queue.

Semaphore Implementation with no Busy waiting (Cont.)

- **Implementation of wait:**

```
wait (S){
        value--;
        if (value < 0) {
                add this process to waiting queue
                block(); }
 }
```

- **Implementation of signal:**

```
Signal (S){
        value++;
        if (value <= 0) {
                remove a process P from the waiting queue
                wakeup(P); }
}
```

Coding Guidelines for µC/OS-II

- Resources
  - User µC/OS-II defined data types for consistency & portability
  - Use statically allocated local variables for preemptive multitasking
  - Use semaphore to protect global variables and resources
- Data transfer
  - Inter-task communication can be achieved by mailbox/queue
  - µC/OS-II & user program all run in privileged mode, use share memory with caution!
- Memory allocation
  - Use µC/OS-II Kernel API: OSMemCreate(), OSMemGet()
- Standard C library
  - Many standard routine works in semihosting mode but not in stand-alone mode. (e.g. printf, fopen)

Starting µC/OS-II

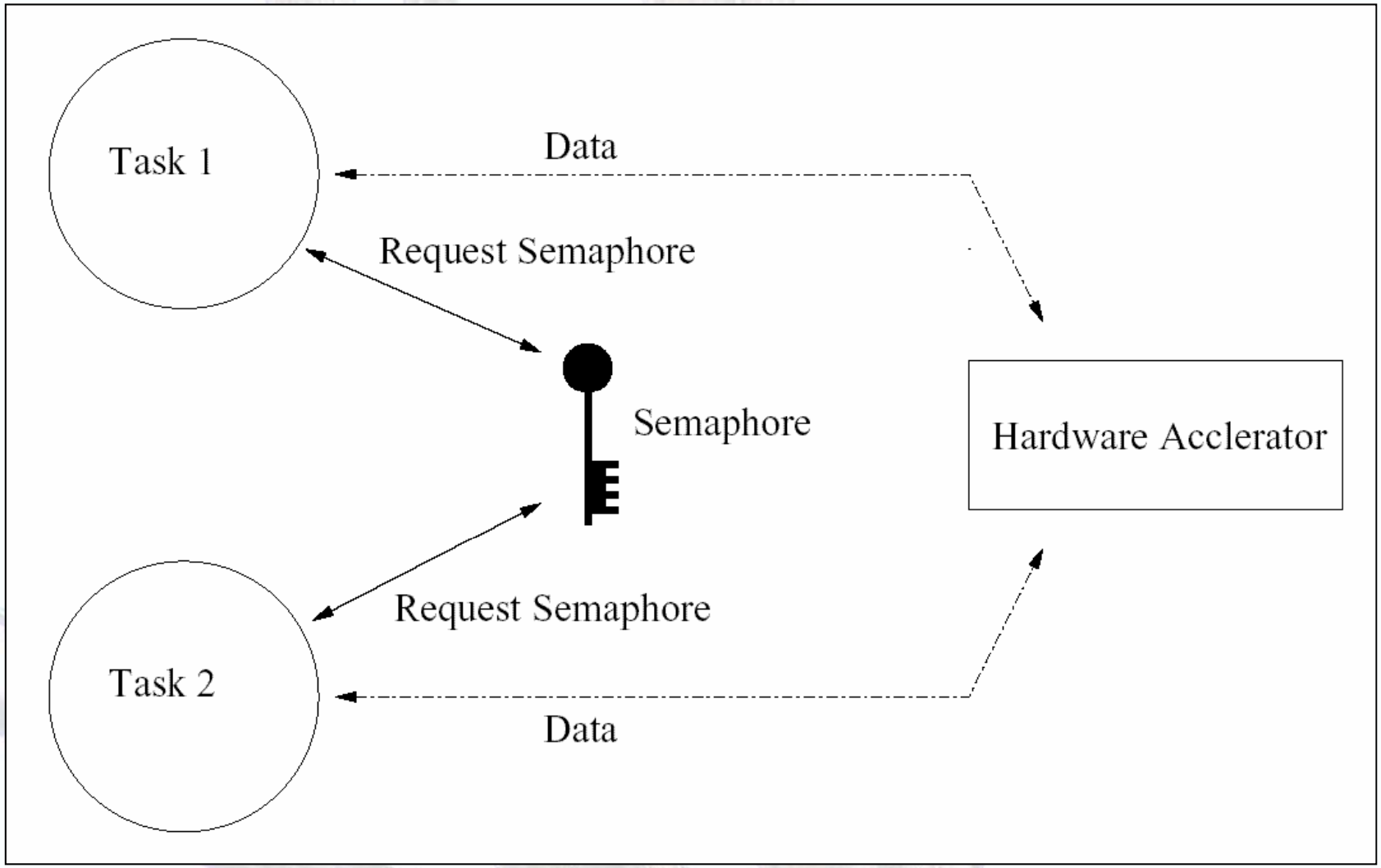
- µC/OS-II is initialized and started in the **main()** function

1. Initialize ARM target
2. Initialize OS
3. OS create/allocate resources
4. Create an initial task with highest priority
5. Create other user tasks
6. OS start scheduling
7. In the initial task, enable global interrupt
8. the initial task deletes itself
9. now, all other tasks runs under the control of OS

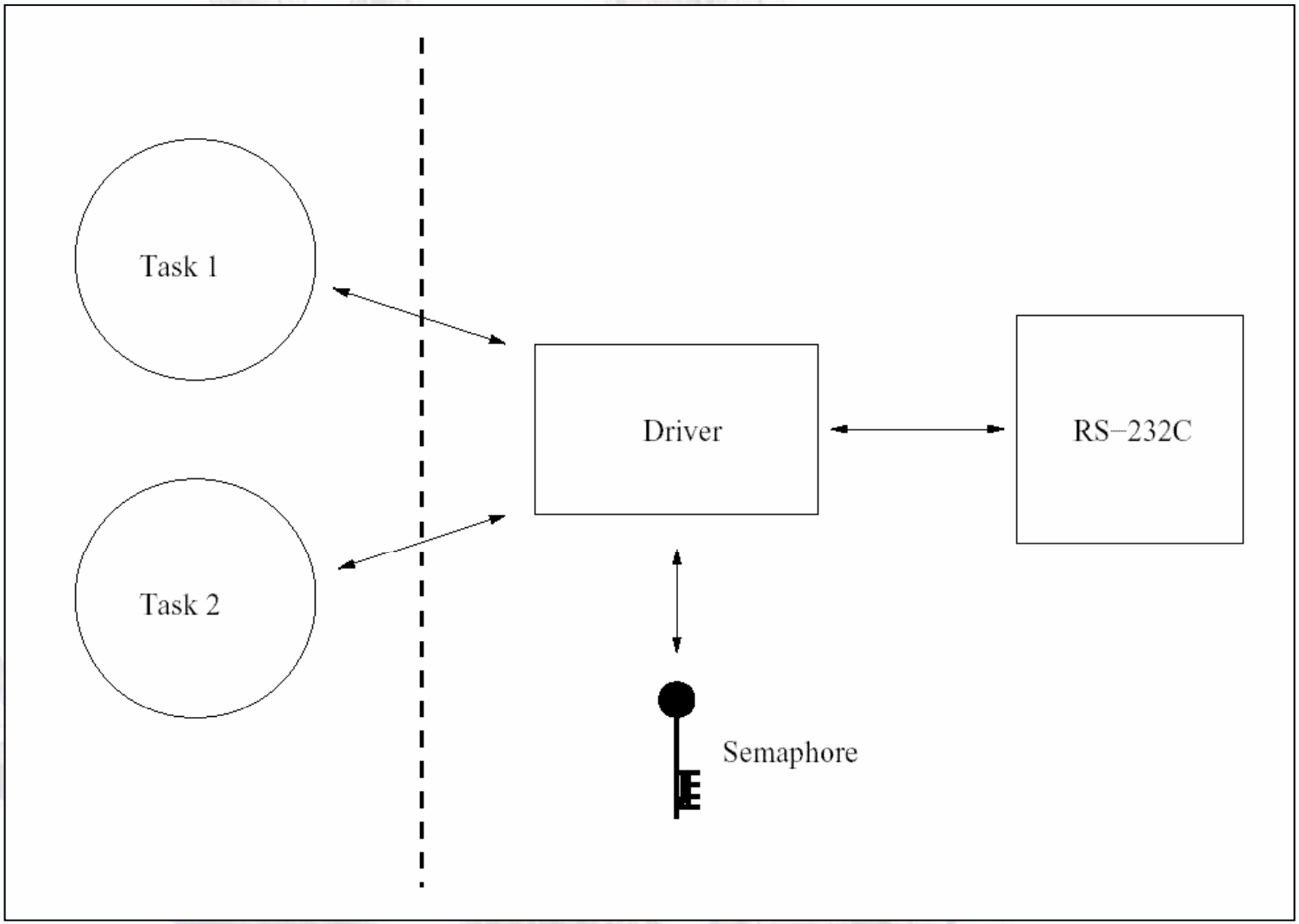
Resource management using Semaphore

- Semaphore is used to represent the status of a resource. To use a shared resource such as I/O port, hardware device or global variable, you must request the semaphore from OS and release the semaphore after access (see Figure)

Resource management using Semaphore



using Semaphore

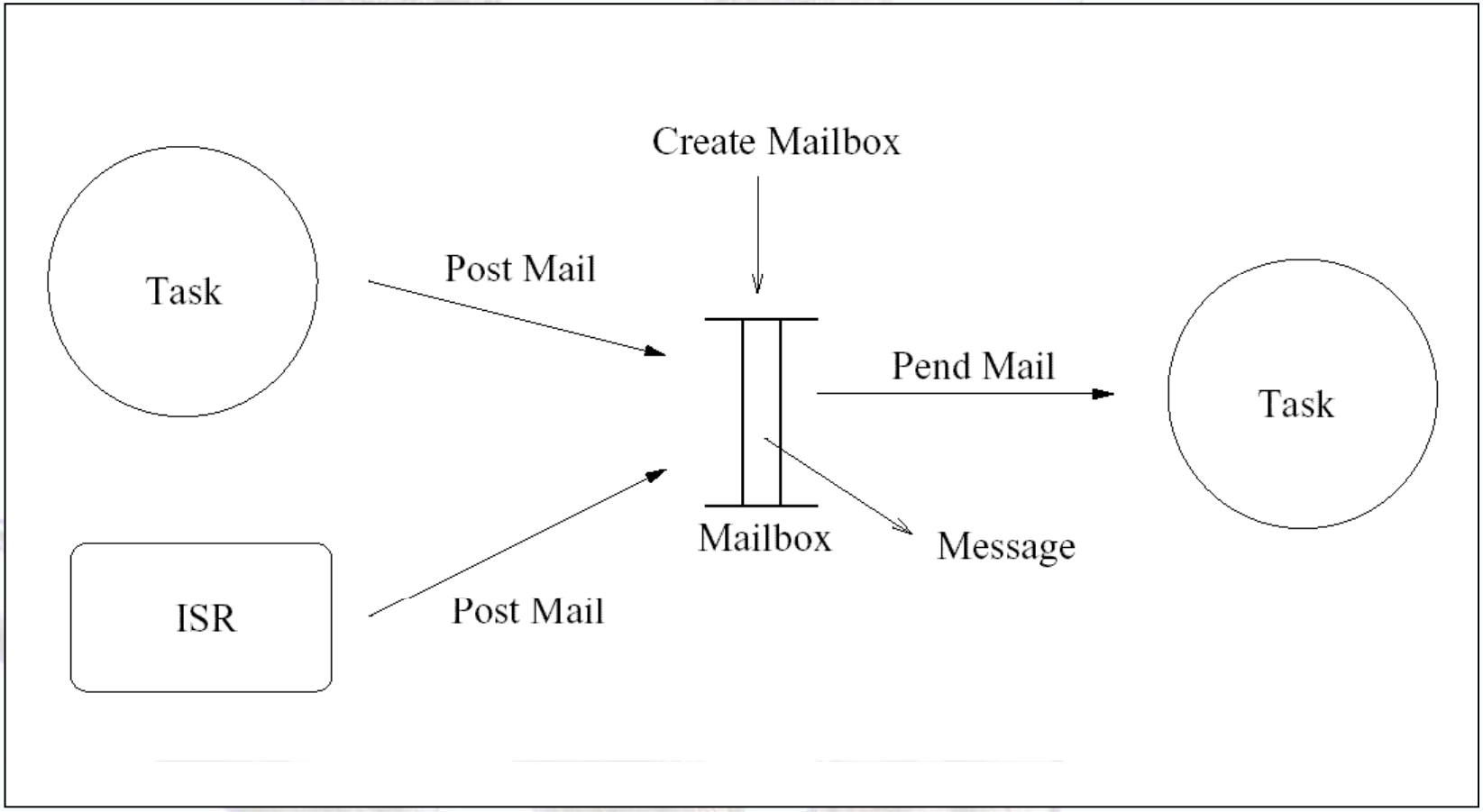


Interprocess Communication (IPC)

- Mechanism for processes to communicate and to synchronize their actions
- Message system – processes communicate with each other without resorting to shared variables
- IPC facility provides two operations:
  - **send**(*message*) – message size fixed or variable
  - **receive**(*message*)
- If *P* and *Q* wish to communicate, they need to:
  - establish a *communication link* between them
  - exchange messages via send/receive
- Implementation of communication link
  - physical (e.g., shared memory, hardware bus)
  - logical (e.g., logical properties)

Inter-Process Communication using Mailbox

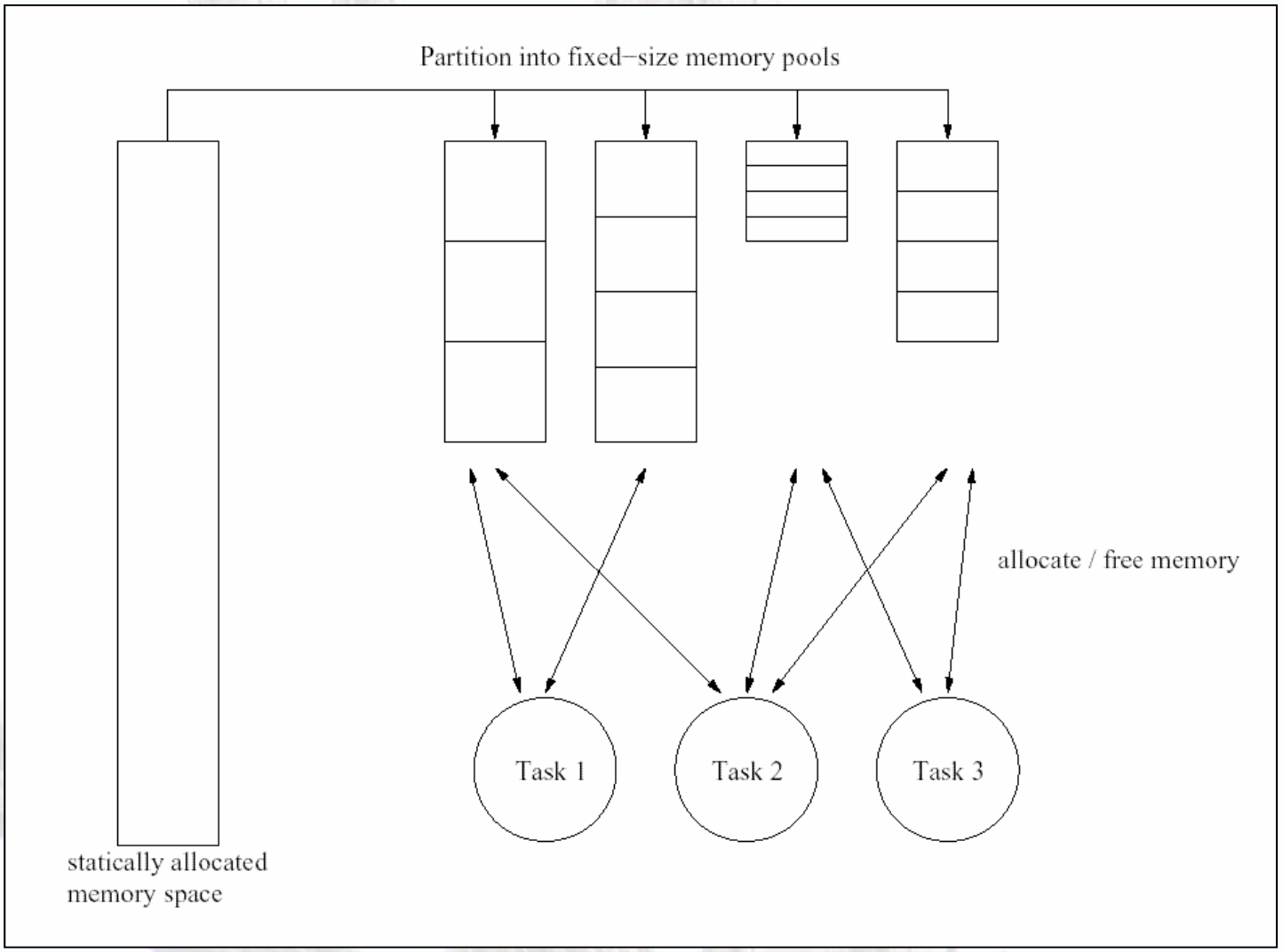
In µC/OS-II, tasks are not allowed to communicate to each other directly. The communication must be done under control of OS through Mailbox (Figure 5). A Queue is an array of mailbox with FIFO or FILO configuration



Mail Box

- A mailbox is for data exchanging between tasks.
  - A mailbox consists of a data pointer and a wait-list.
- OSMboxPend():
  - The message in the mailbox is retrieved.
  - If the mailbox is empty, the task is immediately **blocked** and moved to the wait-list.
  - A time-out value can be specified.
- OSMboxPost():
  - A message is posted in the mailbox.
  - If there is already a message in the mailbox, then an error is returned (not overwritten).
  - If tasks are waiting for a message from the mailbox, then the task with the highest priority is removed from the wait-list and scheduled to run.

Memory Management



任務函數 (1)

- OSTaskCreate()
  - 建立任務
- OSTaskCreateExt()
  - 另一種建立任務的函數
- OSTaskDel()
  - 結束任務
- OSStart()
  - 啟動uCOSII系統，此時系統會由備妥佇列中取出任務執行，此函數永遠不會結束
- OSIntExit()
  - 系統已經離開中斷狀態，此時會從備妥的佇列中取出任務來執行。
- OS_TASK_SW()
  - 系統執行任務切換，此時會從備妥的佇列中取出任務來執行。
- Preempted
  - 目前任務被更高優先權的任無所佔，因而被切換到備妥佇列。
- Interrput
  - 中斷發生了，系統進入中斷服務程式。

任務函數 (2)

- OSMBoxPend()
  - 任務等待訊息郵件盒，因而進入等待狀態。
- OSQPend()
  - 任務等待佇列。
- OSSemPend()
  - 任務等待號誌 (Semaphore)，拿到號誌才能取得對周邊的控制權。
- OSTaskSuspend()
  - 任務自行暫時擱置。
- OSTaskResume()
  - 任務重新開始。
- OSTimeDly()
  - 任務延遲一段時間，以 Time Click (時間節拍) 為單位。
- OSTimeDlyHMSM()
  - 任務延遲一段時間，以時、分、秒為單位。
- OSMBoxPost()
  - 任務收到所等待的訊息佇列盒。
- OSQPost()
  - 任務收到佇列。
- OSSemPost()
  - 任務交出號誌 Semaphore。
- OSTaskDlyResume()
  - 任務延遲取消。

Demo 範例

- Thanks to Jeffery LEE (ST Micro Taiwan)
- You can find the one directory "STM3210B-LK1" in the files. The following is the demo list for porting:

  - LED flashing from left to right and then reverse.
  - Rotating the VR1 changes speed of LED flashing.
  - LCD shows something by scrolling.
  - Pressing KEY2/3 changes words to be displayed on the LCD.
  - Pressing Up/Down of KEY4 changes contrast of the LCD.