Embedded Operating System

Embedded System Software Design

Prof. Ya-Shu Chen
National Taiwan University of Science and Technology

Objectives

•多任務
•調度算法
•資源管理
• 能源管理

- Multitasking
- Scheduling algorithms
- Resource managment
- Power management

Multitasking

- The scheduler switches the attention of CPU among several tasks 調度程序在多個任務之間切換CPU的注意
  - Tasks logically execute concurrently by sharing the CPU 通過共享CPU在邏輯上同時執行任
  - How much CPU share could be obtained by each task depends on the scheduling policy adopted

    每個任務可以獲取多少CPU份額取決於所採用的調度

Task

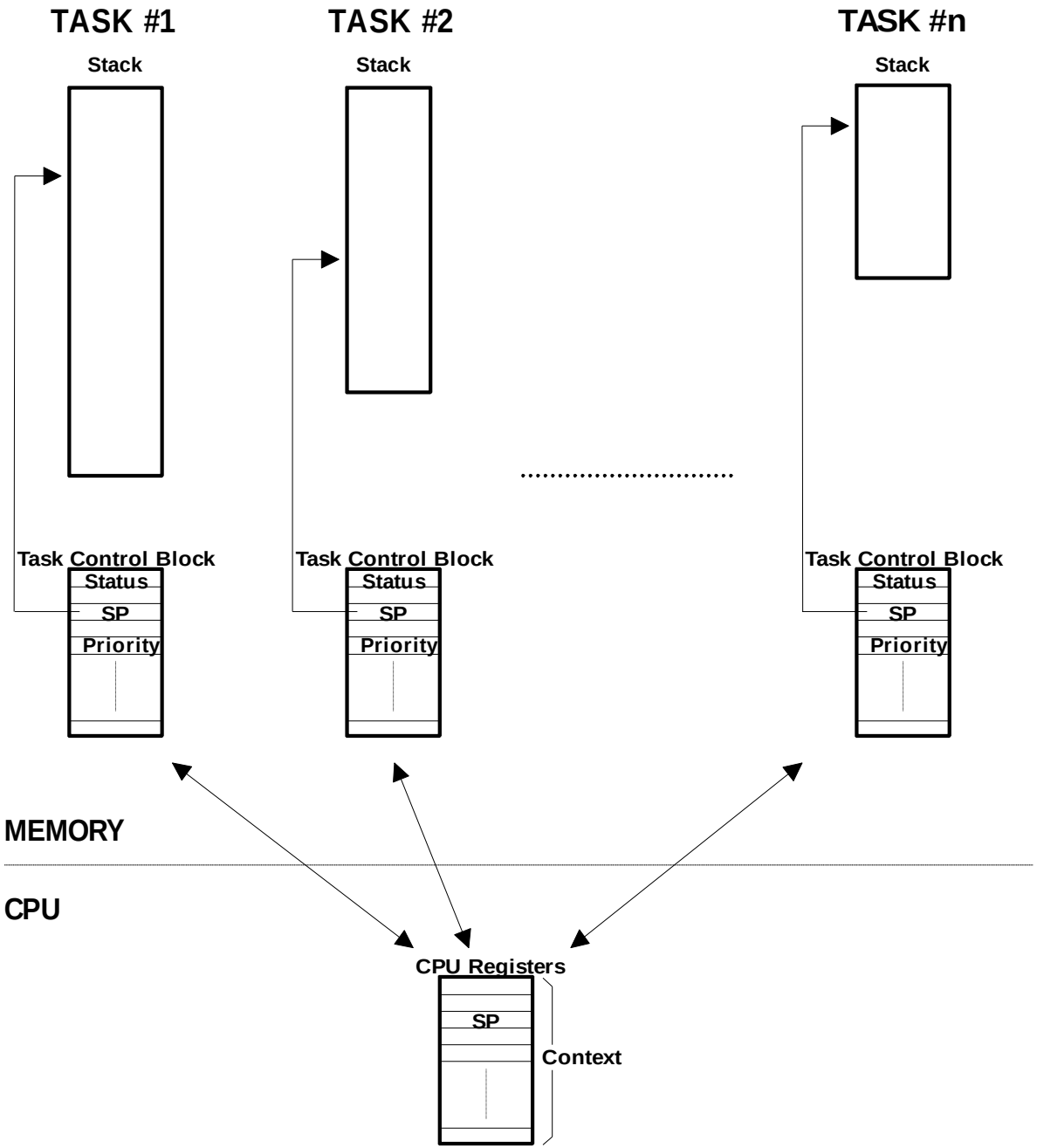
有時稱為過程

–確實構成競爭關係的活躍實

- Sometimes referred to as a process
  - An active entity that does computation

- From the OS point of view, a task is of a priority, a set of registers, its own stack, and some housekeeping information

從OS的角度來看，任務是優先級，一組寄存器，其自身的堆棧以及一些內部管理信息


TASK #1
TASK #2
TASK #n
Stack
Stack
Stack
.....
Task Control Block
Status
SP
Priority
Task Control Block
Status
SP
Priority
Task Control Block
Status
SP
Priority
MEMORY
CPU
CPU Registers
SP
Context

Task

實踐中的線程或進程。 它被視為系統中的活動/可執行實

- A thread or a process in practice. It is considered as an active/executable entity in a system.
- From the perspective of OS, a task is of a priority, a set of registers, its own stack area, and some housekeeping data.

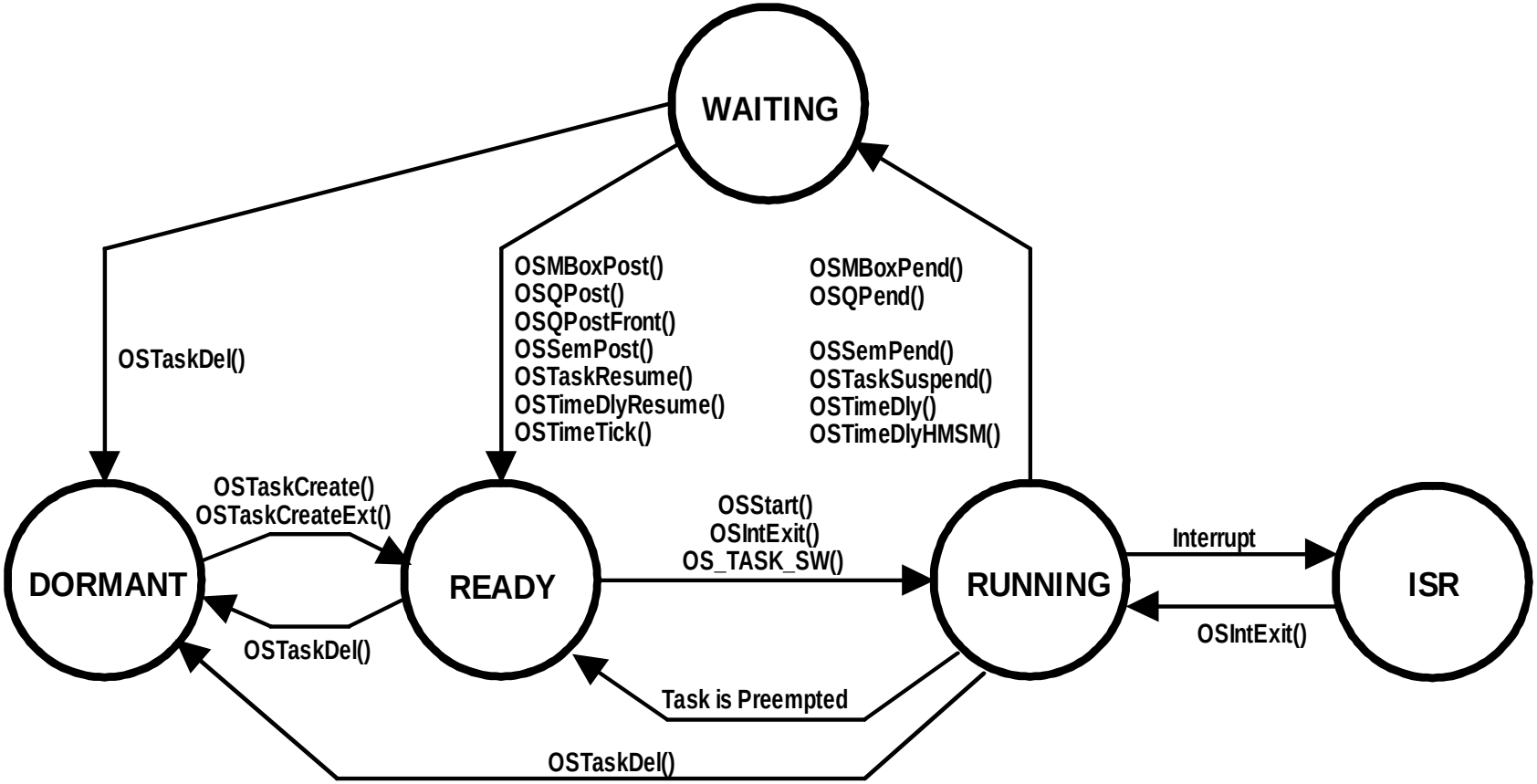
從OS的角度來看，任務是優先級，一組寄存器，其自己的堆棧區域以及一些內部管理數據

- From the perspective of scheduler, a task is of a series of consecutive jobs with regular ready time (for periodic tasks, µC/OS-II).

從調度程序的角度來看，任務是具有一系列準備就緒時間的連續作業（對於定期任務，是µC/ OS-II）。

- There are 5 states under µC/OS-II :
  - Dormant, ready, running, waiting, interrupted.

µC/ OS-II下有5種狀態：
–休眠，準備，運行，等待，中斷


WAITING
OSTaskDel()
OSMBoxPost()
OSQPost()
OSQPostFront()
OSSemPost()
OSTaskResume()
OSTimeDlyResume()
OSTimeTick()
OSMBoxPend()
OSQPend()
OSSemPend()
OSTaskSuspend()
OSTimeDly()
OSTimeDlyHMSM()
OSTaskCreate()
OSTaskCreateExt()
DORMANT
READY
OSStart()
OSIntExit()
OS_TASK_SW()
RUNNING
Interrupt
ISR
OSIntExit()
OSTaskDel()
Task is Preempted
OSTaskDel()

Kernels

內核是多任務系統的一部分，它負責：

- The kernel is a part of a multitasking system, it is responsible for:
  - The management of tasks. –任務管理。
  - Inter-task communication. –任務間通信。

- The kernel imposes additional overheads to task execution. 內核為任務執行增加了額外的開銷。
  - Kernel services take time.
    - Semaphores, message queues, mailboxes, timing controls, and etc…
  - ROM and RAM space are needed.

–內核服務需要時間。

•信號燈，消息隊列，郵箱，定時控件和等等…

–需要ROM和RAM空間。

Context Switch

- It occurs when the scheduler decides to run a different task. 當調度程序決定運行其他任務時，就會發生這種情況。
- The scheduler must save the context of the current task and then load the context of the task-to-run. 調度程序必須保存當前任務的上下文，然後加載要運行的任務的上下文。
  - The context is of a priority, the contents of the registers, the pointers to its stack, and the related housekeeping data.

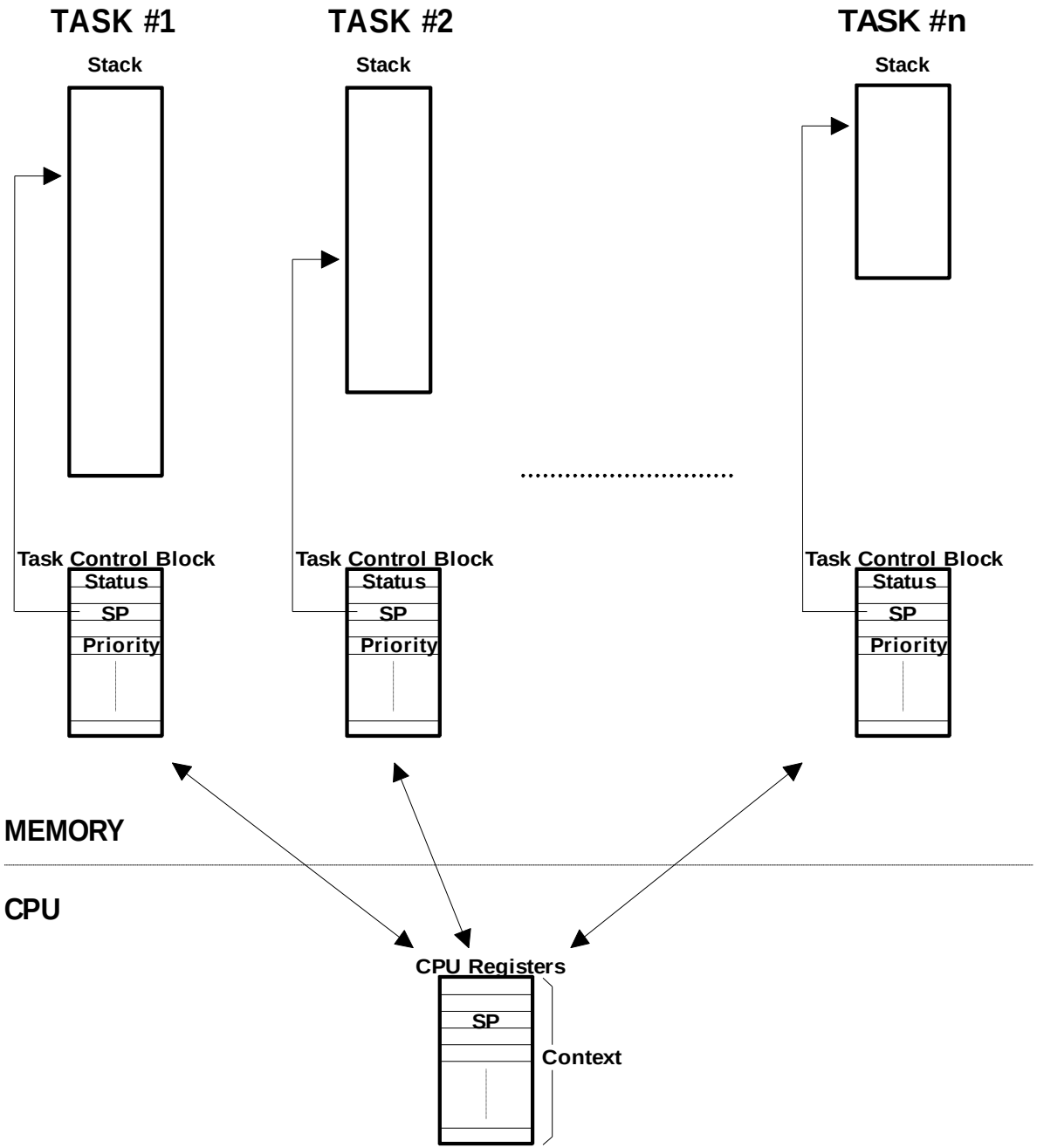
    上下文具有優先級，寄存器的內容，指向其堆棧的指針以及相關的內務處理數據。

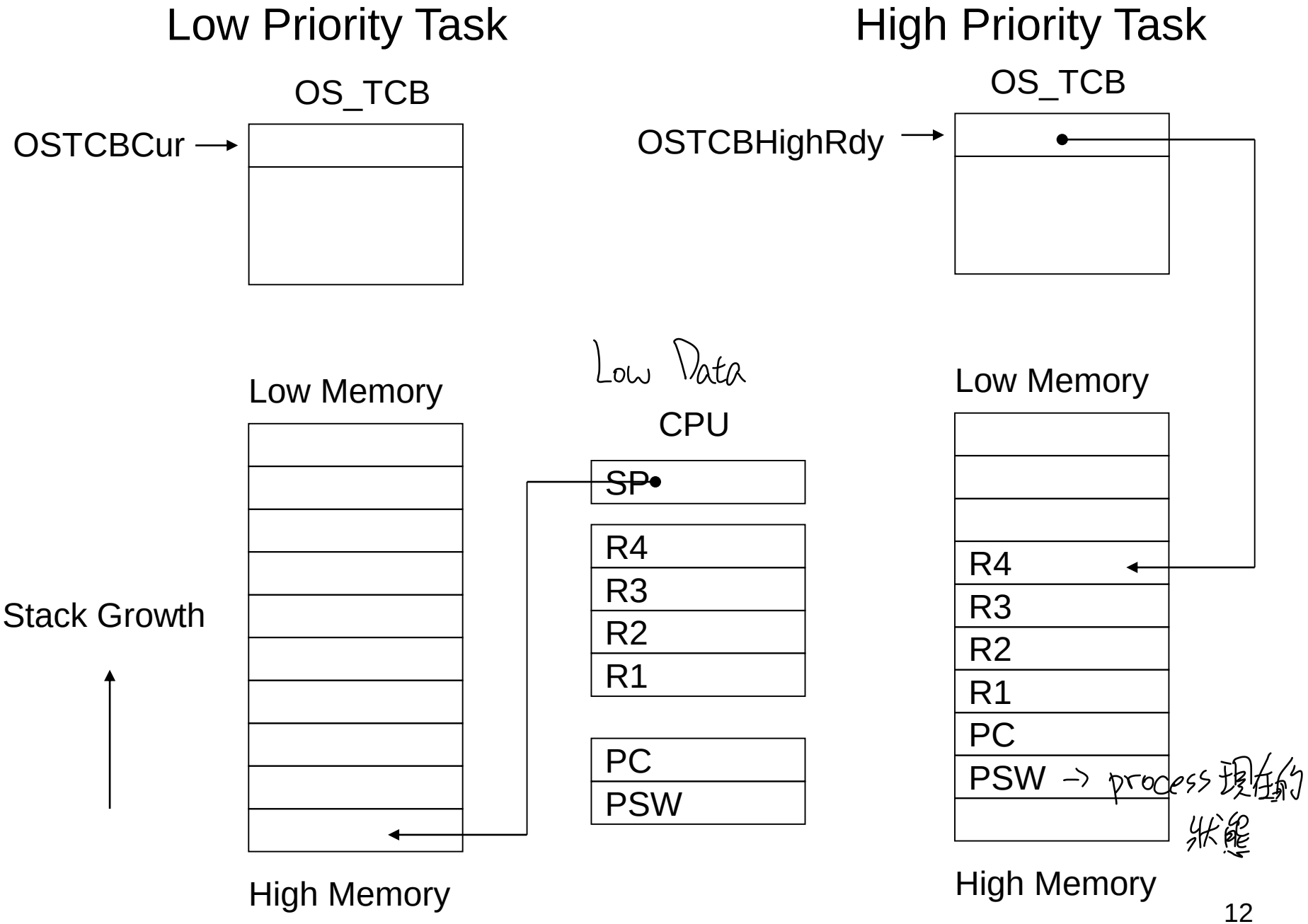
Context Switch

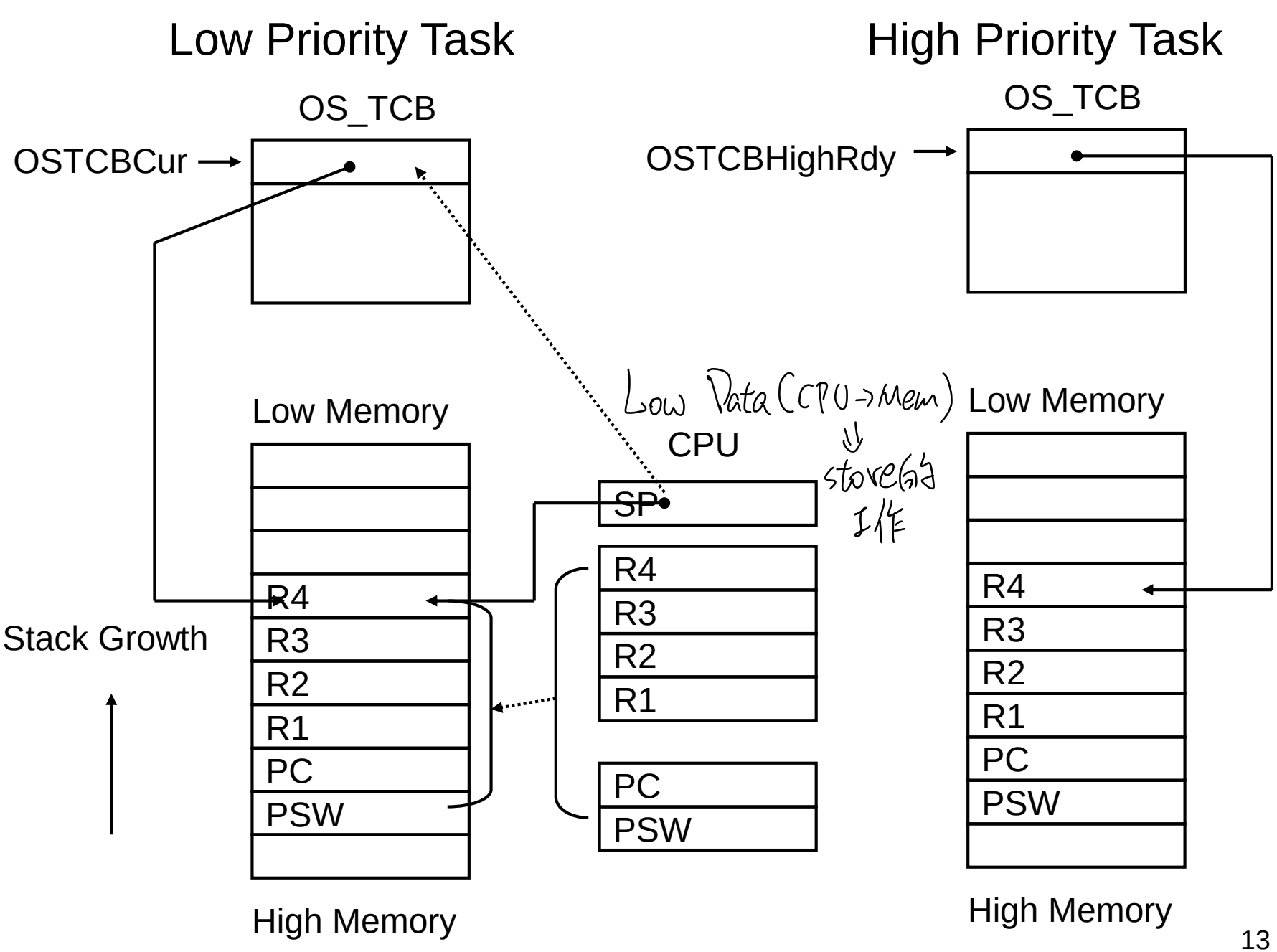
- Context-switches impose overheads on the task executions. 上下文切換給任務執行增加了開銷。
  - A practicable scheduler must not cause intensive context switches. Because modern CPU's have deep pipelines and many registers. 可行的調度程序不得引起密集的上下文切換。 因為現代CPU具有深層次的流水線和許多寄存器。
- For a real-time operating system, we must know how much time it takes to perform a context switch.
  - The overheads of context switch are accounted into high priority tasks. (blocking time, context switch time…)

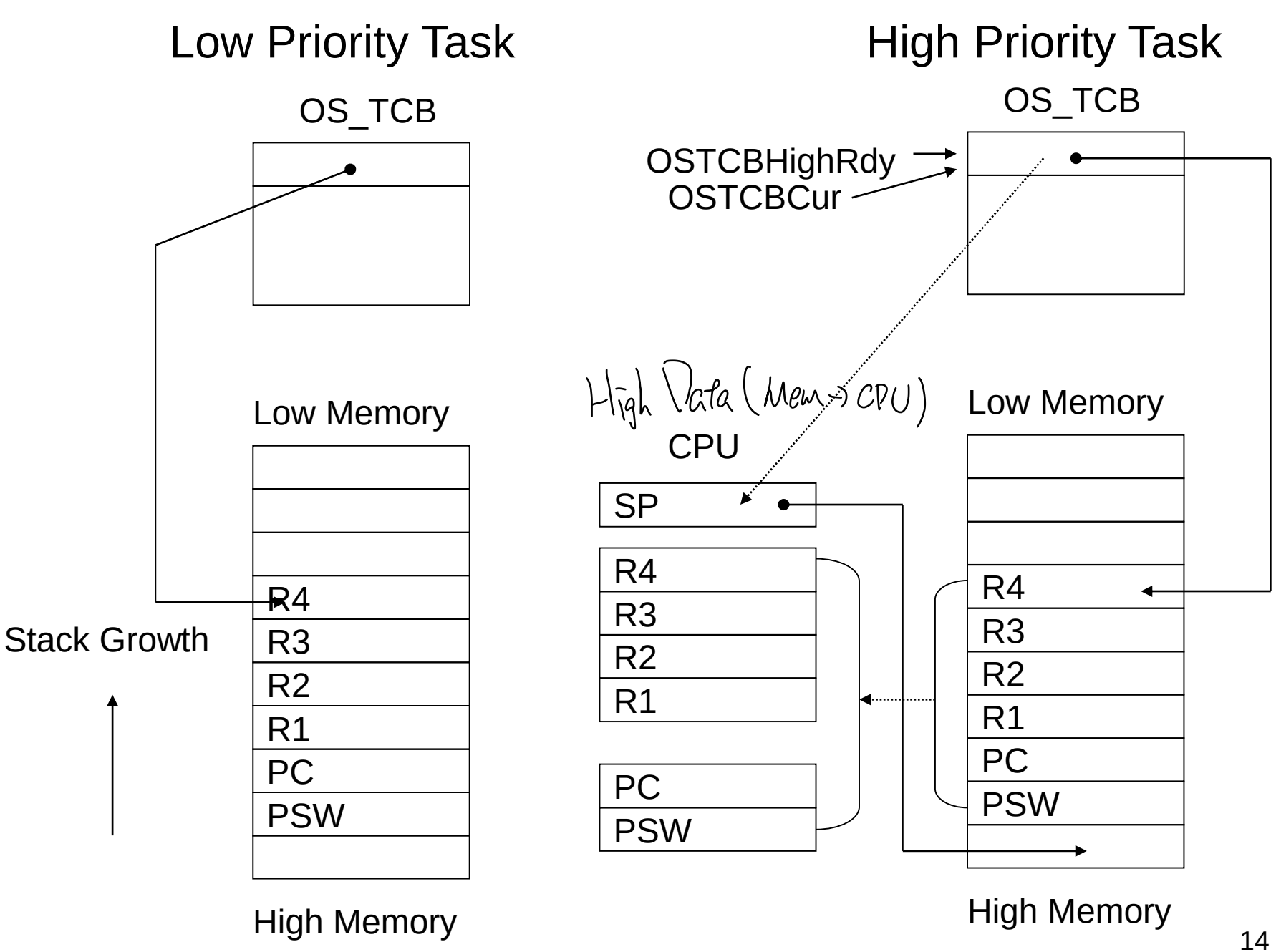
對於實時操作系統，我們必須知道執行上下文切換需要花費多少時間。

–上下文切換的開銷被計入高優先級任務。 （阻止時間，上下文切換時間...）


TASK #1
TASK #2
TASK #n
Stack
Stack
Stack
.....
Task Control Block
Status
SP
Priority
Task Control Block
Status
SP
Priority
Task Control Block
Status
SP
Priority
MEMORY
CPU
CPU Registers
SP
Context


Low Priority Task
OS_TCB
OSTCBCur
Low Memory
Stack Growth
High Memory
Low Data
CPU
SP
R4
R3
R2
R1
PC
PSW
High Priority Task
OS_TCB
OSTCBHighRdy
Low Memory
R4
R3
R2
R1
PC
PSW -> process 現在的狀態
High Memory


Low Priority Task
High Priority Task
OS_TCB
OS_TCB
OSTCBCur
OSTCBHighRdy
Low Memory
Low Data (CPU->Mem)
store的工作
CPU
Low Memory
SP
R4
R3
R2
R1
PC
PSW
R4
R3
R2
R1
PC
PSW
Stack Growth
R4
R3
R2
R1
PC
PSW
High Memory
High Memory


Low Priority Task
OS_TCB
Low Memory
R4
R3
R2
R1
PC
PSW
Stack Growth
High Memory
High Priority Task
OS_TCB
OSTCBHighRdy
OSTCBCur
High Data (Mem → CPU)
CPU
SP
R4
R3
R2
R1
PC
PSW
Low Memory
R4
R3
R2
R1
PC
PSW
High Memory

Low → ISR → Hight Ready → Low → Low Done → High

contextswitch 2次

Non-Preemptive Kernels

僅當任務顯式放棄對CPU的控制時，才發生上下文切換。

- Context switches occur only when tasks explicitly give up control of the CPU.
  - High-priority tasks gain control of the CPU.
  - This procedure must be done frequently to improve the responsiveness.

–高優先級任務獲得對CPU的控制。
–必須經常執行此過程以改善響應

- Events are still handled in ISR's.
  - ISR's always return to the interrupted task.

活動仍在ISR中進行處理。
– ISR始終返回被中斷的任務。

Low → ISR → High ready → High → High Done → Low

contextswitch 2次

Non-Preemptive Kernels

大多數任務是無競爭條件的。

- Most tasks are race-condition free.
  - Non-reentrant codes can be used without protections.
    –不能使用非可重入代碼，沒有保護措施。
    –在某些情況下，仍然需要同步。
  - In some cases, synchronizations are still needed.
- Pros: simple, robust. 優點：簡單，健壯。
- Cons: Not very responsive. There might be lengthy priority inversions.

缺點：反應不太靈敏。 優先級倒置可能會很長。

Preemptive Kernels

搶占式內核的好處是系統更具響應能力。

–任務的執行是確定的。

–優先級高的任務在就緒時立即獲得對CPU的控制。

- The benefit of a preemptive kernel is the system is more responsive.
  - The execution of a task is deterministic.
  - A high-priority task gain control of the CPU instantly when it is ready.

- ISR might not return to the interrupted task.
  - It might return a high-priority task which is ready.

ISR可能不會返回被中斷的任務。

–它可能會返回已準備好的高優先級任務。

- Concurrency among tasks exists. As a result, synchronization mechanisms (semaphores…) must be adopted to prevent from corrupting shared resources.
  - Preemptions, blocking, priority inversions.

.任務之間存在並發性。 結果，必須採用同步機制（信號量）以防止破壞共享資源。

–搶占，阻止，優先級倒置。.

Interrupts

- A hardware event to inform the CPU of an asynchronous event
  - clock tick (triggering scheduling), I/O events, hardware errors.

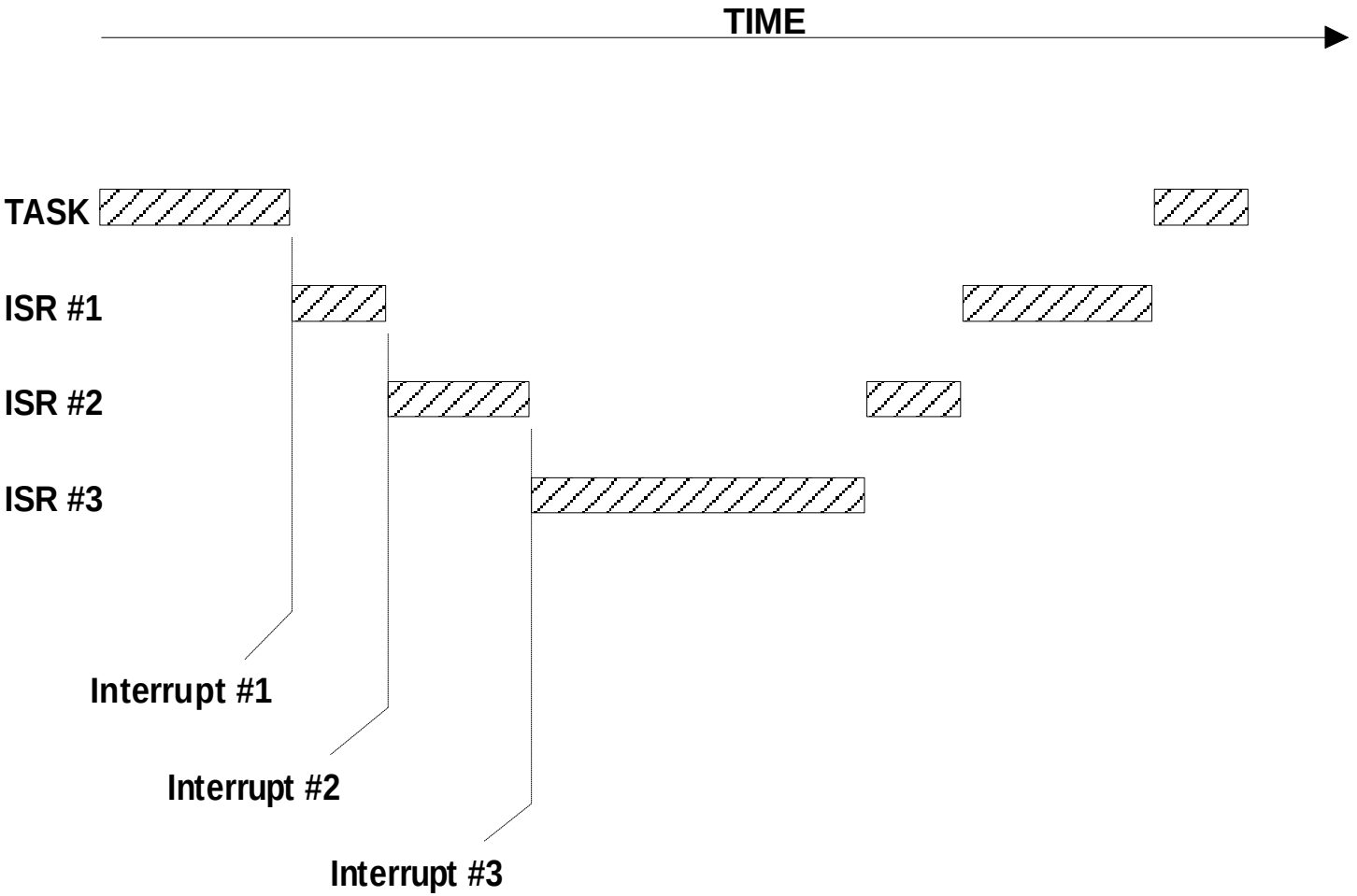
通知CPU異步事件的硬件事件
–時鐘滴答（觸發調度），I / O事件，硬件錯誤。

- The context of the current task is saved and the corresponding interrupt service routine (ISR) is invoked

保存當前任務的上下文，並調用相應的中斷服務程序（ISR）

- The ISR processes the event, and upon completion of the ISR, the program returns to
  - The background for a foreground/background system
  - The interrupted task for a non-preemptive kernel
  - The highest priority task ready to run for a preemptive kernel

ISR處理事件，並在ISR完成後，程序返回到
–前景/背景系統的背景
–非搶占式內核的中斷任務
–可以為搶占式內核運行的最高優先級任務


TIME
TASK
ISR #1
ISR #2
ISR #3
Interrupt #1
Interrupt #2
Interrupt #3

Multiple Interrupts

中斷向量表 → Interrupts 找資料

- A vector table manages devices to be handled by different code. 向量表管理要由不同代碼處理的設備。
- An interrupt with higher priority is executed first

  首先執行優先級更高的中斷

ISR

- ISRs should be as short as possible ISR應該越短越好
- ISR should
  - Recognize the interrupt
  - Get status from the interrupting device
  - Signal a task to perform processing

情監偵應
–識別中斷
–從中斷設備獲取狀態
–向任務發送信號以執行處

- Overhead involved in signaling task
  信令任務涉及的開銷

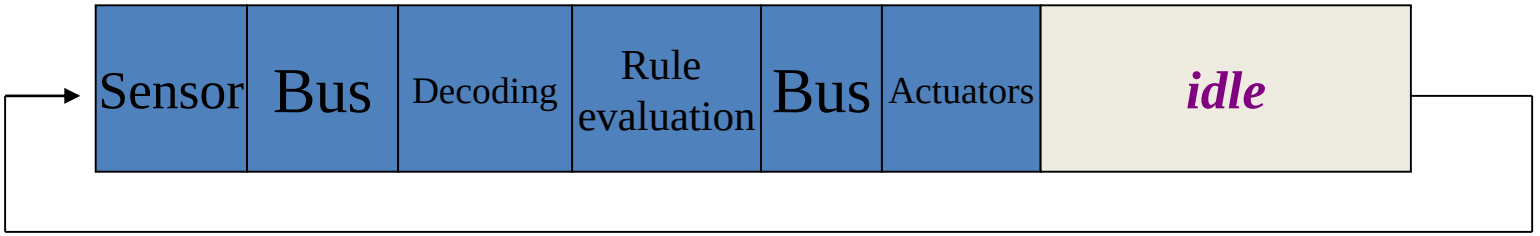
Cyclic Executive

- The system repeatedly exercises a static schedule
  - A table-driven approach

系統反复執行靜態計劃
–表驅動的方法

- Many existing systems still take this approach
  - Easy to debug and easy to visualize
    - Highly deterministic
  - Hard to program, to modify, and to upgrade
    - A program should be divided into many pieces (like an FSM)

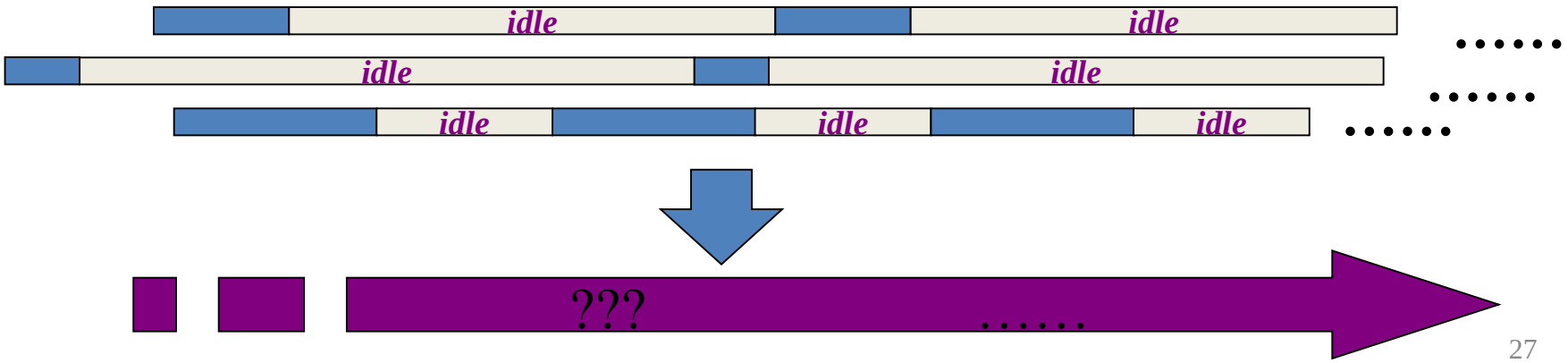
許多現有系統仍採用這種方法
–易於調試且易於可視化
•高度確定性
–難以編程，修改和升級
•程序應分為多個部分（例如FSM）



Cyclic Executive

該表模擬了例程的無限循環
–但是，對於許多複雜的系統來說，一個獨立的循環是不夠的
–應該考慮多個並發循環

- The table emulates an infinite loop of routines
  - However, a single independent loop is not enough to many complicated systems
  - Multiple concurrent loops should be considered
- How large should the table be when there are multiple loops? 當有多個循環時，表應該有多大？



Cyclic Executive

定義：令循環集合的超週期為一個時間間隔，該時間間隔的長度是循環長度的最小乘數

- ***Definition***: Let the hyper-period of a collection of loops be a time interval which's length is the least-common-multiplier of the loops' lengths
  - Let the length of the hyper-period be abbreviated as "h"

  –將超週期的長度縮寫為" h"

- ***Theorem***: The number of routines to be executed in any time interval [t,t+x] is identical to that in [t+h,t+h+x]

定理：在任何時間間隔[t，t + x]中要執行的例程數與[t + h，t + h + x]中的例程數相同

3/3

Round-Robin Scheduling

每個工作要跑多少(quantum)　　quantum用完換下一個

- A quantum is a pre-determined value

- Context switch occurs when:
  - The current task completes
  - The quantum for the current task is reached

•量子是預定值
•在以下情況下發生上下文切換：
–當前任務完成
–達到當前任務的範圍

ex: P_1 53, P_2 17 P_3 68 P_4 24　quantum = 20

3次　1次　4次　2次

(20,20,13)　(17)　(20,20,20,8)　(20,4)

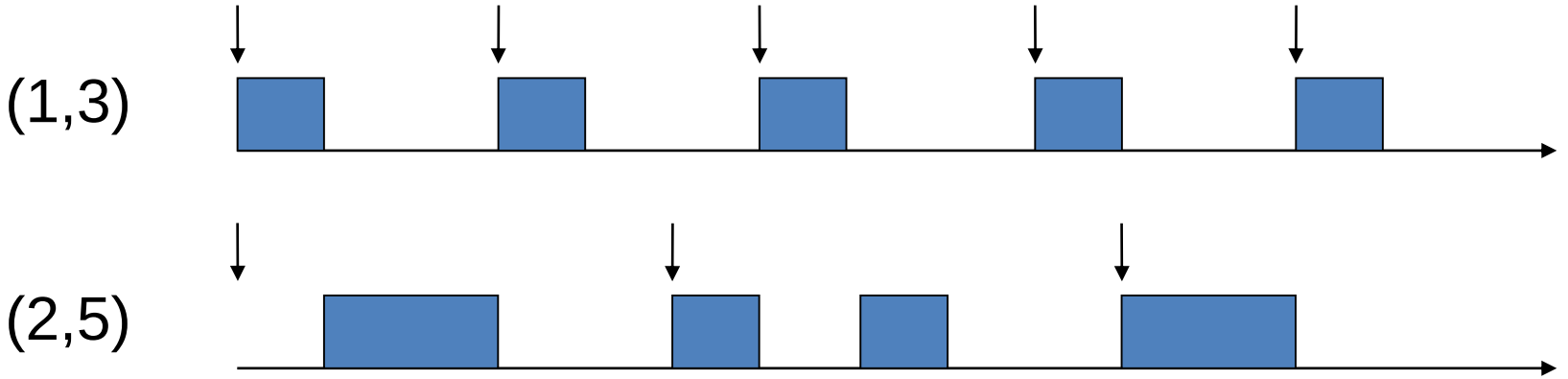
$P_1(1)$ $P_2(1)$ $P_3(1)$ $P_4(1)$ $P_1(2)$ $P_3(2)$ $P_4(2)$ $P_1(3)$ $P_3(3)$ $P_3(4)$

0→20→37→57→77→97→117→121→134→154→162

Rate-Monotonic Scheduling

- Task-level fixed-priority scheduling
  - All jobs inherit its task's priority
  - Usually abbreviated as fixed-priority scheduling
- Tasks' priorities are inversely proportional to their period lengths

•任務級別的固定優先級調度
–所有工作均繼承其任務的優先級
–通常縮寫為固定優先級調度
•任務的優先級與它們的時間長度成反比



Rate-Monotonic Scheduling

任務Ti的關鍵時刻（關鍵實例）

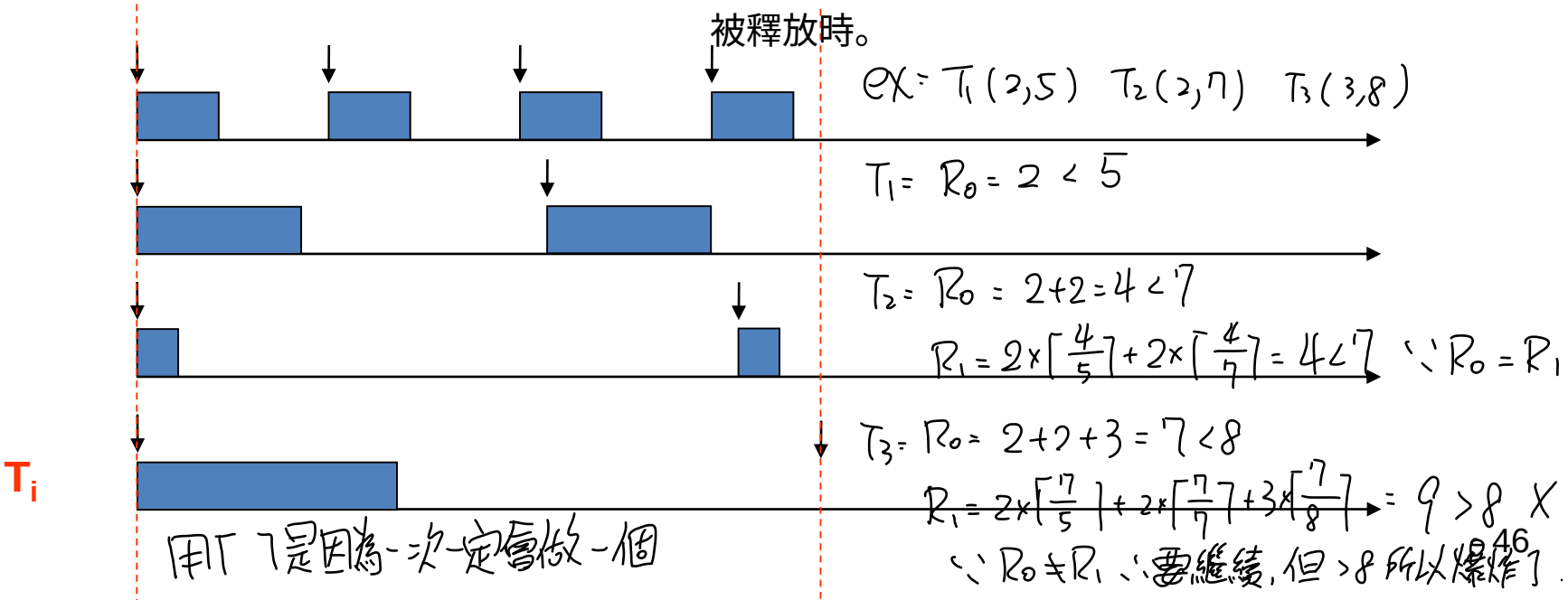
有最長的response time

- Critical instant (critical instance) of task T_i
  - –在Ti的關鍵時刻發布的任務Ti的作業J（i，c）將具有最長的響應時間
  - – A job $J_{i,c}$ of task T_i released at T_i's critical instant would have the longest response time
  - – J（i，c）將是"最難以"在截止日期前完成的任務
  - – $J_{i,c}$ would be the one that is "hardest" to meet its deadline
  - –如果J（i，c）成功滿足其終止日期，則Ti的任何工作　在任何情況下總是成功
  - – If $J_{i,c}$ succeeds in satisfying its deadline, then any job of T_i always succeeds for any cases
    - Since in any other cases deadlines are easier to meet

Rate-Monotonic-Scheduling

- ***Theorem***: A critical instant of any task T_i occurs when one of its job $J_{i,c}$ is released at the same time with a job of every higher-priority task (i.e., in-phase).

定理：任何任務To的關鍵時刻發生在它的一個任務Ji，c與每個更高優先級任務（即同相）的任務同時被釋放時。

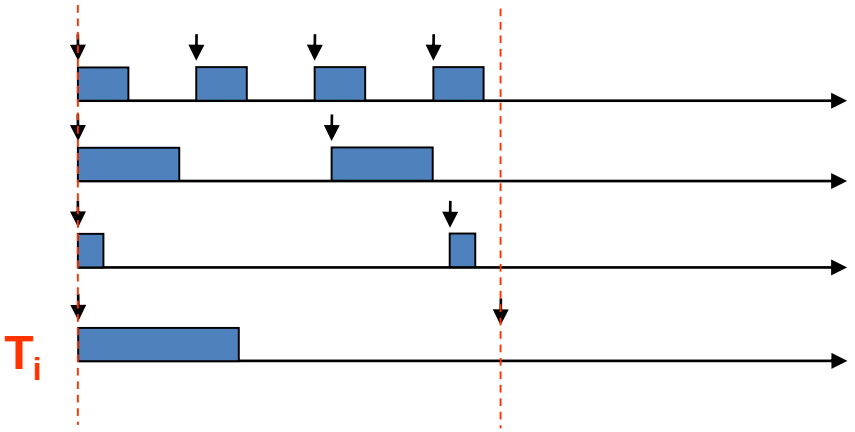


Rate-Monotonic Scheduling

- Response time analysis 響應時間分析
  - The response time of the job of Ti at critical instant can be calculated by the following recursive function Ti的關鍵時刻的響應時間可以通過以下遞歸函數來計算

$$r_0 = \sum_{\forall i} C_i$$

$$r_n = \sum_{\forall i} C_i \left\lceil \frac{r_{n-1}}{p_i} \right\rceil$$



  - Observation: the sequence of r_x, x>=0 may or may not converge 觀察：rx，x> = 0的序列可能會收斂，也可能不會收斂

Rate-Monotonic Scheduling

- ***Theorem***: Given a task set=$\{T_1,T_2,\ldots,T_n\}$, if at critical instant the response time of the first job of task T_i, for each i, converges no later than p_i, then jobs never miss their deadlines

  定理：給定一個任務集= {T1，T2，…，Tn}，如果在關鍵時刻任務Ti的第一個任務的響應時間，對於每個i，最晚在pi之前收斂，然後作業永遠不會錯過其截止日期

- Observations
  - If the task set survives critical instant, then it will survive any task phasing

    如果任務集在關鍵時刻倖免於難，那麼它將在任何任務階段都得以倖存
  - The analysis is an exact schedulability test for RMS
  - Usually referred to as “Rate-Monotonic Analysis”, RMA for short

    該分析是RMS的精確可調度性測試

    通常稱為“速率單調分析”，簡稱RMA

Rate-Monotonic Scheduling

- Definition
  - Utilization factor of task T=(c,p) is defined as

$$\frac{c}{p}$$

  - CPU utilization of a task set $\{T_1,T_2,\ldots,T_n\}$ is

$$U = \sum_{i=1}^{n} \frac{c_i}{p_i}$$

  - Observation: if the total utilization exceeds 1 then the task set is not schedulable

Rate-Monotonic Scheduling

- ***Theorem***: [LL73] Given a task set $\{T_1,T_2,\ldots,T_n\}$. It is schedulable by RMS if

$$\sum_{i=1}^{n}\frac{C_i}{P_i}\leq U(n)=n(2^{1/n}-1)$$

- Observation:
  - If the test succeeds then the task is schedulable
  - A sufficient condition for schedulability

ex: $T_1(1,3)$ $T_2(2,5)$ $\frac{1}{3}+\frac{2}{5}=\frac{11}{15}=0.73$

$R_0=3$

$R_1=1\times\lceil\frac{3}{3}\rceil+2\times\lceil\frac{3}{5}\rceil=3<5$

$T_1(4.5,9)$ $T_2(3,6)$ $\frac{4.5}{9}+\frac{3}{6}=1$

$T_1=$ $R_0=7.5$ $T_2=3<6$

$R_1=4.5\times\lceil\frac{7.5}{9}\rceil+3\times\lceil\frac{7.5}{6}\rceil=10.5>9$

$T_1(1,2)$ $T_2(2,4)$ $\frac{1}{2}+\frac{2}{4}=1$

$T_2=$ $R_0=3$ $T_1=1<2$

$R_1=1\times\lceil\frac{3}{2}\rceil+2\times\lceil\frac{3}{4}\rceil=4$

$R_2=1\times\lceil\frac{4}{2}\rceil+2\times\lceil\frac{4}{4}\rceil=4$

$T_1(2,3)$ $T_2(4,6)$ $\frac{2}{3}+\frac{4}{6}=\frac{8}{6}>1>0.828$

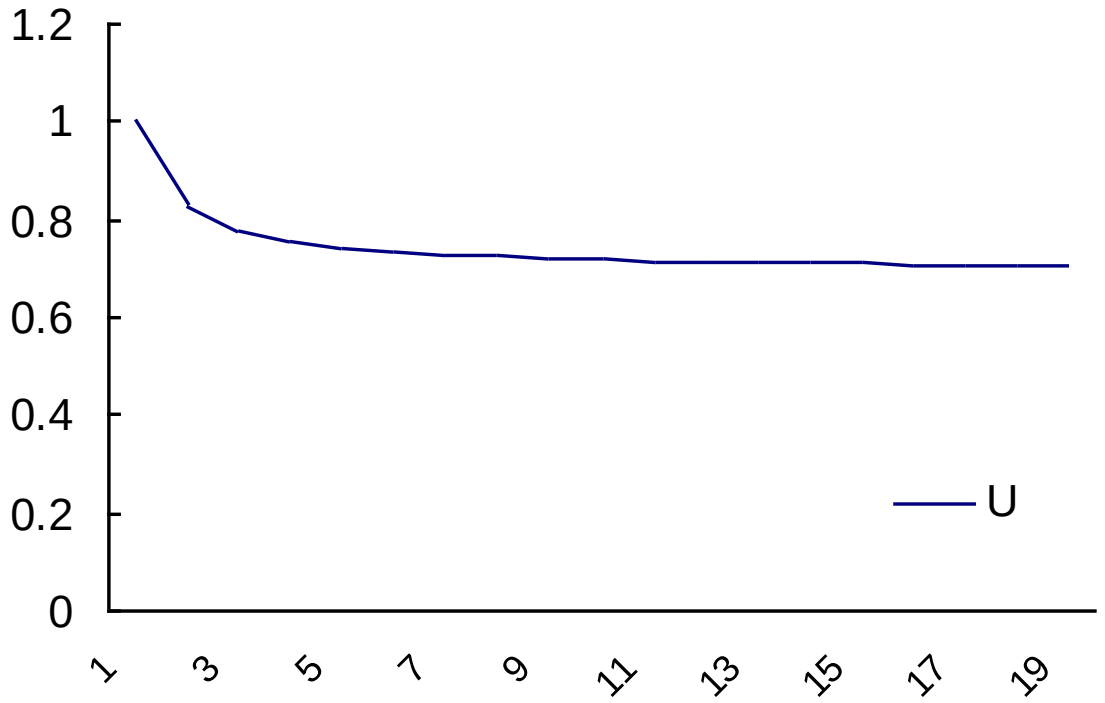
$T_1=2<3$

$T_2=R_0=2+4=6$

$R_1=2\times\lceil\frac{6}{3}\rceil+4\times\lceil\frac{6}{6}\rceil=8>6$

Rate-Monotonic Scheduling

- When x → infinitely large, U(x)→0.68



1	1
2	0.828427
3	0.779763
4	0.756828
5	0.743492
6	0.734772
7	0.728627
8	0.724062
9	0.720538
10	0.717735
11	0.715452
12	0.713557
13	0.711959
14	0.710593
15	0.709412

Earliest-Deadline-First Scheduling

- Definition
  - Feasible
    - A set of tasks is said to be feasible if there is some way to schedule the tasks without any deadline violations
  - Schedulable
    - Given a scheduling algorithm A
    - A set of tasks is said to be schedulable if algorithm A successfully schedule the tasks without any deadline violations
- Observations
  - A feasible task set may not be schedulable by RMS
  - If a task set is schedulable by some algorithm A, then it is feasible

如果可以通過某種方式安排任務而不違反任何截止日期，則認為一組任務是可行的

•給定調度算法A

•如果算法A成功地安排了任務而沒有違反任何期限，則可以計劃一組任務

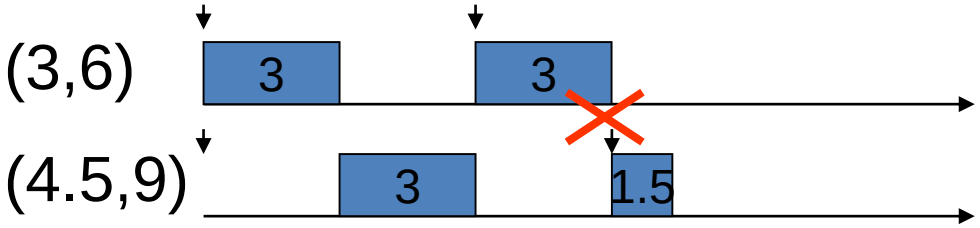
– RMS可能無法安排可行的任務集

–如果某個任務集可以通過某種算法A進行調度，那麼這是可行的

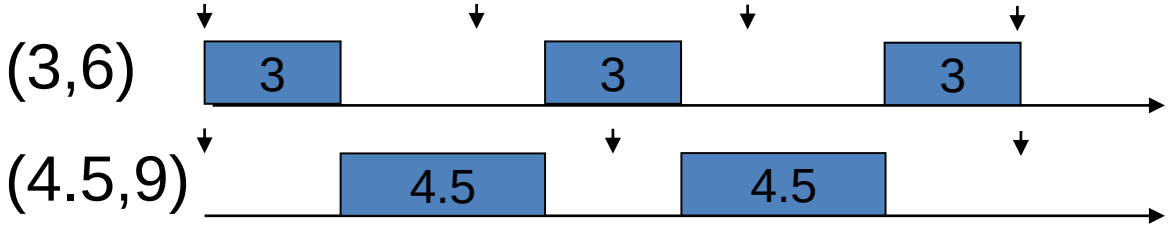
Earliest-Deadline-First Scheduling

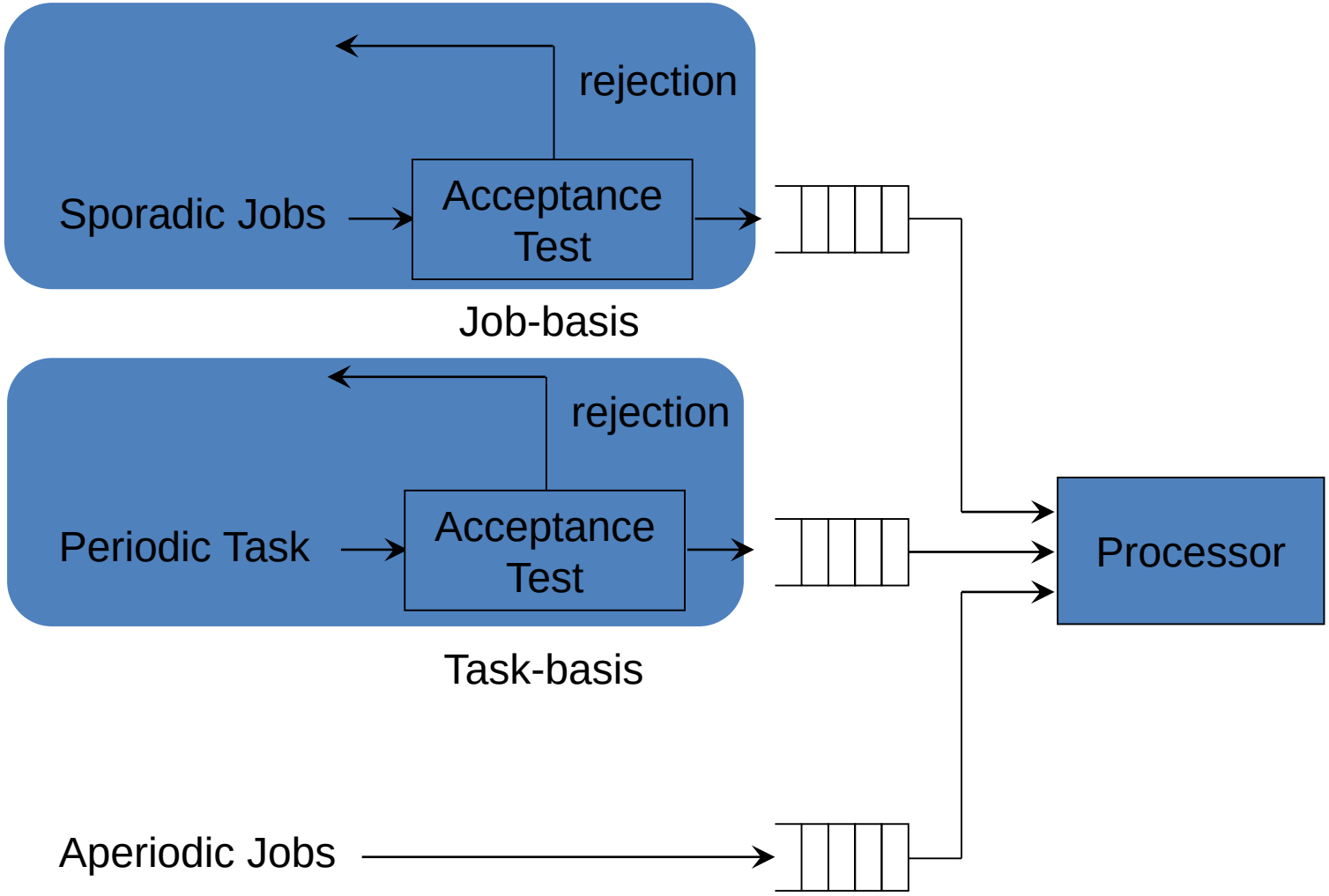
- Example

Not scheduable by RMS



Schedulable by EDF




rejection
Sporadic Jobs
Acceptance Test
Job-basis
rejection
Periodic Task
Acceptance Test
Task-basis
Processor
Aperiodic Jobs

Handling Aperiodic Jobs

- Approaches
  - Background execution
    - Improvement: slack stealing
  - Interrupt-driven execution
    - Improvement: slack stealing
  - Polled execution
    - Improvement: Bandwidth-preserving servers

ex: P63, 64

偷idle time

方法
–後台執行
•改進：偷懶
–中斷驅動的執行
•改進：偷懶
–輪詢執行
•改進：保留帶寬的服務

沒好偷 → 用第三個方法

Handling Aperiodic Jobs

優點：簡單

缺：時間長

- Background execution
  - Handle aperiodic jobs whenever there is no periodic jobs to execute
    - Extremely simple
    - Always produce correct schedule
    - Poor response time

•後台執行

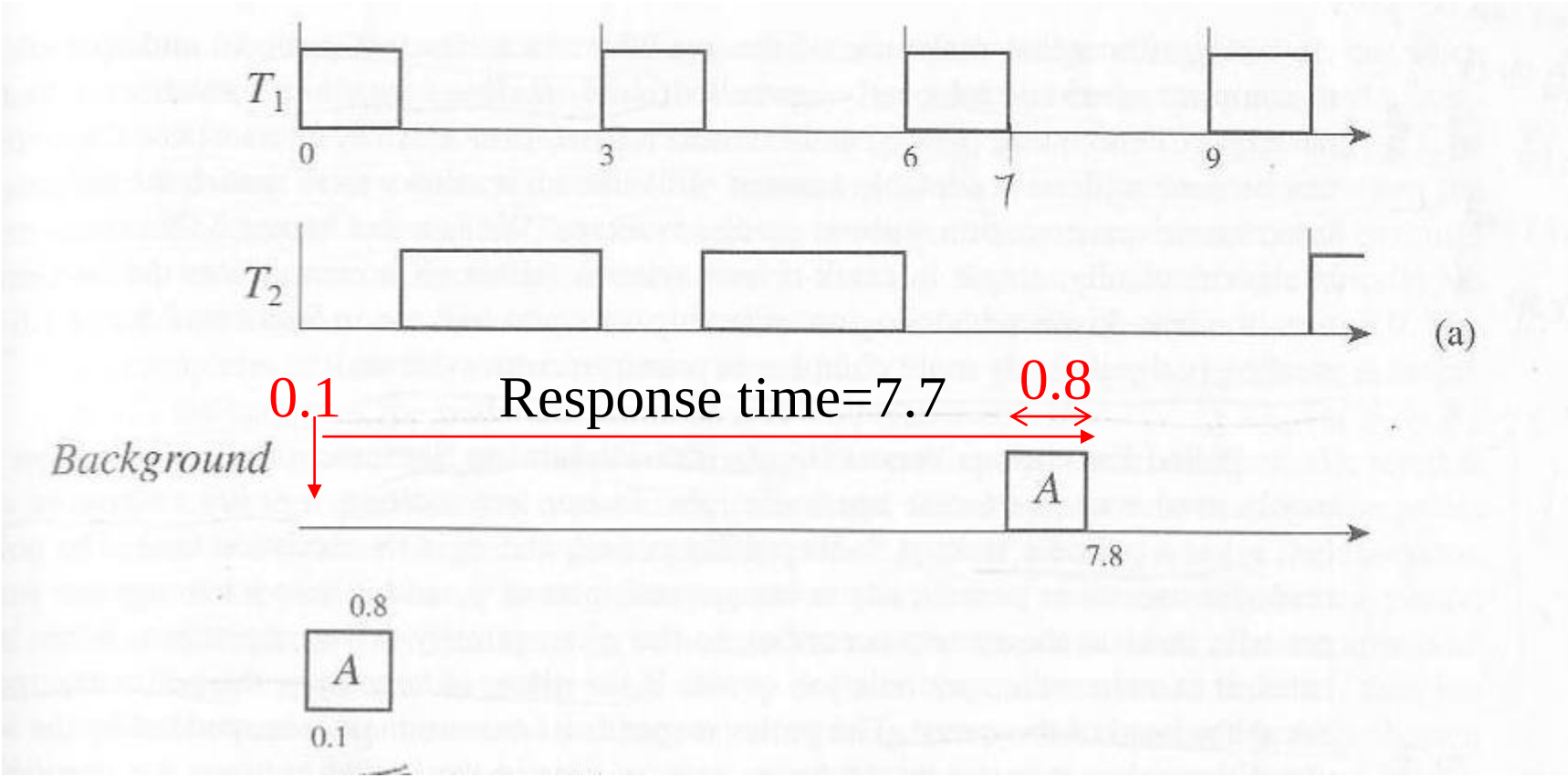
–在沒有工作的情況下處理非定期工作定期執行的工作

•非常簡單

•始終制定正確的時間表

•響應時間短

Background Execution



Handling Aperiodic Jobs

用在非常緊急的作業中
它的執行Time最短，但會引響周期性的工作

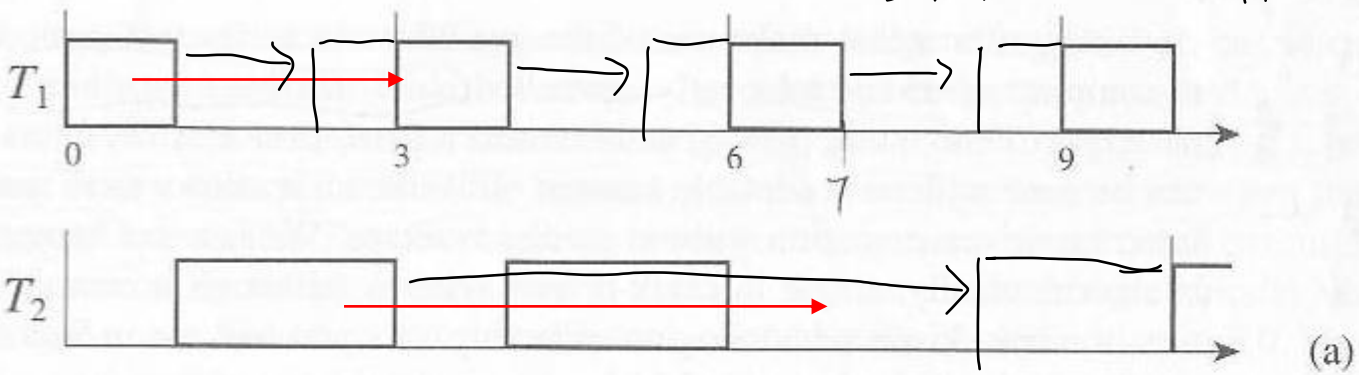
- Interrupt execution
  - An obvious extension to background execution
  - On arrivals, aperiodic jobs immediately interrupts the execution of any periodic jobs
  - Fastest response time
  - Potentially damage the schedulability of periodic jobs

•中斷執行
–後台執行的明顯擴展
–在到達時，非定期作業會立即中斷任何定期作業的執行
–最快的響應時間
–可能會損害定期性的可調度性職位

Interrupt Execution

slack stealing：每個工作都在deadline之前完成
有沒有空檔可以用，但因為工作
時間複雜所以不好用

Execution time=2.1

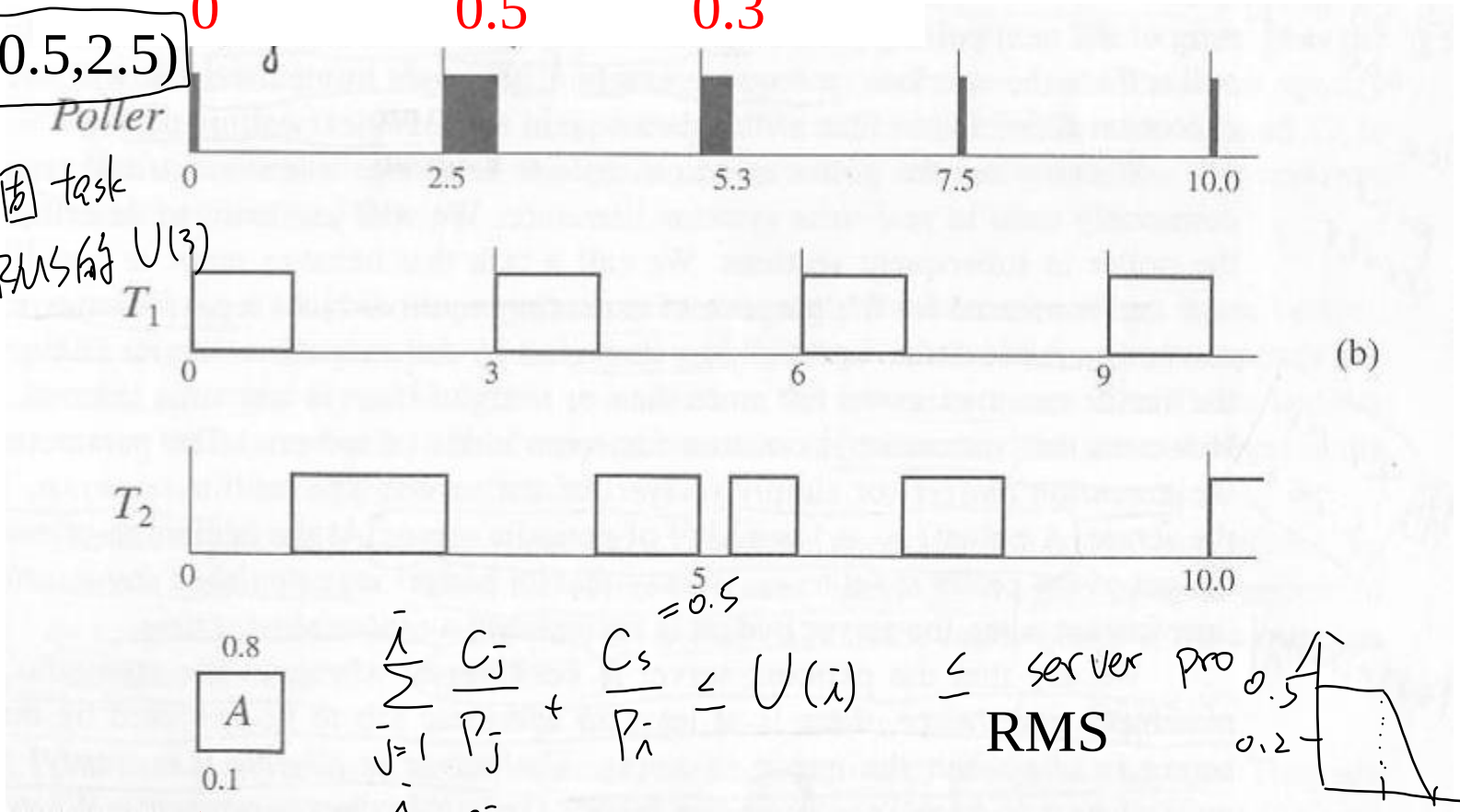


Handling Aperiodic Jobs

- Polled execution
  - –純週期性任務（輪詢服務器），用於服務非週期性作業隊列
  - – A purely periodic task (polling server) to serve a queue of aperiodic jobs
  - –當輪詢服務器獲得對CPU的控制權時，它將為隊列中的非定期作業提供服務
  - – When the polling server gains control of the CPU, it services aperiodic jobs in the queue
  - –如果隊列為空，則輪詢服務器立即掛起
  - – If the queue becomes empty, the polling server suspends immediately
    - •直到下一個輪詢週期才檢查隊列
    - • The queue is not checked until the next polling period

The polling server drops all its budget at time 0 since there is no ready aperiodic jobs

由於沒有現成的非定期作業，因此輪詢服務器會在時間0放棄所有預算



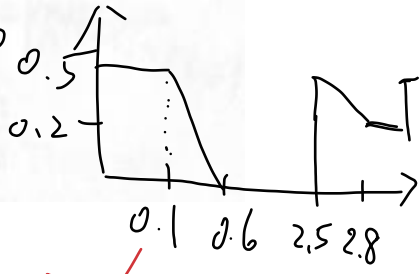
當3個 task 算 RMS 的 U(3)

$\sum_{j=1}^{i} \frac{C_j}{P_j} + \frac{C_s}{P_i} \le U(i) \le$ server pro

$=0.5$

RMS

$\sum_{j=1}^{i} \frac{C_j}{P_j} \le U(i) \ge$ server pro



3/10

Aperiodic job A arrives at time 0.1

Resources

- An entity used by a task.
  - Memory objects
    - Such as tables, global variables …
  - I/O devices.
    - Such as disks, communication transceivers.

•任務使用的實體。
–內存對象
•如表，全局變量…
– I / O設備。
•例如磁盘，通信收發

- A task must gain exclusive access to a shared resource to prevent data (or I/O status) from being corrupted. •任務必須獲得對共享資源的獨占訪問權限，以防止損壞數據（或I / O狀態）。
  - Mutual exclusion. –互斥。

Critical Sections

•部分代碼必須不可分割

- A portion of code must be indivisible
  - –保護共享資源不受競爭條件的破壞
  - – To protect shared resources from being corrupted due to race conditions
  - –可以通過使用中斷啟用/禁用或IPC機制來實現
  - – Could be implemented by using interrupt enable/disable or IPC mechanisms
    - •信號燈，事件，郵箱等
    - • Semaphores, events, mailboxes, etc

Reentrant Functions

可以同時調用可重入函數，而不會破壞任何數據。

- Reentrant functions can be invoked simultaneously without corrupting any data.
  - Reentrant functions use either local variables (on stacks) or synchronization mechanisms (such as semaphores).

可重入函數使用局部變量（在堆棧上）或同步機制（例如信號量）。

```
void strcpy(char *dest, char *src)
  {
      while (*dest++ = *src++) {
          ;
      }
      *dest = NUL;
  }
```

Non-Reentrant Functions

在競爭條件下，不可重入功能可能會破壞共享資源。

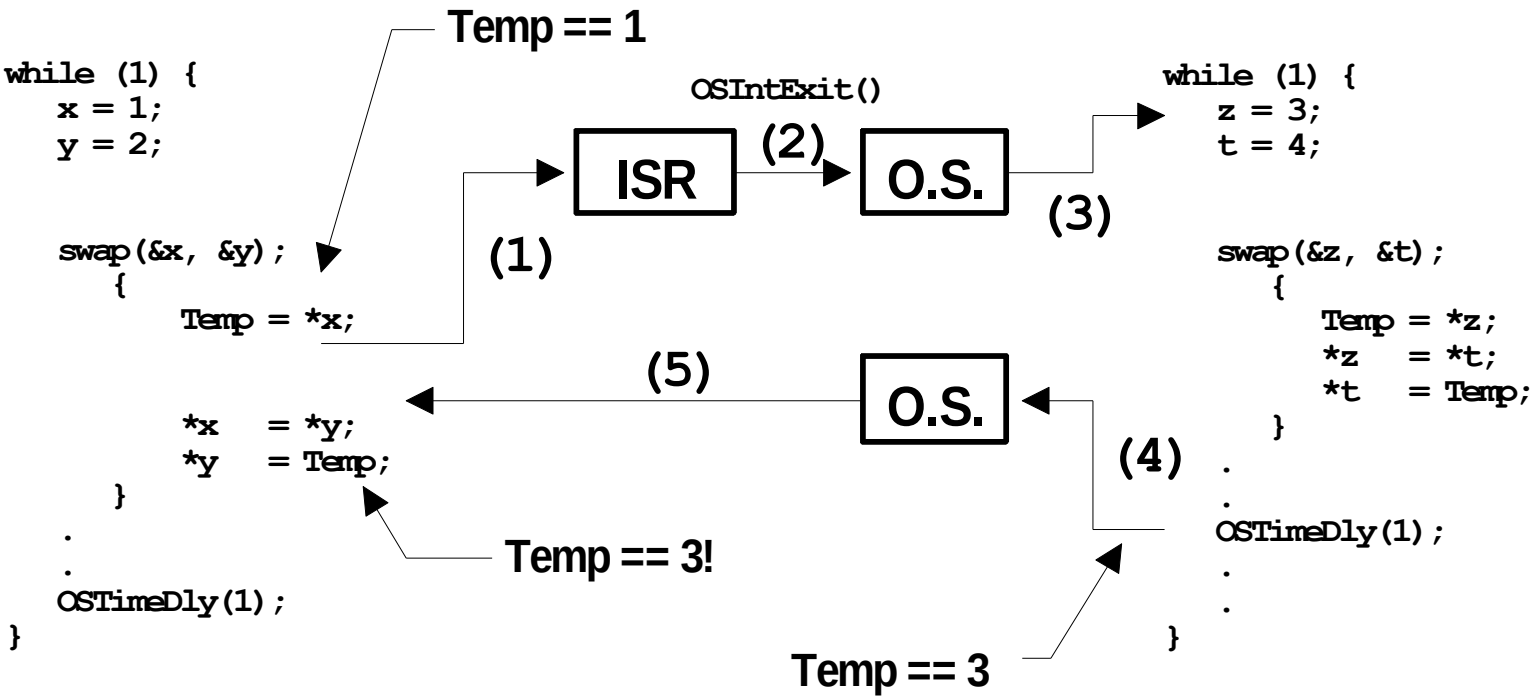
- Non-Reentrant functions might corrupt shared resources under race conditions.

```
int Temp;

  void swap(int *x, int *y)
  {
       Temp = *x;
       *x   = *y;
       *y   = Temp;
  }
```

LOW PRIORITY TASK

HIGH PRIORITY TASK



(1) When swap() is interrupted, TEMP contains 1.

(2) The ISR makes the higher priority task ready to run, so at the completion of the ISR, the kernel is invoked to switch to this task. The high priority task sets TEMP to 3 and swaps the contents of its variables correctly. (i.e., z=4 and t=3).

(3) The high priority task eventually relinquishes control to the low priority task be calling a kernel service to delay itself for one clock tick.

(4) The lower priority task is thus resumed. Note that at this point, TEMP is still set to 3! When the low priority task resumes execution, the task sets y to 3 instead of 1.

Non-Reentrant Functions

使代碼可重入的方法有幾種：

- There are several ways to make the code reentrant:
  - Declare TEMP as a local variable.
  - Disable interrupts and then enable interrupts.
  - Use a semaphore.

–將TEMP聲明為局部變量。
–禁用中斷，然後啟用中斷。
–使用信號量。

Mutual Exclusion

必須採用互斥來保護共享資源。

- Mutual exclusion must be adopted to protect shared resources.
  - Global variables, linked lists, pointers, buffers, and ring buffers.
  - I/O devices. –全局變量，鏈接列表，指針，緩衝區和環形緩衝區。
    – I / O設備。
- When a task is using a resource, the other tasks which are also interested in the resource must not be scheduled to run.

當一個任務正在使用資源時，不得安排對該資源也感興趣的其他任務運行

- Common techniques used are:
  - disable/enable interrupts
  - performing a test-and-set instruction
  - disabling scheduling
  - using synchronization mechanisms (such as semaphores).

常用的技術有：
–禁用/啟用中斷
–執行測試和設置指令
–禁用調度
–使用同步機制（例如信號量）。

保護 Interrupt 會影響到的 global value 料弄 value

Mutual Exclusion

I/O 無法使用

- Disabling/enabling interrupts:
  - OS_ENTER_CRITICAL() and OS_EXIT_CRITICAL()
  - All events are masked since interrupts are disabled.
  - Tasks which do not affect the resources-to-protect are also postponed.
  - Must not disable interrupt before calling system services.

禁用/啟用中斷：

– OS_ENTER_CRITICAL（）和OS_EXIT_CRITICAL（）

–由於禁用了中斷，因此所有事件都被屏蔽。

–不影響要保護資源的任務也被推遲。

–在調用系統服務之前一定不能禁止中斷。

```
void Function (void)
{
    OS_ENTER_CRITICAL();
    .
    .     /* You can access shared data in here */
    .
    OS_EXIT_CRITICAL();
}
```

Mutual Exclusion

- Disabling/Enabling Scheduling:
  - No preemptions could happen while the scheduler is disabled.
  - However, interrupts still happen.
    - ISR's could still corrupt shared data.
    - Once an ISR is done, the interrupted task is always resumed even there are high priority tasks ready.
  - Rescheduling might happen right after the scheduler is re-enabled.
  - Higher overheads!

禁用/啟用調度：
–禁用調度程序後，不會發生搶占。
–但是，中斷仍然會發生。
•ISR仍可能破壞共享數據。
•一旦完成了ISR，即使已準備好高優先級的任務，被中斷的任務也將始終恢復。
–重新啟用調度程序後，可能會立即進行重新調度。
–更高的間接費用！

```
void Function (void)
{
    OSSchedLock();
    .      /* You can access shared data
    .         in here (interrupts are recognized) */
    OSSchedUnlock();
}
```

Mutual Exclusion

信號量：
–由內核提供。
–信號量用於：
•控制對共享資源的訪問。
•發出事件發生的信號。
•允許任務同步其活動。
–對受保護資源不感興趣的高優先級任務仍可以計劃運行。

- Semaphores:
  - Provided by the kernel.
  - Semaphores are used to:
    - Control access to a shared resource.
    - Signal the occurrence of an event.
    - Allow tasks to synchronize their activities.
  - Higher priority tasks which does not interested in the protected resources can still be scheduled to run.

```
OS_EVENT *SharedDataSem;
void Function (void)
{
    INT8U err;
    OSSemPend(SharedDataSem, 0, &err);
    .     /* You can access shared data
    .        in here (interrupts are recognized) */
    OSSemPost(SharedDataSem);
}
```

Semaphores

- Semaphores:
  - Three kinds of semaphores:
    - Counting semaphore (init >1)
    - Binary semaphore (init = 1)
    - Rendezvous semaphore (init = 0)
  - On event posting, a waiting task is released from the waiting queue.
    - The highest-priority task.
    - FIFO (not supported by µC/OS-II)
  - Interrupts and scheduling are still enabled under the use of semaphores.

信號量：
–三種信號量：
•計數信號量（init> 1）
•二進制信號量（初始= 1）
•集合信號量（初始= 0）
–在事件發佈時，從等待隊列中釋放等待任務。
•最高優先級的任務。
•FIFO（µC/ OS-II不支持）
–在使用信號量的情況下，仍允許中斷和調度。

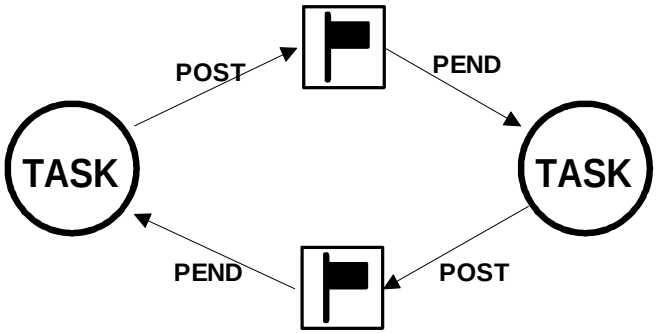
Synchronization

可以使用兩個信號量來集合兩個任務。

- Two semaphores could be used to rendezvous two tasks.
  - It can not be used to synchronize between ISR's and tasks. –不能用於在ISR和任務之間進行同步。
  - For example, a kernel-mode thread could synchronize with a user-mode worker thread which performs complicated jobs.
    –例如，內核模式線程可以與執行複雜作業的用戶模式工作線程同步。

Synchronization

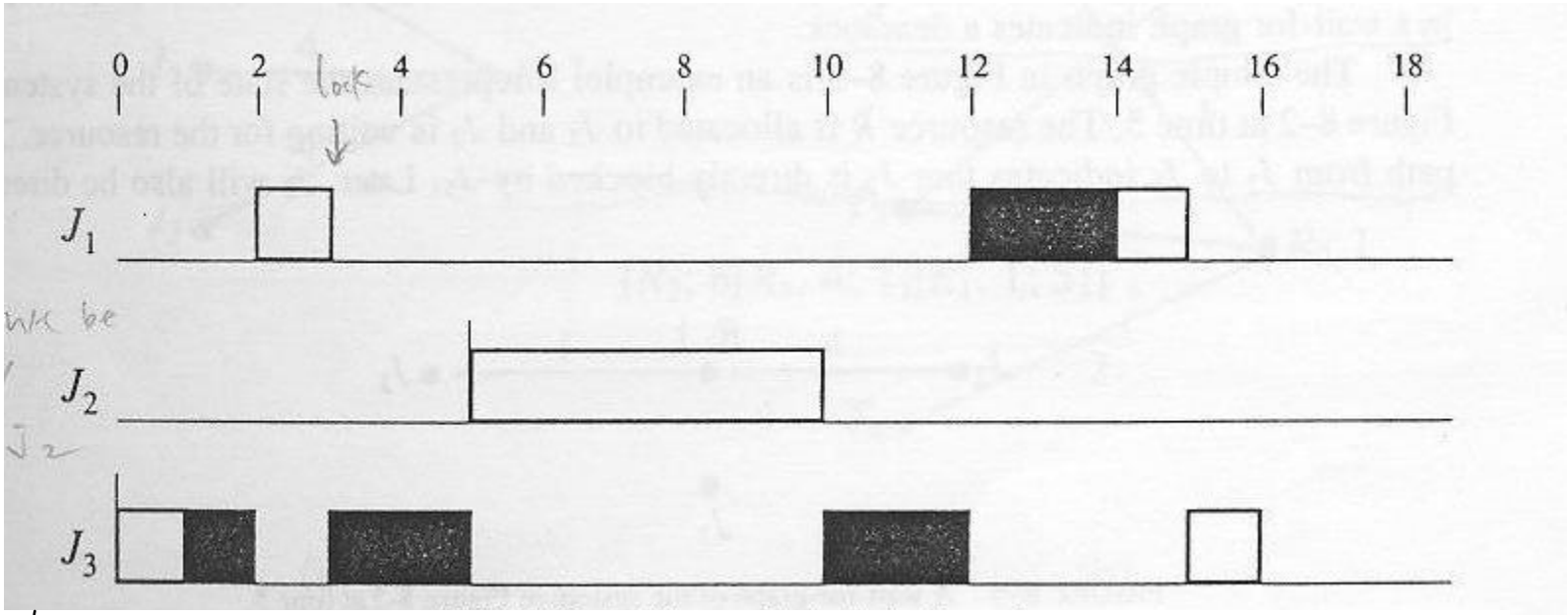
```
Task1()
{
    for (;;) {
            Perform operation;
            Signal task #2;                 (1)
            Wait for signal from task #2;   (2)
            Continue operation;
    }
}
```



** Semaphores are both initialized to 0

```
Task2()
{
    for (;;) {
            Perform operation;
            Signal task #1;                 (3)
            Wait for signal from task #1;   (4)
            Continue operation;
    }
}
```

Priority inversion



1. mutual exclusion
2. circular wait
3. atomic
4. hold and wait

缺一不可

4個皆滿足 => Deadlocks

Deadlocks

- Tasks circularly wait for certain resources which are already locked by another tasks. 任務循環等待已被另一任務鎖定的某些資源。
  - No task could finish executing under such a circumstance. –在這種情況下，任何任務都無法完成執行。
- Deadlocks in static systems can be detected and resolved in advance. 可以提前檢測並解決靜態系統中的死鎖。
- Deadlocks are not easy to detect and resolve in a on-line fashion. 僵局不容易以在線方式檢測和解決。
  - A brute-force way to avoid deadlocks is to set a timeout when acquiring a semaphore.
  - The elegant way is to adopt resource synchronization protocols.
    - Priority Ceiling Protocol (PCP), Stack Resource Policy (SRP)

–避免死鎖的暴力方式是在獲取信號量時設置超時。
–優雅的方法是採用資源同步協議。
•優先天花板協議（PCP），堆棧資源策略（SRP）

Summary

- Scheduling algorithms
- Resource management

•調度算法
• 資源管理

3/24