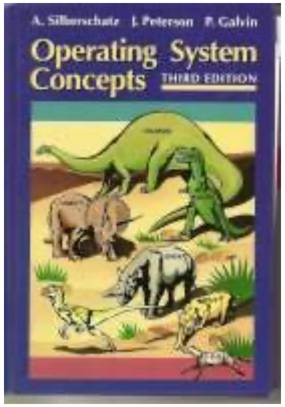
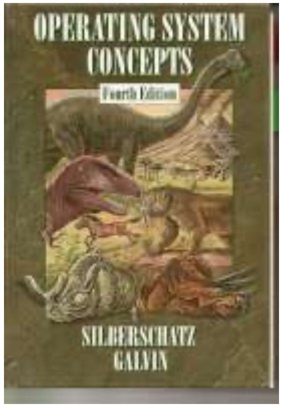
作業系統基本觀念複習 Process & Thread

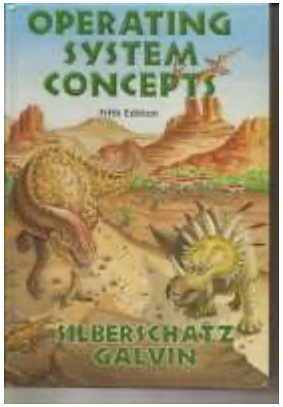
Yiling Lai

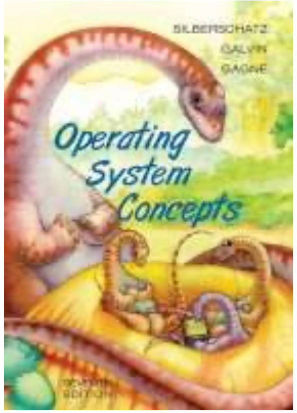
2011/8/24

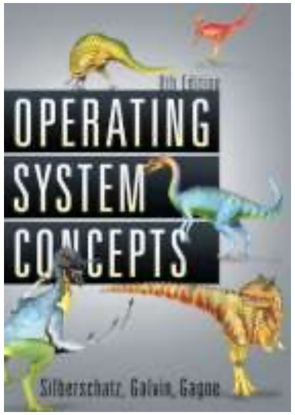
- 恐龍書
  Operating System Concepts
  Silberschatz, Galvin, Gagne


A. Silberschatz J. Peterson P. Galvin
Operating System Concepts
THIRD EDITION







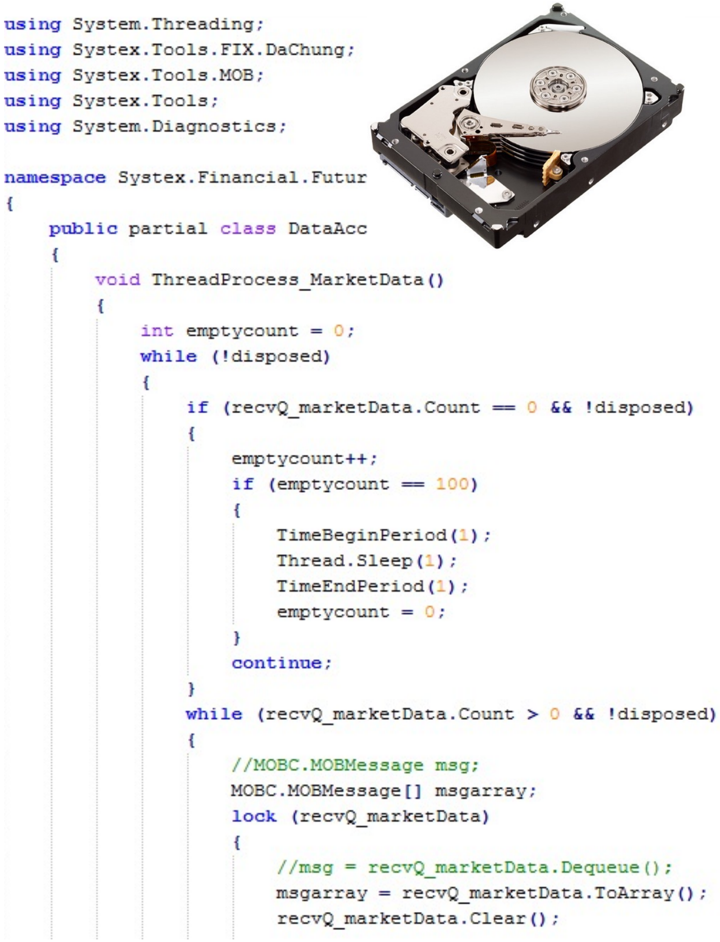


什麼是Process?

- A program in execution. 執行中的程式

什麼是Process?

- A program in execution. 執行中的程式

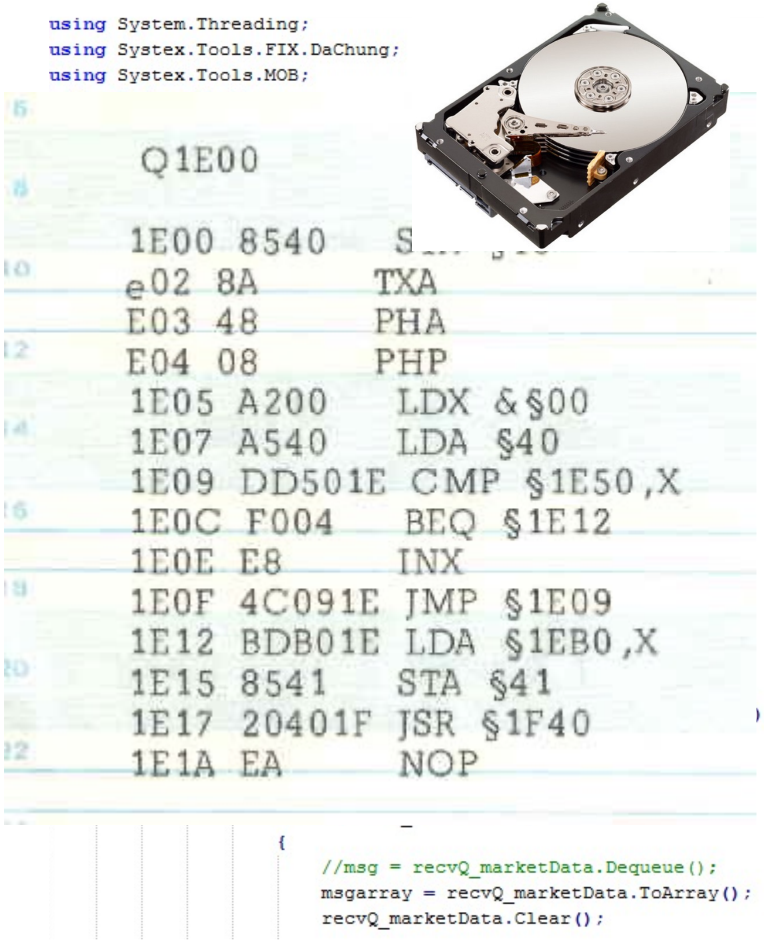


```
using System.Threading;
using Systex.Tools.FIX.DaChung;
using Systex.Tools.MOB;
using Systex.Tools;
using System.Diagnostics;

namespace Systex.Financial.Futur
{
    public partial class DataAcc
    {
        void ThreadProcess_MarketData()
        {
            int emptycount = 0;
            while (!disposed)
            {
                if (recvQ_marketData.Count == 0 && !disposed)
                {
                    emptycount++;
                    if (emptycount == 100)
                    {
                        TimeBeginPeriod(1);
                        Thread.Sleep(1);
                        TimeEndPeriod(1);
                        emptycount = 0;
                    }
                    continue;
                }
                while (recvQ_marketData.Count > 0 && !disposed)
                {
                    //MOBC.MOBMessage msg;
                    MOBC.MOBMessage[] msgarray;
                    lock (recvQ_marketData)
                    {
                        //msg = recvQ_marketData.Dequeue();
                        msgarray = recvQ_marketData.ToArray();
                        recvQ_marketData.Clear();
```

什麼是Process?

- A program in execution. 執行中的程式



```
using System.Threading;
using Systex.Tools.FIX.DaChung;
using Systex.Tools.MOB;
```

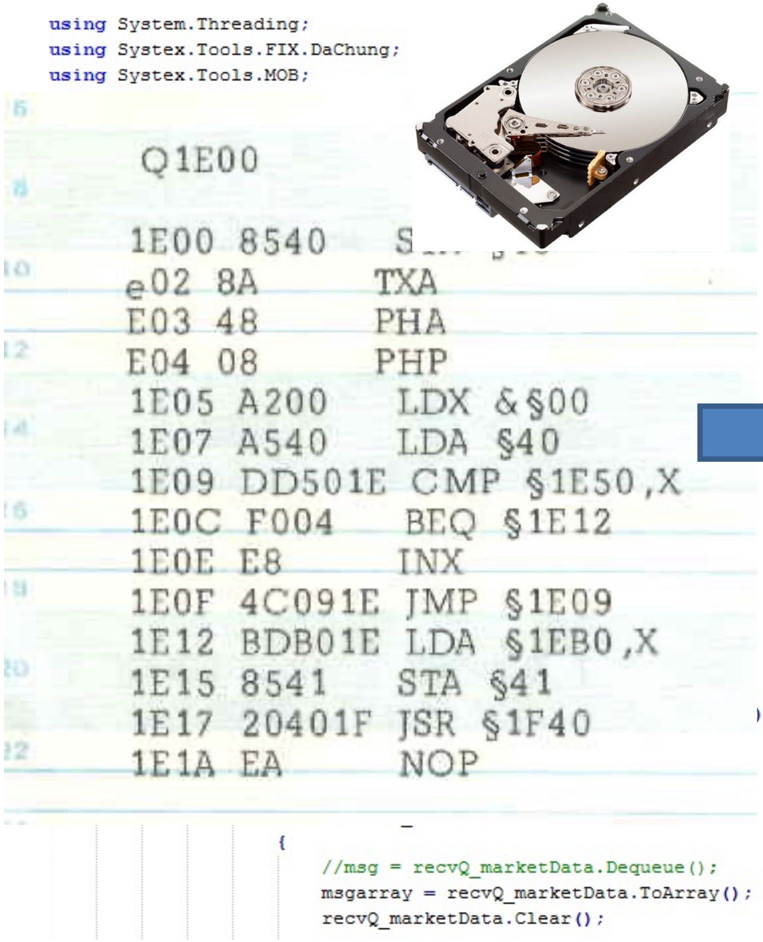
```
Q1E00

1E00 8540   S
e02 8A     TXA
E03 48     PHA
E04 08     PHP
1E05 A200   LDX &$00
1E07 A540   LDA $40
1E09 DD501E CMP $1E50,X
1E0C F004   BEQ $1E12
1E0E E8     INX
1E0F 4C091E JMP $1E09
1E12 BDB01E LDA $1EB0,X
1E15 8541   STA $41
1E17 20401F JSR $1F40
1E1A EA     NOP
```

```
{
    //msg = recvQ_marketData.Dequeue();
    msgarray = recvQ_marketData.ToArray();
    recvQ_marketData.Clear();
```

什麼是Process?

- A program in execution. 執行中的程式

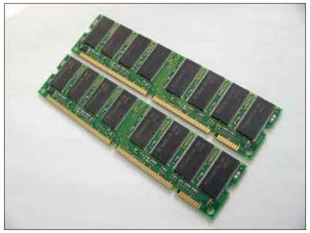


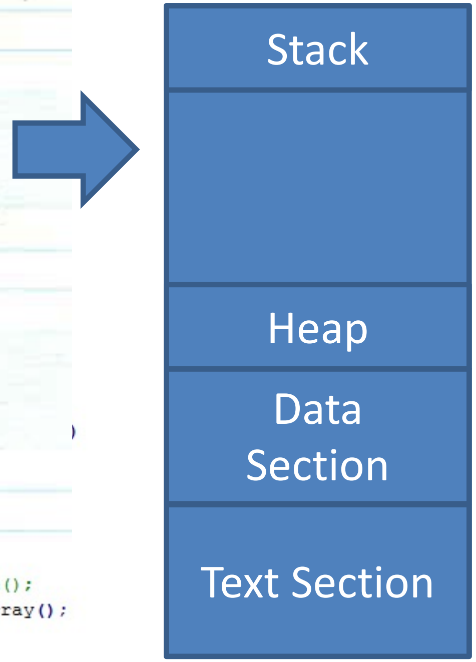
```
using System.Threading;
using Systex.Tools.FIX.DaChung;
using Systex.Tools.MOB;
```

```
 Q1E00

 1E00 8540    S
e02  8A       TXA
E03  48       PHA
E04  08       PHP
 1E05 A200    LDX &§00
 1E07 A540    LDA §40
 1E09 DD501E CMP §1E50,X
 1E0C F004    BEQ §1E12
 1E0E E8      INX
 1E0F 4C091E JMP §1E09
 1E12 BDB01E LDA §1EB0,X
 1E15 8541    STA §41
 1E17 20401F JSR §1F40
 1E1A EA      NOP
```

```
{
    //msg = recvQ_marketData.Dequeue();
    msgarray = recvQ_marketData.ToArray();
    recvQ_marketData.Clear();
```

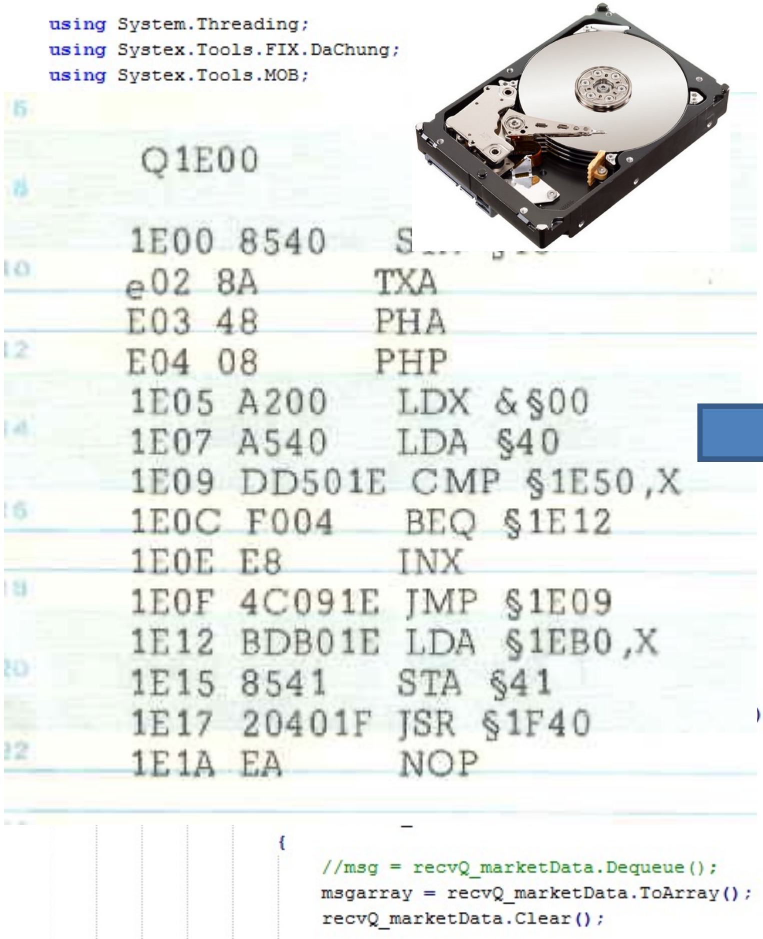




Program counter

什麼是Process?

- A program in execution. 執行中的程式

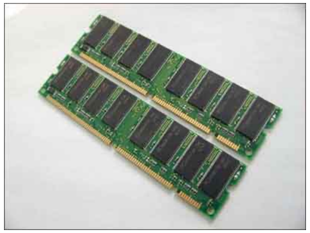


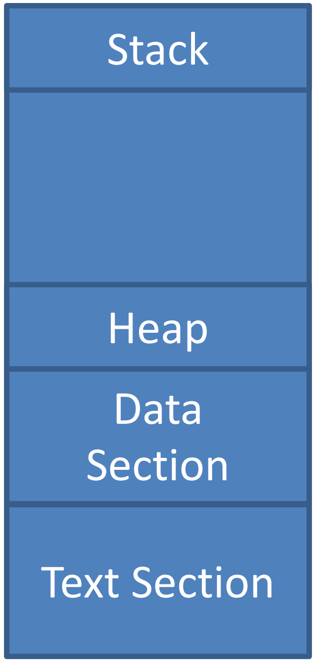
```
using System.Threading;
using Systex.Tools.FIX.DaChung;
using Systex.Tools.MOB;
```

```
     Q1E00

      1E00 8540    S
     e02 8A       TXA
     E03 48       PHA
     E04 08       PHP
      1E05 A200    LDX &§00
      1E07 A540    LDA §40
      1E09 DD501E CMP §1E50,X
      1E0C F004     BEQ §1E12
      1E0E E8       INX
      1E0F 4C091E JMP §1E09
      1E12 BDB01E LDA §1EB0,X
      1E15 8541     STA §41
      1E17 20401F JSR §1F40
      1E1A EA       NOP
```

```
{
    //msg = recvQ_marketData.Dequeue();
    msgarray = recvQ_marketData.ToArray();
    recvQ_marketData.Clear();
```



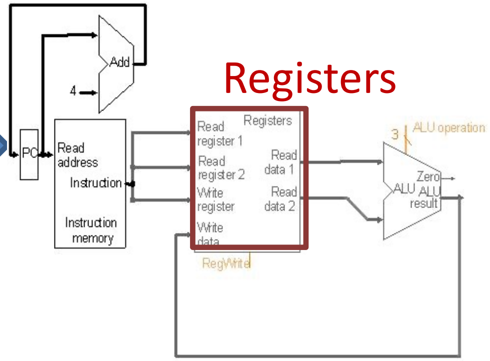


Stack

Heap

Data Section

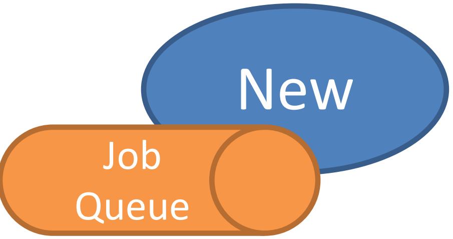
Text Section



Registers

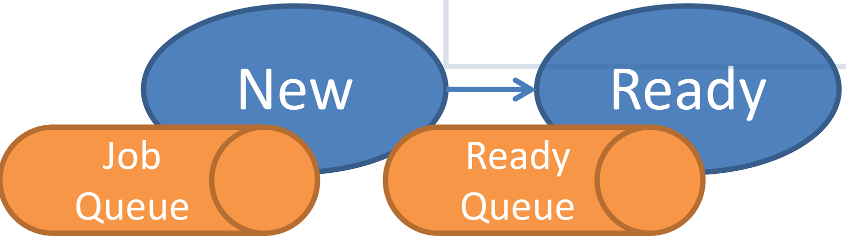
Program counter

Process的一生 (STD)

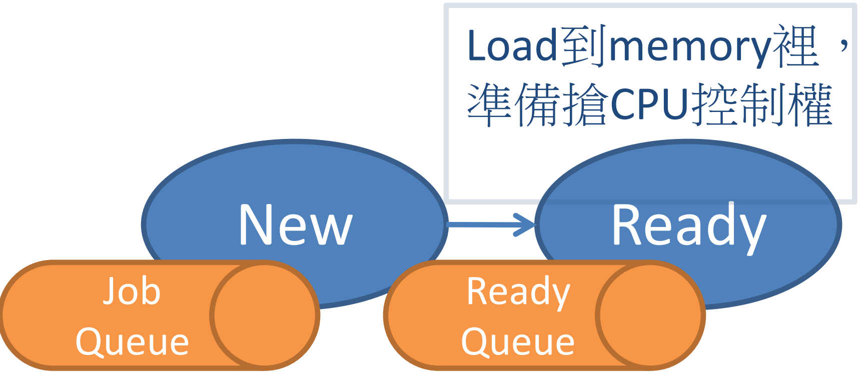


Process的一生 (STD)

Load到memory裡，準備搶CPU控制權



Process的一生 (STD)

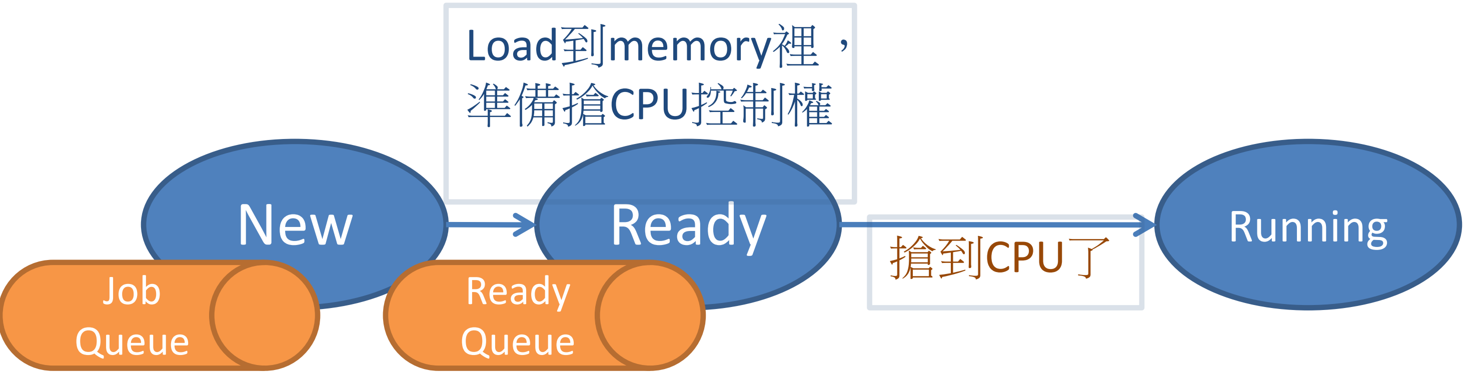


Load到memory裡，準備搶CPU控制權

PCB

Process ID
Process State
PC
CPU Registers
….

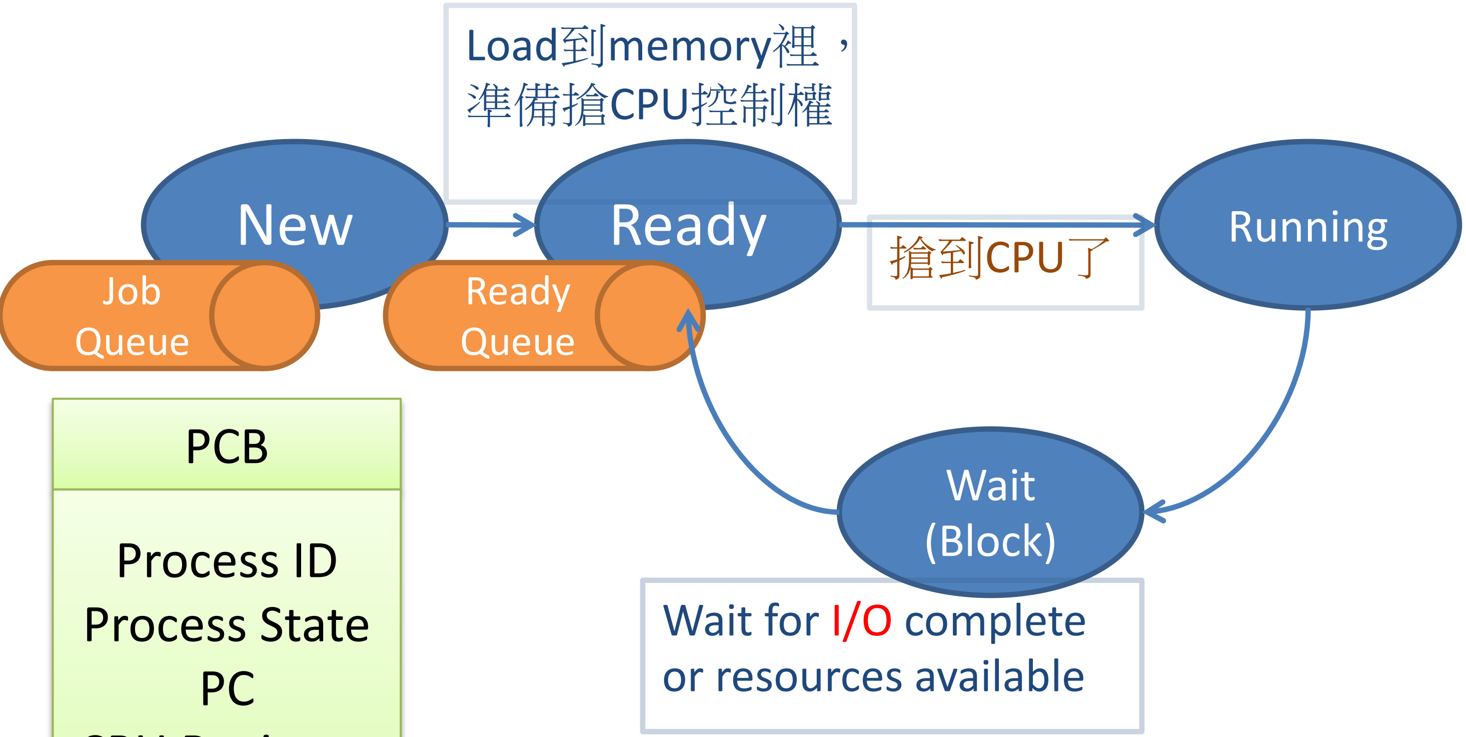
Process的一生 (STD)


Load到memory裡，準備搶CPU控制權

搶到CPU了

PCB
Process ID Process State PC CPU Registers ….

Process的一生 (STD)



Load到memory裡，準備搶CPU控制權

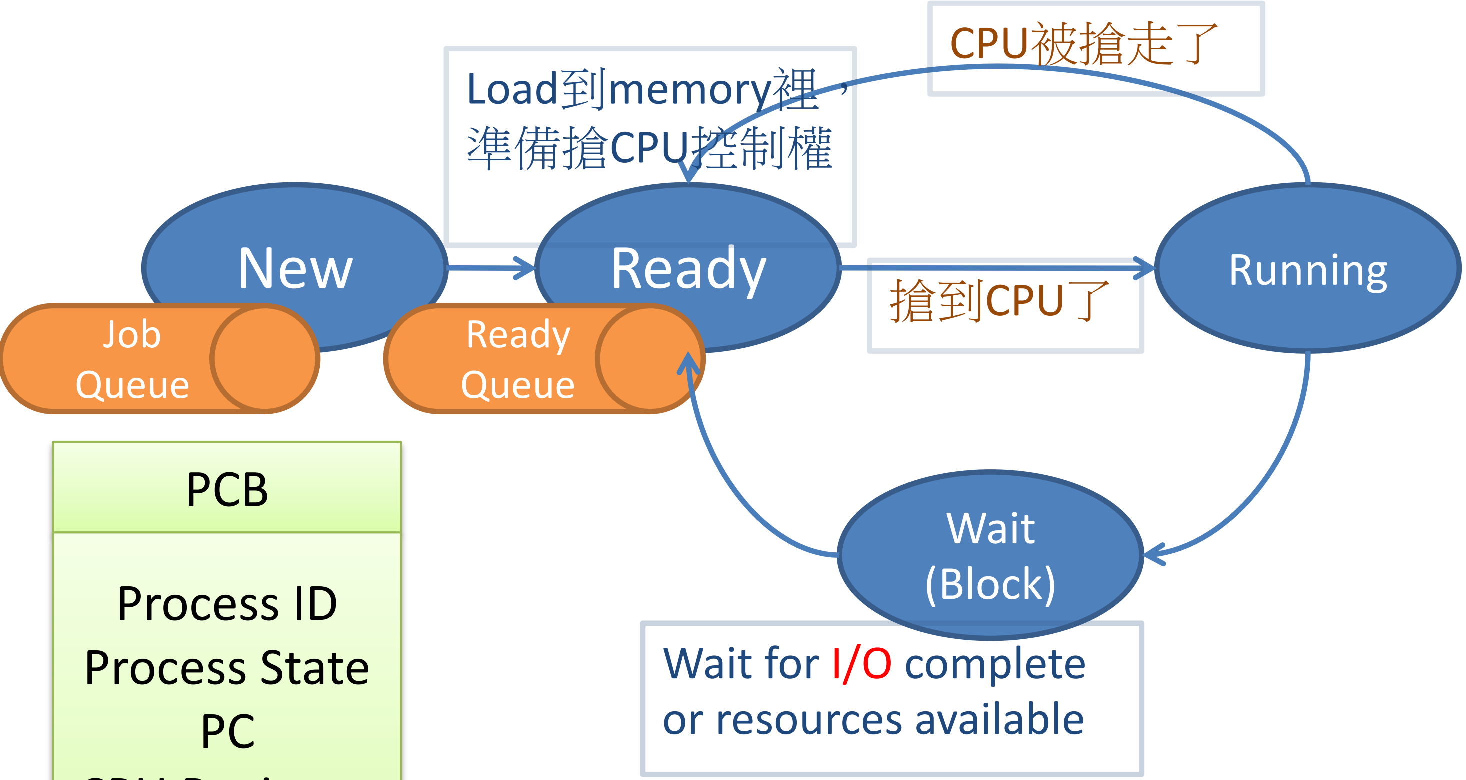
搶到CPU了

Wait for I/O complete or resources available

PCB

Process ID
Process State
PC
CPU Registers
....

Process的一生 (STD)



New

Ready

Running

Wait
(Block)

Job
Queue

Ready
Queue

Load到memory裡，
準備搶CPU控制權

CPU被搶走了

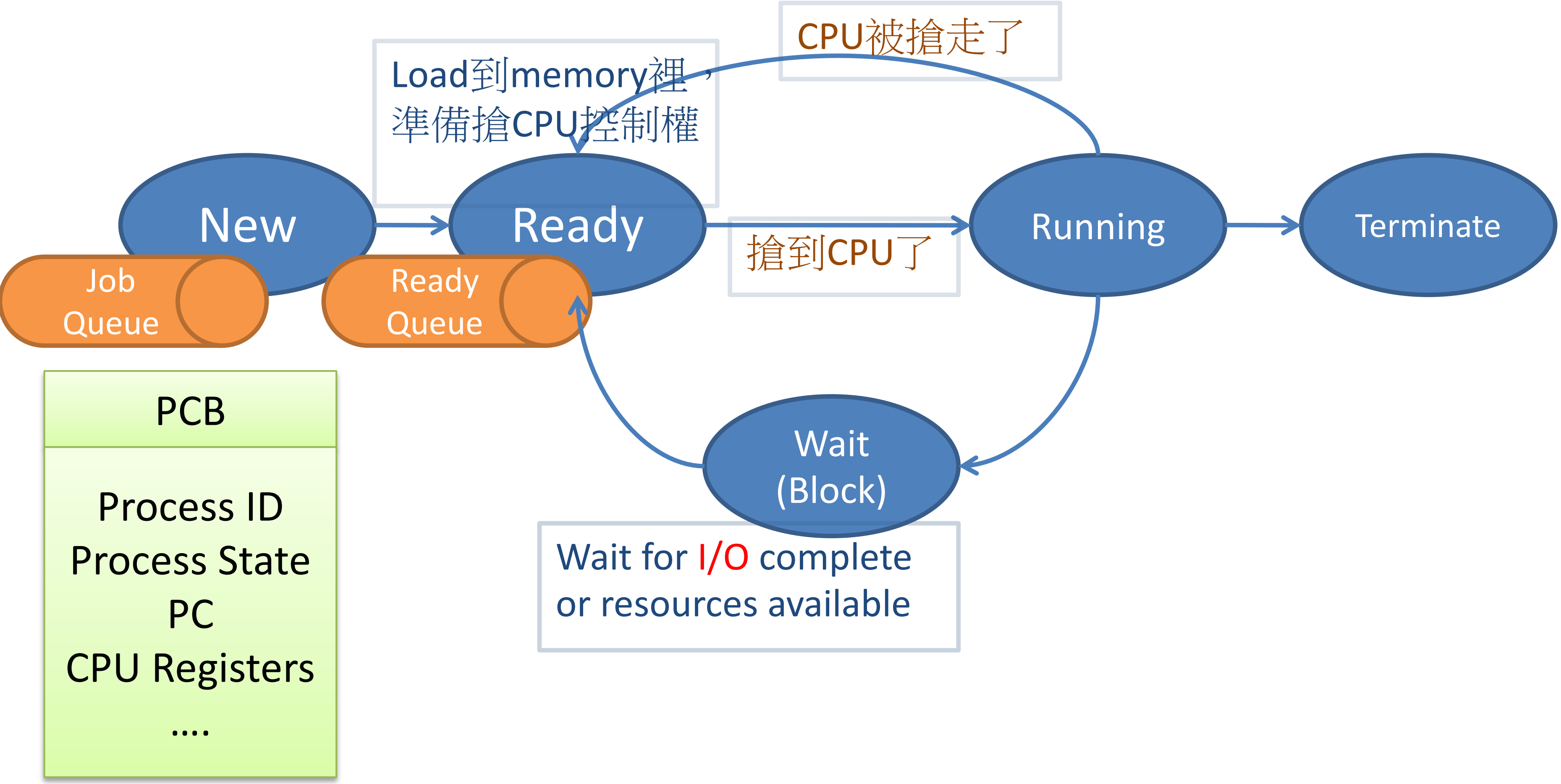
搶到CPU了

Wait for I/O complete
or resources available

PCB

Process ID
Process State
PC
CPU Registers
….

Process的一生 (STD)



Load到memory裡，準備搶CPU控制權

CPU被搶走了

New

Ready

Running

Terminate

搶到CPU了

Job Queue

Ready Queue

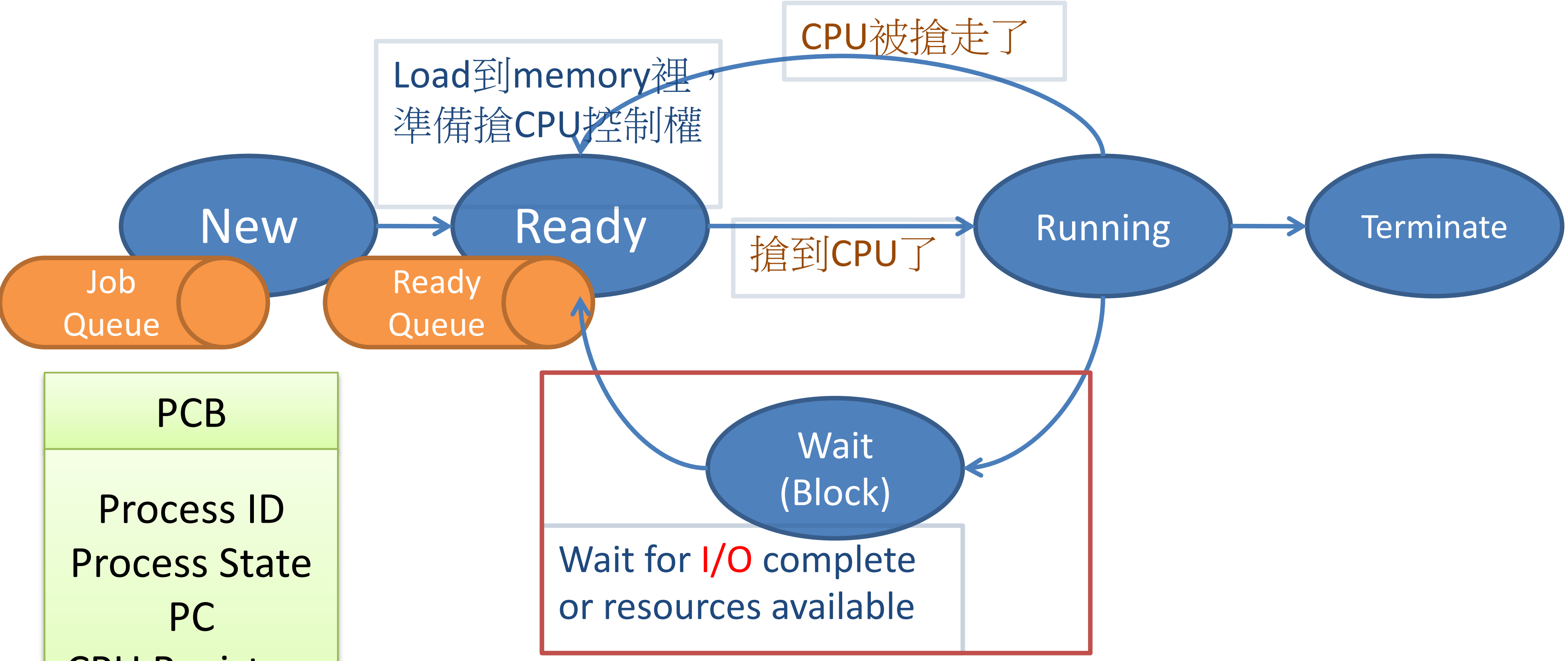
Wait (Block)

Wait for I/O complete or resources available

PCB

Process ID
Process State
PC
CPU Registers
….

Process的一生 (STD)



CPU被搶走了

Load到memory裡，準備搶CPU控制權

New

Ready

Running

Terminate

Job Queue

Ready Queue

搶到CPU了

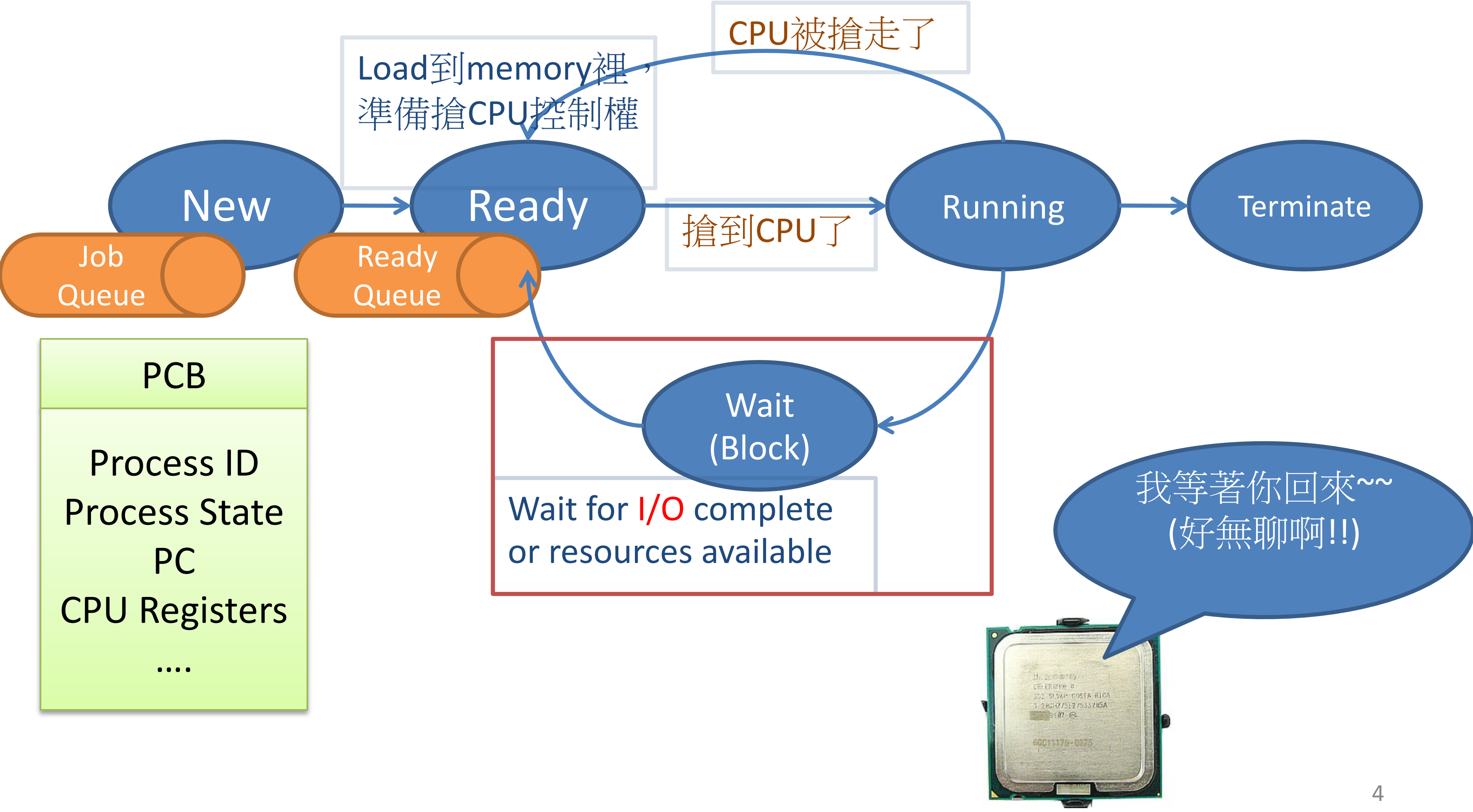
Wait (Block)

Wait for I/O complete or resources available

PCB

Process ID
Process State
PC
CPU Registers
….

Process的一生 (STD)



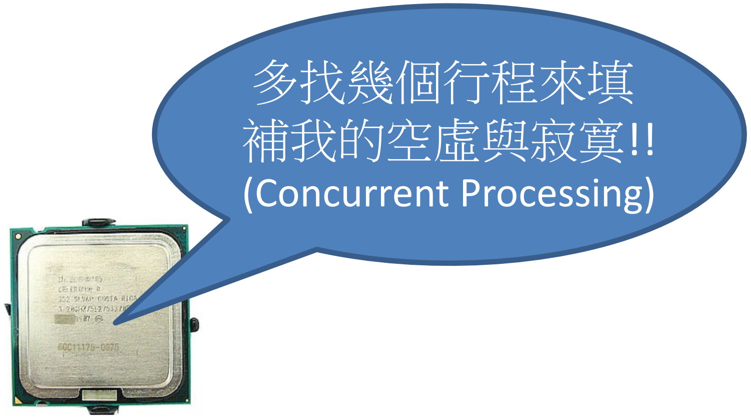
PCB

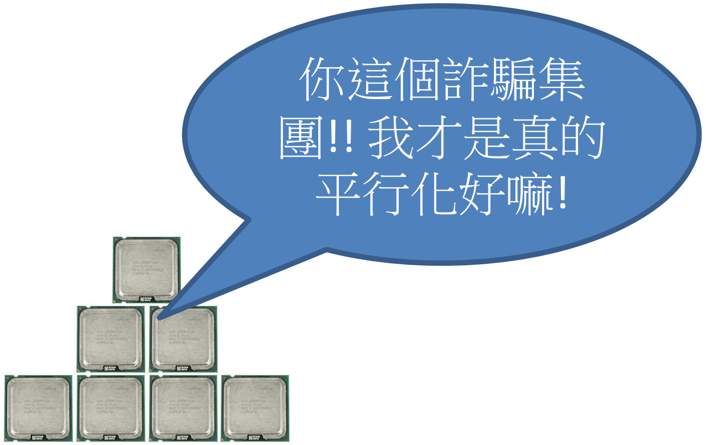
Process ID
Process State
PC
CPU Registers
….

多找幾個行程來填
補我的空虛與寂寞!!
(Concurrent Processing)

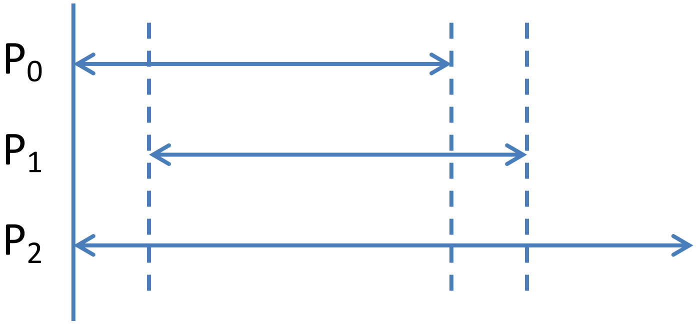
多找幾個行程來填
補我的空虛與寂寞!!
(Concurrent Processing)

你這個詐騙集
團!! 我才是真的
平行化好嘛!





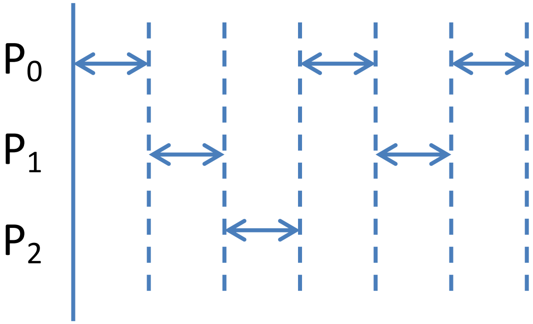
Parallel Processing



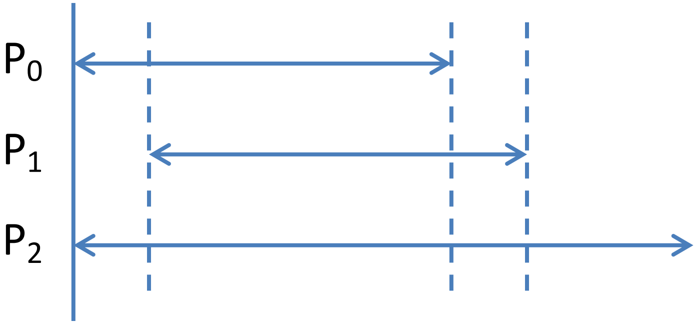
多找幾個行程來填補我的空虛與寂寞!!
(Concurrent Processing)

你這個詐騙集團!! 我才是真的平行化好嘛!

Concurrnt Processing



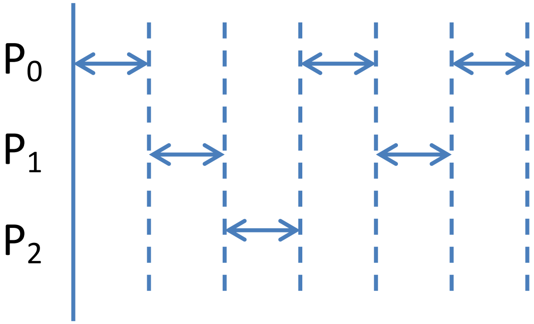
Parallel Processing



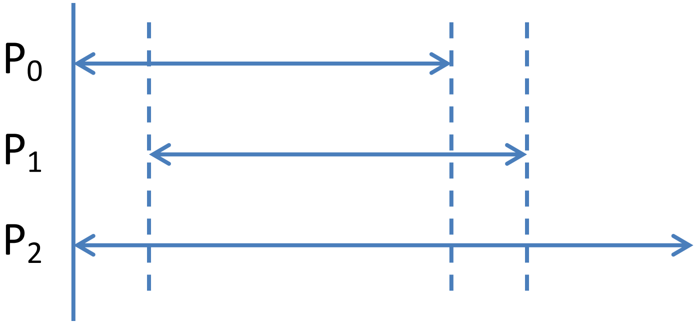
多找幾個行程來填
補我的空虛與寂寞!!
(Concurrent Processing)

你這個詐騙集
團!! 我才是真的
平行化好嘛!

Concurrnt Processing



Parallel Processing



大家輪流使用CPU，但是怎麼輪？？

我就知道沒有我果然不行，交給我吧!!

1. CPU的使用率(Utilization)要最大
2. 工作產能(Throughput)要高
3. Process等待的時間要短
4. 完成時間(Turnaround time)要短
5. 資源利用率也要大
6. 要公平
7. …

1. CPU的使用率(Utilization)要最大
2. 工作產能(Throughput)要高
3. Process等待的時間要短
4. 完成時間(Turnaround time)要短
5. 資源利用率也要大
6. 要公平
7. …

- FIFO: First come first out
- SJF: Shortest Job First
- SRJF: Shortest Remaining Time Job First
- Priority Scheduling
- RR: Round Robin
- …..

1. CPU的使用率(Utilization)要最大
2. 工作產能(Throughput)要高
3. Process等待的時間要短
4. 完成時間(Turnaround time)要短
5. 資源利用率也要大
6. 要公平
7. …

- FIFO: First come first out
- SJF: Shortest Job First
- SRJF: Shortest Remaining Time Job First
- Priority Scheduling
- RR: Round Robin
- …..

Scheduling Algorithm

FIFO: 誰先來誰先做

SJF: 你要3個小時？我只要3分鐘，先給我用一下吧~

(SJF: shortest time job first)

Process	CPU Time
P_0	14
P_1	5
P_2	2

1. CPU的使用率要最大
2. 工作產能要高
3. Process等待的時間要短
4. 完成時間要短
5. 資源利用率也要大
6. 要公平

FIFO：

SJF：

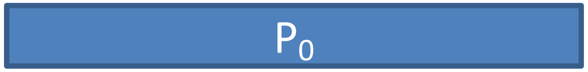
FIFO: 誰先來誰先做

SJF: 你要3個小時？我只要3分鐘，先給我用一下吧~

(SJF: shortest time job first)

Process	CPU Time
P_0	14
P_1	5
P_2	2

FIFO：



SJF：

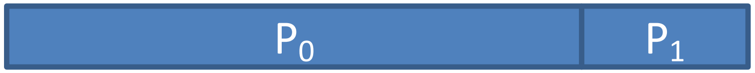
1. CPU的使用率要最大
2. 工作產能要高
3. Process等待的時間要短
4. 完成時間要短
5. 資源利用率也要大
6. 要公平

FIFO: 誰先來誰先做
SJF: 你要3個小時？我只要3分鐘，先給我用一下吧~
(SJF: shortest time job first)

Process	CPU Time
P_0	14
P_1	5
P_2	2

1. CPU的使用率要最大
2. 工作產能要高
3. Process等待的時間要短
4. 完成時間要短
5. 資源利用率也要大
6. 要公平

FIFO：

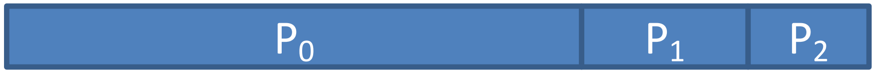


SJF：

FIFO: 誰先來誰先做
SJF: 你要3個小時？我只要3分鐘，先給我用一下吧~
(SJF: shortest time job first)

Process	CPU Time
P_0	14
P_1	5
P_2	2

FIFO：



P_0 P_1 P_2

1. CPU的使用率要最大
2. 工作產能要高
3. Process等待的時間要短
4. 完成時間要短
5. 資源利用率也要大
6. 要公平

SJF：

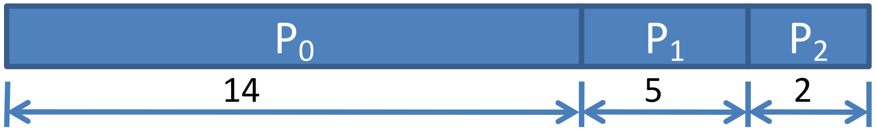
FIFO: 誰先來誰先做

SJF: 你要3個小時？我只要3分鐘，先給我用一下吧~

(SJF: shortest time job first)

Process	CPU Time
P_0	14
P_1	5
P_2	2

FIFO：



平均等待時間 = (0+ 14+19)/3 = 11

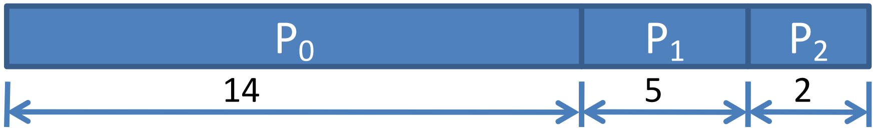
1. CPU的使用率要最大
2. 工作產能要高
3. Process等待的時間要短
4. 完成時間要短
5. 資源利用率也要大
6. 要公平

SJF：

FIFO: 誰先來誰先做
SJF: 你要3個小時？我只要3分鐘，先給我用一下吧~
(SJF: shortest time job first)

Process	CPU Time
P_0	14
P_1	5
P_2	2

FIFO：



平均等待時間 = (0+ 14+19)/3 = 11

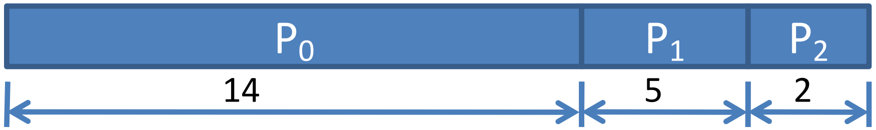
1. CPU的使用率要最大
2. 工作產能要高
3. Process等待的時間要短
4. 完成時間要短
5. 資源利用率也要大
6. 要公平

SJF：

FIFO: 誰先來誰先做
SJF: 你要3個小時？我只要3分鐘，先給我用一下吧~
(SJF: shortest time job first)

Process	CPU Time
P_0	14
P_1	5
P_2	2

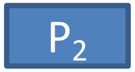
FIFO：



平均等待時間 = (0+ 14+19)/3 = 11

1. CPU的使用率要最大
2. 工作產能要高
3. Process等待的時間要短
4. 完成時間要短
5. 資源利用率也要大
6. 要公平

SJF：

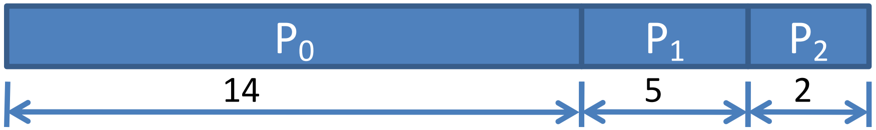


FIFO: 誰先來誰先做
SJF: 你要3個小時？我只要3分鐘，先給我用一下吧~
(SJF: shortest time job first)

Process	CPU Time
P_0	14
P_1	5
P_2	2

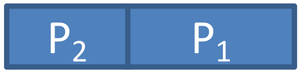
1. CPU的使用率要最大
2. 工作產能要高
3. Process等待的時間要短
4. 完成時間要短
5. 資源利用率也要大
6. 要公平

FIFO：



平均等待時間 = (0+ 14+19)/3 = 11

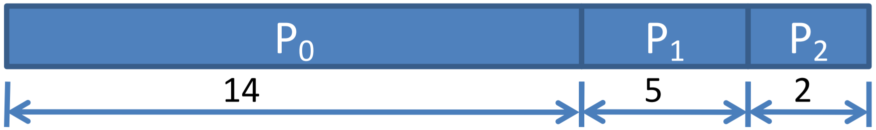
SJF：



FIFO: 誰先來誰先做
SJF: 你要3個小時？我只要3分鐘，先給我用一下吧~
(SJF: shortest time job first)

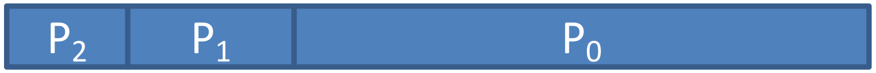
Process	CPU Time
P_0	14
P_1	5
P_2	2

FIFO：



平均等待時間 = (0+ 14+19)/3 = 11

SJF：



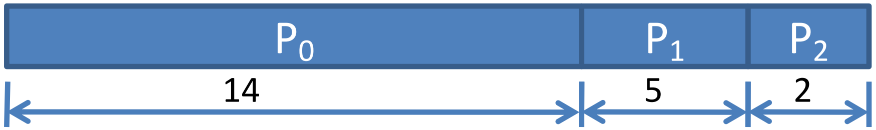
1. CPU的使用率要最大
2. 工作產能要高
3. Process等待的時間要短
4. 完成時間要短
5. 資源利用率也要大
6. 要公平

FIFO: 誰先來誰先做
SJF: 你要3個小時？我只要3分鐘，先給我用一下吧~
(SJF: shortest time job first)

Process	CPU Time
P_0	14
P_1	5
P_2	2

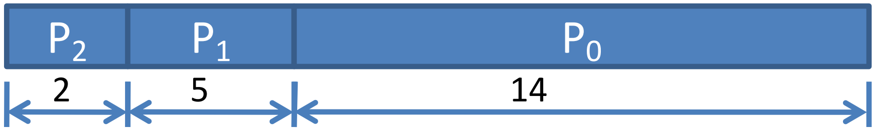
1. CPU的使用率要最大
2. 工作產能要高
3. Process等待的時間要短
4. 完成時間要短
5. 資源利用率也要大
6. 要公平

FIFO：



平均等待時間 = (0+ 14+19)/3 = 11

SJF：



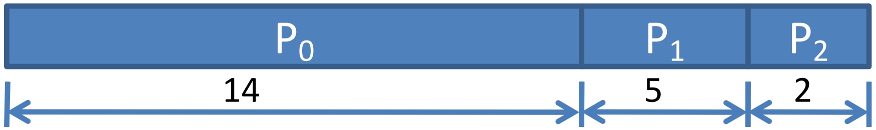
平均等待時間 = (0+2+7)/3 = 3

FIFO: 誰先來誰先做
SJF: 你要3個小時？我只要3分鐘，先給我用一下吧~
(SJF: shortest time job first)

Process	CPU Time
P_0	14
P_1	5
P_2	2

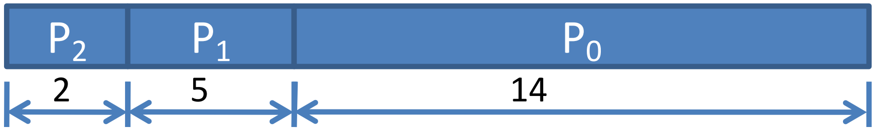
1. CPU的使用率要最大
2. 工作產能要高
3. Process等待的時間要短
4. 完成時間要短
5. 資源利用率也要大
6. 要公平

FIFO：



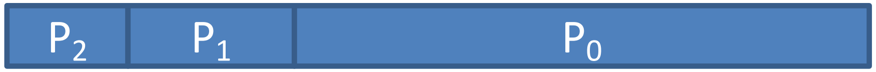
平均等待時間 = (0+ 14+19)/3 = 11

SJF：

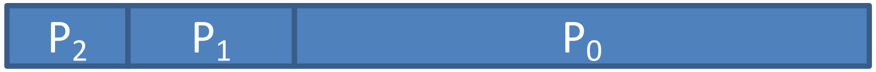


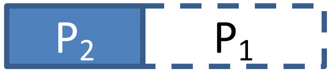
平均等待時間 = (0+2+7)/3 = 3

Process	CPU Time
P_0	14
P_1	5
P_2	2

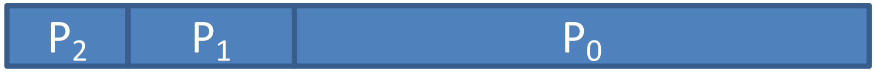


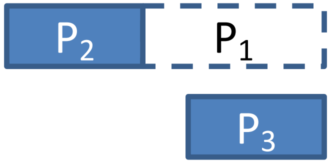
Process	CPU Time
P_0	14
P_1	5
P_2	2


P_2 P_1 P_0

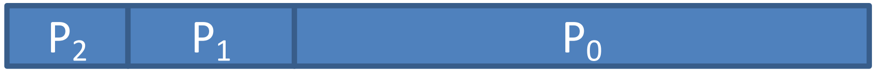

P_2 P_1

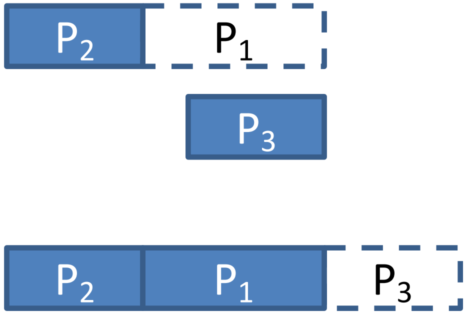
Process	CPU Time
P_0	14
P_1	5
P_2	2



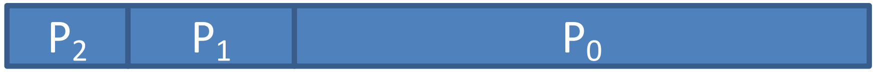


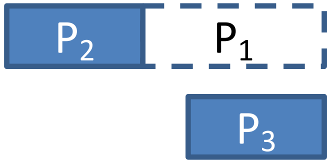
Process	CPU Time
P_0	14
P_1	5
P_2	2

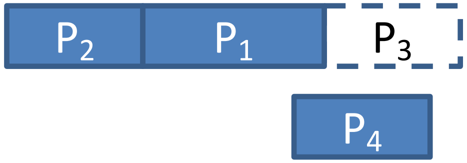




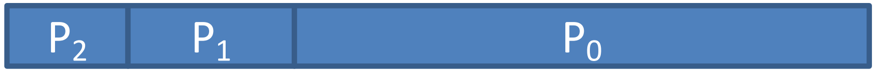
Process	CPU Time
P_0	14
P_1	5
P_2	2

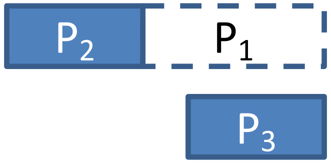


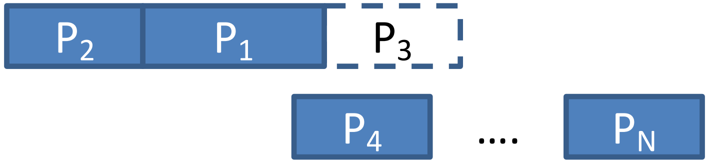




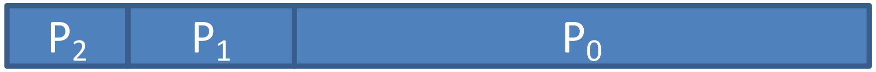
Process	CPU Time
P_0	14
P_1	5
P_2	2

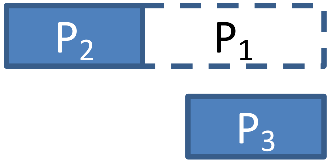




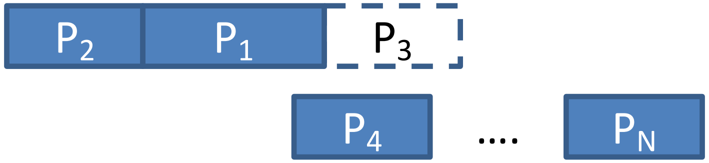


Process	CPU Time
P_0	14
P_1	5
P_2	2

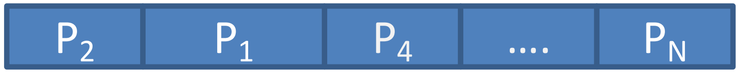

P_2 P_1 P_0


P_2 P_1

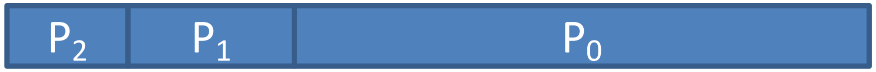
P_3

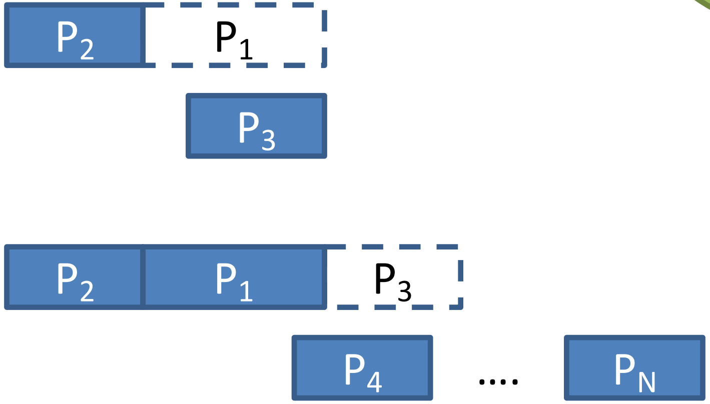

P_2 P_1 P_3

P_4 …. P_N


P_2 P_1 P_4 …. P_N

Process	CPU Time
P_0	14
P_1	5
P_2	2

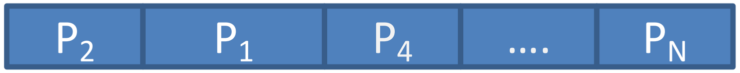

P_2 P_1 P_0


P_2 P_1

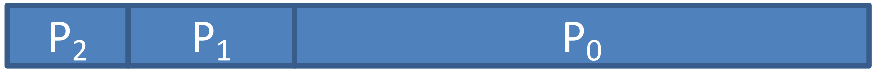
P_3

P_2 P_1 P_3

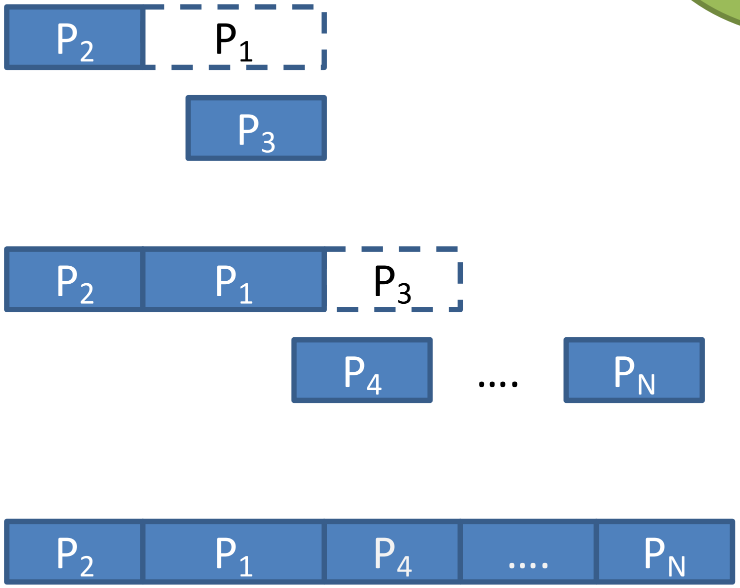
P_4 …. P_N


P_2 P_1 P_4 …. P_N

Process	CPU Time
P_0	14
P_1	5
P_2	2



P_2 P_1 P_0



P_2 P_1

P_3

P_2 P_1 P_3

P_4 …. P_N

P_2 P_1 P_4 …. P_N

P_0

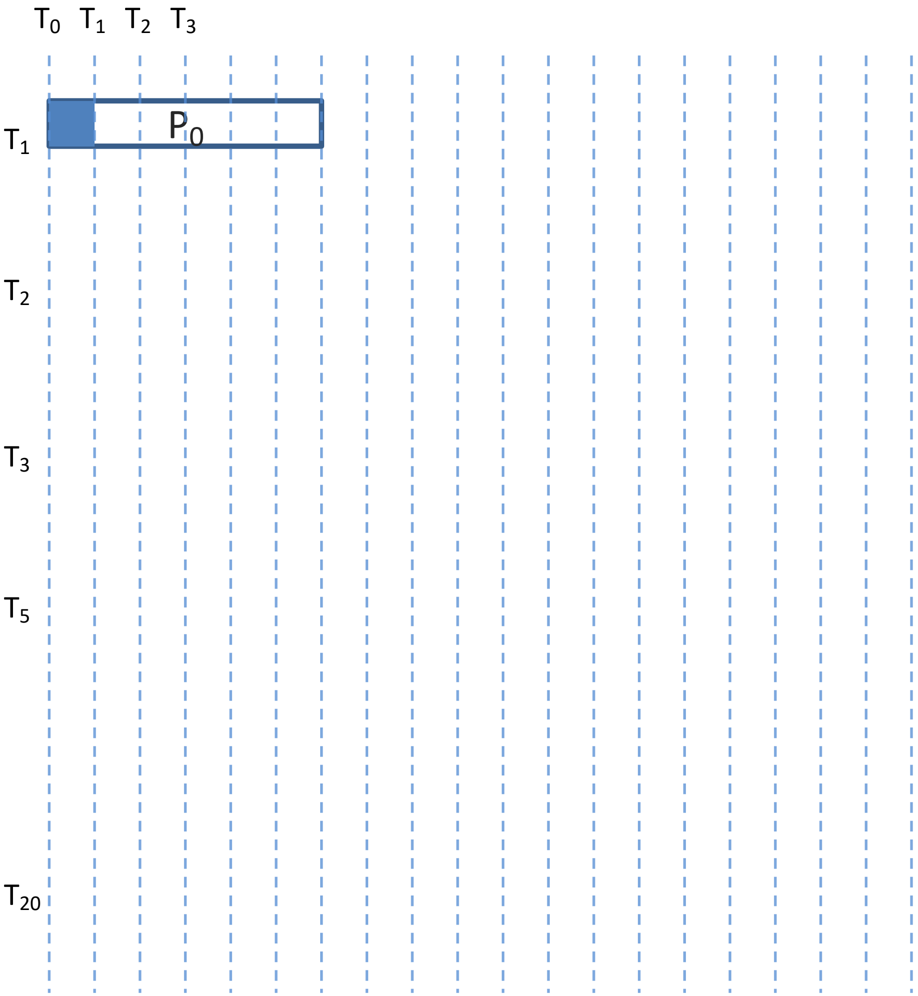
Starvation 飢餓：Process因為長期無法取得完工所需的全部資源，以致形成indefinite blocking之現象。

解法：Aging Tech等…

SRJF: Shortest Remaining Time Job First

- 可插隊的。

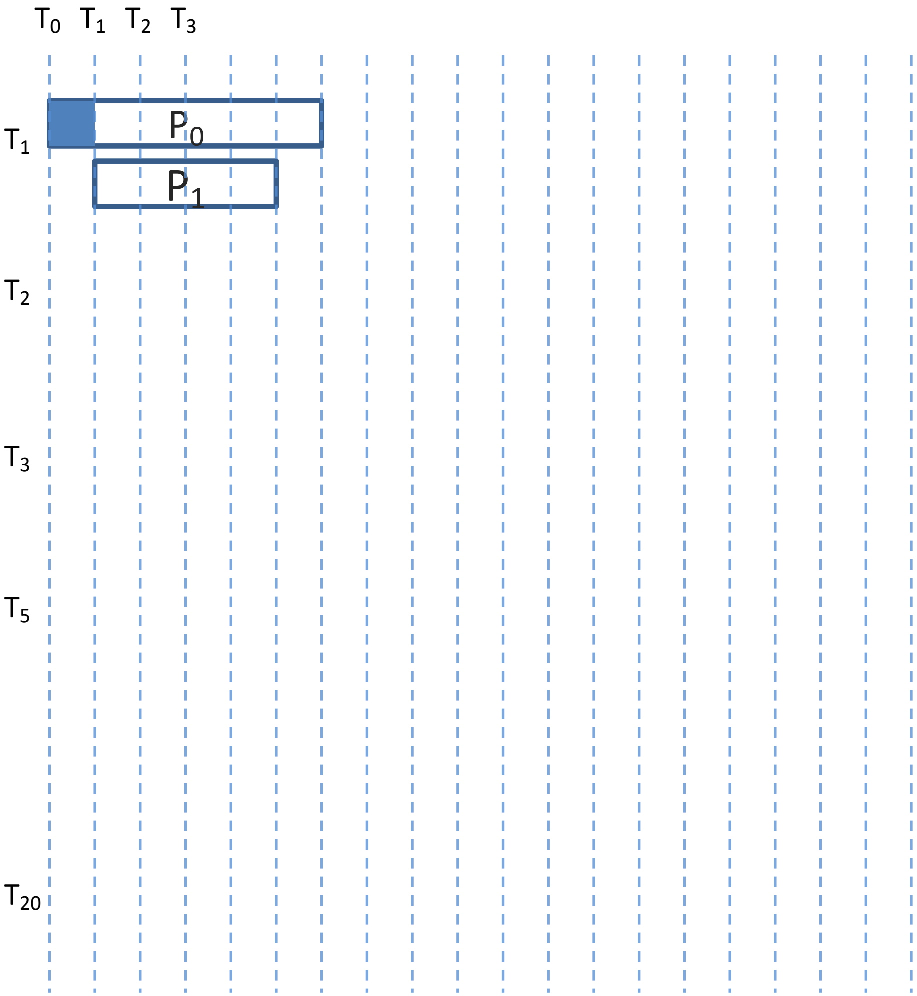
Process	Arrival Time	CPU Time
P_0	0	6
P_1	1	4
P_2	2	7
P_3	3	3



SRJF: Shortest Remaining Time Job First

- 可插隊的。

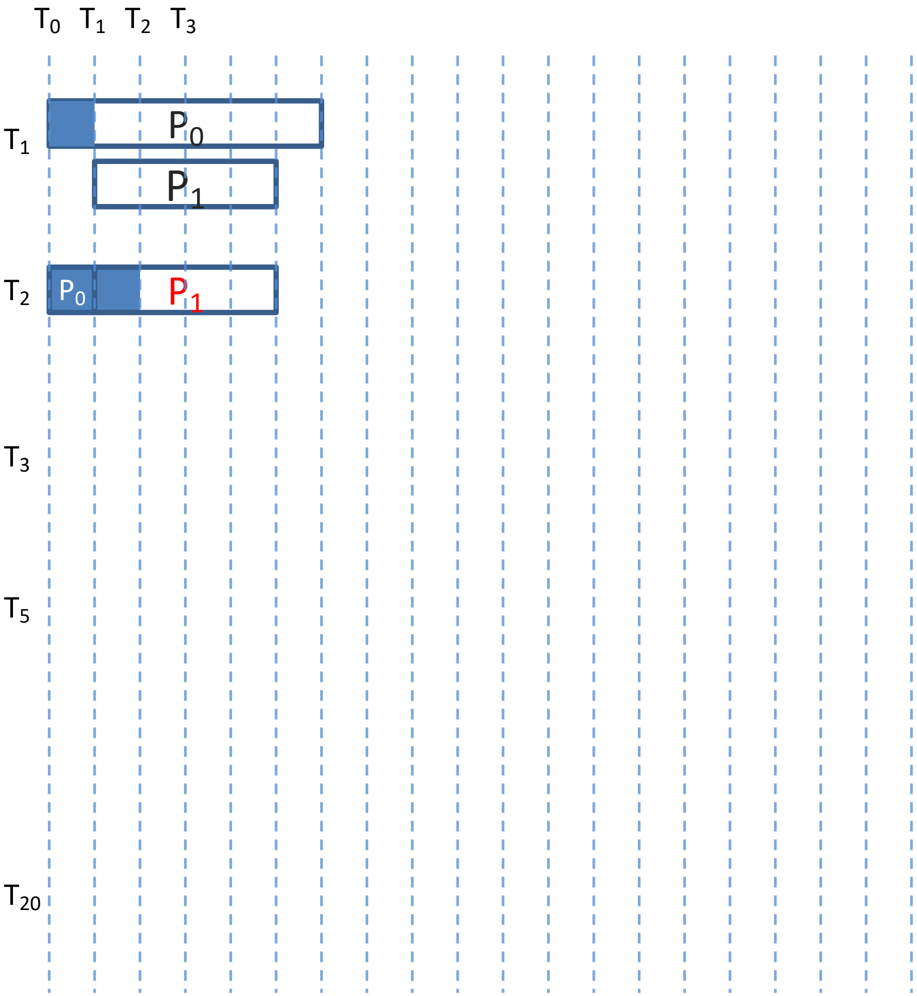
Process	Arrival Time	CPU Time
P_0	0	6
P_1	1	4
P_2	2	7
P_3	3	3



SRJF: Shortest Remaining Time Job First

- 可插隊的。

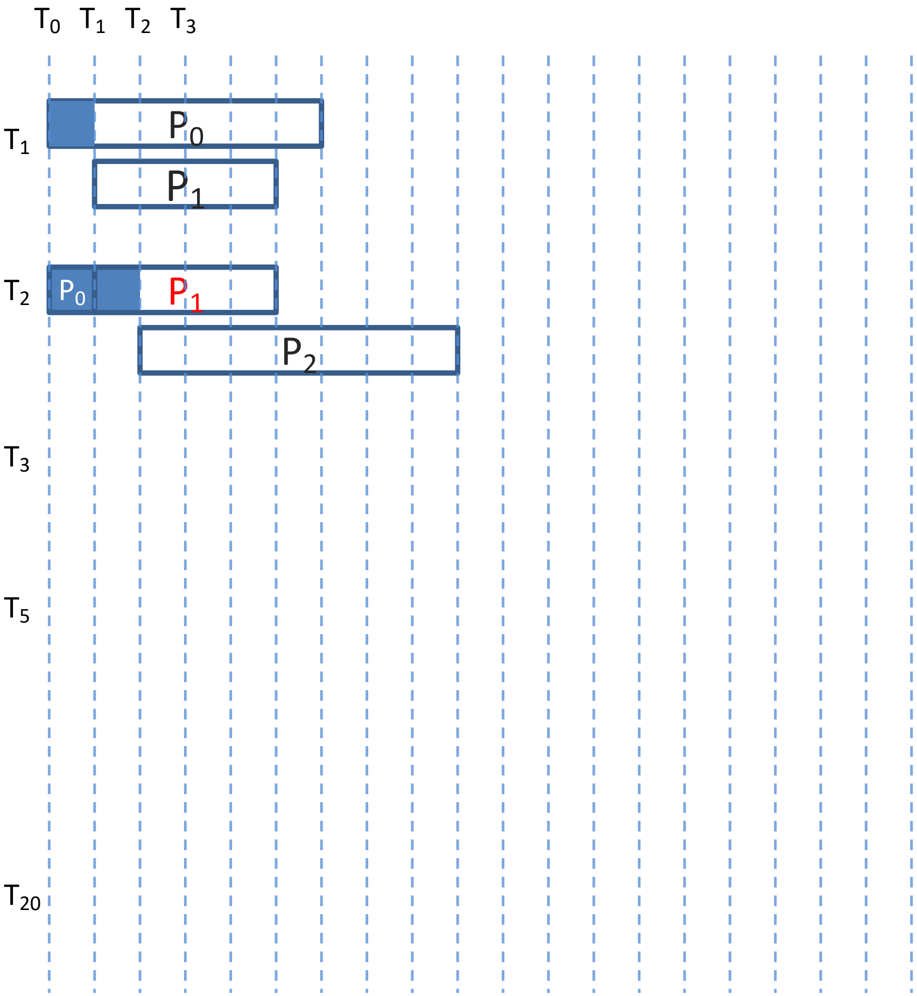
Process	Arrival Time	CPU Time
P_0	0	6
P_1	1	4
P_2	2	7
P_3	3	3



SRJF: Shortest Remaining Time Job First

- 可插隊的。

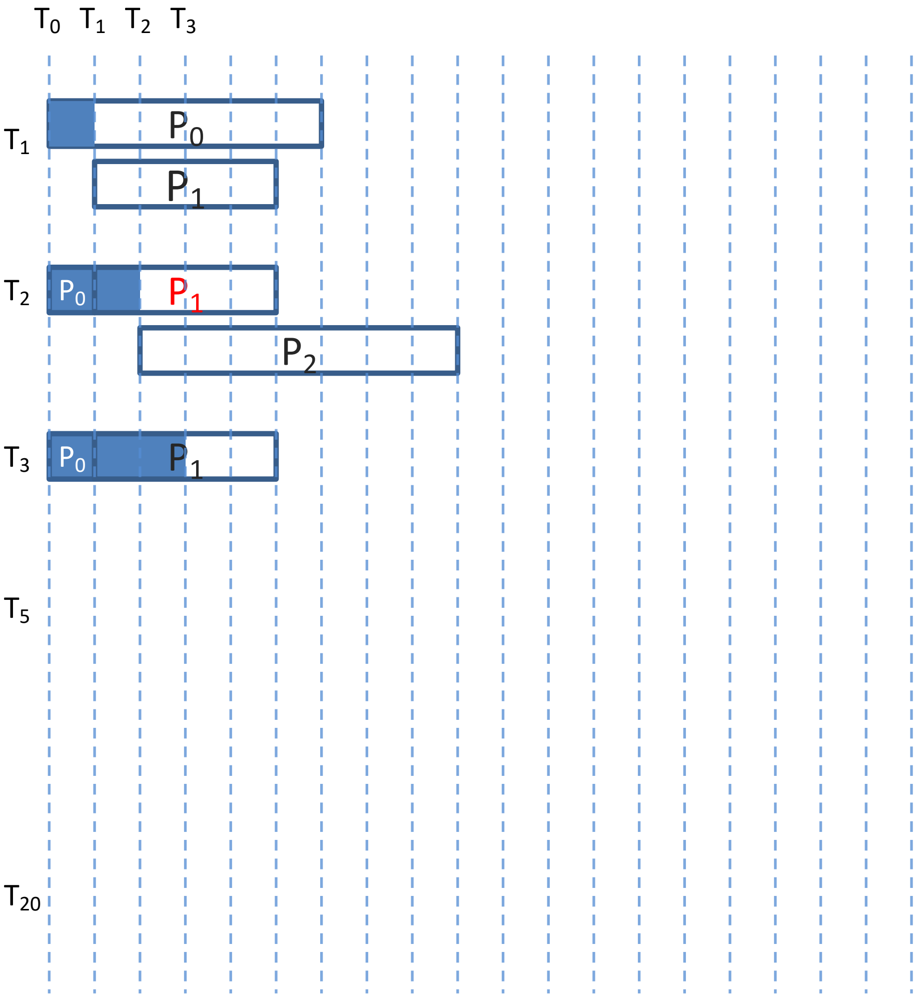
Process	Arrival Time	CPU Time
P_0	0	6
P_1	1	4
P_2	2	7
P_3	3	3



SRJF: Shortest Remaining Time Job First

- 可插隊的。

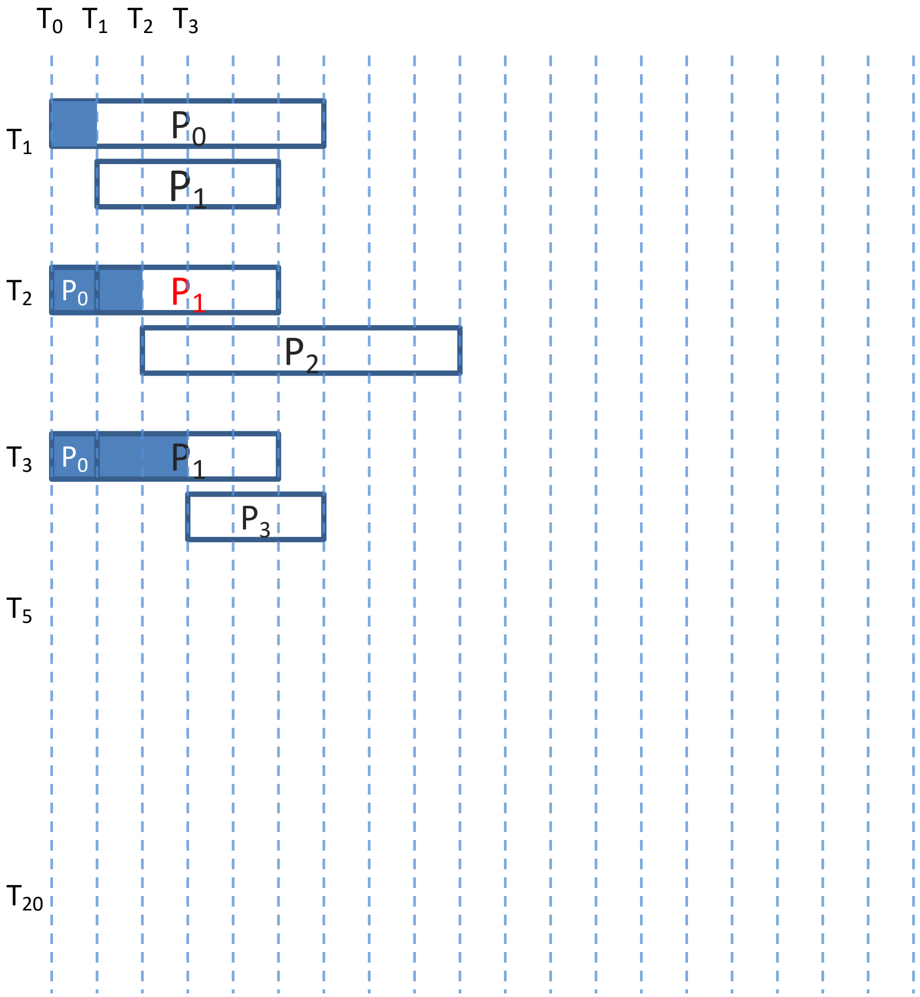
Process	Arrival Time	CPU Time
P_0	0	6
P_1	1	4
P_2	2	7
P_3	3	3



SRJF: Shortest Remaining Time Job First

- 可插隊的。

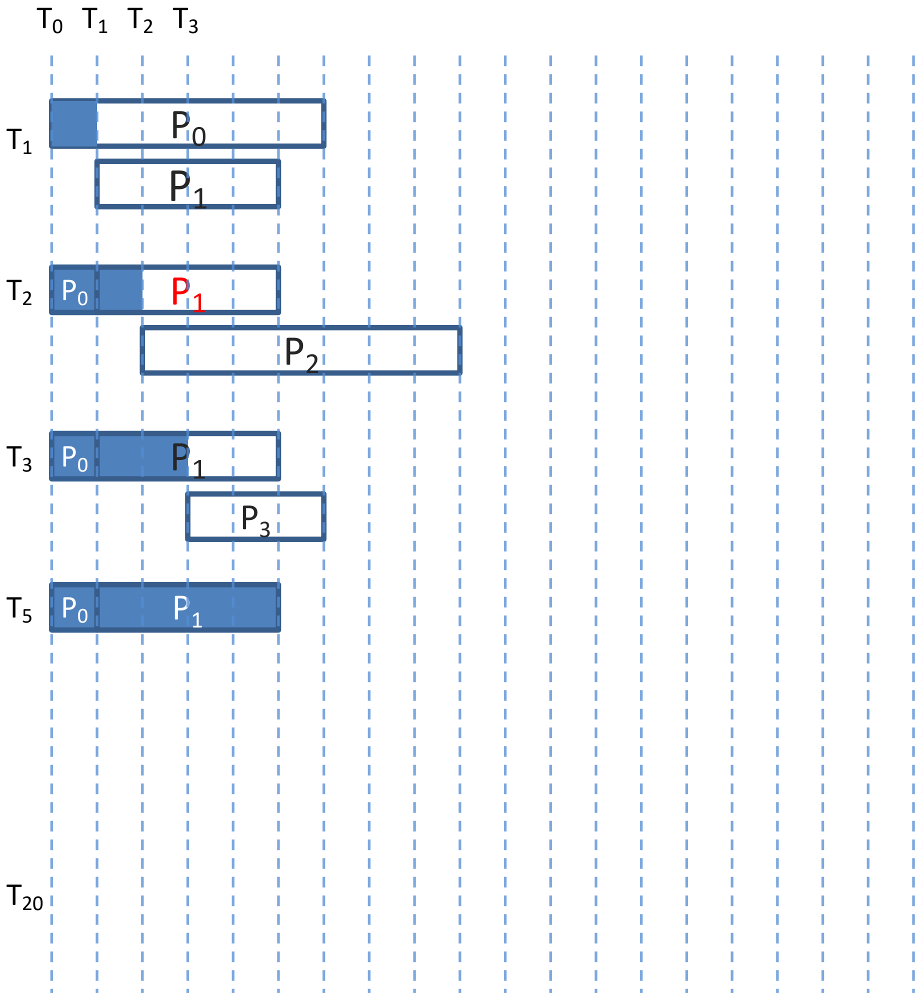
Process	Arrival Time	CPU Time
P_0	0	6
P_1	1	4
P_2	2	7
P_3	3	3



SRJF: Shortest Remaining Time Job First

- 可插隊的。

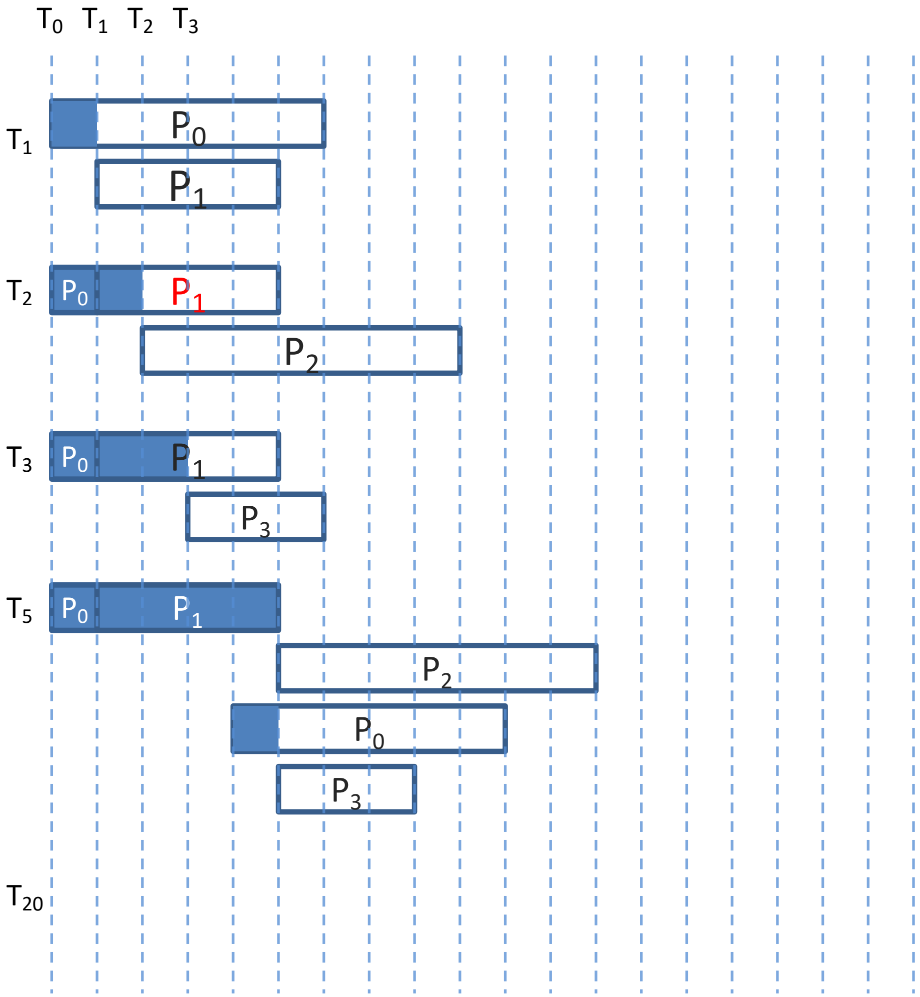
Process	Arrival Time	CPU Time
P_0	0	6
P_1	1	4
P_2	2	7
P_3	3	3



SRJF: Shortest Remaining Time Job First

- 可插隊的。

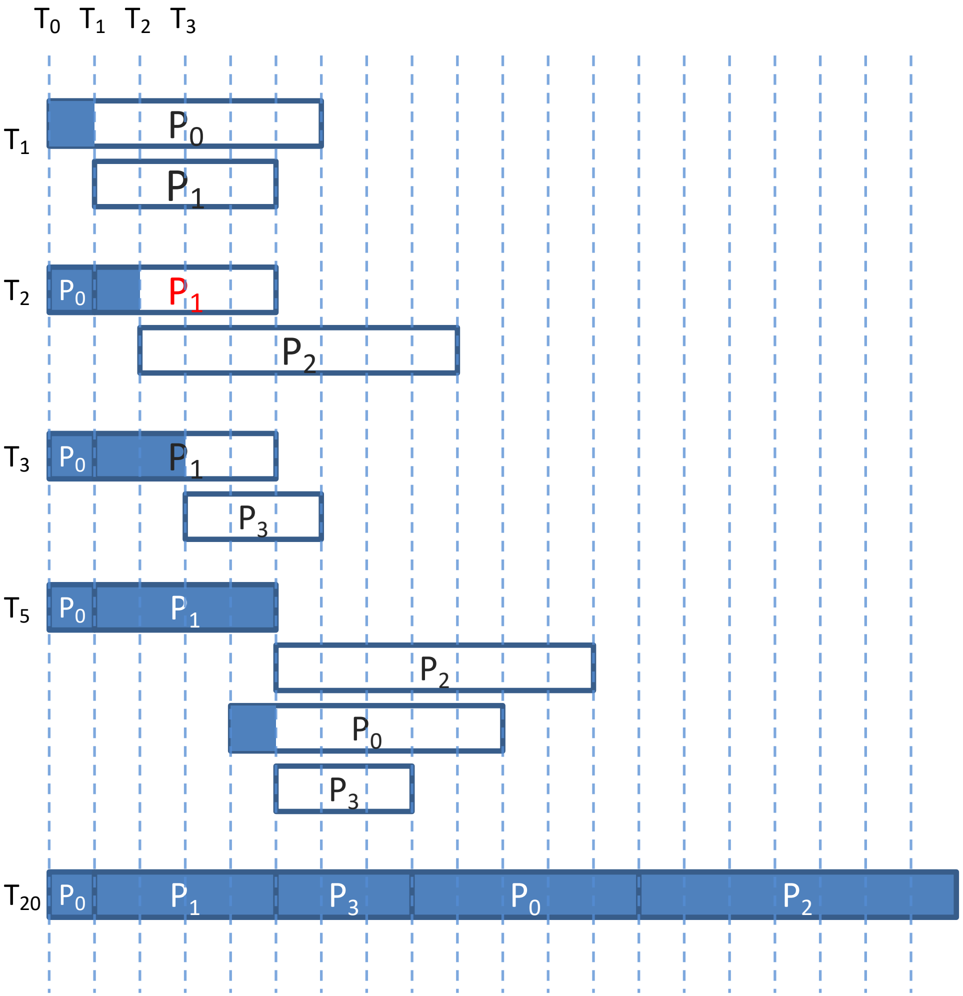
Process	Arrival Time	CPU Time
P_0	0	6
P_1	1	4
P_2	2	7
P_3	3	3

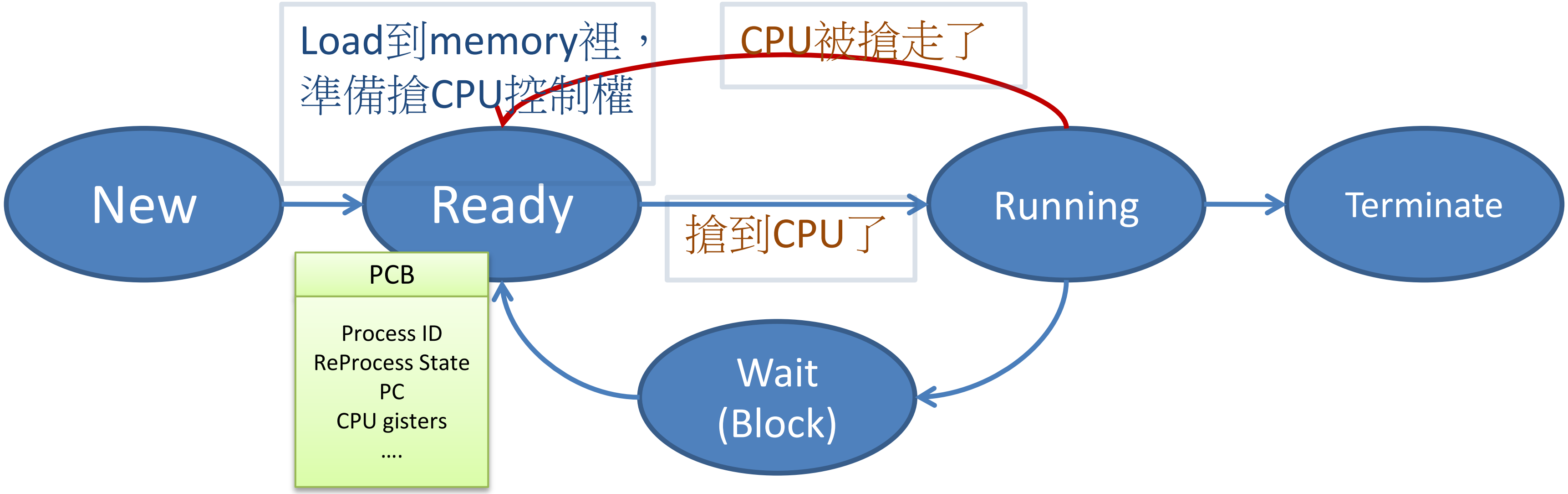


SRJF: Shortest Remaining Time Job First

- 可插隊的。

Process	Arrival Time	CPU Time
P_0	0	6
P_1	1	4
P_2	2	7
P_3	3	3





New

Ready

Running

Terminate

Wait
(Block)

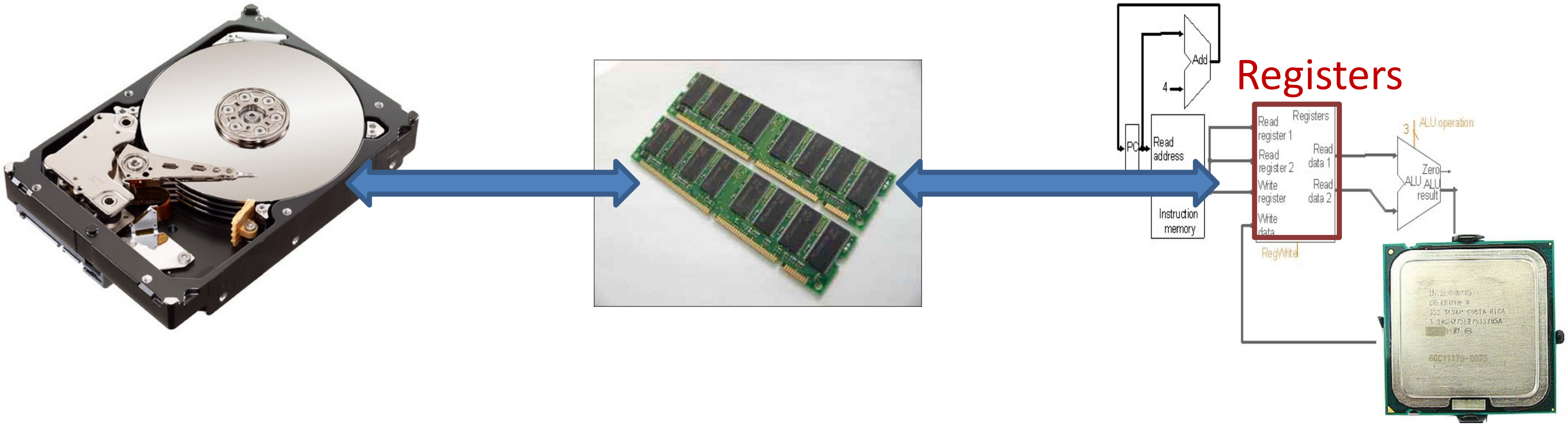
Load到memory裡，
準備搶CPU控制權

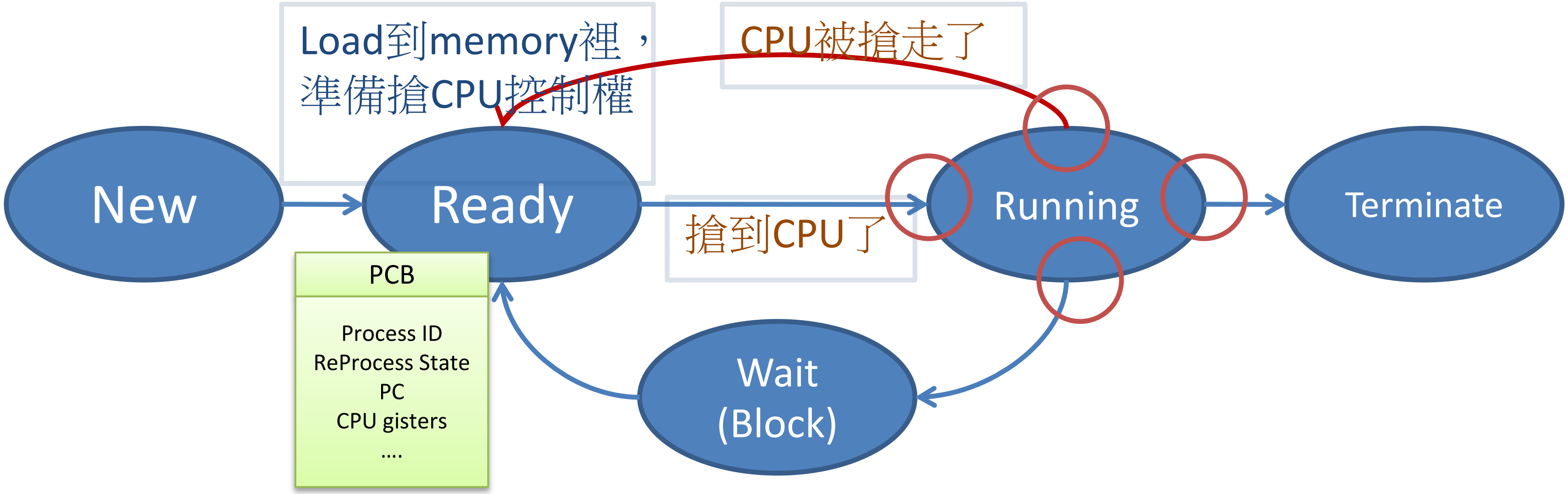
CPU被搶走了

搶到CPU了

PCB

Process ID
ReProcess State
PC
CPU gisters
....





New

Ready

Running

Terminate

Wait
(Block)

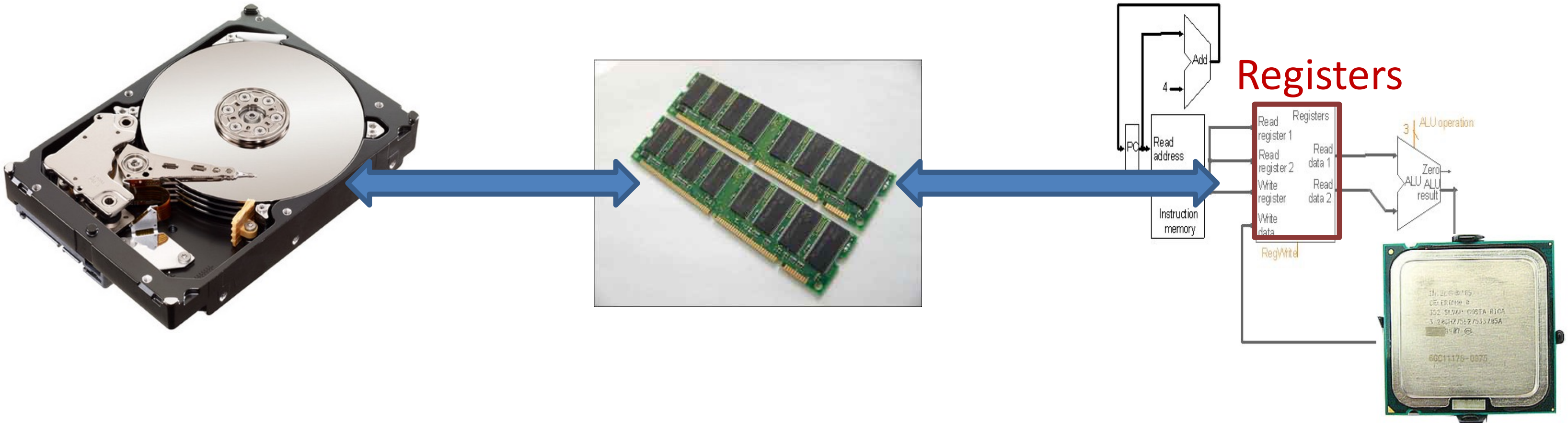
Load到memory裡，
準備搶CPU控制權

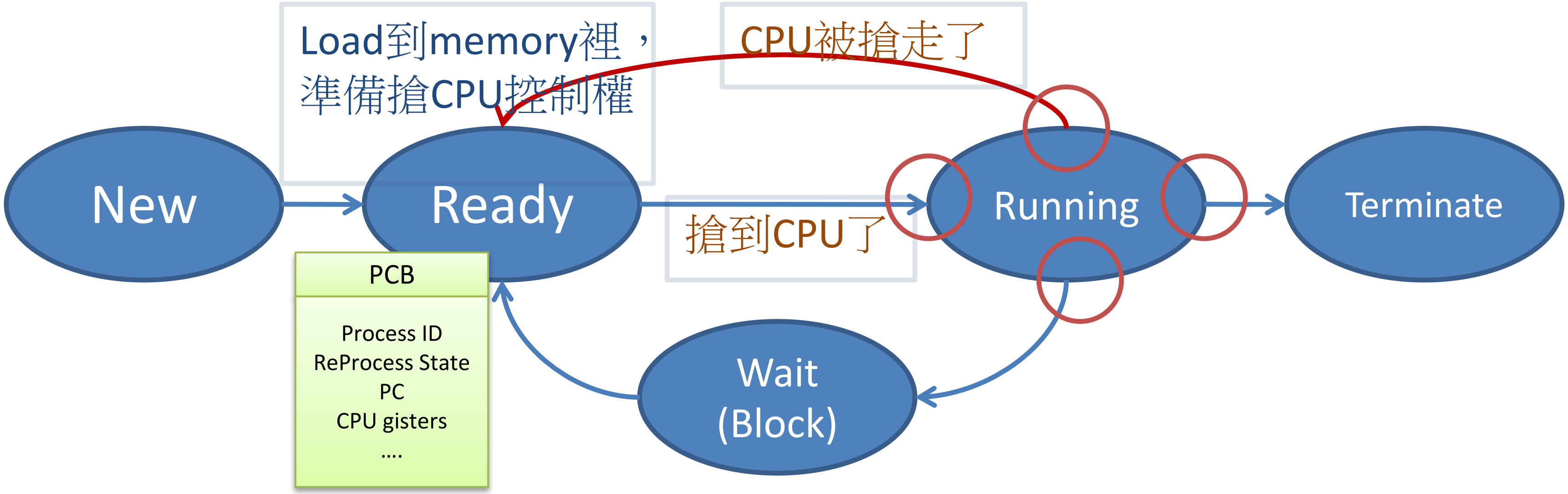
CPU被搶走了

搶到CPU了

PCB

Process ID
ReProcess State
PC
CPU gisters
....





Load到memory裡，準備搶CPU控制權

CPU被搶走了

New

Ready

搶到CPU了

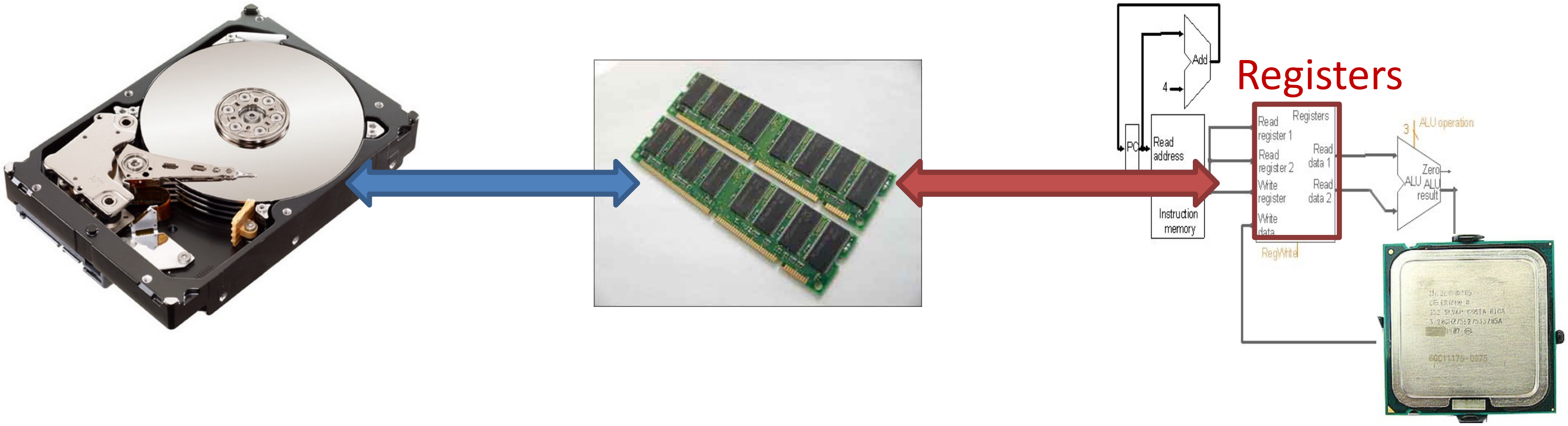
Running

Terminate

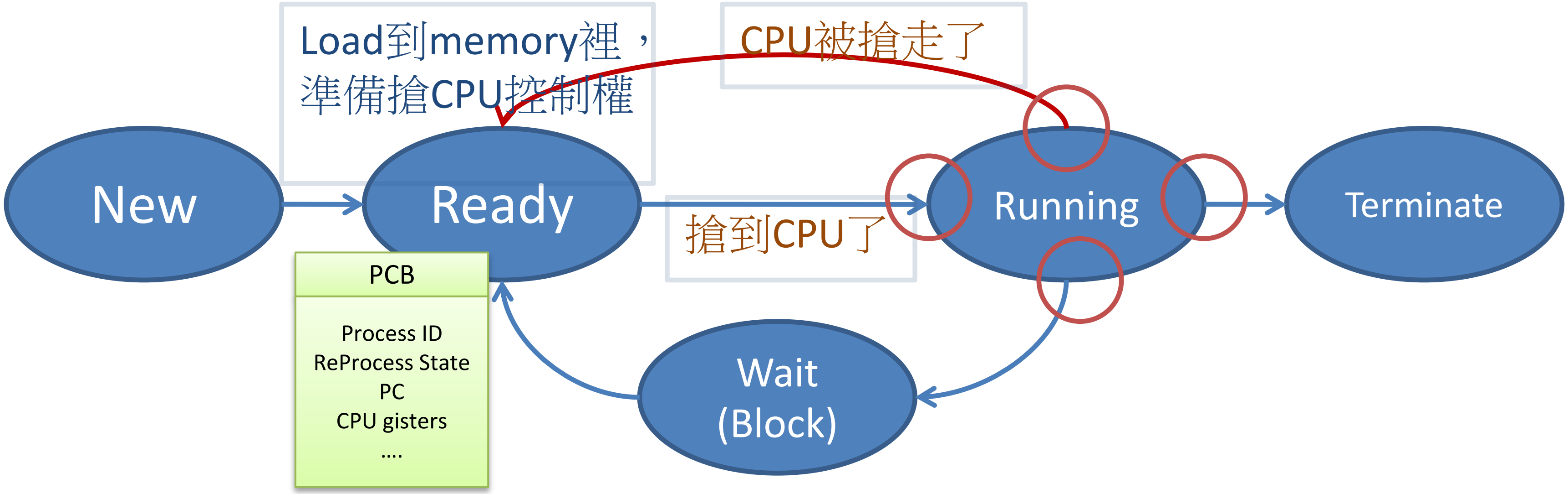
Wait (Block)

PCB

Process ID
ReProcess State
PC
CPU gisters
....



Registers



Load到memory裡，準備搶CPU控制權

CPU被搶走了

New

Ready

搶到CPU了

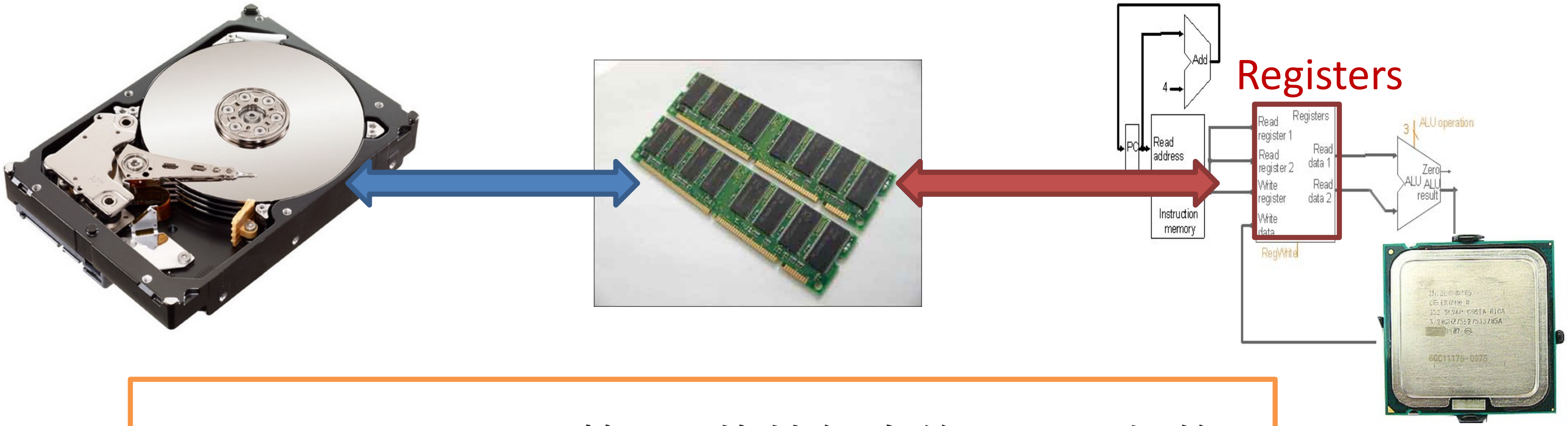
Running

Terminate

Wait (Block)

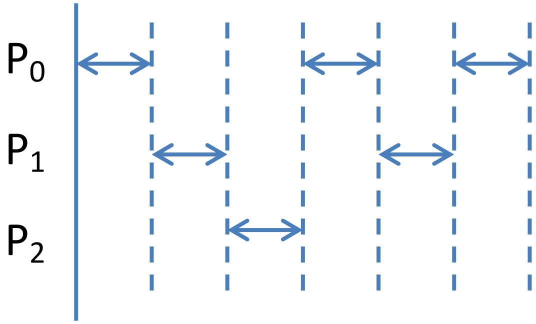
PCB

Process ID
ReProcess State
PC
CPU gisters
….

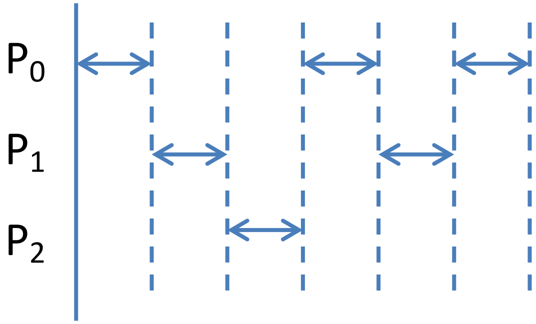


Context Switching：若CPU將執行中的Process切換給其他Process使用時，必須保存目前執行中Process的狀態，並載入欲執行Process的狀態資訊。

問題是搬家是需要成本的…

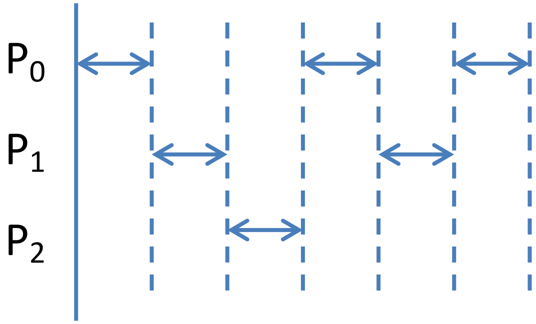


問題是搬家是需要成本的…



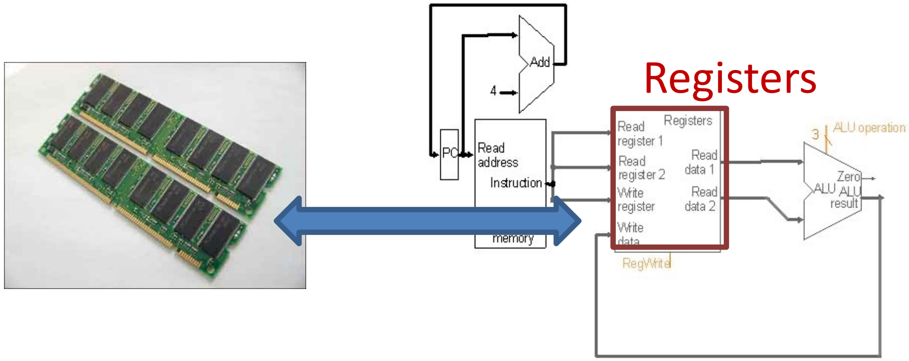
太多Process並行效能反而不如預期…

問題是搬家是需要成本的…

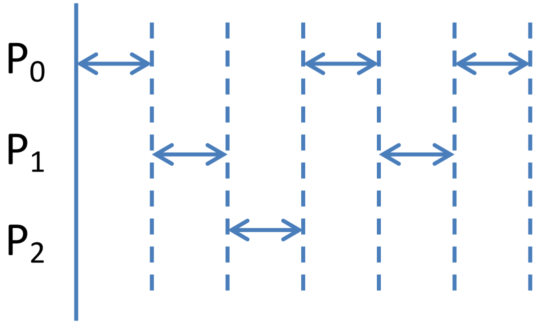


太多Process並行效能反而不如預期…

解法1：提供多套Registers

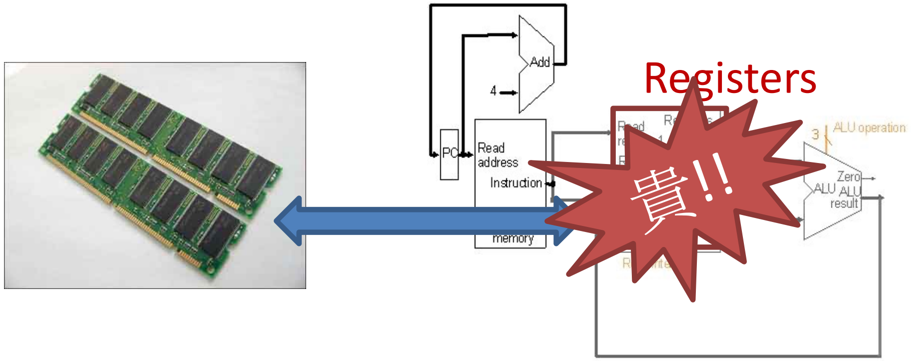


問題是搬家是需要成本的...

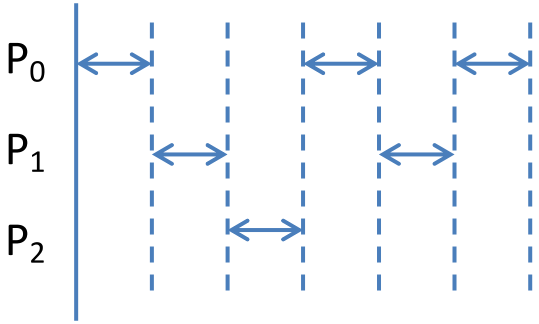


太多Process並行效能反而不如預期...

解法1：提供多套Registers



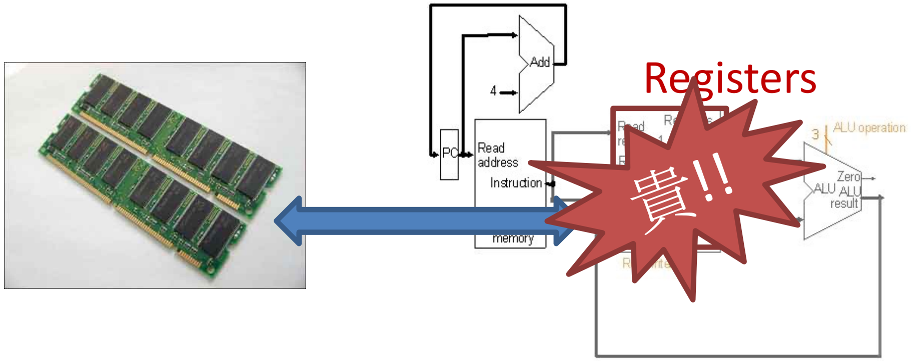
問題是搬家是需要成本的...



太多Process並行效能反而不如預期...

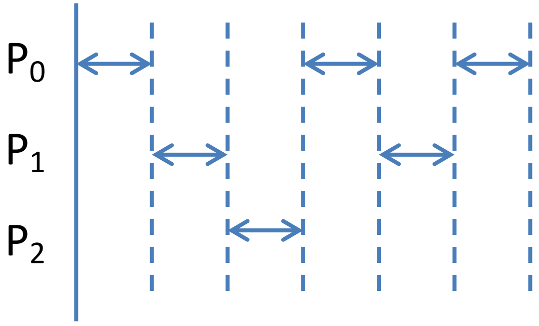
解法1：提供多套Registers

解法2：改用Thread~



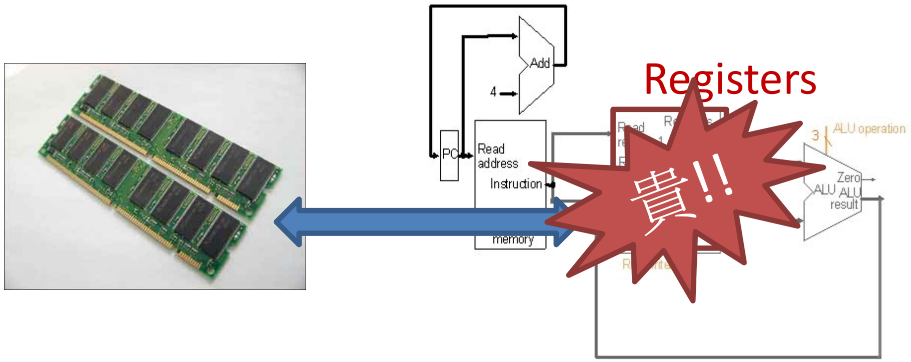
Thread: Light weight process. 是CPU分配資源的最小單位，而同一個Process內的threads共享code section, data section, 跟一些OS資源

問題是搬家是需要成本的…



太多Process並行效能反而不如預期…

解法1：提供多套Registers



解法2：改用Thread~

Thread: Light weight process. 是CPU分配資源的最小單位，而同一個Process內的threads共享code section, data section, 跟一些OS資源

context switching 的負擔較小

建立Thread也是有成本的

→ 用空間換取時間~~

建立Thread也是有成本的

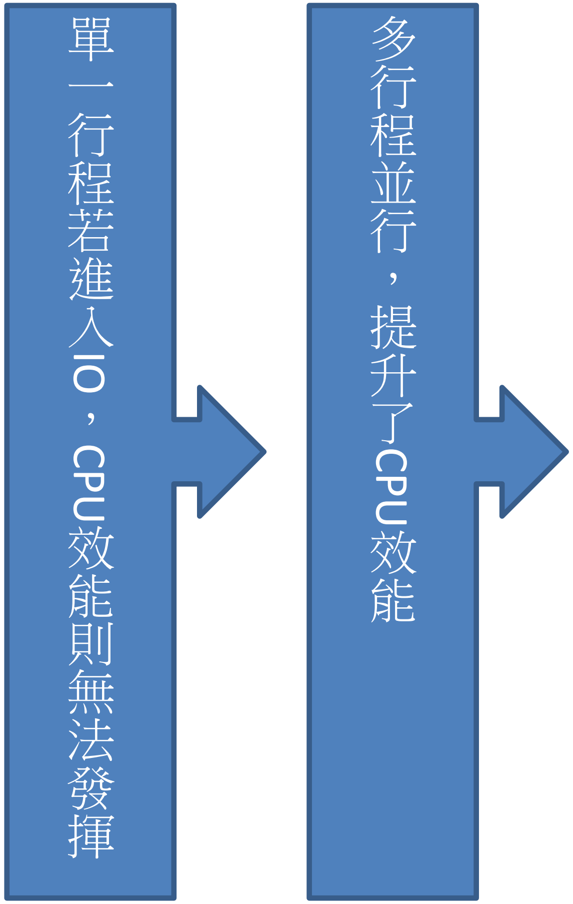
→ 用空間換取時間~~

Thread Pool: 在Process建立之初，預先建立多條threads置於thread pool中，當需要使用時，就從thread pool中取出使用，用完再還給thread pool。

到目前為止…

單一行程若進入IO，CPU效能則無法發揮

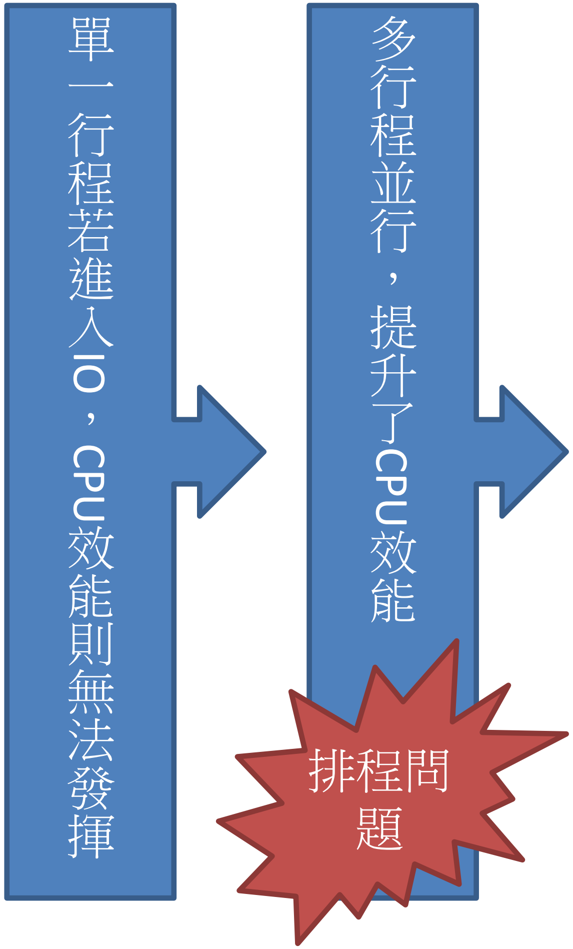
到目前為止…



單一行程若進入IO，CPU效能則無法發揮

多行程並行，提升了CPU效能

到目前為止…

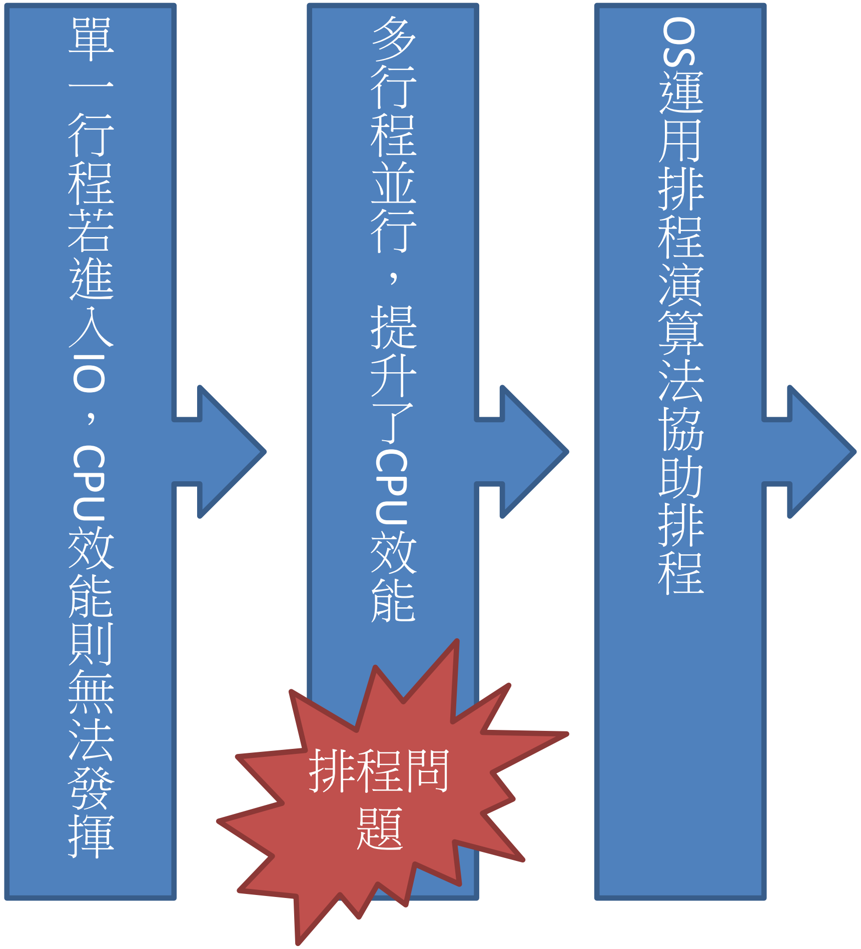


單一行程若進入IO，CPU效能則無法發揮

多行程並行，提升了CPU效能

排程問題

到目前為止…



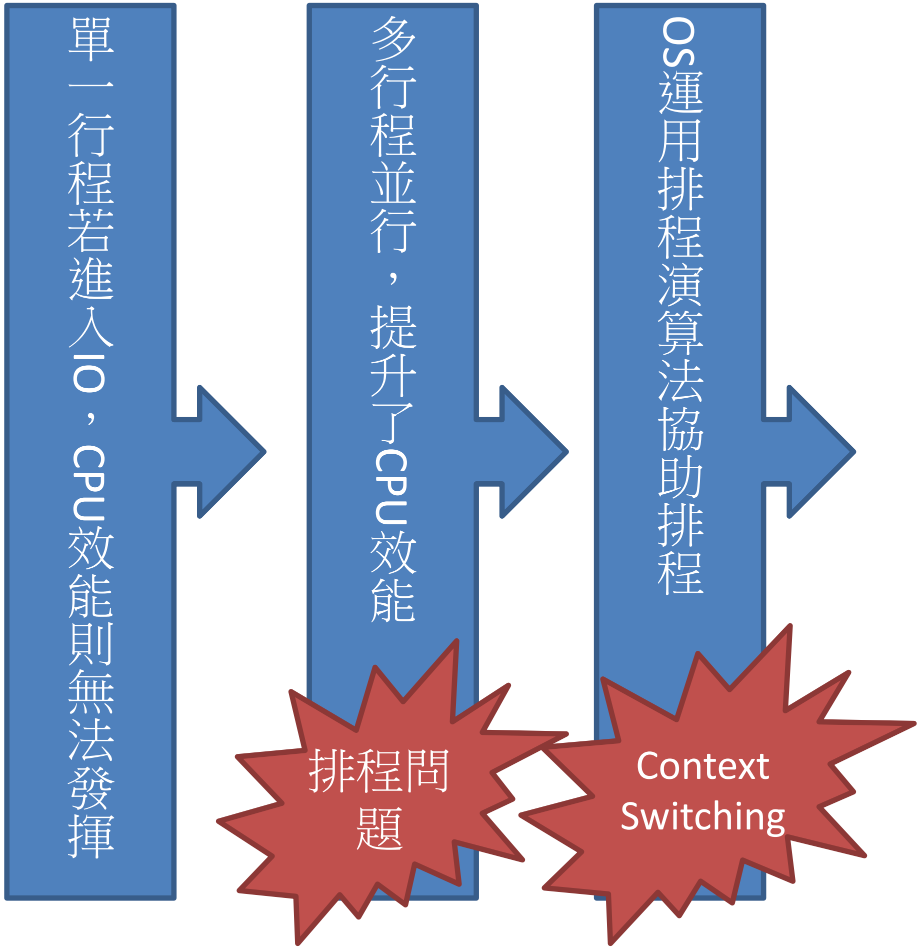
單一行程若進入IO，CPU效能則無法發揮

多行程並行，提升了CPU效能

排程問題

OS運用排程演算法協助排程

到目前為止…



單一行程若進入IO，CPU效能則無法發揮

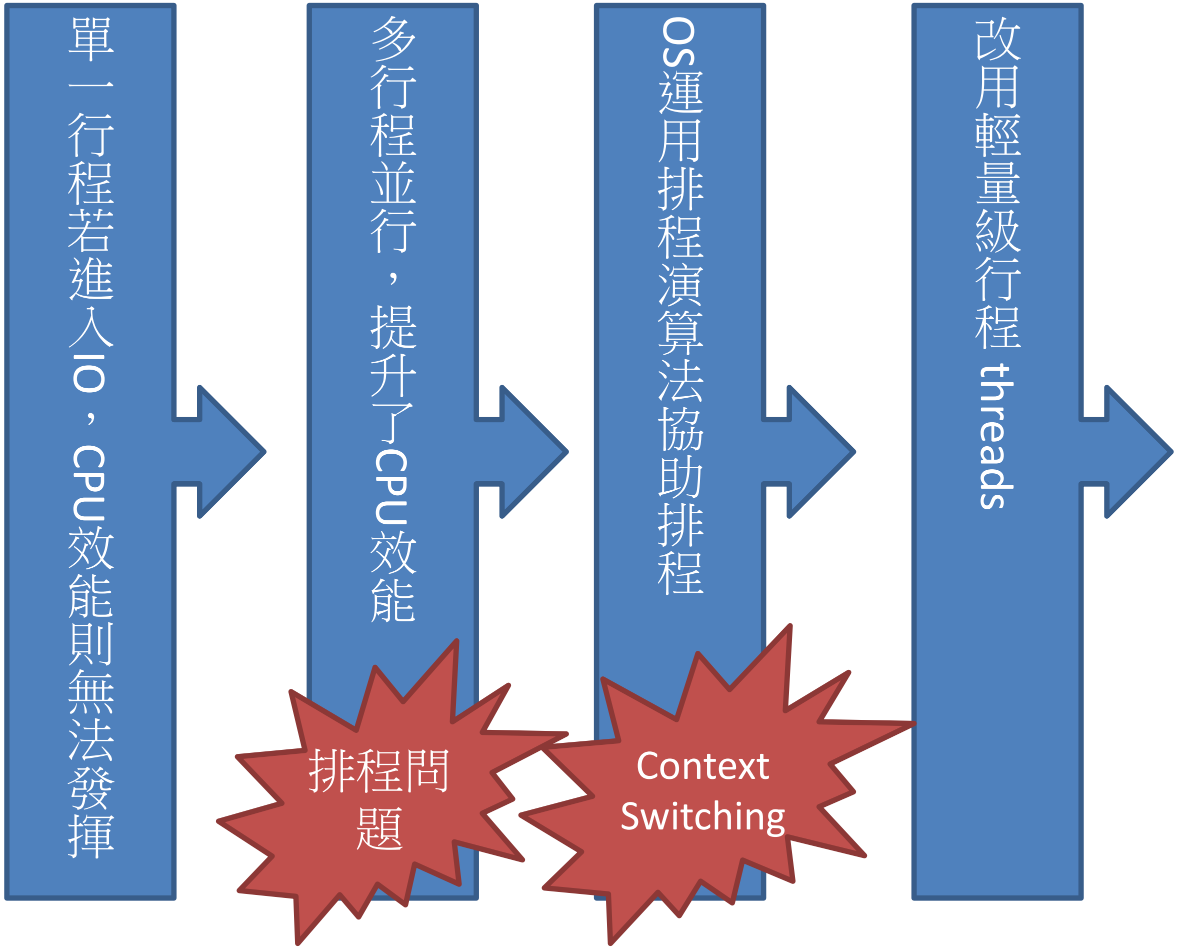
多行程並行，提升了CPU效能

OS運用排程演算法協助排程

排程問題

Context Switching

到目前為止…



單一行程若進入IO，CPU效能則無法發揮

多行程並行，提升了CPU效能

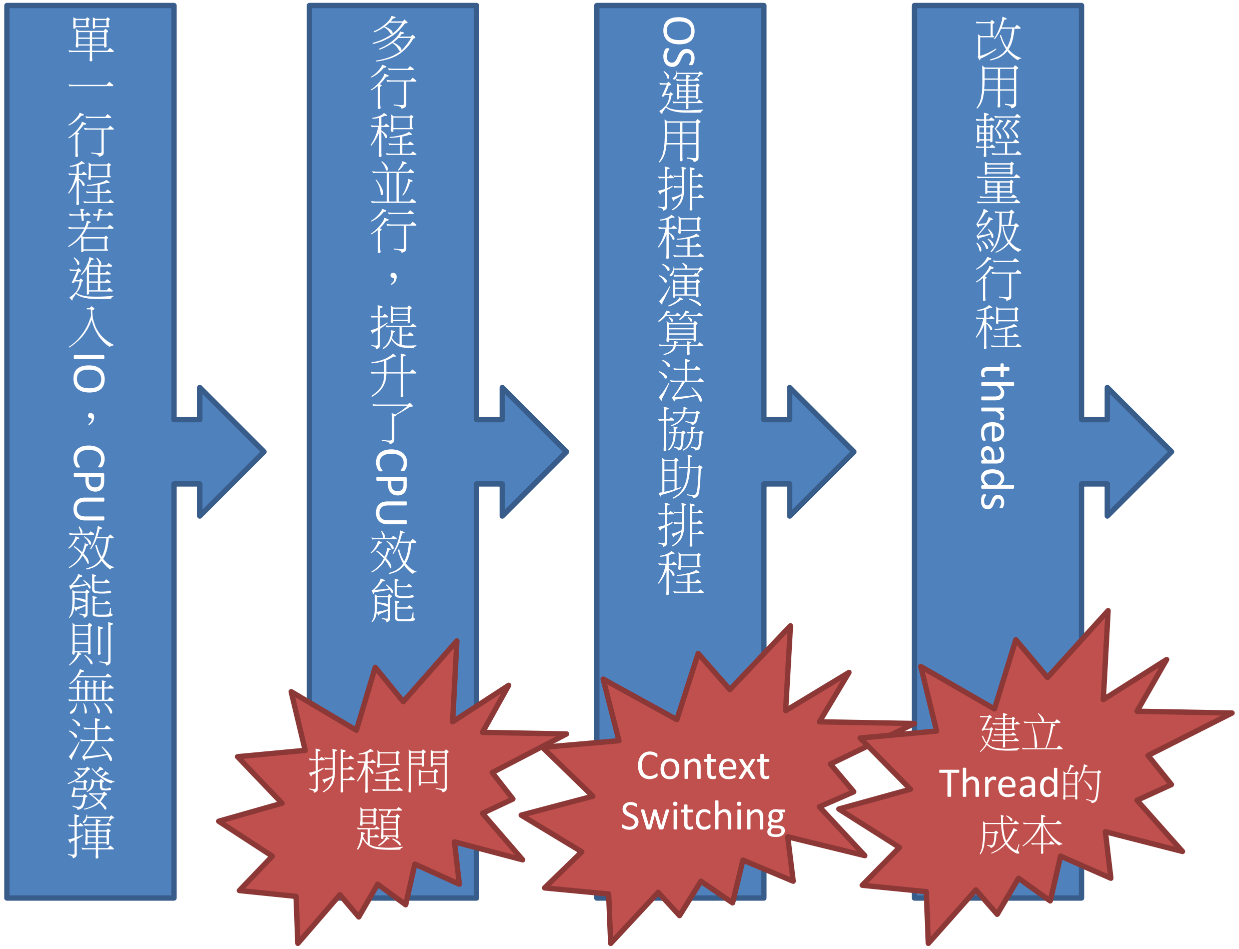
OS運用排程演算法協助排程

改用輕量級行程 threads

排程問題

Context Switching

到目前為止…



單一行程若進入IO，CPU效能則無法發揮

多行程並行，提升了CPU效能

OS運用排程演算法協助排程

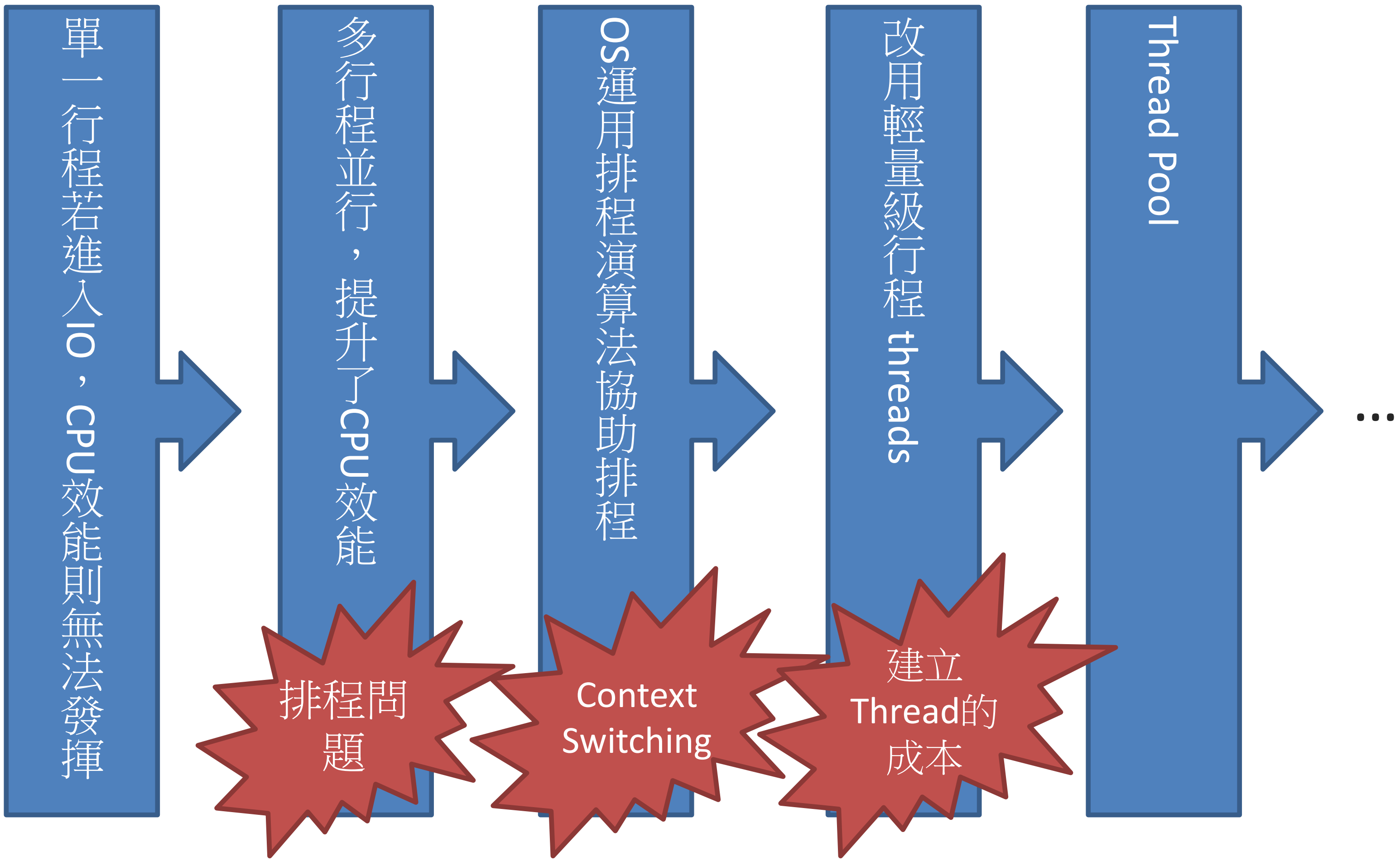
改用輕量級行程 threads

排程問題

Context Switching

建立Thread的成本

到目前為止…



單一行程若進入IO，CPU效能則無法發揮

多行程並行，提升了CPU效能

排程問題

OS運用排程演算法協助排程

Context Switching

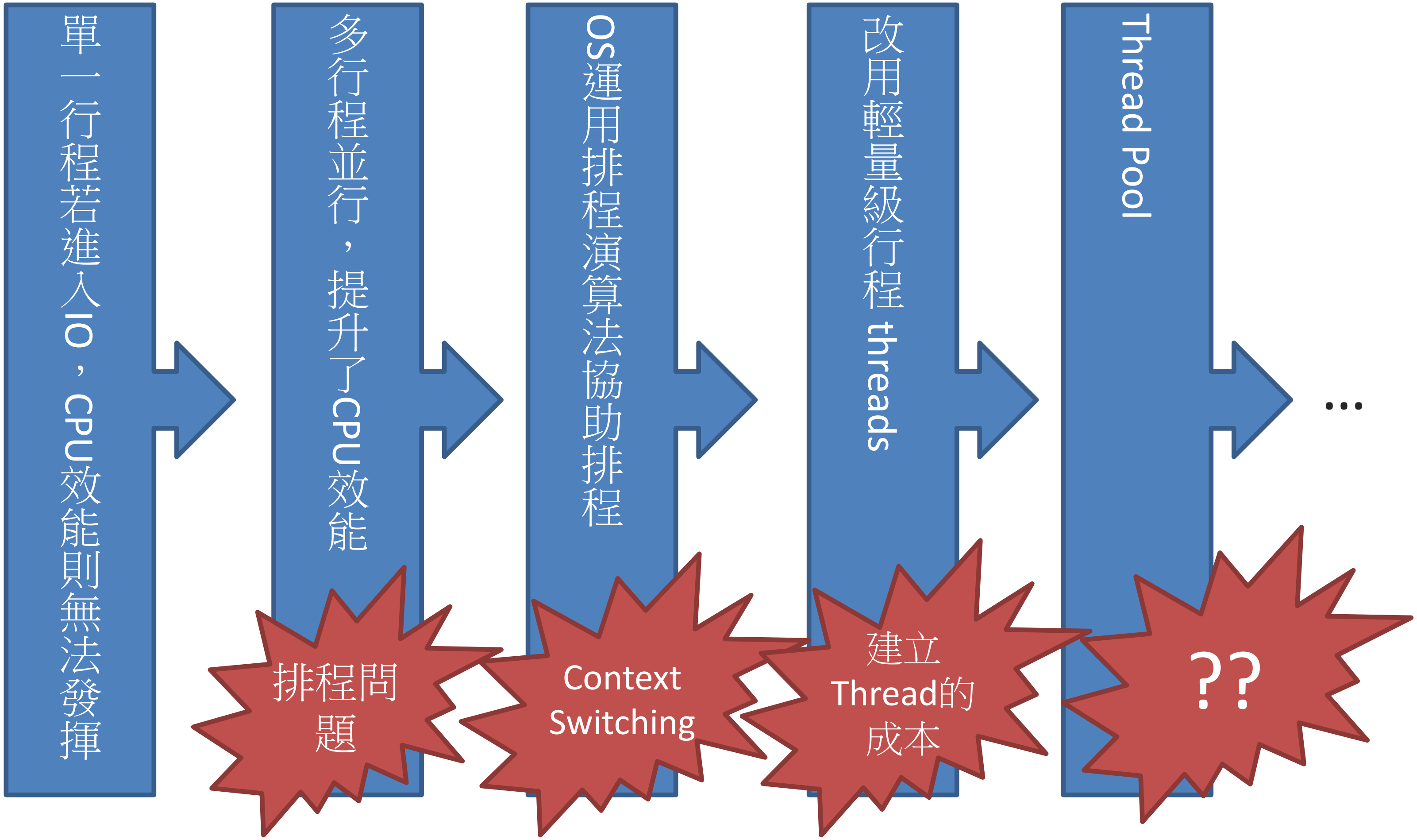
改用輕量級行程 threads

建立Thread的成本

Thread Pool

…

到目前為止…



單一行程若進入IO，CPU效能則無法發揮

多行程並行，提升了CPU效能

OS運用排程演算法協助排程

改用輕量級行程 threads

Thread Pool

…

排程問題

Context Switching

建立Thread的成本

??

Multi-Processing / Multi-Threading (1)

- 系統的資源通常是有限的
  - CPU週期、記憶體空間、檔案、I/O Device…

當Process需要資源時…

Multi-Processing / Multi-Threading (1)

- 系統的資源通常是有限的
  - CPU週期、記憶體空間、檔案、I/O Device…

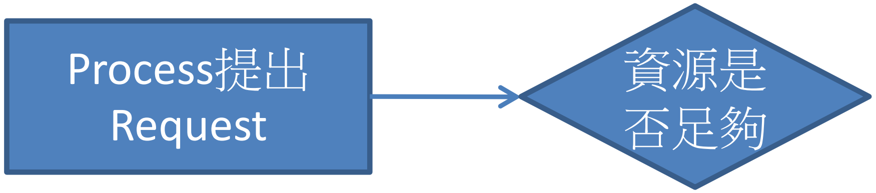
當Process需要資源時…

Process提出
Request

Multi-Processing / Multi-Threading (1)

- 系統的資源通常是有限的
  - CPU週期、記憶體空間、檔案、I/O Device...

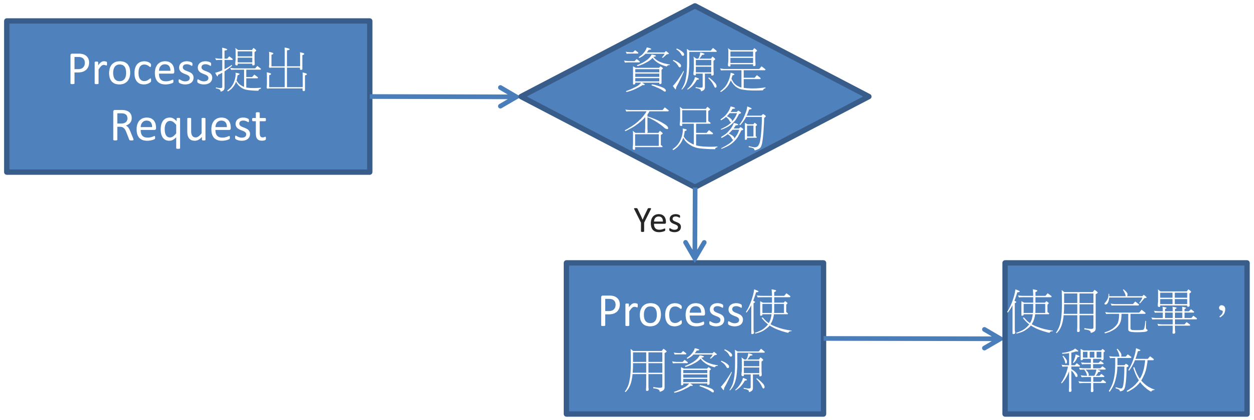
當Process需要資源時...



Multi-Processing / Multi-Threading (1)

- 系統的資源通常是有限的
  - CPU週期、記憶體空間、檔案、I/O Device…

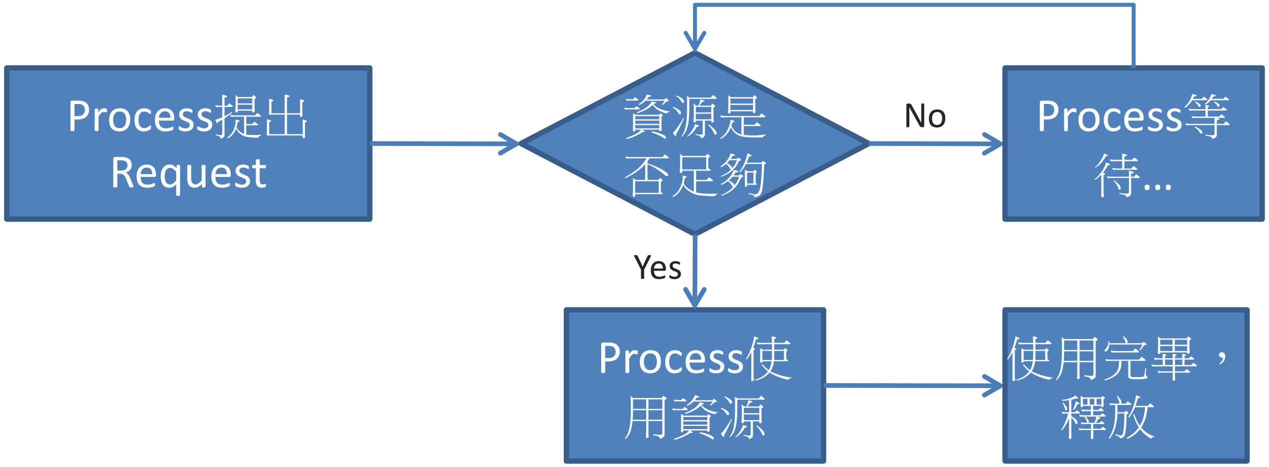
當Process需要資源時…



Multi-Processing / Multi-Threading (1)

- 系統的資源通常是有限的
  - CPU週期、記憶體空間、檔案、I/O Device…

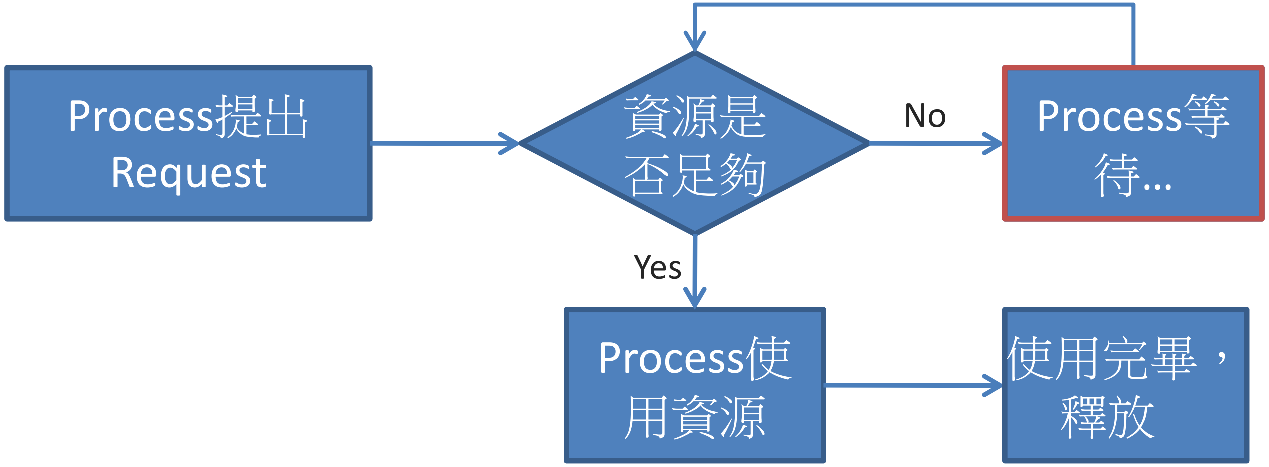
當Process需要資源時…



Multi-Processing / Multi-Threading (1)

- 系統的資源通常是有限的
  - CPU週期、記憶體空間、檔案、I/O Device…

當Process需要資源時…



Process提出Request

資源是否足夠

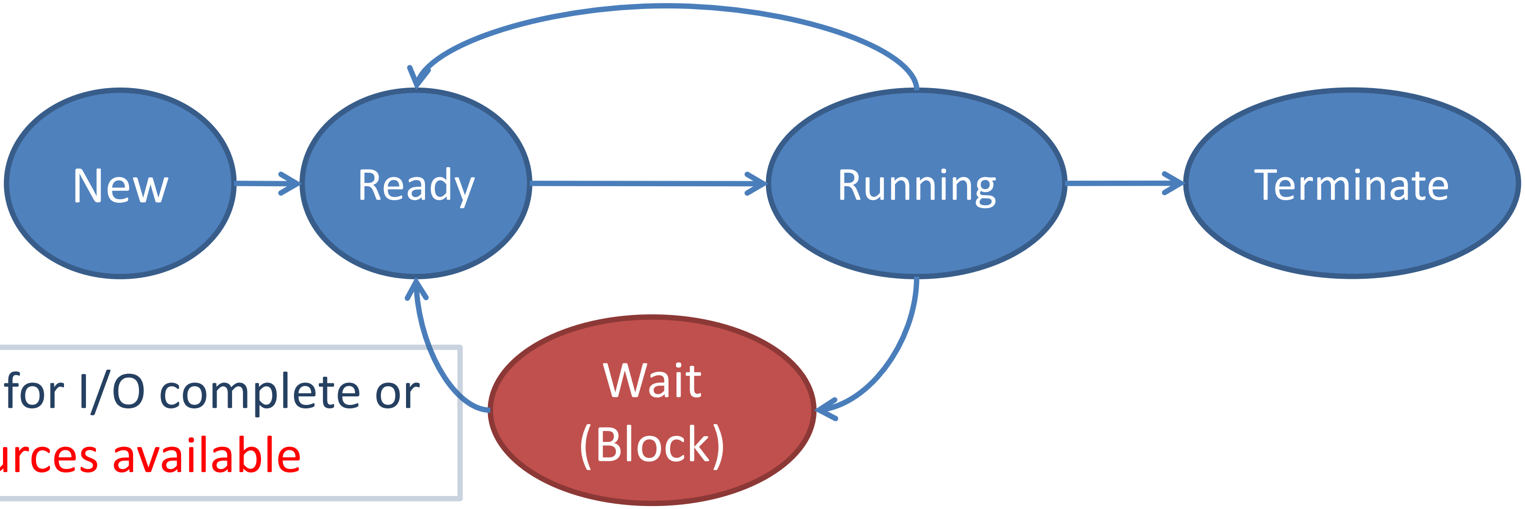
No

Process等待…

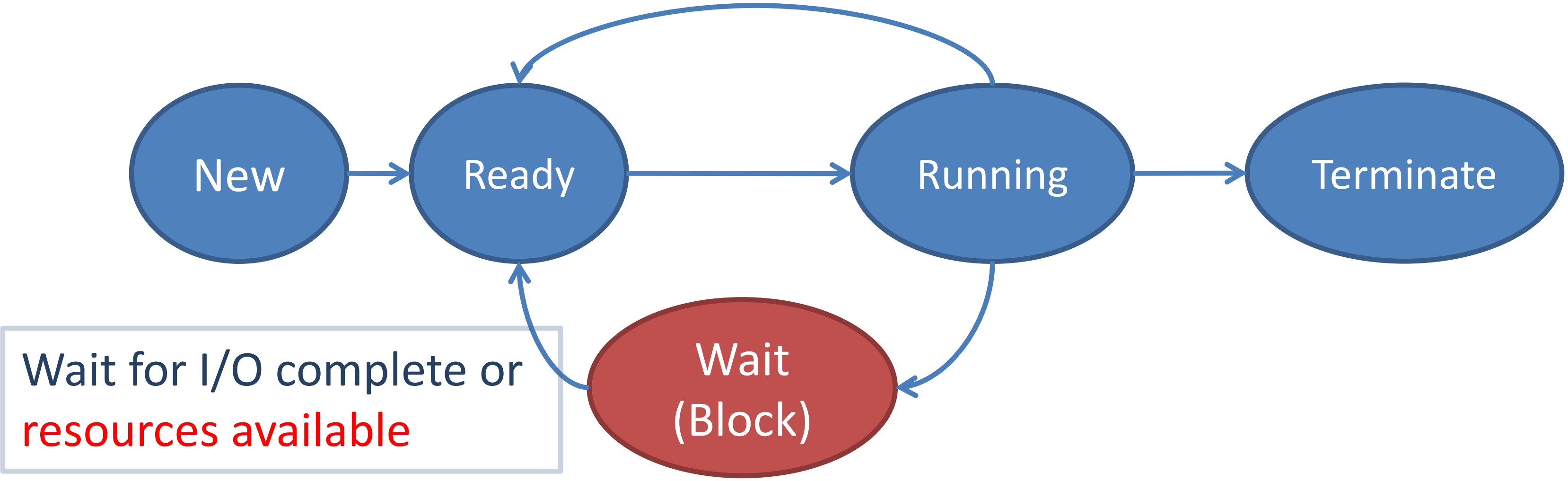
Yes

Process使用資源

使用完畢，釋放



Wait for I/O complete or resources available



New

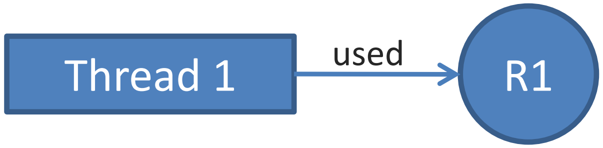
Ready

Running

Terminate

Wait (Block)

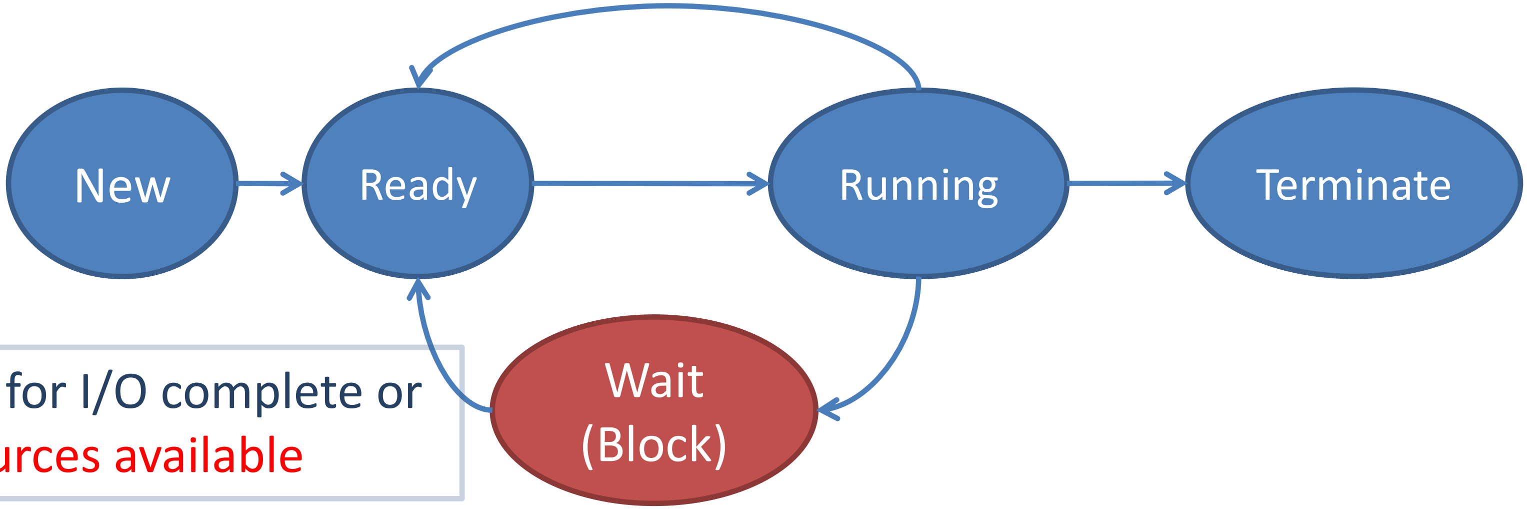
Wait for I/O complete or resources available



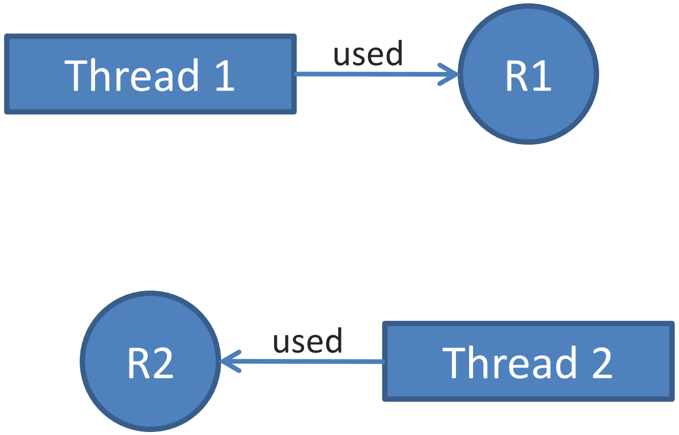
Thread 1

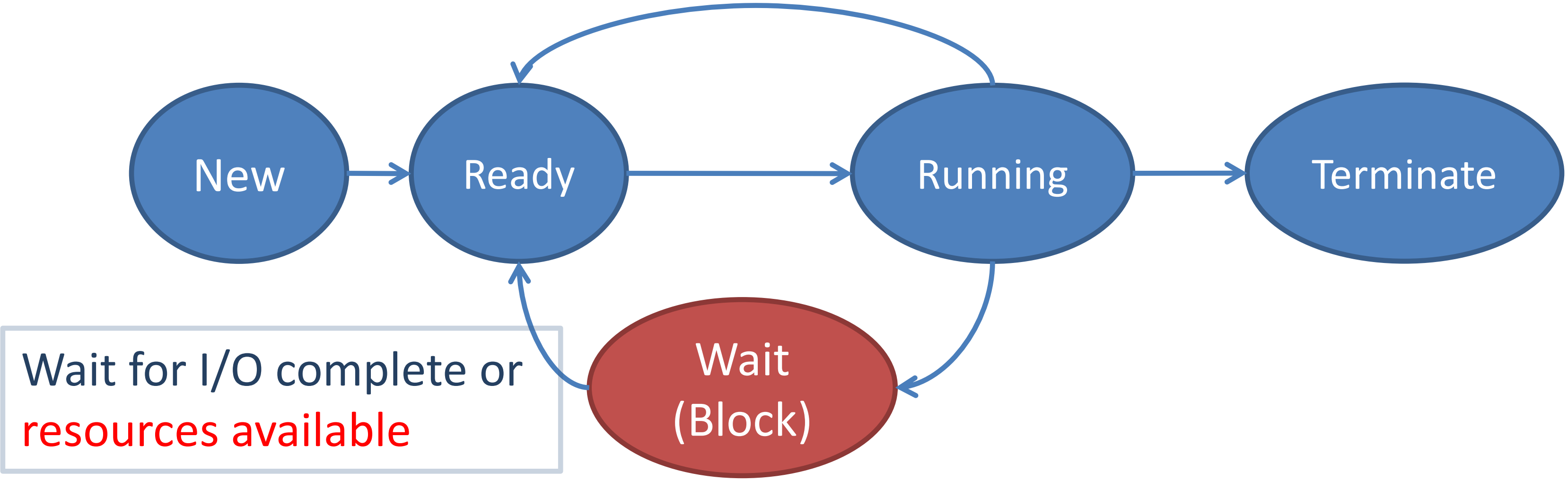
used

R1

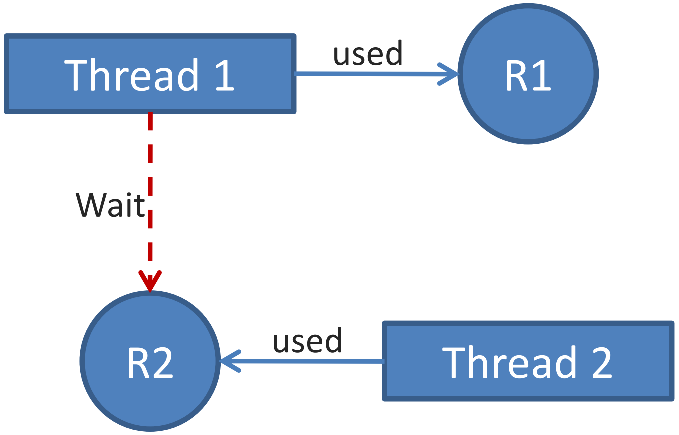


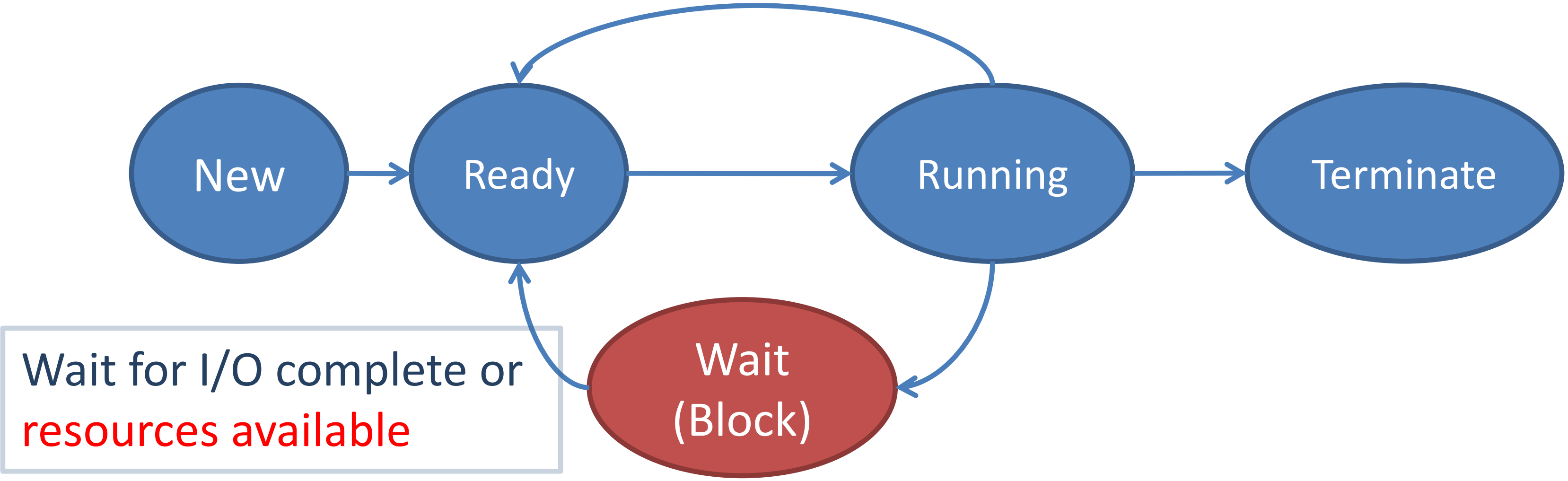
Wait for I/O complete or resources available



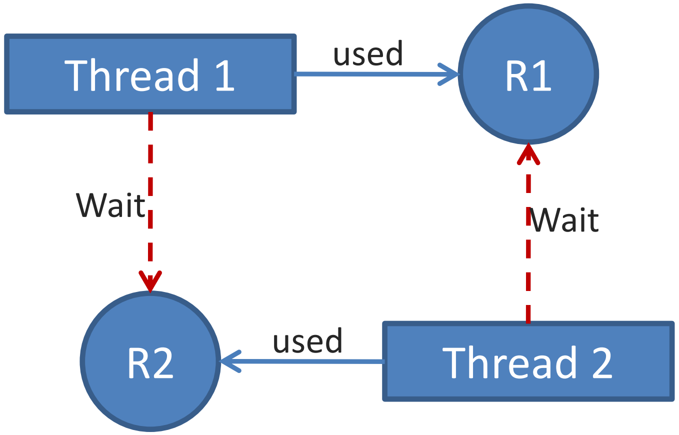


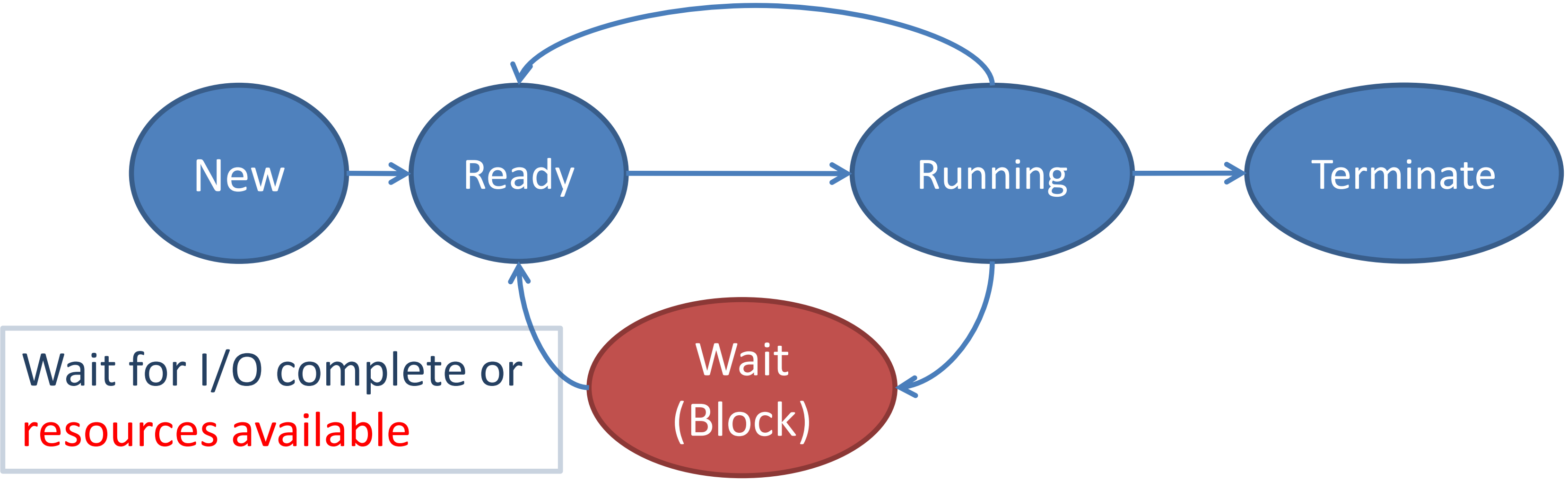
Wait for I/O complete or resources available



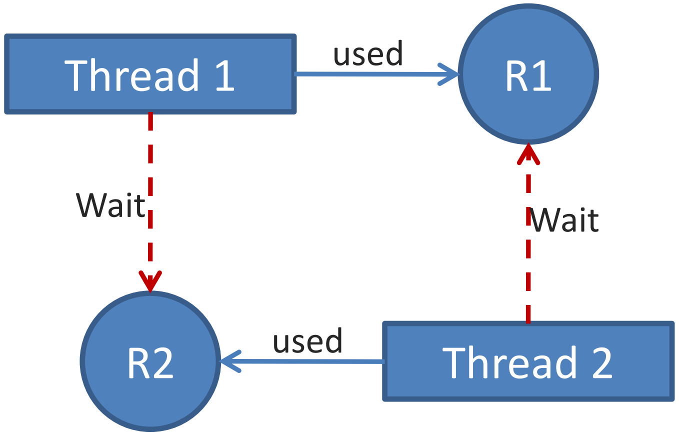


Wait for I/O complete or resources available



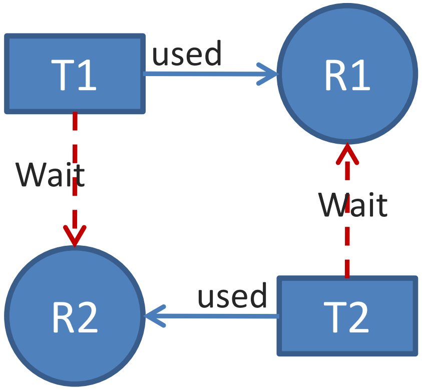


Wait for I/O complete or resources available



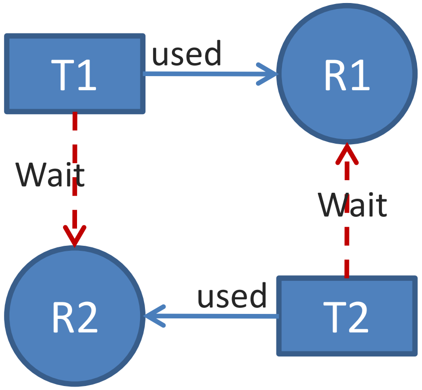
Deadlock: 系統中存在一組Processes，彼此形成circular waiting的情況，使得Processes皆無法繼續往下執行，導致CPU利用度及產能急速下降。

Deadlock 處理策略



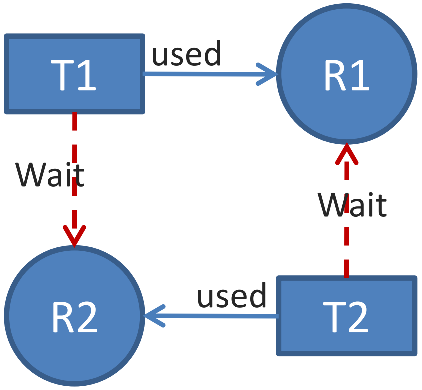
Deadlock 處理策略

- Deadlock形成的必要條件：
  - 資源的Mutual Exclusion
  - Hold & Wait
  - No Preemptive
  - Circular waiting



Deadlock 處理策略

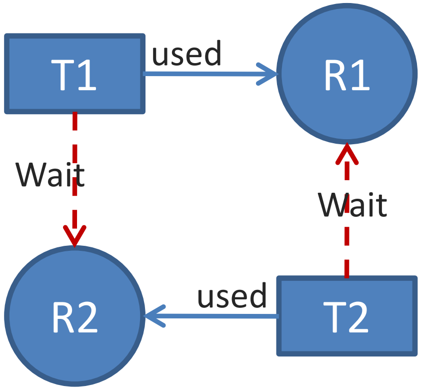
- Deadlock形成的必要條件：
  - 資源的Mutual Exclusion
  - Hold & Wait
  - No Preemptive
  - Circular waiting



- 處理策略：
  - Deadlock Prevention
  - Deadlock Avoidance
  - Deadlock Detection & Recovery

Deadlock 處理策略

- Deadlock形成的必要條件：
  - 資源的Mutual Exclusion
  - Hold & Wait
  - No Preemptive
  - Circular waiting

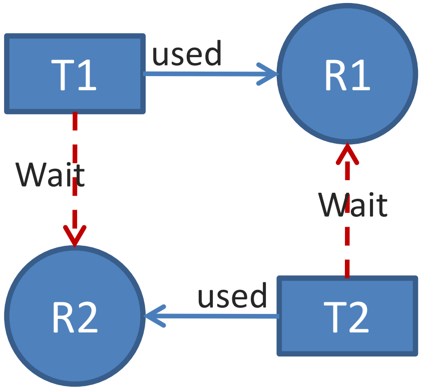


- 處理策略：
  - – Deadlock Prevention
  - – Deadlock Avoidance
  - – Deadlock Detection & Recovery

預防勝於治療

Deadlock 處理策略

- Deadlock形成的必要條件：
  - 資源的Mutual Exclusion
  - Hold & Wait
  - No Preemptive
  - Circular waiting



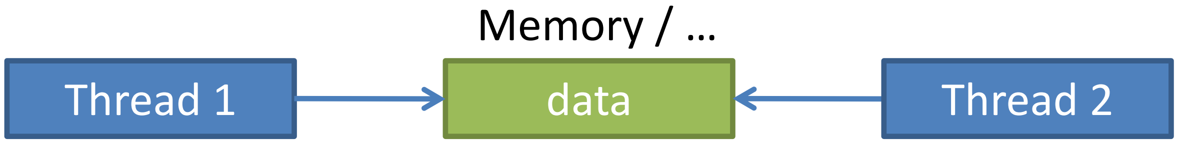
- 處理策略：
  - – Deadlock Prevention
  - – Deadlock Avoidance
  - – Deadlock Detection & Recovery

預防勝於治療

見到棺材才掉淚

Multi-Processing / Multi-Threading (2)

- Communication
  - Shared Memory



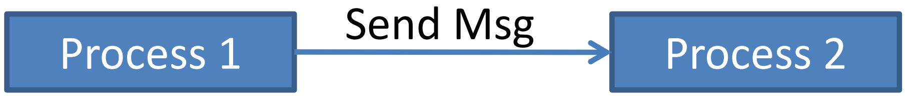
Memory / …

Thread 1

data

Thread 2

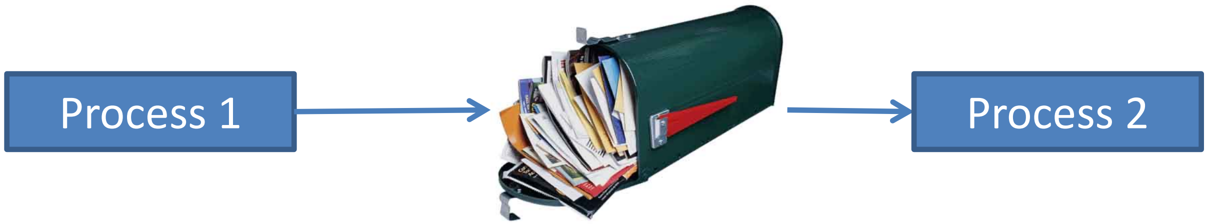
  - Message Passing



Process 1

Send Msg

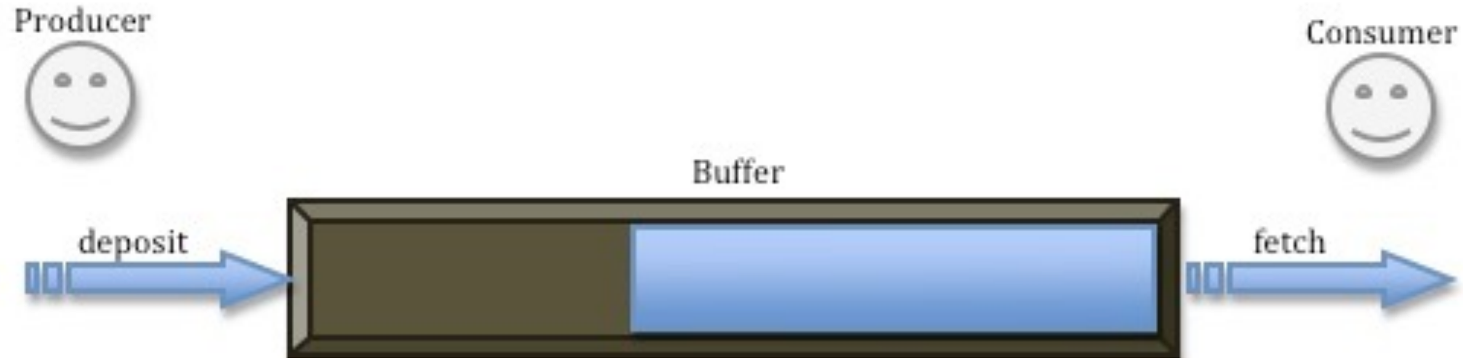
Process 2



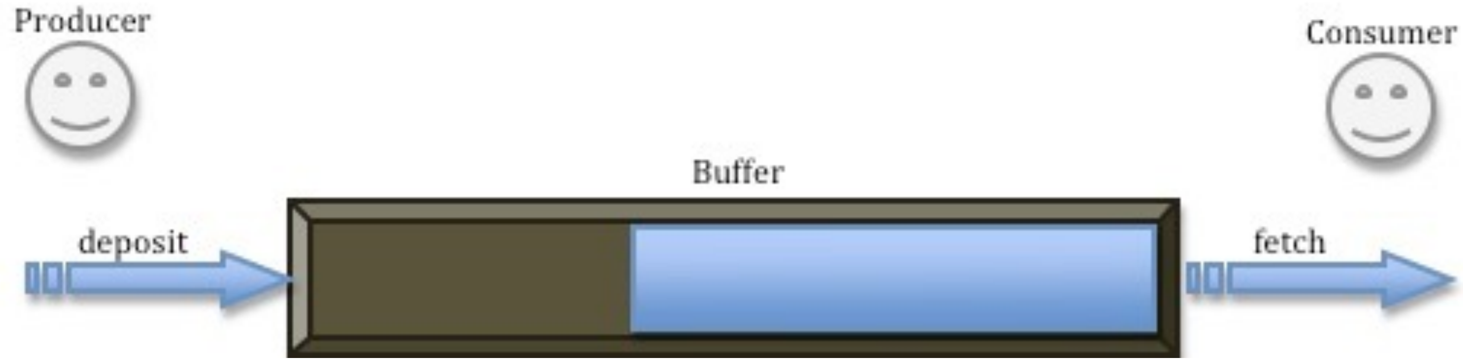
Process 1

Process 2

Producer & Consumer



Producer & Consumer

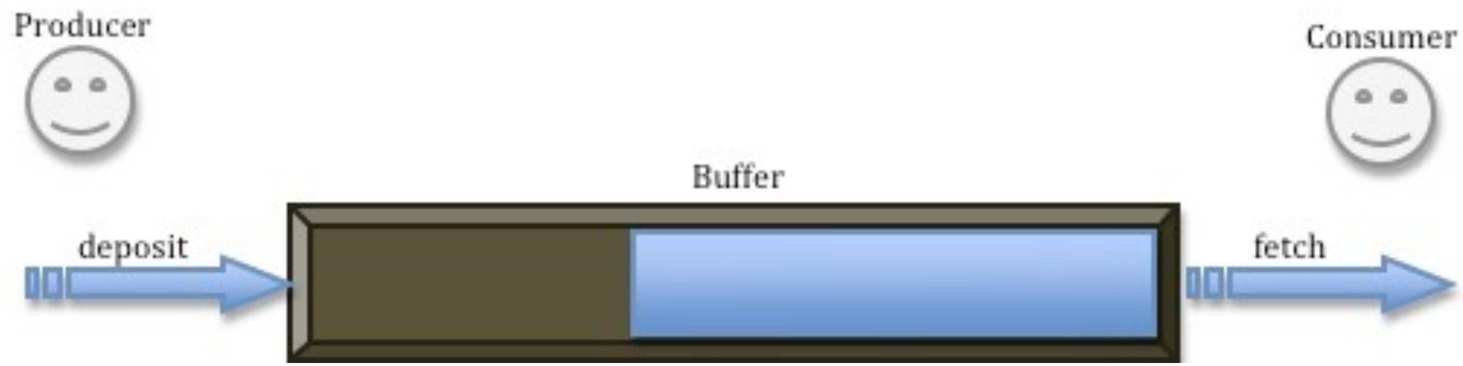


Bounded buffer Producer/Consumer Problem:

(1) 當Buffer滿時，則Producer必須等待。

(2) 當Buffer空時，則Consumer必須等待。

Producer & Consumer



Bounded buffer Producer/Consumer Problem:

(1) 當Buffer滿時，則Producer必須等待。

(2) 當Buffer空時，則Consumer必須等待。

→ 採用一個共用變數count來記錄buffer裡item的個數

Producer:

```
   生產一個item;
   while(count==size)
     do no-op;
   buffer[in] = item;
   in = (in+1) % size;
   count = count+1;
```

Consumer:

```
 while(count==0)
    do no-op;
 item = buffer[out];
 out = (out+1)  mod size;
 count = count-1;
```

Producer:

```
   生產一個item;
   while(count==size)
     do no-op;
   buffer[in] = item;
   in = (in+1) % size;
   count = count+1;
```

Consumer:

```
 while(count==0)
    do no-op;
 item = buffer[out];
 out = (out+1)  mod size;
 count = count-1;
```

1. Producer跟 Consumer是concurrent的
2. 共享count變數

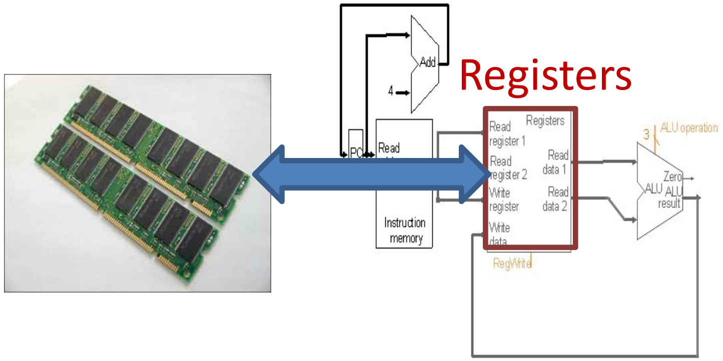
Producer:

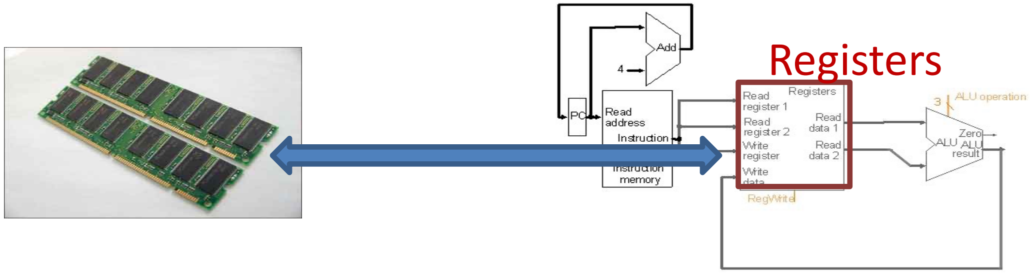
```
   生產一個item;
   while(count==size)
     do no-op;
   buffer[in] = item;
   in = (in+1) % size;
   count = count+1;
```

Consumer:

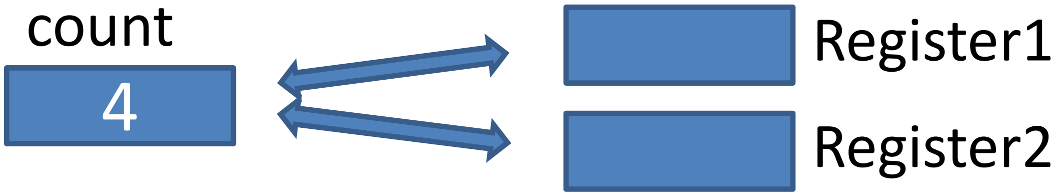
```
 while(count==0)
    do no-op;
 item = buffer[out];
 out = (out+1)  mod size;
 count = count-1;
```

1. Producer跟 Consumer是concurrent的
2. 共享count變數





1. Producer跟 Consumer是concurrent的
2. 共享count變數
3. CPU可被搶奪的 (e.g. SRJF)



count

4

Register1

Register2

Producer (Thread1)

`count = count+1;`

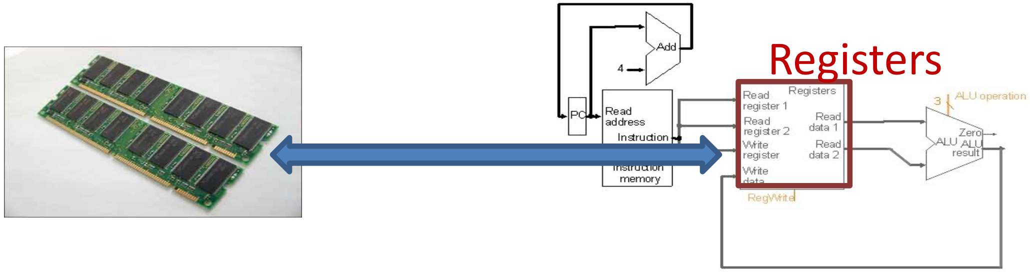
Register1 = count ...(1)
Register1 = Register1 + 1 ...(2)
count = Register1 ...(3)

Consumer (Thread2)

`count = count-1;`

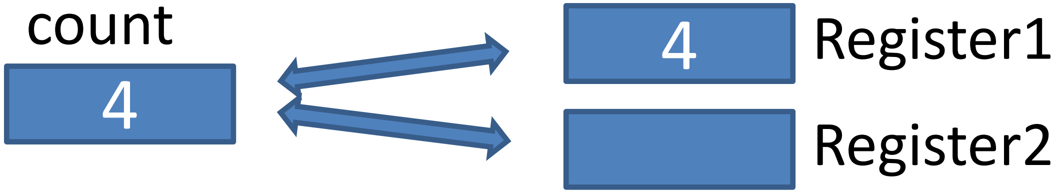
Register2 = count ...(4)
Register2 = Register2 – 1 ...(5)
count = Register2 ...(6)

Not Atomic!!



1. Producer跟 Consumer 是concurrent的
2. 共享count變數
3. CPU可被搶奪的

   (e.g. SRJF)



Producer (Thread1)

```
count = count+1;
```

Register1 = count ...(1)

Register1 = Register1 + 1 ...(2)

count = Register1 ...(3)

Consumer (Thread2)

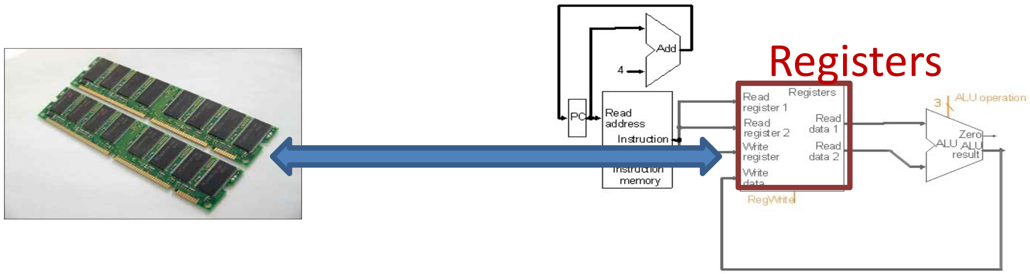
```
count = count-1;
```

Register2 = count ...(4)

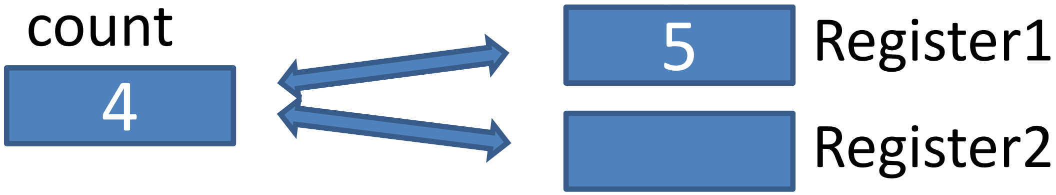
Register2 = Register2 – 1 ...(5)

count = Register2 ...(6)

Not Atomic!!



1. Producer跟 Consumer是concurrent的
2. 共享count變數
3. CPU可被搶奪的 (e.g. SRJF)



count

4

5 Register1

Register2

Producer (Thread1)

```
count = count+1;
```

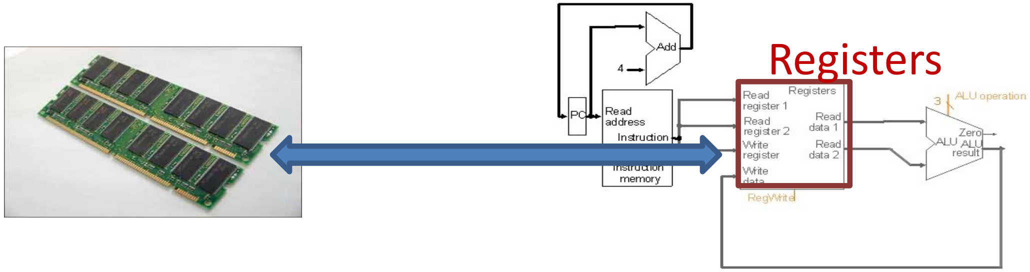
Register1 = count	…(1)
Register1 = Register1 + 1	…(2)
count = Register1	…(3)

Consumer (Thread2)

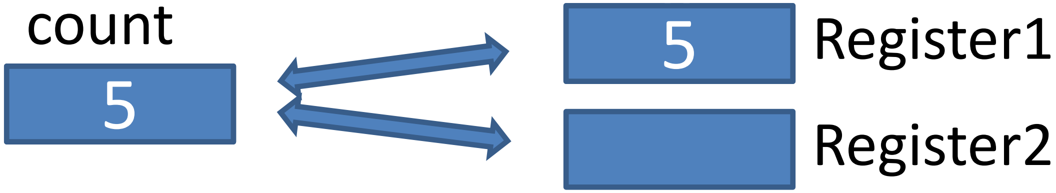
```
count = count-1;
```

Register2 = count	…(4)
Register2 = Register2 – 1	…(5)
count = Register2	…(6)

Not Atomic!!



1. Producer跟 Consumer是concurrent的
2. 共享count變數
3. CPU可被搶奪的 (e.g. SRJF)



Producer (Thread1)

```
count = count+1;
```

Register1 = count ...(1)

Register1 = Register1 + 1 ...(2)

count = Register1 ...(3)

Consumer (Thread2)

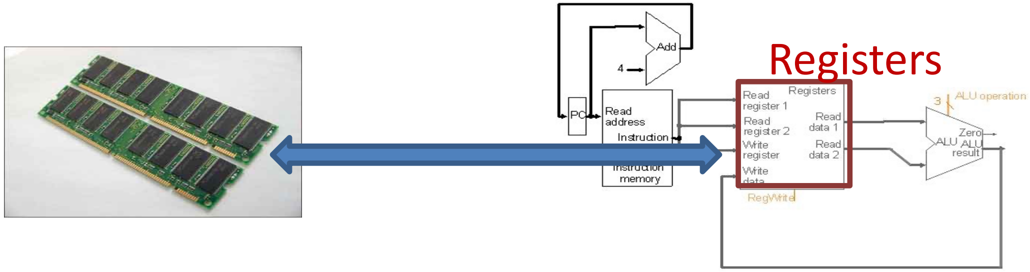
```
count = count-1;
```

Register2 = count ...(4)

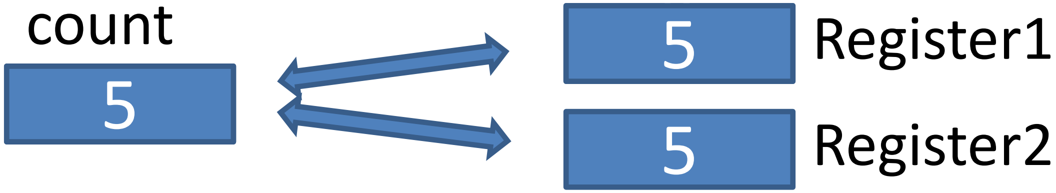
Register2 = Register2 – 1 ...(5)

count = Register2 ...(6)

Not Atomic!!



1. Producer跟 Consumer 是concurrent的
2. 共享count變數
3. CPU可被搶奪的 (e.g. SRJF)



Producer (Thread1)

```
count = count+1;
```

Register1 = count ...(1)

Register1 = Register1 + 1 ...(2)

count = Register1 ...(3)

Consumer (Thread2)

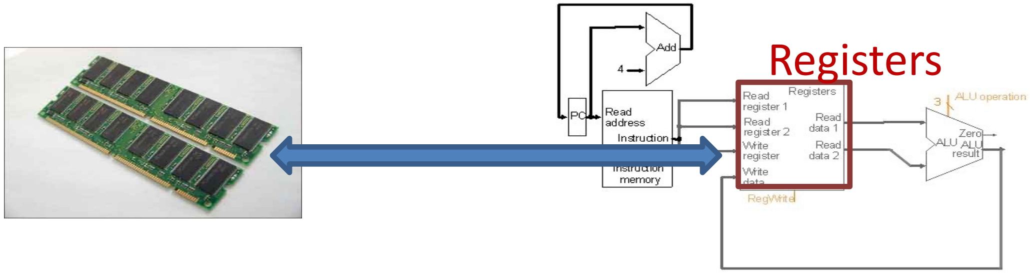
```
count = count-1;
```

Register2 = count ...(4)

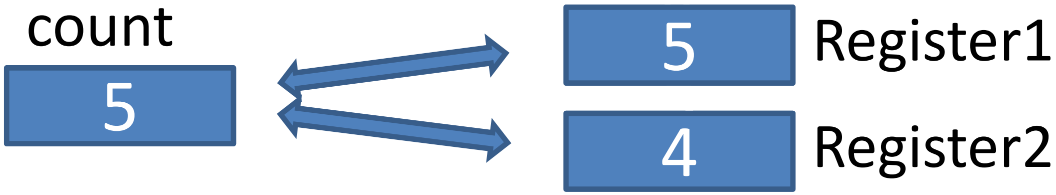
Register2 = Register2 – 1 ...(5)

count = Register2 ...(6)

Not Atomic!!



1. Producer跟 Consumer是concurrent的
2. 共享count變數
3. CPU可被搶奪的 (e.g. SRJF)



count

5

5 Register1

4 Register2

Producer (Thread1)

```
count = count+1;
```

Register1 = count ...(1)

Register1 = Register1 + 1 ...(2)

count = Register1 ...(3)

Consumer (Thread2)

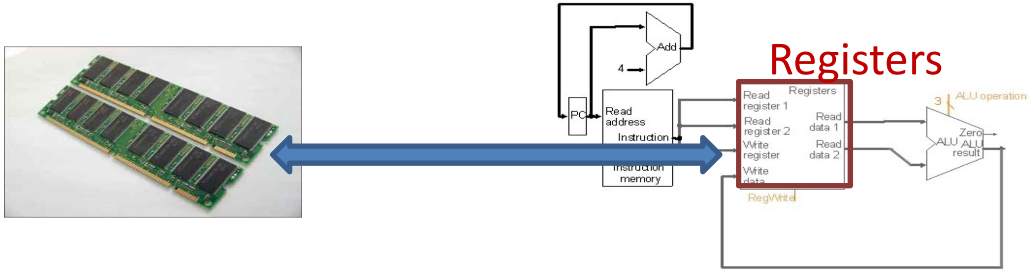
```
count = count-1;
```

Register2 = count ...(4)

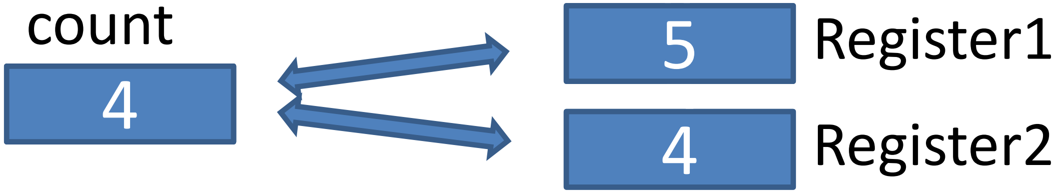
Register2 = Register2 – 1 ...(5)

count = Register2 ...(6)

Not Atomic!!



1. Producer跟 Consumer 是concurrent的
2. 共享count變數
3. CPU可被搶奪的 (e.g. SRJF)



Producer (Thread1)

```
count = count+1;
```

Register1 = count ...(1)

Register1 = Register1 + 1 ...(2)

count = Register1 ...(3)

Consumer (Thread2)

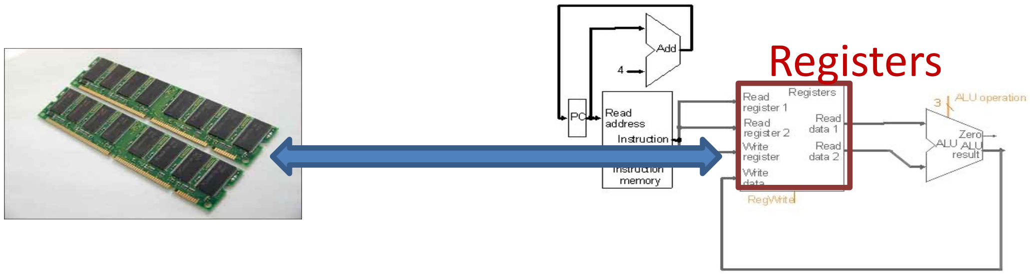
```
count = count-1;
```

Register2 = count ...(4)

Register2 = Register2 – 1 ...(5)

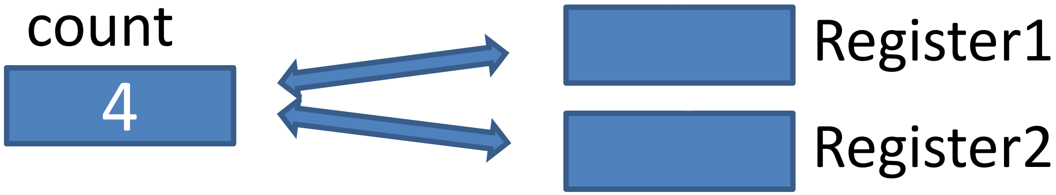
count = Register2 ...(6)

Not Atomic!!



1. Producer跟 Consumer 是concurrent的
2. 共享count變數
3. CPU可被搶奪的

   (e.g. SRJF)



Producer (Thread1)

```
count = count+1;
```

Register1 = count ...(1)

Register1 = Register1 + 1 ...(2)

count = Register1 ...(5)

Consumer (Thread2)

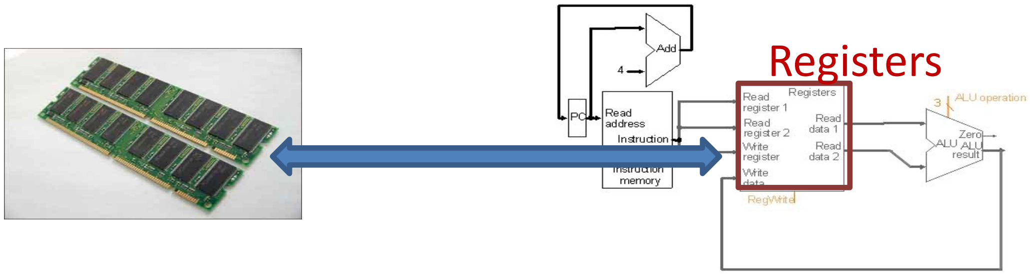
```
count = count-1;
```

Register2 = count ...(3)

Register2 = Register2 – 1 ...(4)

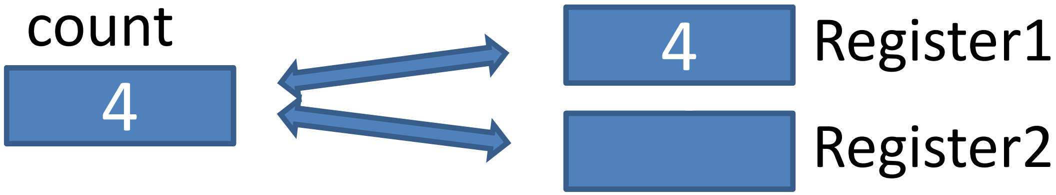
count = Register2 ...(6)

Not Atomic!!



1. Producer跟 Consumer是concurrent的
2. 共享count變數
3. CPU可被搶奪的

(e.g. SRJF)



count

4

4 Register1

Register2

Producer (Thread1)

```
count = count+1;
```

Register1 = count ...(1)

Register1 = Register1 + 1 ...(2)

count = Register1 ...(5)

Consumer (Thread2)

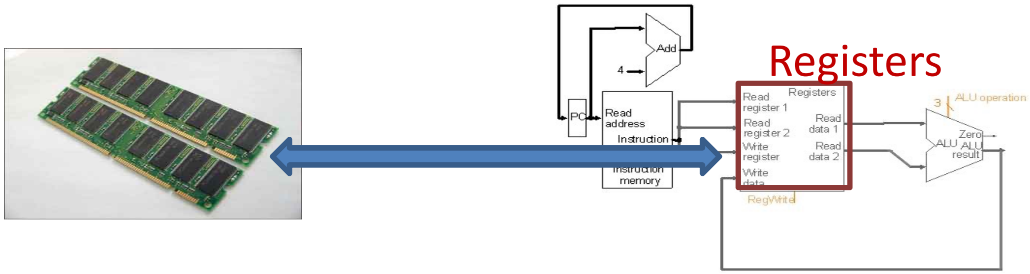
```
count = count-1;
```

Register2 = count ...(3)

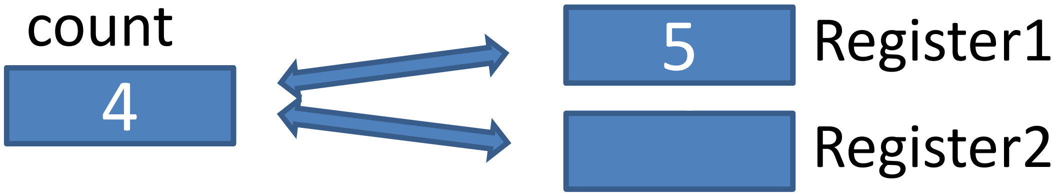
Register2 = Register2 – 1 ...(4)

count = Register2 ...(6)

Not Atomic!!



1. Producer跟 Consumer 是concurrent的
2. 共享count變數
3. CPU可被搶奪的 (e.g. SRJF)



Producer (Thread1)

```
count = count+1;
```

Register1 = count ...(1)

Register1 = Register1 + 1 ...(2)

count = Register1 ...(5)

Consumer (Thread2)

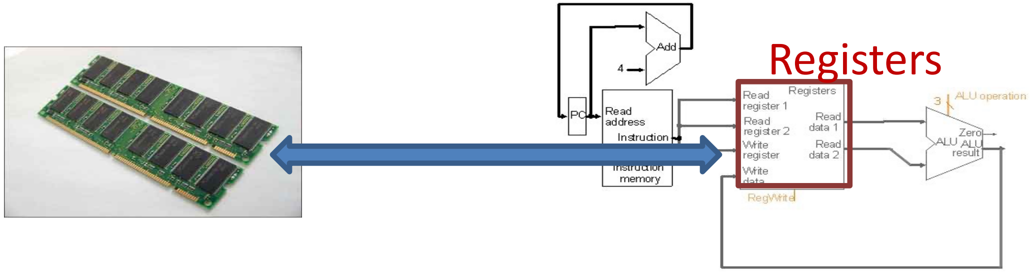
```
count = count-1;
```

Register2 = count ...(3)

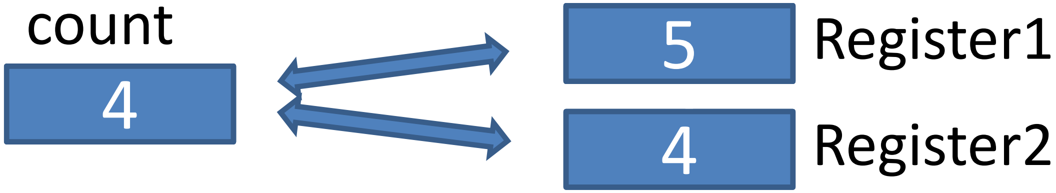
Register2 = Register2 – 1 ...(4)

count = Register2 ...(6)

Not Atomic!!



1. Producer跟 Consumer是concurrent的
2. 共享count變數
3. CPU可被搶奪的 (e.g. SRJF)



count

4

5 Register1

4 Register2

Producer (Thread1)

```
count = count+1;
```

Register1 = count ...(1)

Register1 = Register1 + 1 ...(2)

count = Register1 ...(5)

Consumer (Thread2)

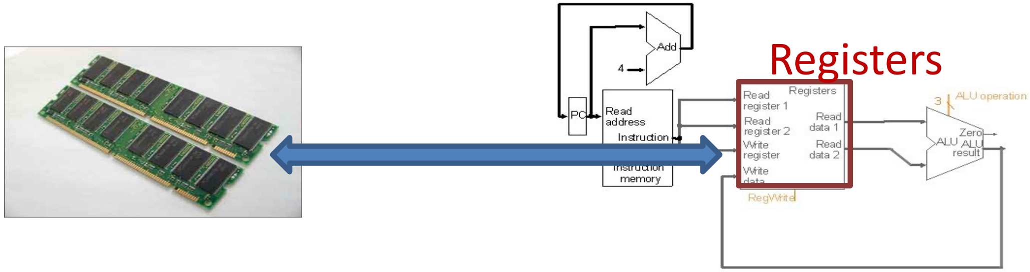
```
count = count-1;
```

Register2 = count ...(3)

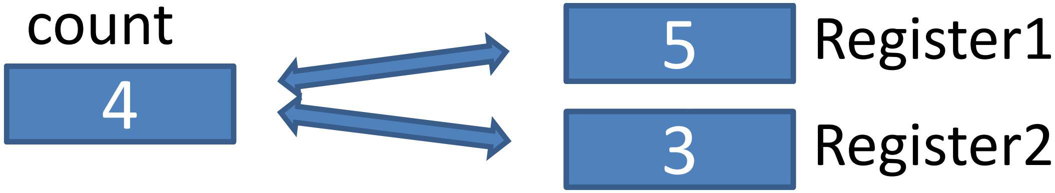
Register2 = Register2 – 1 ...(4)

count = Register2 ...(6)

Not Atomic!!



1. Producer跟 Consumer是concurrent的
2. 共享count變數
3. CPU可被搶奪的 (e.g. SRJF)



Producer (Thread1)

```
count = count+1;
```

Register1 = count ...(1)

Register1 = Register1 + 1 ...(2)

count = Register1 ...(5)

Consumer (Thread2)

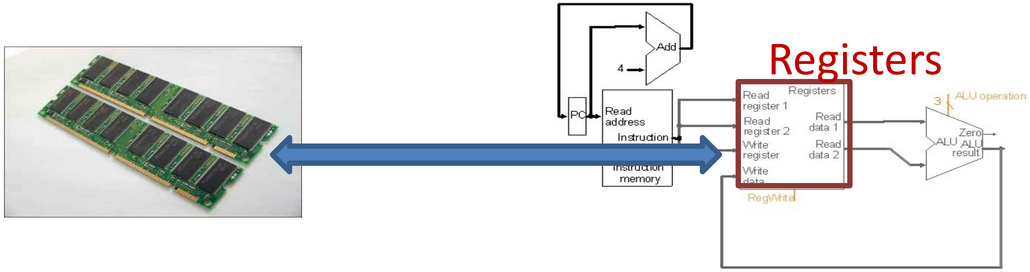
```
count = count-1;
```

Register2 = count ...(3)

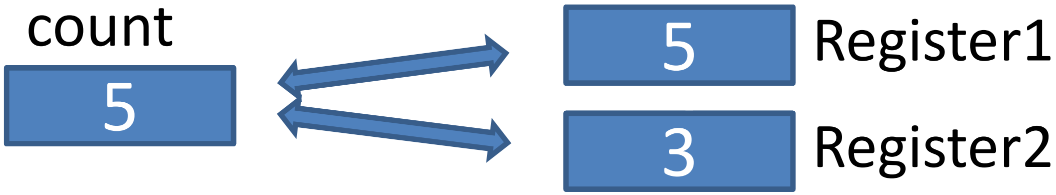
Register2 = Register2 – 1 ...(4)

count = Register2 ...(6)

Not Atomic!!



1. Producer跟 Consumer 是concurrent的
2. 共享count變數
3. CPU可被搶奪的 (e.g. SRJF)



Producer (Thread1)

```
count = count+1;
```

Register1 = count ...(1)

Register1 = Register1 + 1 ...(2)

count = Register1 ...(5)

Consumer (Thread2)

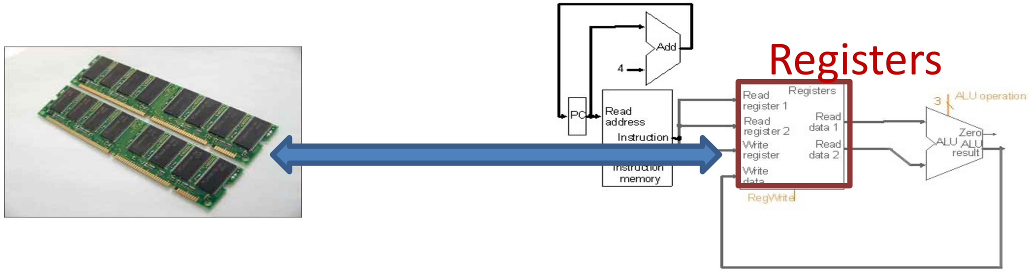
```
count = count-1;
```

Register2 = count ...(3)

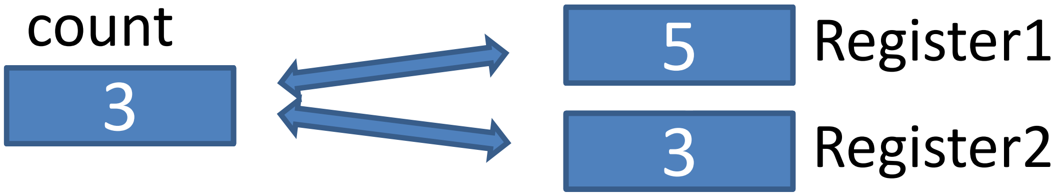
Register2 = Register2 – 1 ...(4)

count = Register2 ...(6)

Not Atomic!!



1. Producer跟 Consumer是concurrent的
2. 共享count變數
3. CPU可被搶奪的 (e.g. SRJF)


count
3
5
Register1
3
Register2

Producer (Thread1)

```
count = count+1;
```

Register1 = count ...(1)
Register1 = Register1 + 1 ...(2)
count = Register1 ...(5)

Consumer (Thread2)

```
count = count-1;
```

Register2 = count ...(3)
Register2 = Register2 – 1 ...(4)
count = Register2 ...(6)

Not Atomic!!

Race Condition

- 在Share Memory溝通方式下，共享變數的值會因為Processes執行的順序不同而有所不同。

```
Producer
count = count+1;
```

```
Consumer:
count = count-1;
```

Race Condition

- 在Share Memory溝通方式下，共享變數的值會因為Processes執行的順序不同而有所不同。

```
Producer
count = count+1;
```

```
Consumer:
count = count-1;
```

保證同一時間只有一個Process在存取共享變數

Critical Section

```
Repeat
    …
    Entry section
    C.S.
    Exit section
    …
Until false
```

Critical Section

```
Repeat
    …
    Entry section
    C.S.
    Exit section
    …
Until false
```

- Mutual Exclusion: 在任何時間點最多允許一個process在其C.S.內活動

Critical Section

```
Repeat
    …
    Entry section
    C.S.
    Exit section
    …
Until false
```

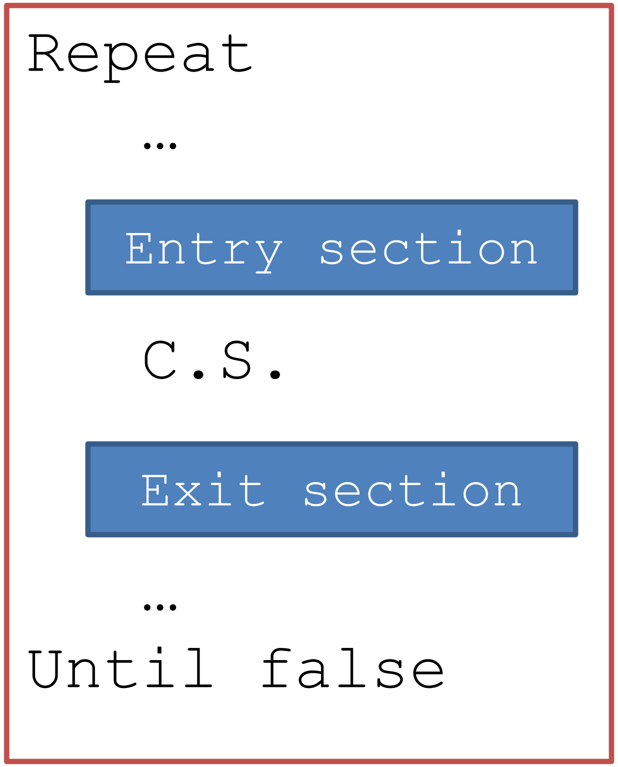
- Mutual Exclusion: 在任何時間點最多允許一個process在其C.S.內活動
- Progress
  - 不想進入C.S.的Process不能阻礙其他process進入
  - 在有限的時間內必須從想進入C.S.的processes中決定出一個Process進入C.S.

Critical Section

```
Repeat
    …
    Entry section
    C.S.
    Exit section
    …
Until false
```

- Mutual Exclusion: 在任何時間點最多允許一個process在其C.S.內活動
- Progress
  - 不想進入C.S.的Process不能阻礙其他process進入
  - 在有限的時間內必須從想進入C.S.的processes中決定出一個Process進入C.S.
- Bounded Waiting: Process從提出申請要進入C.S.到其獲准進入C.S.的這段時間是有限的

Critical Section



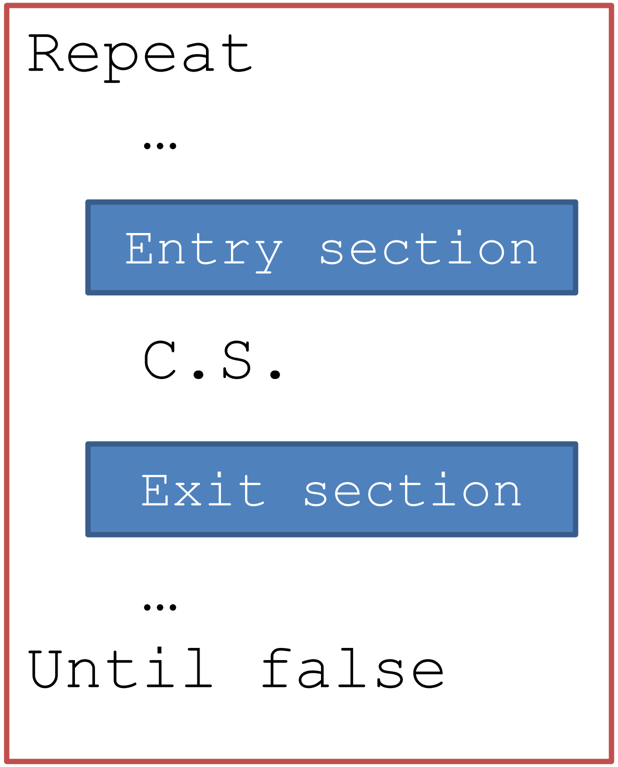
```
Repeat
    …
    Entry section
    C.S.
    Exit section
    …
Until false
```

- Mutual Exclusion: 在任何時間點最多允許一個process在其C.S.內活動
- Progress
  - 不想進入C.S.的Process不能阻礙其他process進入
  - 在有限的時間內必須從想進入C.S.的processes中決定出一個Process進入C.S.

No Deadlock

- Bounded Waiting: Process從提出申請要進入C.S.到其獲准進入C.S.的這段時間是有限的

Critical Section



```
Repeat
    …
    Entry section
    C.S.
    Exit section
    …
Until false
```

- Mutual Exclusion: 在任何時間點最多允許一個process在其C.S.內活動
- Progress
  - 不想進入C.S.的Process不能阻礙其他process進入
  - 在有限的時間內必須從想進入C.S.的processes中決定出一個Process進入C.S.

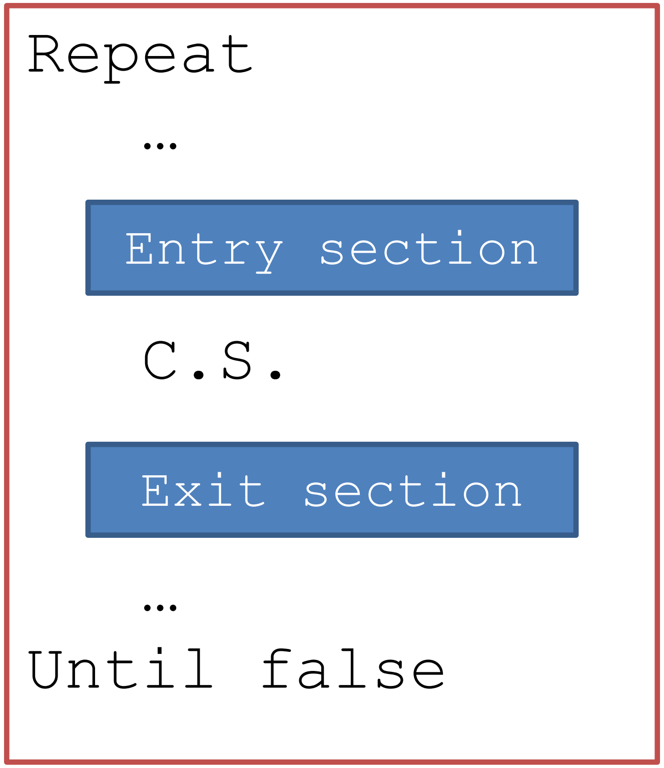
No Deadlock

- Bounded Waiting: Process從提出申請要進入C.S.到其獲准進入C.S.的這段時間是有限的

No Starvation

C.S. Design – design entry/exit section

- Software Solution
- Hardware Solution
- Semaphore
- Monitor
- Critical Region
- …



```
Repeat
     …
    Entry section
     C.S.
    Exit section
     …
Until false
```

Multi-Processing / Multi-Threading (3)

- 同步問題

如果我們希望A敘述
一定要在B敘述之前
執行，怎麼做？

P_1	P_2
…	…
A	B
…	…

Multi-Processing / Multi-Threading (3)

- 同步問題

如果我們希望A敘述一定要在B敘述之前執行，怎麼做？

P_1	P_2
…	…
A	B
…	…

```
Wait(s){
        while(s<=0) do no-op;
         s = s - 1;
}
```

```
Signal(s){
         s++;
}
```

Multi-Processing / Multi-Threading (3)

- 同步問題

如果我們希望A敘述一定要在B敘述之前執行，怎麼做？

```
Wait(s){
        while(s<=0) do no-op;
         s = s - 1;
}
```

```
Signal(s){
         s++;
}
```

P_1	P_2
…	…
A	B
…	…

P_1	P_2
…	…
A	wait(s)
Signal(s)	B
…	…

Multi-Processing / Multi-Threading (3)

- 同步問題

如果我們希望A敘述一定要在B敘述之前執行，怎麼做？

P_1	P_2
…	…
A	B
…	…

```
Wait(s){
        while(s<=0) do no-op;
         s = s - 1;
}
```

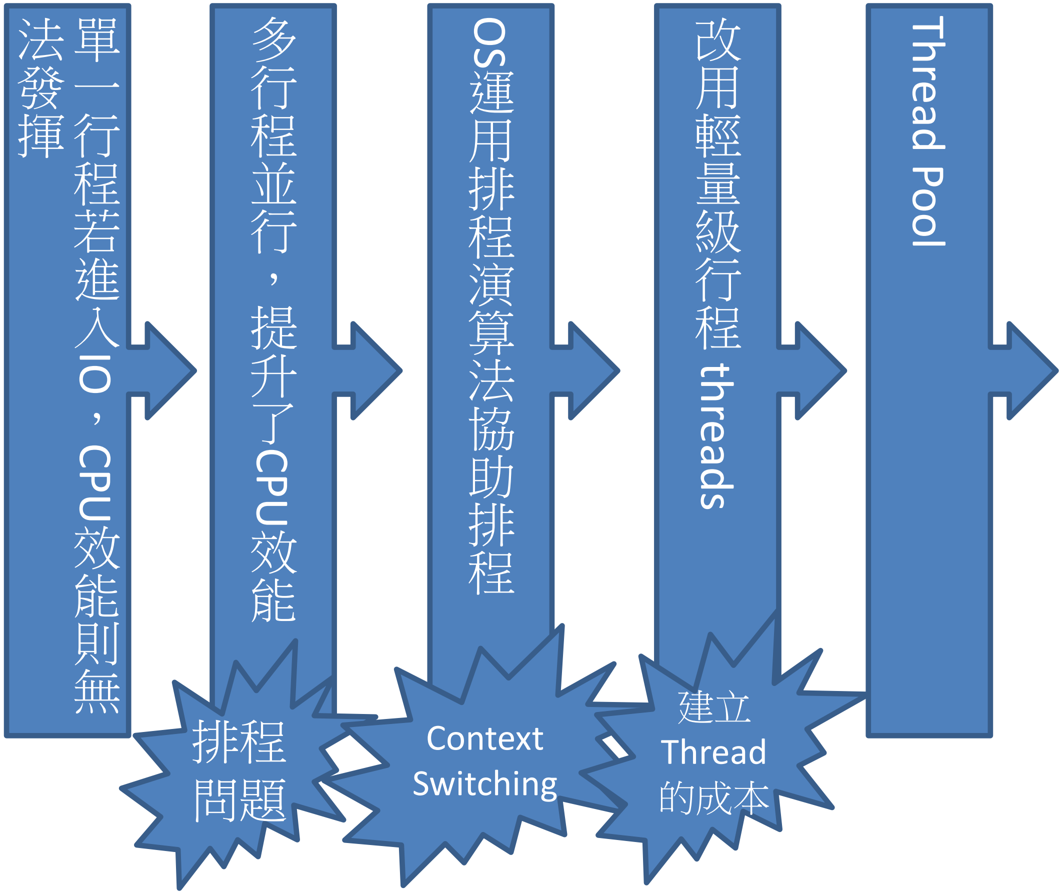
```
Signal(s){
         s++;
}
```

Semaphore

P_1	P_2
…	…
A	wait(s)
Signal(s)	B
…	…

Synchronization

- 著名的同步問題
  - Dining philosophers problem
  - Producer-consumer problem
  - Readers-writers problem
  - Sleeping barber problem
- 解法：
  - Semaphore
  - Monitor
  - Critical Region



單一行程若進入IO，CPU效能則無法發揮

多行程並行，提升了CPU效能

OS運用排程演算法協助排程

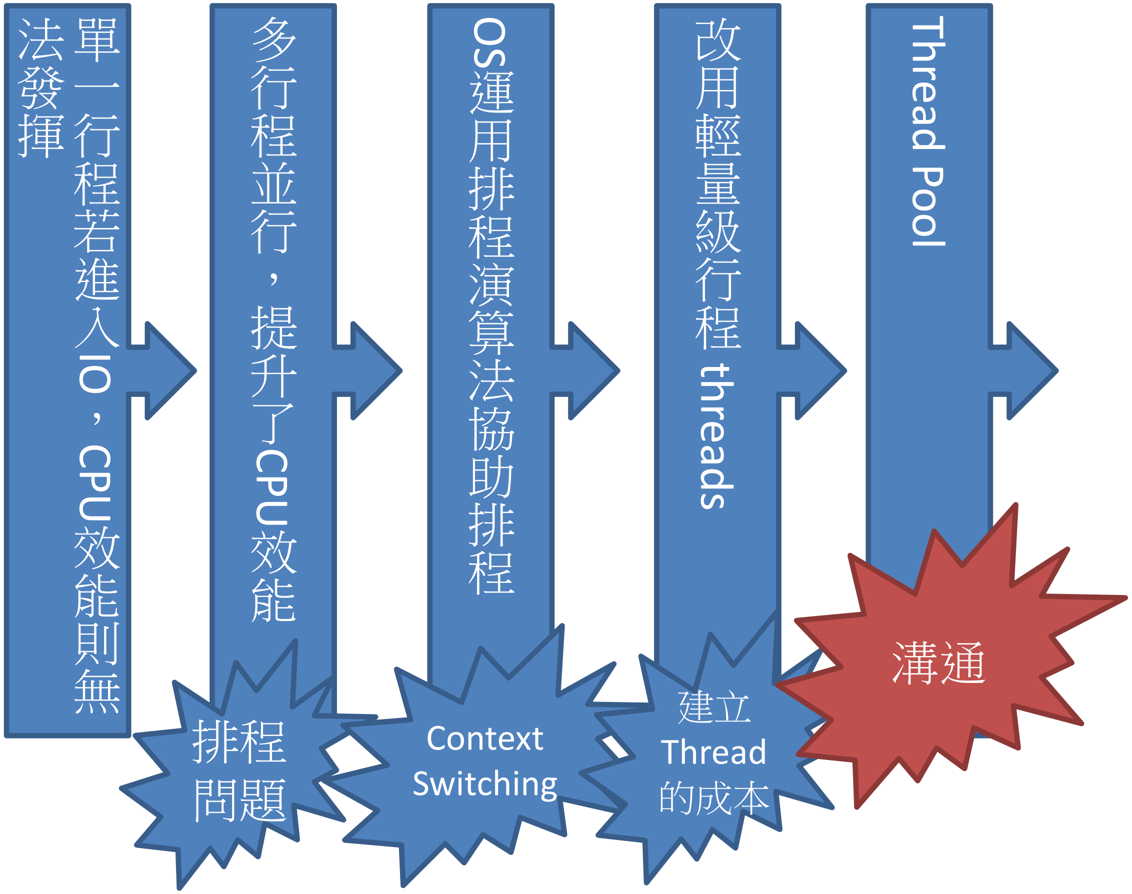
改用輕量級行程 threads

Thread Pool

排程問題

Context Switching

建立Thread的成本



單一行程若進入IO，CPU效能則無法發揮

多行程並行，提升了CPU效能

OS運用排程演算法協助排程

改用輕量級行程threads

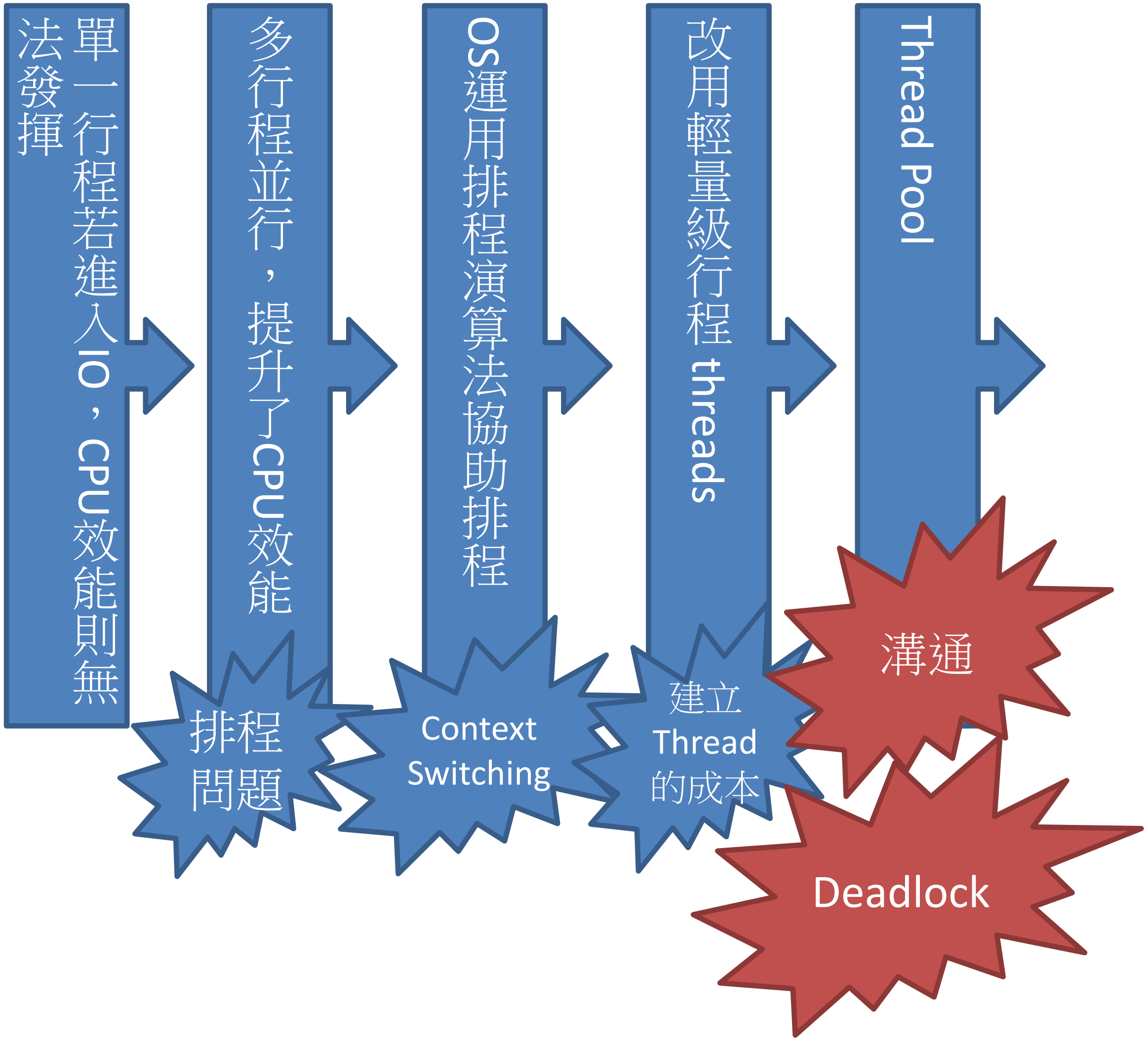
Thread Pool

排程問題

Context Switching

建立Thread的成本

溝通



單一行程若進入IO，CPU效能則無法發揮

多行程並行，提升了CPU效能

OS運用排程演算法協助排程

改用輕量級行程threads

Thread Pool

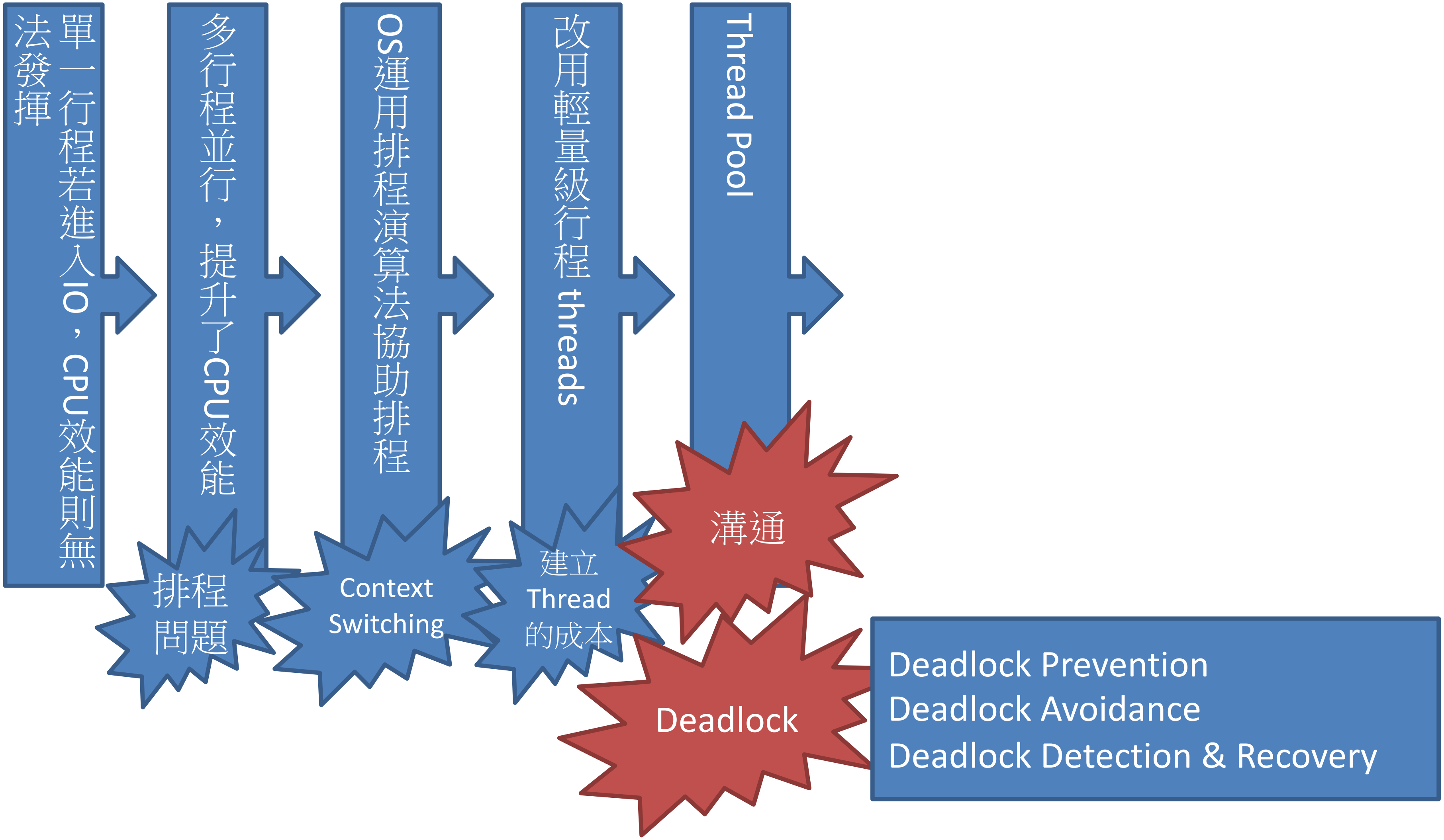
排程問題

Context Switching

建立Thread的成本

溝通

Deadlock



單一行程若進入IO，CPU效能則無法發揮

多行程並行，提升了CPU效能

OS運用排程演算法協助排程

改用輕量級行程threads

Thread Pool

排程問題

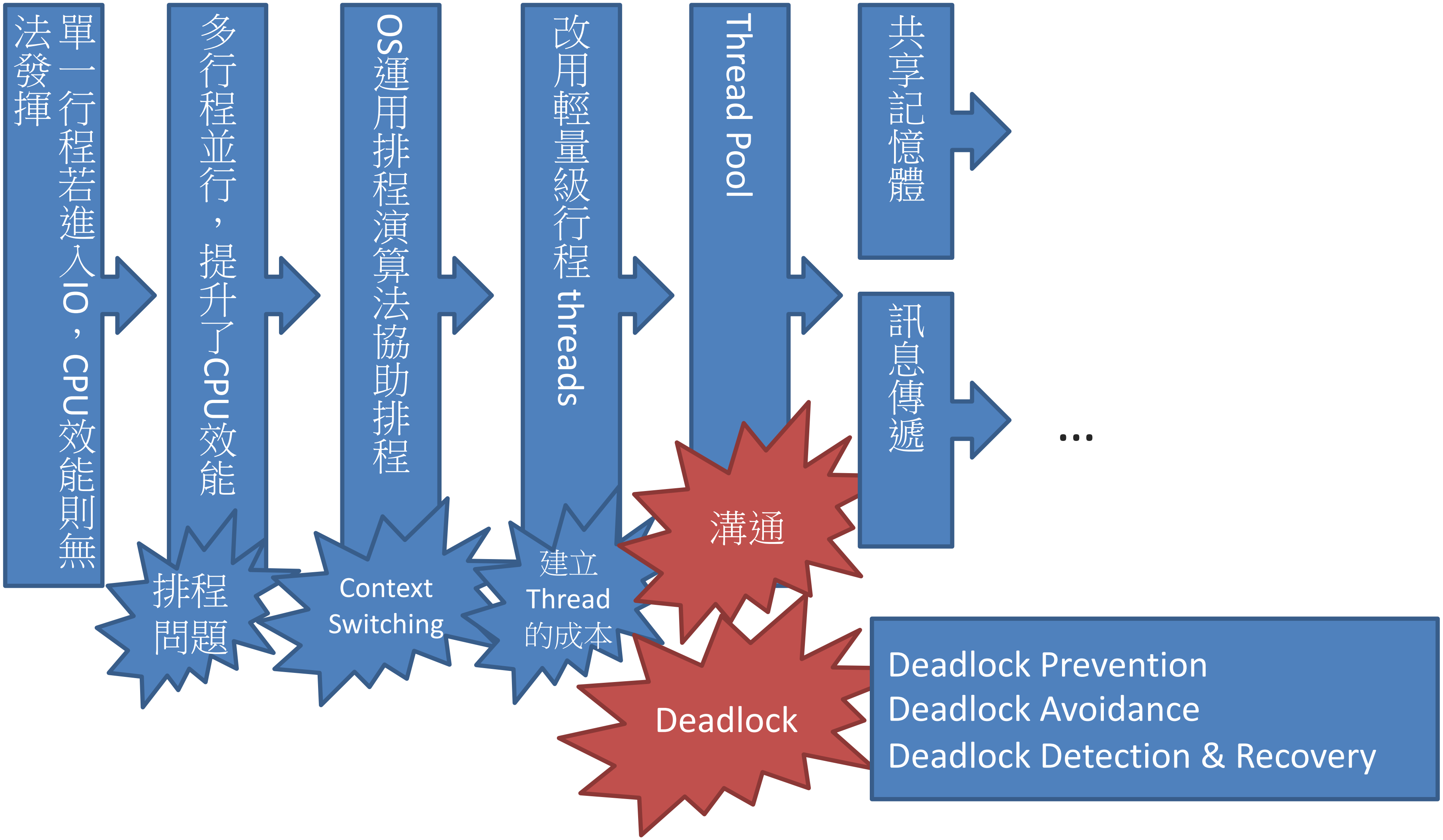
Context Switching

建立Thread的成本

溝通

Deadlock

Deadlock Prevention
Deadlock Avoidance
Deadlock Detection & Recovery



單一行程若進入IO，CPU效能則無法發揮

多行程並行，提升了CPU效能

OS運用排程演算法協助排程

改用輕量級行程 threads

Thread Pool

共享記憶體

訊息傳遞

…

排程問題

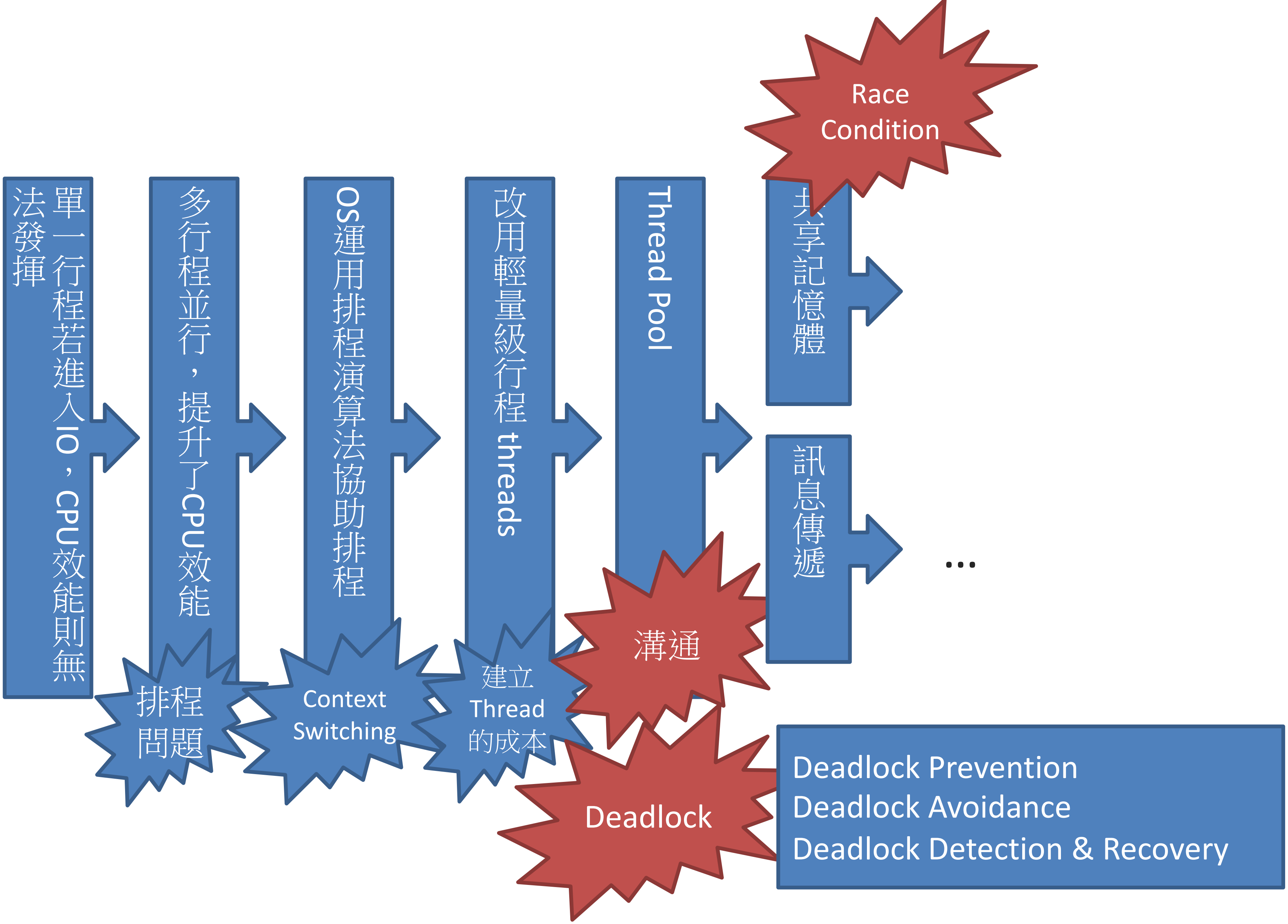
Context Switching

建立Thread的成本

溝通

Deadlock

Deadlock Prevention
Deadlock Avoidance
Deadlock Detection & Recovery



單一行程若進入IO，CPU效能則無法發揮

多行程並行，提升了CPU效能

OS運用排程演算法協助排程

改用輕量級行程 threads

Thread Pool

共享記憶體

訊息傳遞

…

排程問題

Context Switching

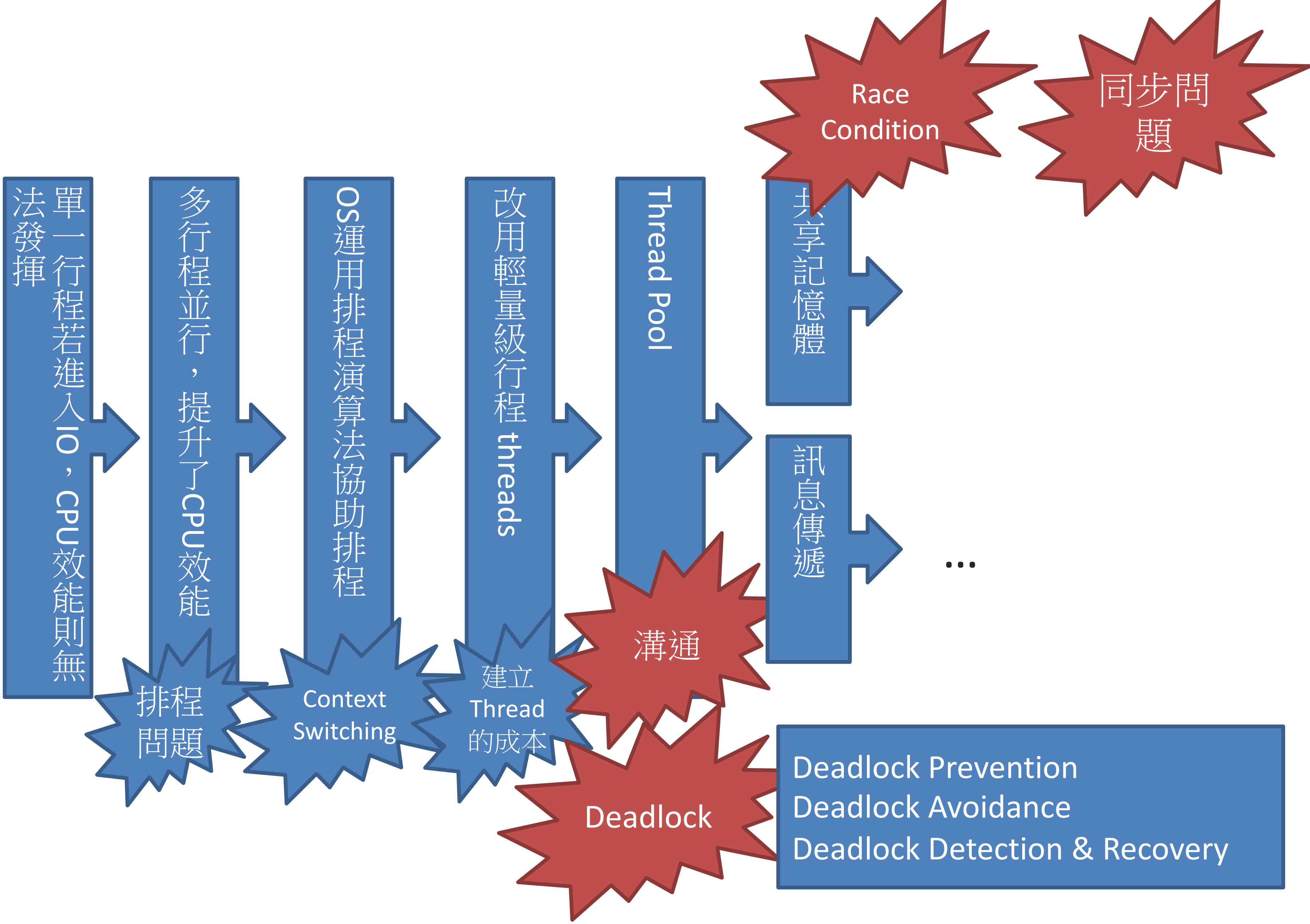
建立Thread的成本

溝通

Race Condition

Deadlock

Deadlock Prevention
Deadlock Avoidance
Deadlock Detection & Recovery



單一行程若進入IO，CPU效能則無法發揮

排程問題

多行程並行，提升了CPU效能

Context Switching

OS運用排程演算法協助排程

建立Thread的成本

改用輕量級行程 threads

Thread Pool

溝通

共享記憶體

Race Condition

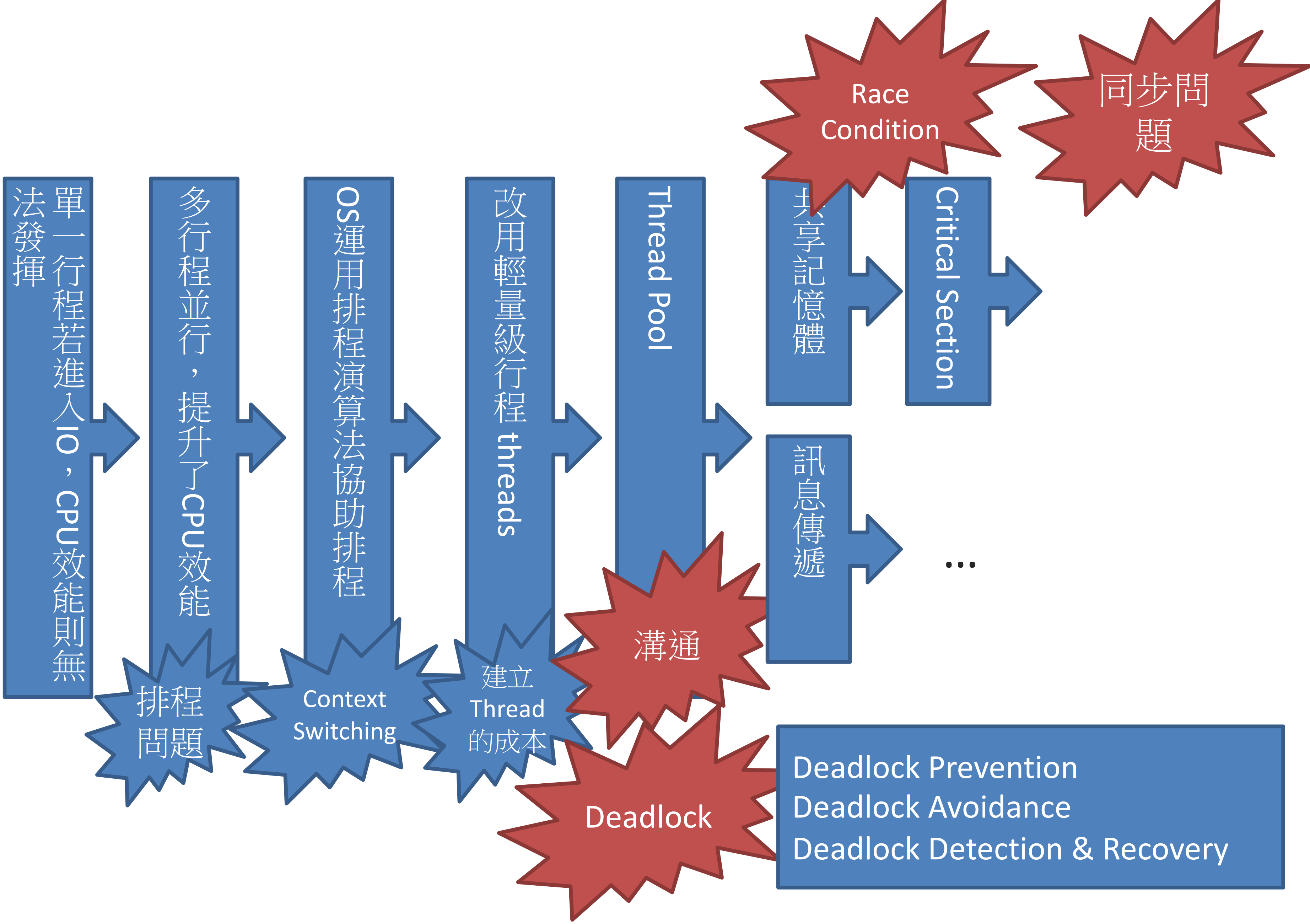
同步問題

訊息傳遞

…

Deadlock

Deadlock Prevention
Deadlock Avoidance
Deadlock Detection & Recovery



單一行程若進入IO，CPU效能則無法發揮

排程問題

多行程並行，提升了CPU效能

Context Switching

OS運用排程演算法協助排程

建立Thread的成本

改用輕量級行程threads

Thread Pool

溝通

Deadlock

共享記憶體

Race Condition

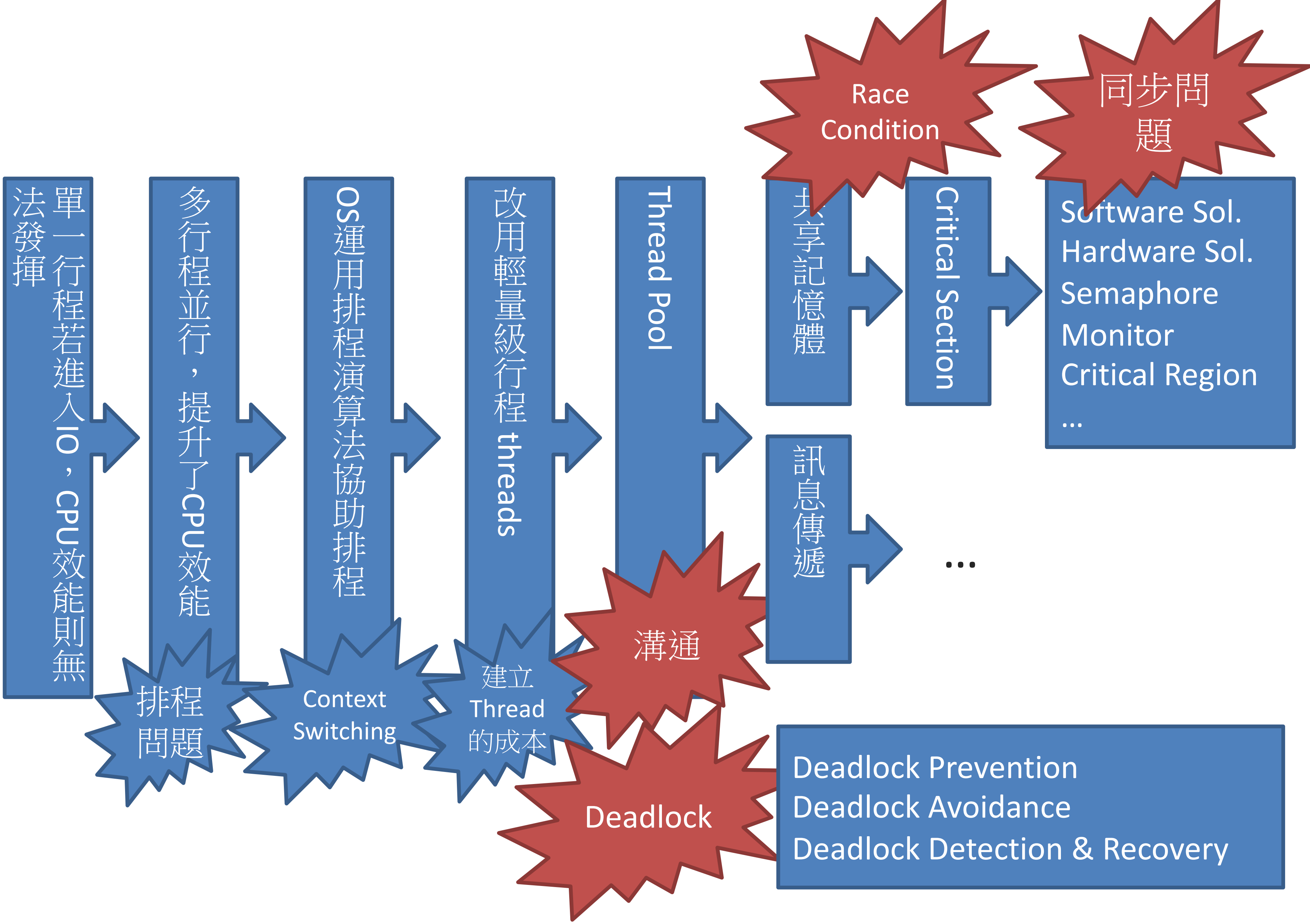
Critical Section

同步問題

訊息傳遞

...

Deadlock Prevention
Deadlock Avoidance
Deadlock Detection & Recovery



單一行程若進入IO，CPU效能則無法發揮

排程問題

多行程並行，提升了CPU效能

Context Switching

OS運用排程演算法協助排程

建立Thread的成本

改用輕量級行程 threads

Thread Pool

溝通

共享記憶體

Race Condition

Critical Section

同步問題

Software Sol.
Hardware Sol.
Semaphore
Monitor
Critical Region
…

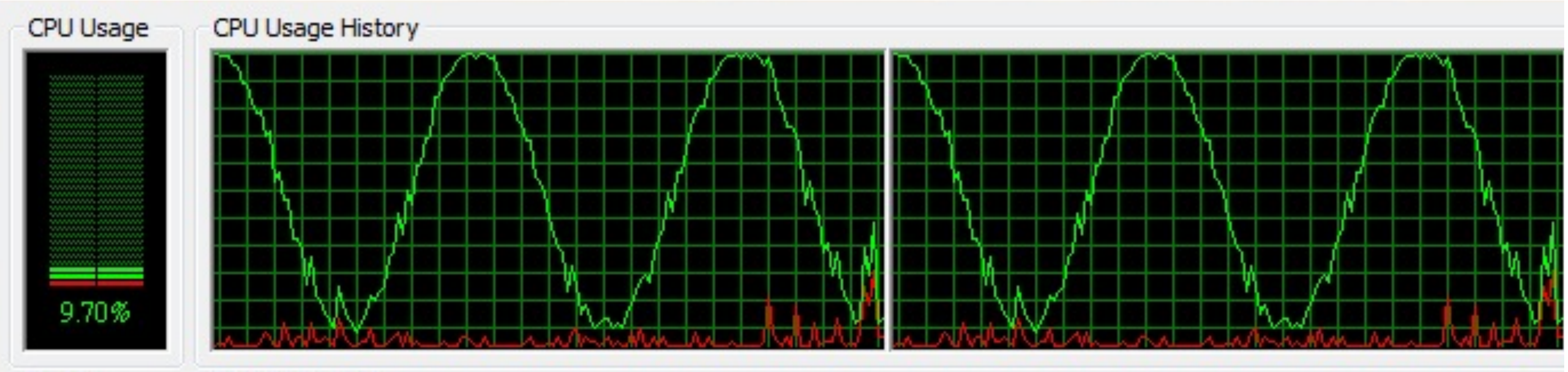
訊息傳遞

…

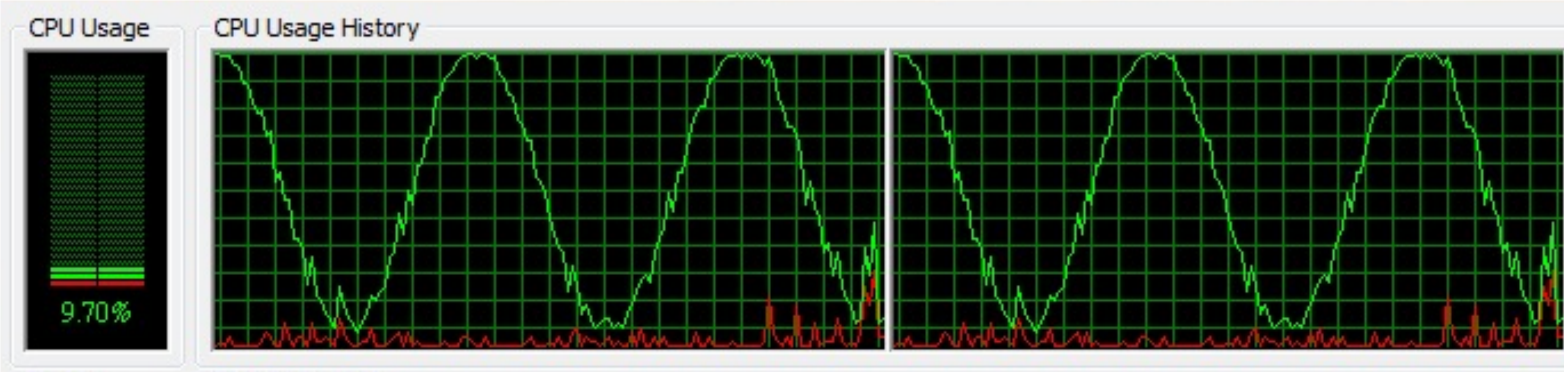
Deadlock

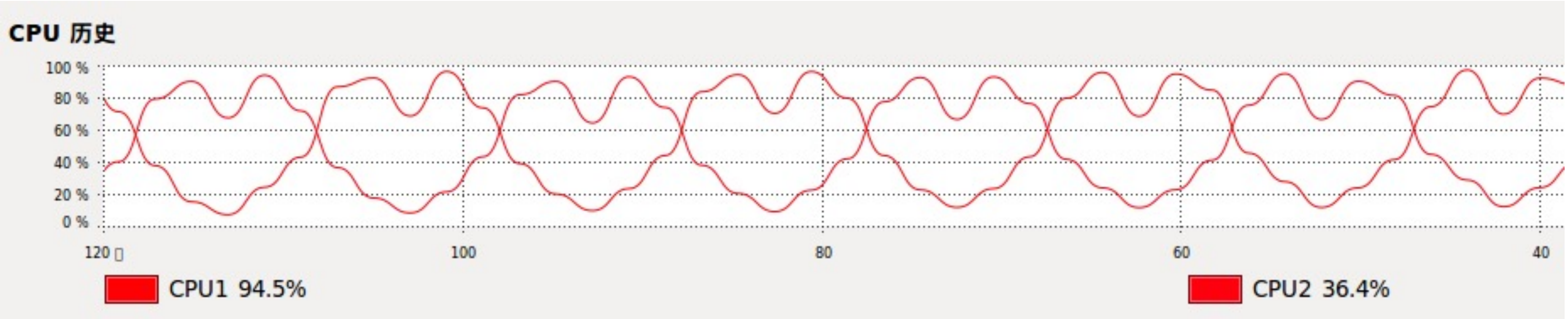
Deadlock Prevention
Deadlock Avoidance
Deadlock Detection & Recovery

補充



補充


CPU Usage
9.70%
CPU Usage History


CPU 历史
100 %
80 %
60 %
40 %
20 %
0 %
120
100
80
60
40
🟥 CPU1 94.5%
🟥 CPU2 36.4%