Embedded System (ES)

Lecturer: Dr. Cheng-Kai Lu
Phone: (02)7749-3554
Office: TD302/BAIR Lab
Email: cklu@ntnu.edu.tw

國立臺灣師範大學
NATIONAL TAIWAN NORMAL UNIVERSITY

Outline

In this lecture, we will cover:

- Review on the **important points** which were covered in the previous lecture
- Assembly Instructions
- Real-Time System & Real-Time Operating System

Recap_Calculating Baud

- Two 32 bit registers UARTIBRD and UARTFBRD
- **BRDI = integer portion, BRDF = fractional portion**
- **Baud Rate = UARTSysClk/((BRD) * ClkDiv)**
  - UARTSysClk = 16Mhz
  - ClkDiv = **16 with HSE bit = 0 (8 with HSE bit = 1)**
  - Baud Rate used = 115200
- BRDI = (int)(BRD)
- BRDF = (int)(fraction of BRD)*64 +.5)

Recap_Example BRDI and BRDF

- Set a baud rate of 9600 bps for 16Mhz SysClk, HSE = 0
- BRD = 16,000,000 / (16 * 9600) = 104.16666
- BRDI = 104
- BRDF = .1666*64+.5 = 11.16666 = 11

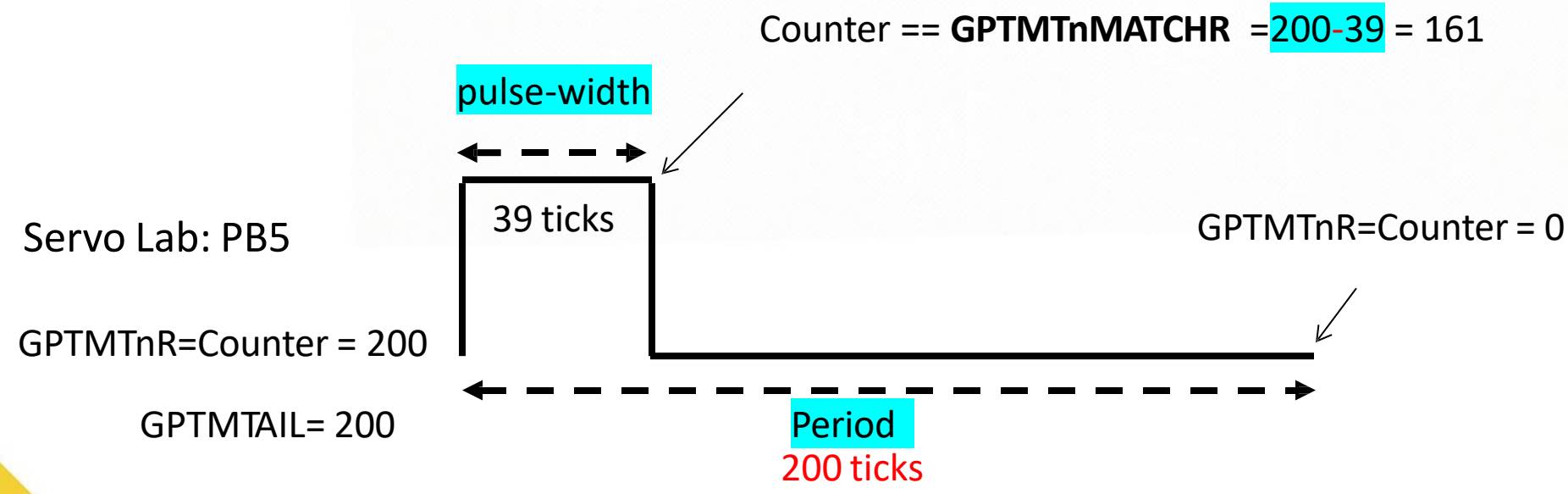
*** Baud Rate = UARTSysClk/((BRD) * ClkDiv)**

Recap_Pulse Width Modulation (PWM)

Parameters: Period and Pulse Width

Duty Cycle = Pulse width / Period

Programming: How to set the two parameters?



Recap: Major Classes of Assembly Instructions

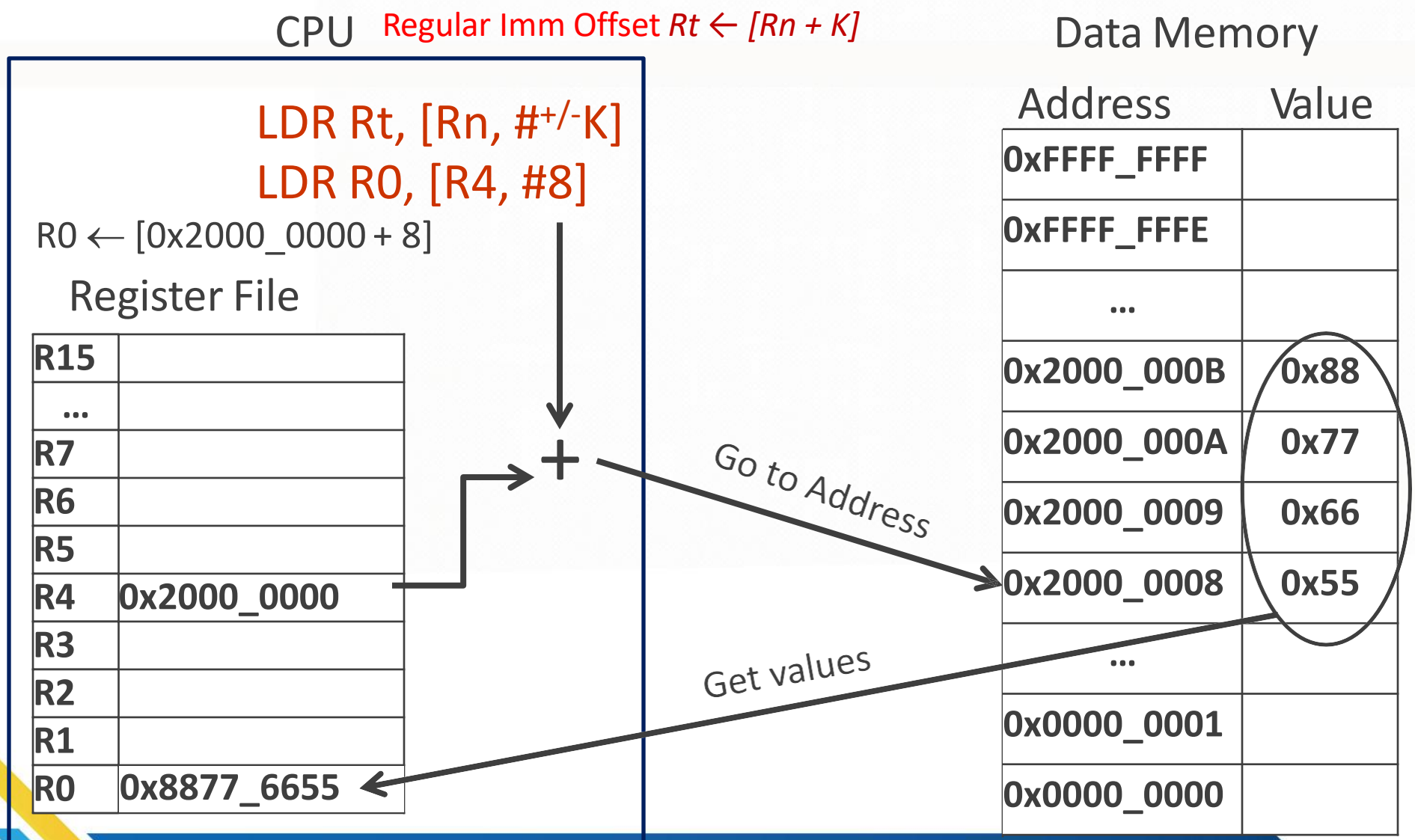
- Data Movement
  - Move data between registers
  - Move data in & out of Memory
  - Different addressing modes
- Logic & Arithmetic
  - Addition, subtraction, etc.
  - AND, OR, bit shift, etc.
- Control Flow
  - Control which sections of code should be executed (e.g. In C “IF”, “CASE”, “WHILE”, etc.
  - Function Calls

Assembly Instructions

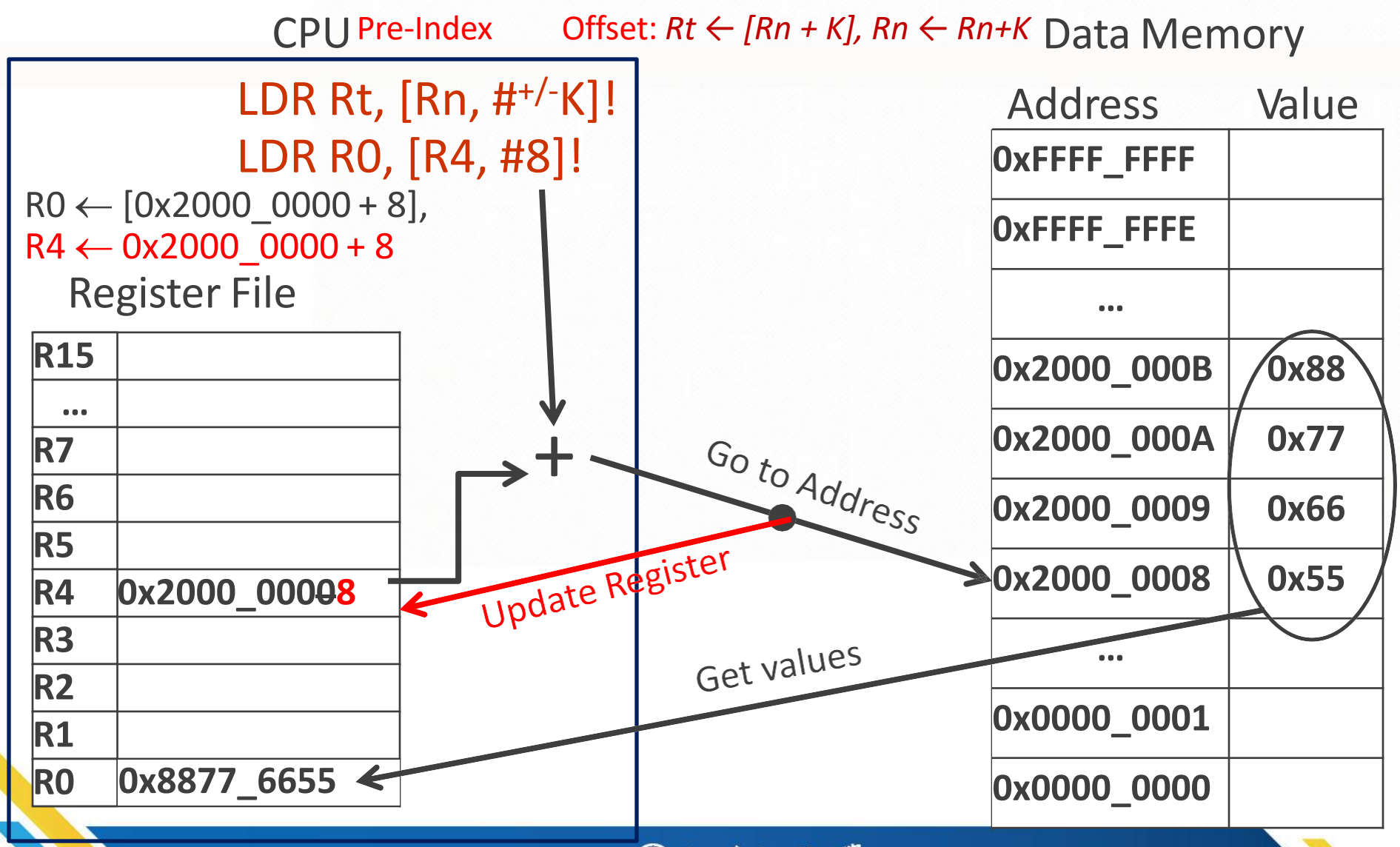
Load/Store: Addressing modes

- Immediate offset
  - LDR Rt, [Rn, #K] Regular Imm Offset *Rt ← [Rn + K]*
  - LDR Rt, [Rn, #K]! Pre-Index Offset: *Rt ← [Rn + K], Rn ← Rn+K*
  - LDR Rt, [Rn], #K Post-Index Offset: *Rt ← [Rn], Rn ← Rn+K*
- Register offset
  - LDR Rt, [Rn, Rm, LSL #n] *Rt ← [Rn + (Rm << n)]*
- PC-Relative
  - LDR Rt, [PC, #K] *Rt ← [PC + K]*
- PUSH/POP Addressing mode
  - Loads/Stores a list of registers to the stack
- Multiple Register Addressing mode
  - Loads/Stores a list of registers
- Exclusive Addressing mode
  - Used to guarantee a single source is accessing a memory

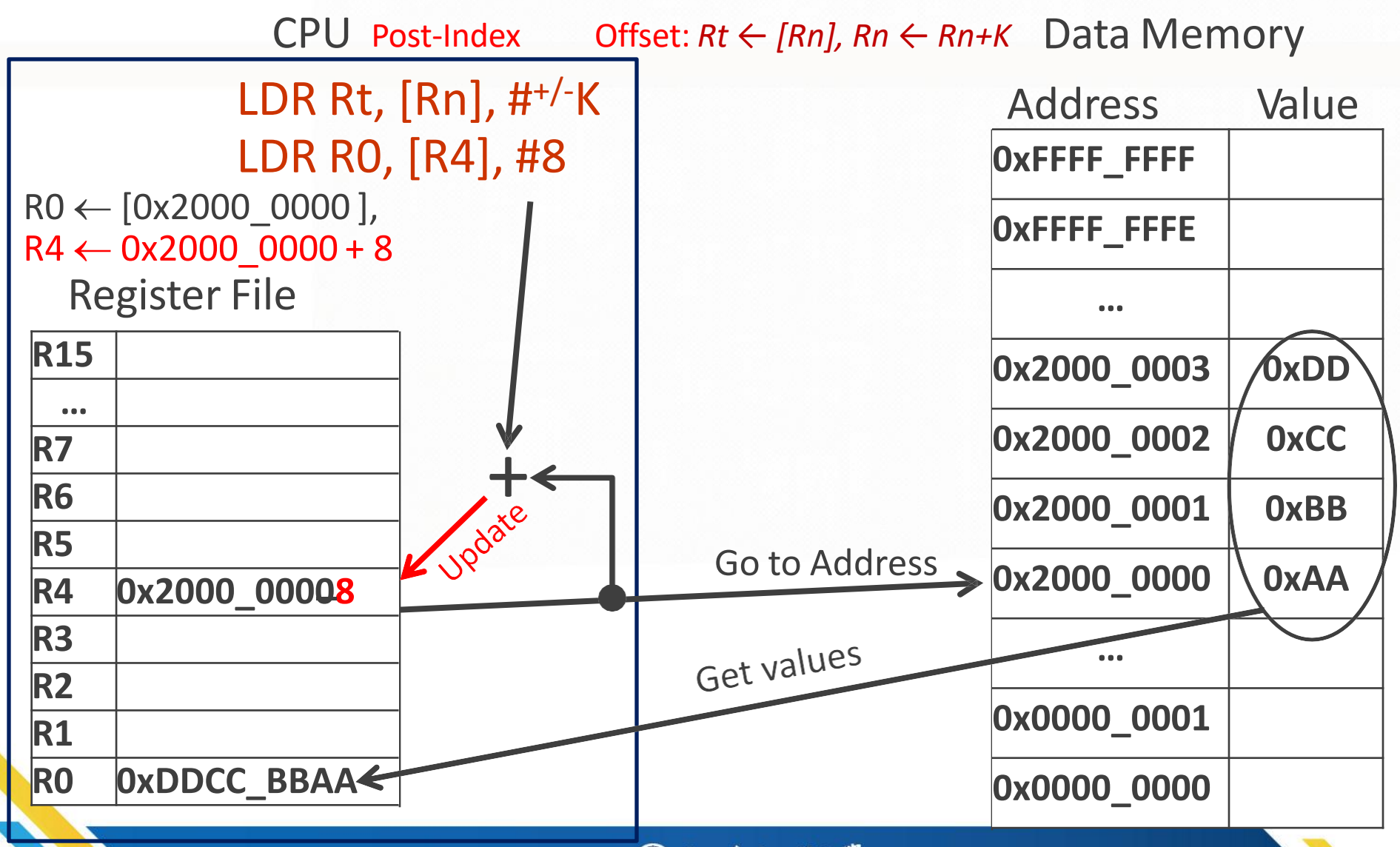
Normal Immediate Offset Addressing mode



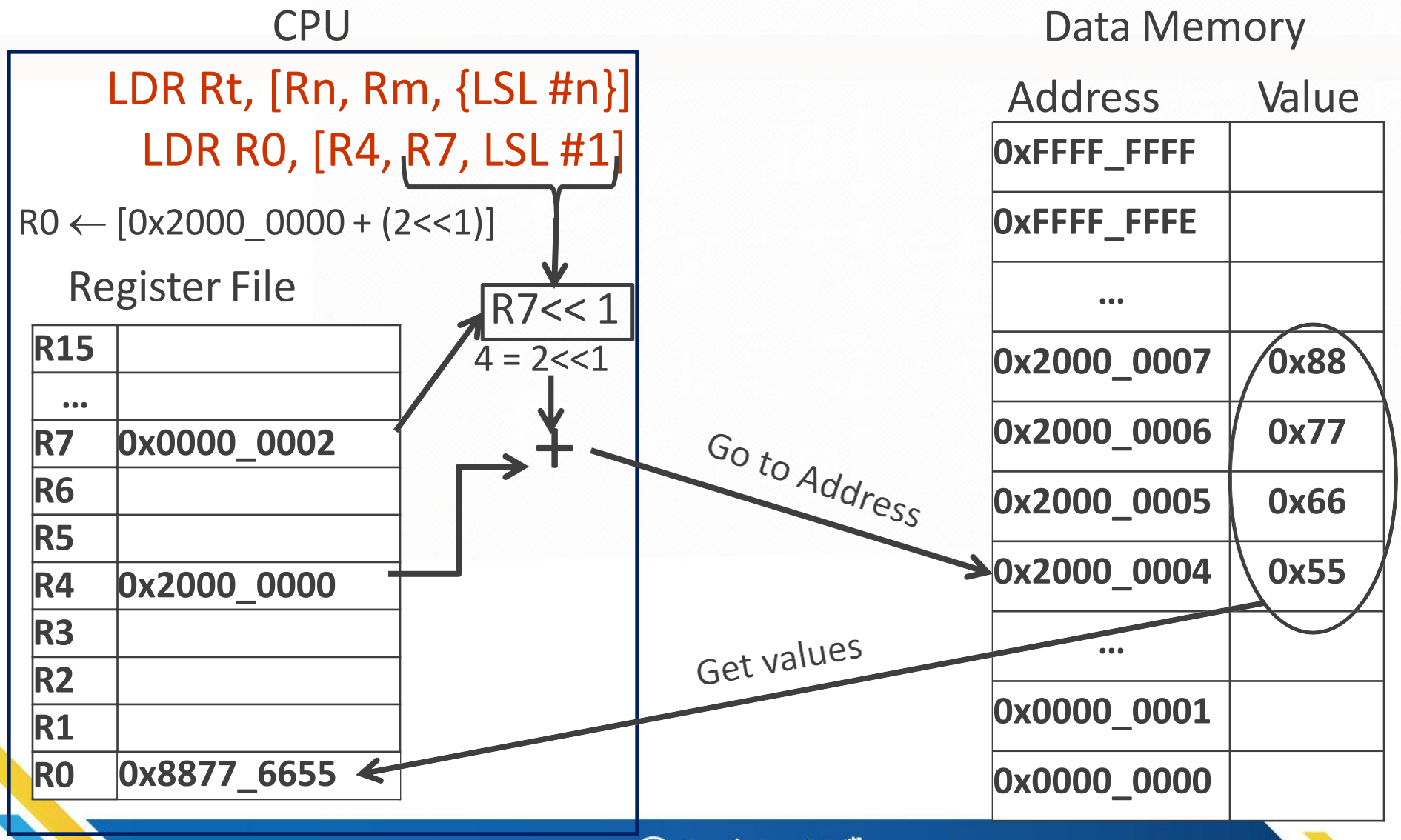
Pre-Index Immediate Offset Addressing mode



Post-Index Immediate Offset Addressing mode



Register offset addressing mode



Exercise: Array Access

```
int A[50]; // at 0x1000_A000
int B[50]; // at 0x1000_B000

B[0] = A[0];
B[1] = A[1];
```

Register File

R15	
…	
R7	
R6	
R5	
R4	
R3	
R2	
R1	
R0	

Data Memory

	…	
B[1]	0x1000_B004	
	0x1000_B003	
	0x1000_B002	
	0x1000_B001	
B[0]	0x1000_B000	
	…	
	0x1000_A007	0xDD
	0x1000_A006	0xCC
	0x1000_A005	0xBB
A[1]	0x1000_A004	0xAA
	0x1000_A003	0x55
	0x1000_A002	0x44
	0x1000_A001	0x33
A[0]	0x1000_A000	0x22

Exercise: Array Access

```
int A[50]; // at 0x1000_A000
int B[50]; // at 0x1000_B000

B[0] = A[0];
B[1] = A[1];

;Get A address
MOVW R0, 0xA000
```

Register File

R15	
…	
R7	
R6	
R5	
R4	
R3	
R2	
R1	
R0	0x0000_A000

Data Memory

	…	
B[1]	0x1000_B004	
	0x1000_B003	
	0x1000_B002	
	0x1000_B001	
B[0]	0x1000_B000	
	…	
	0x1000_A007	0xDD
	0x1000_A006	0xCC
	0x1000_A005	0xBB
A[1]	0x1000_A004	0xAA
	0x1000_A003	0x55
	0x1000_A002	0x44
	0x1000_A001	0x33
A[0]	0x1000_A000	0x22

Exercise: Array Access

```
int A[50]; // at 0x1000_A000
int B[50]; // at 0x1000_B000

B[0] = A[0];
B[1] = A[1];

;Get A address
MOVW R0, 0xA000
MOVT R0, 0x1000
```

Register File

R15	
…	
R7	
R6	
R5	
R4	
R3	
R2	
R1	
R0	0x1000_A000

Data Memory

	…	
B[1]	0x1000_B004	
	0x1000_B003	
	0x1000_B002	
	0x1000_B001	
B[0]	0x1000_B000	
	…	
	0x1000_A007	0xDD
	0x1000_A006	0xCC
	0x1000_A005	0xBB
A[1]	0x1000_A004	0xAA
	0x1000_A003	0x55
	0x1000_A002	0x44
	0x1000_A001	0x33
A[0]	0x1000_A000	0x22

Exercise: Array Access

```
int A[50]; // at 0x1000_A000
int B[50]; // at 0x1000_B000

B[0] = A[0];
B[1] = A[1];

;Get A address
MOVW R0, 0xA000
MOVT R0, 0x1000
;Get B address
MOVW R1, 0xB000
```

Register File

R15	
…	
R7	
R6	
R5	
R4	
R3	
R2	
R1	0x0000_B000
R0	0x1000_A000

Data Memory

	…	
B[1]	0x1000_B004	
	0x1000_B003	
	0x1000_B002	
	0x1000_B001	
B[0]	0x1000_B000	
	…	
	0x1000_A007	0xDD
	0x1000_A006	0xCC
	0x1000_A005	0xBB
A[1]	0x1000_A004	0xAA
	0x1000_A003	0x55
	0x1000_A002	0x44
	0x1000_A001	0x33
A[0]	0x1000_A000	0x22

Exercise: Array Access

```
int A[50]; // at 0x1000_A000
int B[50]; // at 0x1000_B000

B[0] = A[0];
B[1] = A[1];

;Get A address
MOVW R0, 0xA000
MOVT R0, 0x1000
;Get B address
MOVW R1, 0xB000
MOVT R1, 0x1000
```

Register File

R15	
…	
R7	
R6	
R5	
R4	
R3	
R2	
R1	0x1000_B000
R0	0x1000_A000

Data Memory

	…	
B[1]	0x1000_B004	
	0x1000_B003	
	0x1000_B002	
	0x1000_B001	
B[0]	0x1000_B000	
	…	
	0x1000_A007	0xDD
	0x1000_A006	0xCC
	0x1000_A005	0xBB
A[1]	0x1000_A004	0xAA
	0x1000_A003	0x55
	0x1000_A002	0x44
	0x1000_A001	0x33
A[0]	0x1000_A000	0x22

Exercise: Array Access

```
int A[50]; // at 0x1000_A000
int B[50]; // at 0x1000_B000

B[0] = A[0];
B[1] = A[1];

;Get A address
MOVW R0, 0xA000
MOVT R0, 0x1000
;Get B address
MOVW R1, 0xB000
MOVT R1, 0x1000

LDR R2, [R0, #0];Get A[0]
```

Register File

R15	
…	
R7	
R6	
R5	
R4	
R3	
R2	0x5544_3322
R1	0x1000_B000
R0	0x1000_A000

Data Memory

	…	
B[1]	0x1000_B004	
	0x1000_B003	
	0x1000_B002	
	0x1000_B001	
B[0]	0x1000_B000	
	…	
	0x1000_A007	0xDD
	0x1000_A006	0xCC
	0x1000_A005	0xBB
A[1]	0x1000_A004	0xAA
	0x1000_A003	0x55
	0x1000_A002	0x44
	0x1000_A001	0x33
A[0]	0x1000_A000	0x22

Exercise: Array Access

```
int A[50]; // at 0x1000_A000
int B[50]; // at 0x1000_B000

B[0] = A[0];
B[1] = A[1];

;Get A address
MOVW R0, 0xA000
MOVT R0, 0x1000
;Get B address
MOVW R1, 0xB000
MOVT R1, 0x1000

LDR R2, [R0, #0];Get A[0]
STR R2, [R1, #0];Set B[0]
```

Register File

R15	
…	
R7	
R6	
R5	
R4	
R3	
R2	0x5544_3322
R1	0x1000_B000
R0	0x1000_A000

Data Memory

	…	
B[1]	0x1000_B004	
	0x1000_B003	0x55
	0x1000_B002	0x44
	0x1000_B001	0x33
B[0]	0x1000_B000	0x22
	…	
	0x1000_A007	0xDD
	0x1000_A006	0xCC
	0x1000_A005	0xBB
A[1]	0x1000_A004	0xAA
	0x1000_A003	0x55
	0x1000_A002	0x44
	0x1000_A001	0x33
A[0]	0x1000_A000	0x22

Exercise: Array Access

```
int A[50]; // at 0x1000_A000
int B[50]; // at 0x1000_B000
```

```
B[0] = A[0];
B[1] = A[1];

;Get A address
MOVW R0, 0xA000
MOVT R0, 0x1000
;Get B address
MOVW R1, 0xB000
MOVT R1, 0x1000

LDR R2, [R0, #0];Get A[0]
STR R2, [R1, #0];Set B[0]
LDR R2, [R0, #4];Get A[1]
```

Register File

R15	
…	
R7	
R6	
R5	
R4	
R3	
R2	0xDDCC_BBAA
R1	0x1000_B000
R0	0x1000_A000

Data Memory

	…	
B[1]	0x1000_B004	
	0x1000_B003	0x55
	0x1000_B002	0x44
	0x1000_B001	0x33
B[0]	0x1000_B000	0x22
	…	
	0x1000_A007	0xDD
	0x1000_A006	0xCC
	0x1000_A005	0xBB
A[1]	0x1000_A004	0xAA
	0x1000_A003	0x55
	0x1000_A002	0x44
	0x1000_A001	0x33
A[0]	0x1000_A000	0x22

Exercise: Array Access

```
int A[50]; // at 0x1000_A000
int B[50]; // at 0x1000_B000
```

```
B[0] = A[0];
B[1] = A[1];

;Get A address
MOVW R0, 0xA000
MOVT R0, 0x1000
;Get B address
MOVW R1, 0xB000
MOVT R1, 0x1000

LDR R2, [R0, #0];Get A[0]
STR R2, [R1, #0];Set B[0]
LDR R2, [R0, #4];Get A[1]
STR R2, [R1, #4];Set B[1]
```

Register File

R15	
…	
R7	
R6	
R5	
R4	
R3	
R2	0xDDCC_BBAA
R1	0x1000_B000
R0	0x1000_A000

Data Memory

	…	
B[1]	0x1000_B004	0xAA
	0x1000_B003	0x55
	0x1000_B002	0x44
	0x1000_B001	0x33
B[0]	0x1000_B000	0x22
	…	
	0x1000_A007	0xDD
	0x1000_A006	0xCC
	0x1000_A005	0xBB
A[1]	0x1000_A004	0xAA
	0x1000_A003	0x55
	0x1000_A002	0x44
	0x1000_A001	0x33
A[0]	0x1000_A000	0x22

Exercise: Array Access (Better?)

```
int A[50]; // at 0x1000_A000
int B[50]; // at 0x1000_B000

B[0] = A[0];
B[1] = A[1];

;Get A address
MOVW R0, 0xA000
MOVT R0, 0x1000
;Get B address
MOVW R1, 0xB000
MOVT R1, 0x1000
```

Register File

R15	
…	
R7	
R6	
R5	
R4	
R3	
R2	
R1	0x1000_B000
R0	0x1000_A000

Data Memory

	…	
B[1]	0x1000_B004	
	0x1000_B003	
	0x1000_B002	
	0x1000_B001	
B[0]	0x1000_B000	
	…	
	0x1000_A007	0xDD
	0x1000_A006	0xCC
	0x1000_A005	0xBB
A[1]	0x1000_A004	0xAA
	0x1000_A003	0x55
	0x1000_A002	0x44
	0x1000_A001	0x33
A[0]	0x1000_A000	0x22

Exercise: Array Access (Better?)

```
int A[50]; // at 0x1000_A000
int B[50]; // at 0x1000_B000

B[0] = A[0];
B[1] = A[1];

;Get A address
MOVW R0, 0xA000
MOVT R0, 0x1000
;Get B address
MOVW R1, 0xB000
MOVT R1, 0x1000

LDR R2, [R0], #4;Get A[0]
```

Register File

R15	
…	
R7	
R6	
R5	
R4	
R3	
R2	0x5544_3322
R1	0x1000_B000
R0	0x1000_A000

Data Memory

	…	
B[1]	0x1000_B004	
	0x1000_B003	
	0x1000_B002	
	0x1000_B001	
B[0]	0x1000_B000	
	…	
	0x1000_A007	0xDD
	0x1000_A006	0xCC
	0x1000_A005	0xBB
A[1]	0x1000_A004	0xAA
	0x1000_A003	0x55
	0x1000_A002	0x44
	0x1000_A001	0x33
A[0]	0x1000_A000	0x22

Exercise: Array Access (Better?)

```
int A[50]; // at 0x1000_A000
int B[50]; // at 0x1000_B000

B[0] = A[0];
B[1] = A[1];

;Get A address
MOVW R0, 0xA000
MOVT R0, 0x1000
;Get B address
MOVW R1, 0xB000
MOVT R1, 0x1000

LDR R2, [R0], #4;Get A[0]
```

Register File

R15	
…	
R7	
R6	
R5	
R4	
R3	
R2	0x5544_3322
R1	0x1000_B000
R0	0x1000_A004

Data Memory

	…	
B[1]	0x1000_B004	
	0x1000_B003	
	0x1000_B002	
	0x1000_B001	
B[0]	0x1000_B000	
	…	
	0x1000_A007	0xDD
	0x1000_A006	0xCC
	0x1000_A005	0xBB
A[1]	0x1000_A004	0xAA
	0x1000_A003	0x55
	0x1000_A002	0x44
	0x1000_A001	0x33
A[0]	0x1000_A000	0x22

Exercise: Array Access (Better?)

```
int A[50]; // at 0x1000_A000
int B[50]; // at 0x1000_B000

B[0] = A[0];
B[1] = A[1];

;Get A address
MOVW R0, 0xA000
MOVT R0, 0x1000
;Get B address
MOVW R1, 0xB000
MOVT R1, 0x1000

LDR R2, [R0], #4;Get A[0]
STR R2, [R1], #4;Set B[0]
```

Register File

R15	
…	
R7	
R6	
R5	
R4	
R3	
R2	0x5544_3322
R1	0x1000_B000
R0	0x1000_A004

Data Memory

	…	
B[1]	0x1000_B004	
	0x1000_B003	0x55
	0x1000_B002	0x44
	0x1000_B001	0x33
B[0]	0x1000_B000	0x22
	…	
	0x1000_A007	0xDD
	0x1000_A006	0xCC
	0x1000_A005	0xBB
A[1]	0x1000_A004	0xAA
	0x1000_A003	0x55
	0x1000_A002	0x44
	0x1000_A001	0x33
A[0]	0x1000_A000	0x22

Exercise: Array Access (Better?)

```
int A[50]; // at 0x1000_A000
int B[50]; // at 0x1000_B000

B[0] = A[0];
B[1] = A[1];

;Get A address
MOVW R0, 0xA000
MOVT R0, 0x1000
;Get B address
MOVW R1, 0xB000
MOVT R1, 0x1000

LDR R2, [R0], #4;Get A[0]
STR R2, [R1], #4;Set B[0]
```

Register File

R15	
…	
R7	
R6	
R5	
R4	
R3	
R2	0x5544_3322
R1	0x1000_B004
R0	0x1000_A004

Data Memory

	…	
B[1]	0x1000_B004	
	0x1000_B003	0x55
	0x1000_B002	0x44
	0x1000_B001	0x33
B[0]	0x1000_B000	0x22
	…	
	0x1000_A007	0xDD
	0x1000_A006	0xCC
	0x1000_A005	0xBB
A[1]	0x1000_A004	0xAA
	0x1000_A003	0x55
	0x1000_A002	0x44
	0x1000_A001	0x33
A[0]	0x1000_A000	0x22

Exercise: Array Access (Better?)

```
int A[50]; // at 0x1000_A000
int B[50]; // at 0x1000_B000
```

```
B[0] = A[0];
B[1] = A[1];

;Get A address
MOVW R0, 0xA000
MOVT R0, 0x1000
;Get B address
MOVW R1, 0xB000
MOVT R1, 0x1000

LDR R2, [R0], #4;Get A[0]
STR R2, [R1], #4;Set B[0]
LDR R2, [R0], #4;Get A[1]
```

Register File

R15	
…	
R7	
R6	
R5	
R4	
R3	
R2	0xDDCC_BBAA
R1	0x1000_B004
R0	0x1000_A004

Data Memory

	…	
B[1]	0x1000_B004	
	0x1000_B003	0x55
	0x1000_B002	0x44
	0x1000_B001	0x33
B[0]	0x1000_B000	0x22
	…	
	0x1000_A007	0xDD
	0x1000_A006	0xCC
	0x1000_A005	0xBB
A[1]	0x1000_A004	0xAA
	0x1000_A003	0x55
	0x1000_A002	0x44
	0x1000_A001	0x33
A[0]	0x1000_A000	0x22

Exercise: Array Access (Better?)

```
int A[50]; // at 0x1000_A000
int B[50]; // at 0x1000_B000
```

```
B[0] = A[0];
B[1] = A[1];

;Get A address
MOVW R0, 0xA000
MOVT R0, 0x1000
;Get B address
MOVW R1, 0xB000
MOVT R1, 0x1000

LDR R2, [R0], #4;Get A[0]
STR R2, [R1], #4;Set B[0]
LDR R2, [R0], #4;Get A[1]
```

Register File

R15	
…	
R7	
R6	
R5	
R4	
R3	
R2	0xDDCC_BBAA
R1	0x1000_B004
R0	0x1000_A008

Data Memory

	…	
B[1]	0x1000_B004	
	0x1000_B003	0x55
	0x1000_B002	0x44
	0x1000_B001	0x33
B[0]	0x1000_B000	0x22
	…	
	0x1000_A007	0xDD
	0x1000_A006	0xCC
	0x1000_A005	0xBB
A[1]	0x1000_A004	0xAA
	0x1000_A003	0x55
	0x1000_A002	0x44
	0x1000_A001	0x33
A[0]	0x1000_A000	0x22

Exercise: Array Access (Better?)

```
int A[50]; // at 0x1000_A000
int B[50]; // at 0x1000_B000
```

```
B[0] = A[0];
B[1] = A[1];

;Get A address
MOVW R0, 0xA000
MOVT R0, 0x1000
;Get B address
MOVW R1, 0xB000
MOVT R1, 0x1000

LDR R2, [R0], #4;Get A[0]
STR R2, [R1], #4;Set B[0]
LDR R2, [R0], #4;Get A[1]
STR R2, [R1], #4;Set B[1]
```

Register File

R15	
…	
R7	
R6	
R5	
R4	
R3	
R2	0xDDCC_BBAA
R1	0x1000_B004
R0	0x1000_A008

Data Memory

	…	
B[1]	0x1000_B004	0xAA
	0x1000_B003	0x55
	0x1000_B002	0x44
	0x1000_B001	0x33
B[0]	0x1000_B000	0x22
	…	
	0x1000_A007	0xDD
	0x1000_A006	0xCC
	0x1000_A005	0xBB
A[1]	0x1000_A004	0xAA
	0x1000_A003	0x55
	0x1000_A002	0x44
	0x1000_A001	0x33
A[0]	0x1000_A000	0x22

Exercise: Array Access (Better?)

```
int A[50]; // at 0x1000_A000
int B[50]; // at 0x1000_B000
```

```
B[0] = A[0];
B[1] = A[1];

;Get A address
MOVW R0, 0xA000
MOVT R0, 0x1000
;Get B address
MOVW R1, 0xB000
MOVT R1, 0x1000

LDR R2, [R0], #4;Get A[0]
STR R2, [R1], #4;Set B[0]
LDR R2, [R0], #4;Get A[1]
STR R2, [R1], #4;Set B[1]
```

Register File

R15	
…	
R7	
R6	
R5	
R4	
R3	
R2	0xDDCC_BBAA
R1	0x1000_B008
R0	0x1000_A008

Data Memory

	…	
B[1]	0x1000_B004	0xAA
	0x1000_B003	0x55
	0x1000_B002	0x44
	0x1000_B001	0x33
B[0]	0x1000_B000	0x22
	…	
	0x1000_A007	0xDD
	0x1000_A006	0xCC
	0x1000_A005	0xBB
A[1]	0x1000_A004	0xAA
	0x1000_A003	0x55
	0x1000_A002	0x44
	0x1000_A001	0x33
A[0]	0x1000_A000	0x22

Usage Summary

Mode	Syntax	Base Updated?	When to Use
Regular Offset	[R2, #offset]	❌ No	Access with fixed offset, base remains unchanged
Pre-indexed	[R2, #offset]!	✅ Yes	When you want to shift pointer before access
Post-indexed	[R2], #offset	✅ Yes	When you want to shift pointer after access
Register Offset	[R2, R3]	❌ No	When offset varies and is in a register

Array Access (Is one better?)

```
int A[50]; // A starts at 0x1000_A000
int B[50]; // B Starts at 0x1000_B000

B[0] = A[0];
B[1] = A[1];
```

...

...

VS.

```
LDR R2, [R0, #0];Get A[0]
STR R2, [R1, #0];Set B[0]
LDR R2, [R0, #4];Get A[1]
STR R2, [R1, #4];Set B[1]
```

```
LDR R2, [R0], #4;Get A[0]
STR R2, [R1], #4;Set B[0]
LDR R2, [R0], #4;Get A[1]
STR R2, [R1], #4;Set B[1]
```

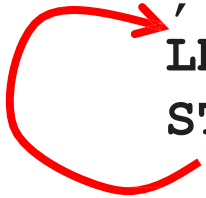
Array Access (Is one better?)

```
int A[50]; // A starts at 0x1000_A000
int B[50]; // B Starts at 0x1000_B000

for(n=0;n<50;n++)
{
  B[n] = A[n];
}
```

```
                ; Place Base Addresses in R0 and R1
                …
                …
                ; B[n] = A[n]
                LDR R2, [R0], #4 ;Get A[n]
                STR R2, [R1], #4 ;Set B[n]
```

Repeat 50 times



Conclusion

Aspect	Immediate Offset	Post-Indexed Offset
Memory Access Pattern	Explicit address with offset	Auto-increment after access
Code Simplicity	More verbose	More compact
Efficiency	Slightly less efficient	Better loop performance
Use Case	Random access (A[2], A[5]...)	Sequential access (A[0] to A[49])

Exercise(1): Pointer Access

```
int *pInt; // at 0x1000_0000
int a;     // at 0x1000_A000
```

```
a = *pInt;
```

Steps:

1. Load the contents of the pointer variable (i.e. **`pInt`**)
2. Load the contents of the dereferenced address (i.e. **`*pInt`**)
3. Store to **`a`** the contents of the dereferenced address

Register File

R15	
…	
R7	
R6	
R5	
R4	
R3	
R2	
R1	
R0	

Data Memory

	0x1000_A003	
	0x1000_A002	
	0x1000_A001	
a	0x1000_A000	
	…	
	0x1000_0003	0x00
	0x1000_0002	0x00
	0x1000_0001	0xFA
pInt	0x1000_0000	0x00
	…	
	0x0000_FA03	0x55
	0x0000_FA02	0x44
	0x0000_FA01	0x33
	0x0000_FA00	0x22

Real-Time System &
Real-Time Operating System

Real-Time System

Soft versus Hard Real Time

- Real Time: A software system with specific speed or response time requirements.
- Soft Real Time: If the deadlines are not met, performance is considered low.
- Hard Real Time: A computer system in which at least one task must meet deadlines in time. If the deadlines are not met, the system has failed.

Real-Time System

Super Hard Real Time:

- Mostly periodic tasks: Tasks run at regular, fixed intervals.
- Very strict timing: Missing a deadline is unacceptable. It can lead to system failure.

Symbol	Meaning
P	Period of the task (how often it is triggered)
T	OS timer tick (how often the scheduler checks the task queue)
C	Computation time (how long the task takes to finish)
D	Deadline (when the task must be completed)

Two conditions:
$P = T = D$, $C \ll P$.

This design gives a **huge safety margin**, critical in systems where **failure is not an option**.

Terminology in Real-time Operating System

- State
  - A unique operating condition of the system.
- Task
  - A single thread of execution through a group of related states.
- Task Manager
  - Responsible for maintaining the current state of each task.
  - Responsible for providing each task with execution time.

Real-time Operating System

- A real-time operating system (RTOS) is a multi-tasking operating system intended for real-time applications.
  - Mainly used in embedded applications.
  - Facilitates the creation of a real-time system.
  - Tool for the real-time software developer.
  - Provides a layer abstraction between the hardware and software.
  - Allows increasingly complex systems to be developed in less time.
  - Continues the evolution microcontroller system design.

Programs or Processes

A *Program* or *Process* is a unit of computation with the following attributes:

- Code/instructions
- Data
- Context/State
- Resources (memory, I/O devices, semaphores)

Note: A *Subroutine* or *Function* is a unit of computation with code, data, and context, but no managed resources.

Programs or Processes

Note: Processor Context is governed by several state **variables** possessed by the CPU of a computer. Once they (i.e., the state variables) are specified, we know the **exact state of the CPU**. These include:

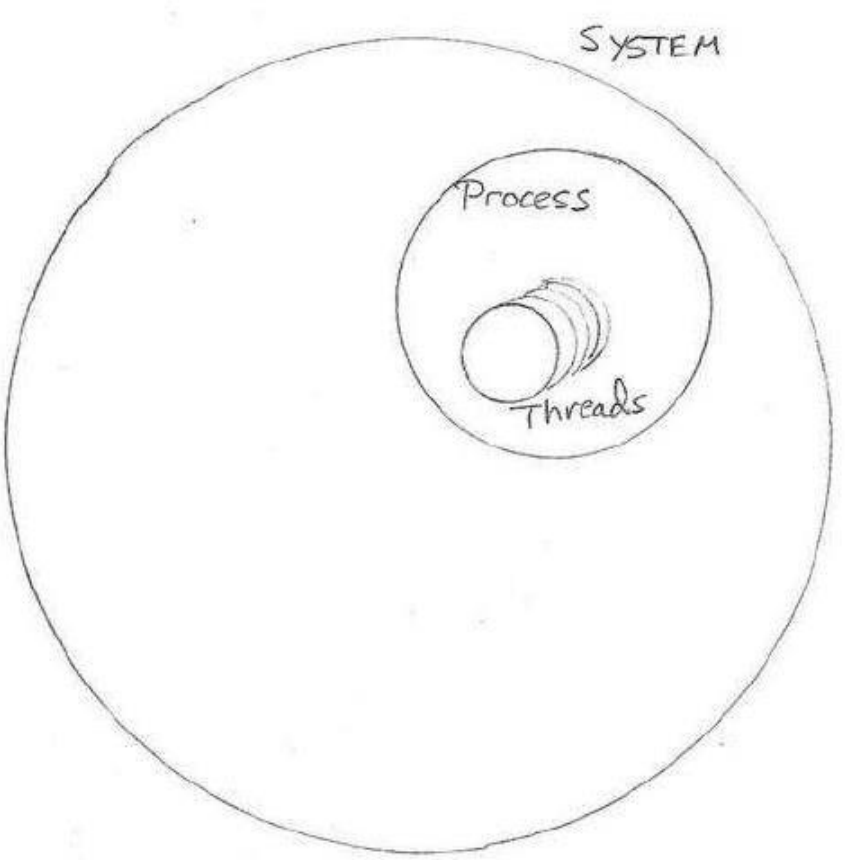
- Program Counter (location in memory of next instruction)
- Value of each register
- Processor Status Flags
- Stack Pointer

Threads or Tasks

- A *thread* is a unit of computation with code and context, but no private data. Threads may even share code with each other.
- Thread(s) is/are owned by a program/process.
- Threads are like pieces within a program which can be scheduled by the OS independently.
- A thread is usually implemented in C by writing a function.

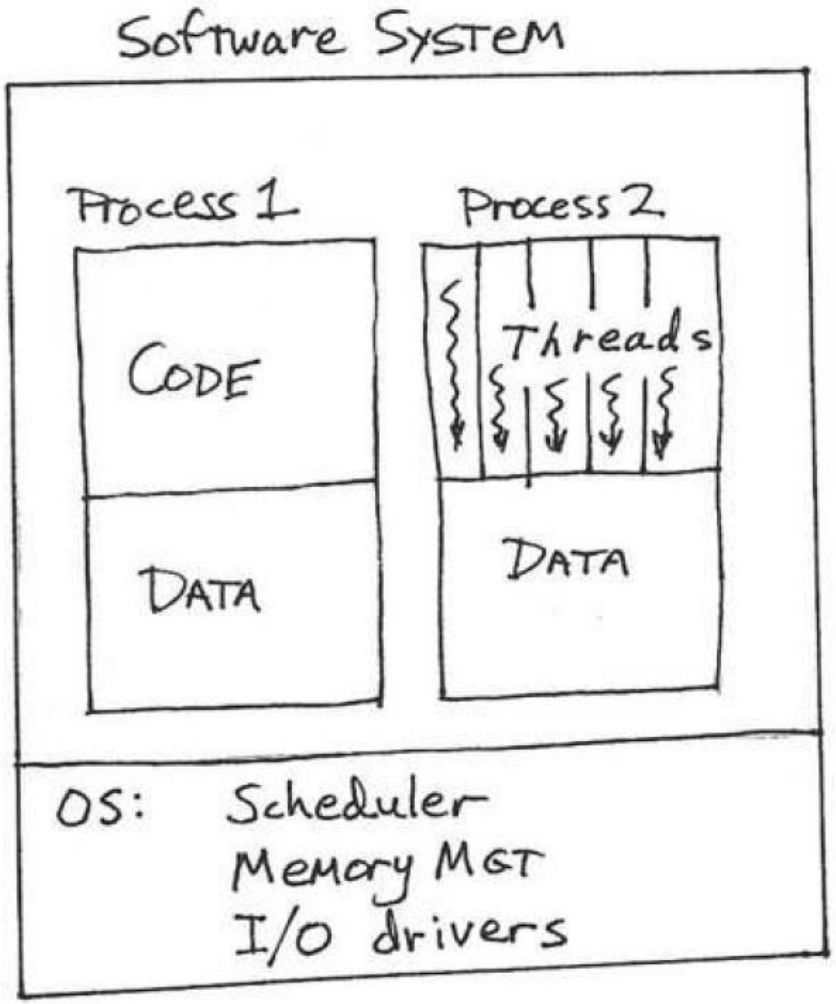
Threads or Tasks

- Thread(s) is/are owned by a program/process.
- Threads are like pieces within a program which can be scheduled by the OS independently.



Threads or Tasks

- A complete software system with two processes, namely Process 1 and Process 2.
- Process 1 is a "normal" single-threaded process. Process 2 is a multi-threaded process.
- Scheduler manages 6 units: Process 1 and the 5 threads of process 2.



Threads or Tasks

- We now know that a thread constitutes a smaller unit of computation.
- But, why use thread(s)? What are the advantages?
  - ✓ Since threads have a smaller context than programs/processes, context switching is faster
  - ✓ Only save/restore CPU state
  - ✓ No need to change memory setup

Threads or Tasks

- So far, a thread/threads has/have been defined and explained.
- What about a task/tasks?
  - ✓ Task (basic notion in RTOS) = Thread (lightweight process)
  - ✓ A task may communicate with other tasks
  - ✓ A task may use system resources such as memory blocks
- We may have timing constraints for tasks.

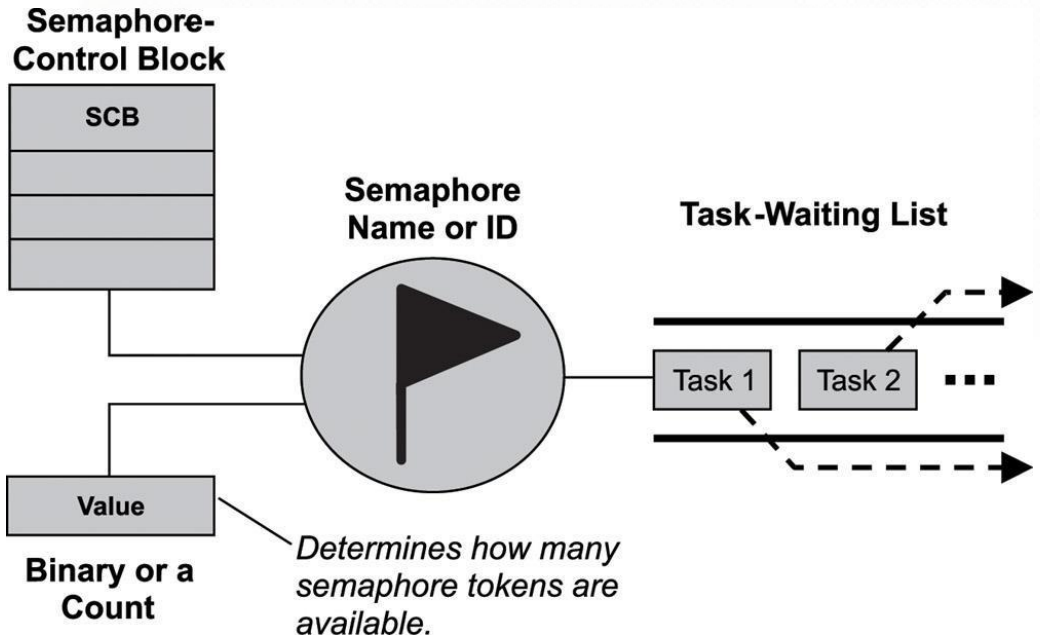
Semaphore_Introduction

- Multiple concurrent threads of execution within an application must be able to synchronize their execution and coordinate mutually exclusive access to shared resources.

- To address these requirements, RTOS kernels provide a semaphore object and associated semaphore management services.

Semaphores_Definition

- A *semaphore* (sometimes called a *semaphore token*) is a kernel object that one or more threads of execution can acquire or release for the purposes of synchronization or mutual exclusion.
- When a semaphore is first created, the kernel assigns to it an associated semaphore control block (SCB), a unique ID, a value (binary or a count), and a task-waiting list.



Semaphores Definition

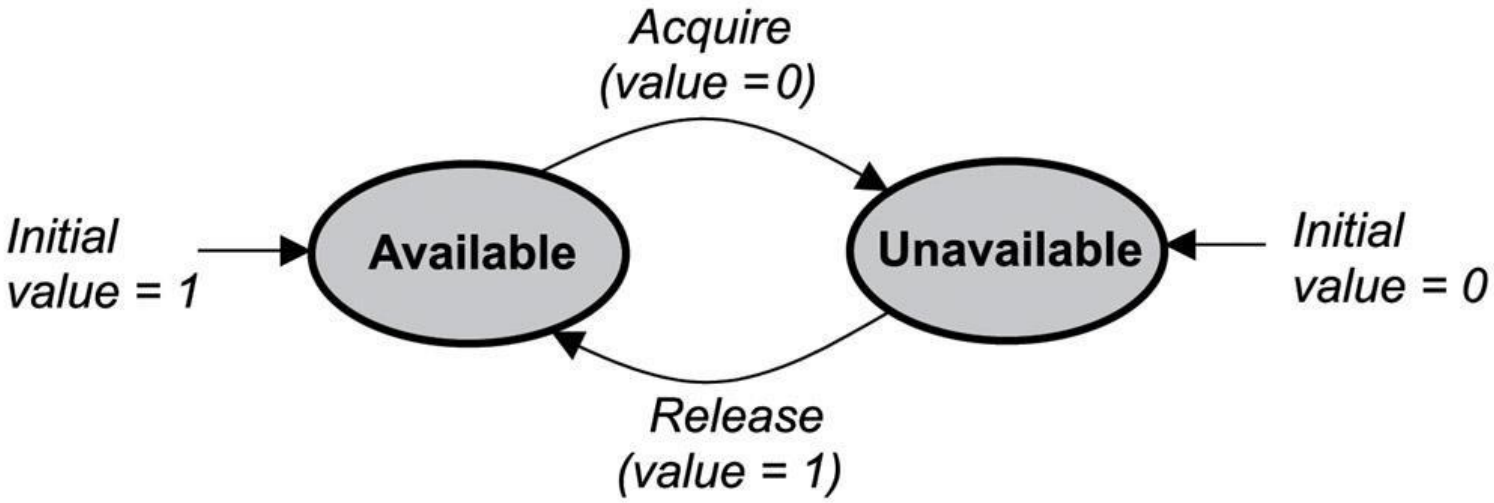
- A semaphore is like a key that allows a task to carry out some operation or to access a resource.
- The kernel tracks the number of times a semaphore has been acquired or released by maintaining a token count, which is initialized to a value when the semaphore is created.
- As a task acquires the semaphore, the token count is decremented; as a task releases the semaphore, the count is incremented.
- If the token count reaches 0, the semaphore has no tokens left.
- A requesting task, therefore, cannot acquire the semaphore, and the task blocks if it chooses to wait for the semaphore to become available.
- The task-waiting list tracks all tasks blocked while waiting on an unavailable semaphore.
- These blocked tasks are kept in the task-waiting list in either first in/first out (FIFO) order or highest priority first order.
- When an unavailable semaphore becomes available, the kernel allows the first task in the task-waiting list to acquire it.

Types of Semaphores

- A kernel can support many different types of semaphores, including
  - **binary,**
  - **counting, and**
  - **mutual-exclusion (mutex) semaphores.**

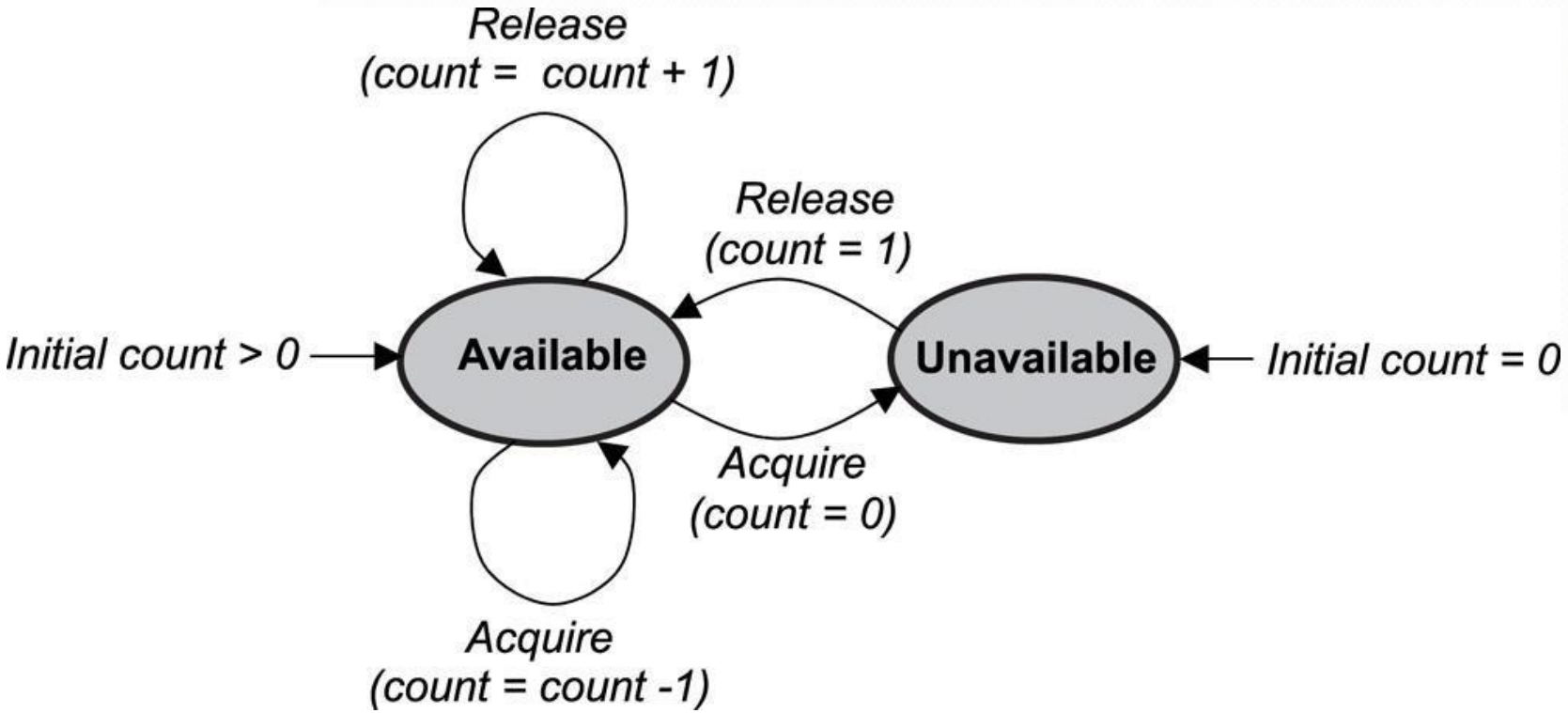
Binary Semaphores

- A *binary semaphore* can have a value of either 0 or 1.
- When a binary semaphore's value is 0, the semaphore is considered *unavailable* (or *empty)*; when the value is 1, the binary semaphore is considered *available* (or *full*).



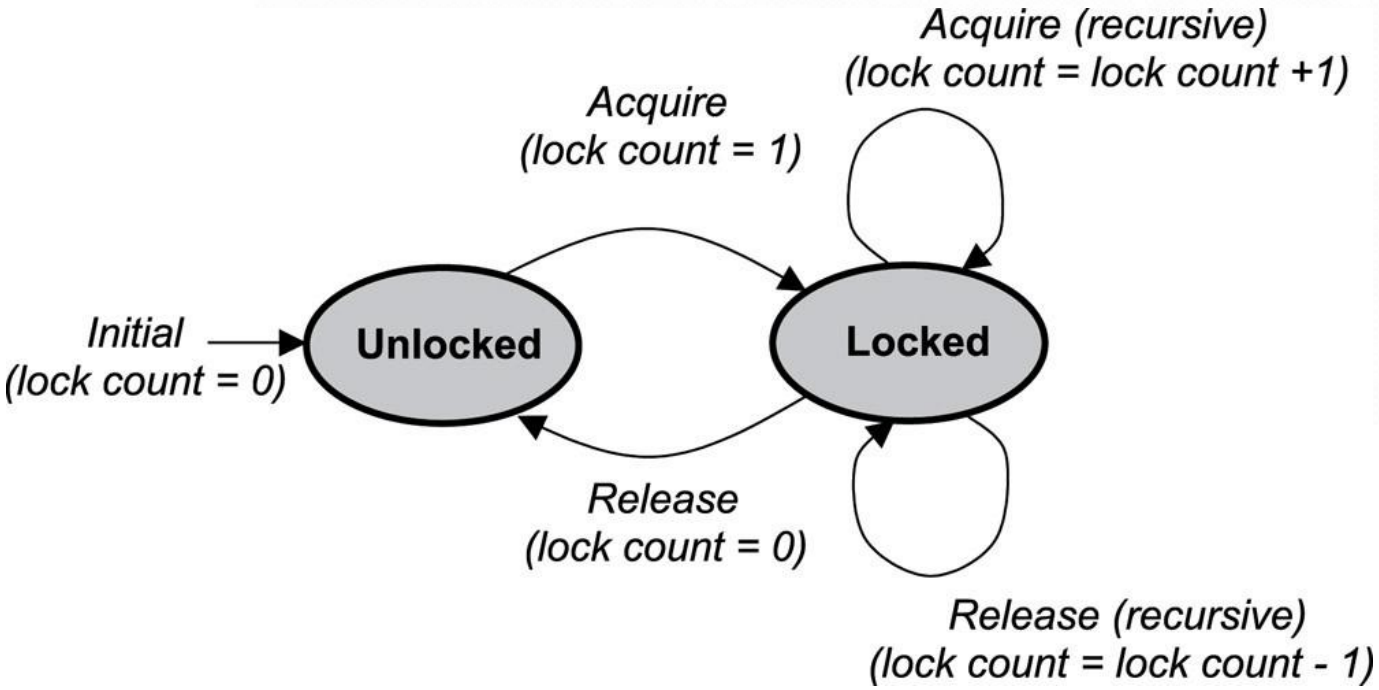
Counting Semaphores

- A *counting semaphore* uses a count to allow it to be acquired or released multiple times.
- As with binary semaphores, counting semaphores are global resources that can be shared by all tasks that need them.



Mutual Exclusion (Mutex) Semaphores

- *A mutual exclusion (mutex) semaphore* is a special binary semaphore that supports ownership, recursive access, task deletion safety, and one or more protocols for avoiding problems inherent to mutual exclusion.



Mutex Semaphores

- As opposed to the available and unavailable states in binary and counting semaphores, the states of a mutex are *unlocked* or *locked* (0 or 1, respectively).
- A mutex is initially created in the unlocked state, in which it can be acquired by a task. After being acquired, the mutex moves to the locked state.
- Conversely, when the task releases the mutex, the mutex returns to the unlocked state. Some kernels might use the terms *lock* and *unlock* for a mutex instead of *acquire* and *release*.
- Depending on the implementation, a mutex can support additional features not found in binary or counting semaphores.
- These key differentiating features include ownership, recursive locking, task deletion safety, and priority inversion avoidance protocols.

Mutex Ownership

- *Ownership* of a mutex is gained when a task first locks the mutex by acquiring it.
- Conversely, a task loses ownership of the mutex when it unlocks it by releasing it.
- When a task owns the mutex, it is not possible for any other task to lock or unlock that mutex.
- Contrast this concept with the binary semaphore, which can be released by any task, even a task that did not originally acquire the semaphore.

Recursive Locking (*recursive mutex*)

- Many mutex implementations also support *recursive locking*, which allows the task that owns the mutex to acquire it multiple times in the locked state.
- Depending on the implementation, recursion within a mutex can be automatically built into the mutex, or it might need to be enabled explicitly when the mutex is first created.
- The mutex with recursive locking is called a *recursive mutex* .
- This type of mutex is most useful when a task requiring exclusive access to a shared resource calls one or more routines that also require access to the same resource.
- A recursive mutex allows nested attempts to lock the mutex to succeed, rather than cause *deadlock* , which is a condition in which two or more tasks are blocked and are waiting on mutually locked resources.

Task Deletion Safety

- Some mutex implementations also have built-in *task deletion safety*.
- Premature task deletion is avoided by using *task deletion locks* when a task locks and unlocks a mutex.
- Enabling this capability within a mutex ensures that while a task owns the mutex, the task cannot be deleted.
- Typically protection from premature deletion is enabled by setting the appropriate initialization options when creating the mutex.

Priority Inversion Avoidance

- Priority inversion commonly happens in poorly designed real-time embedded applications.
- Priority inversion occurs when a higher priority task is blocked and is waiting for a resource being used by a lower priority task, which has itself been preempted by an unrelated medium-priority task.
- In this situation, the higher priority task's priority level has effectively been inverted to the lower priority task's level.

- Enabling certain protocols that are typically built into mutexes can help avoid priority inversion.

Priority Inversion Avoidance

- Two common protocols used for avoiding priority inversion include:
- **priority inheritance protocol**—ensures that the priority level of the lower priority task that has acquired the mutex is raised to that of the higher priority task that has requested the mutex when inversion happens. The priority of the raised task is lowered to its original value after the task releases the mutex that the higher priority task requires.
- **ceiling priority protocol**—ensures that the priority level of the task that acquires the mutex is automatically set to the highest priority of all possible tasks that might request that mutex when it is first acquired until it is released.
- When the mutex is released, the priority of the task is lowered to its original value.

Priority Inversion Avoidance

Priority Inheritance Protocol

T1: Low-priority task holds mutex M
T3: High-priority task requests M

→
T1's priority becomes equal to T3's
→ T1 runs, finishes, releases M
→ T3 continues

*Prevents medium-priority tasks from interrupting T1 and blocking T3.

Ceiling Priority Protocol

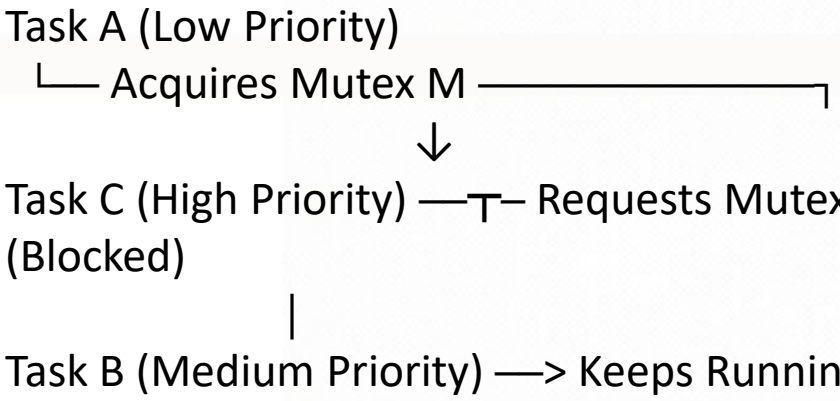
Mutex M might be used by T1, T2, and T3. Highest priority = T3.

If T1 acquires M, its priority is instantly raised to T3.

T2 and T3 cannot preempt T1 until it releases M.

*This is a proactive approach, not reactive like priority inheritance.

Exercise (2)



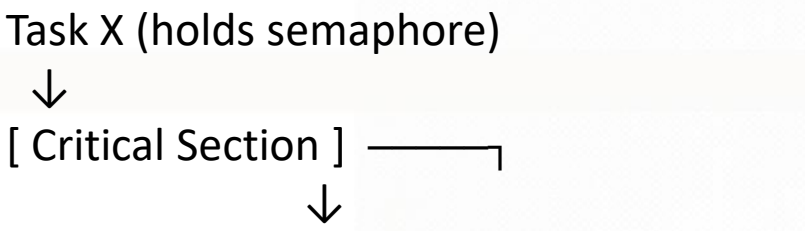
Question:

1.What problem is occurring in the above scenario?

2.Which protocol can solve it and how?

3.How would a semaphore be used differently in this case to avoid the issue?

Exercise (optional)



Question:

1.What issue arises if Task X is deleted while holding a semaphore?

2.Suggest a strategy to ensure safe task deletion related to mutual exclusion.

3.What additional protection could you implement?

Answers:

1.Issue: The semaphore remains **locked forever**, causing **deadlock** for other tasks needing it. This is a **resource leak** due to unsafe task deletion.

2.Safe Deletion Strategy:

1. **Check and release all held resources** before deletion.
2. Use a **cleanup handler** or **hook** in the RTOS to safely release semaphores during deletion.

3.Additional Protection:

1. Use **recursive mutexes** if reentrancy is needed.
2. Use a **watchdog or resource monitor** to detect stuck mutexes and auto-recover.

Typical Semaphore Operations

- Typical operations that developers might want to perform with the semaphores in an application include:
  - creating and deleting semaphores
  - acquiring and releasing semaphores
  - clearing a semaphore's task-waiting list
  - getting semaphore information.

Creating and Deleting Semaphores

Operation	Description
Create	Creates a semaphore
Delete	Deletes a semaphore

- Several things must be considered when creating and deleting semaphores.
- If a kernel supports different types of semaphores, different calls might be used for creating binary, counting, and mutex semaphores, as follows:
  - **Binary**: specify the initial semaphore state and the task-waiting order.
  - **Counting**: specify the initial semaphore count and the task-waiting order.
  - **Mutex**: specify the task-waiting order and enable task deletion safety, recursion, and priority-inversion avoidance protocols, if supported.

Acquiring and Releasing Semaphores

Operation	Description
Acquire	Acquire a semaphore token
Release	Release a semaphore token

- The operations for acquiring and releasing a semaphore might have different names, depending on the kernel: for example, *take* and *give* , *sm_p* and *sm_v* , *pend* and *post* , and *lock* and *unlock* . Regardless of the name, they all effectively acquire and release semaphores.
- Tasks typically make a request to acquire a semaphore in one of the following ways:
  - **Wait forever**—task remains blocked until it is able to acquire a semaphore.
  - **Wait with a timeout**—task remains blocked until it is able to acquire a semaphore or until a set interval of time, called the *timeout interval* , passes. At this point, the task is removed from the semaphore's task-waiting list and put in either the ready state or the running state.
  - **Do not wait (Non-blocking)**—task makes a request to acquire a semaphore token, but, if one is not available, the task does not block.

Summary

Mode	Blocks?	Timeout?	Use Case
Wait forever	Yes	No	Critical section must be accessed
Wait with timeout	Yes	Yes	Time-sensitive operations
Do not wait	No	No	Optional or best-effort tasks

Clearing Semaphore Task-Waiting Lists

Operation	Description
Flush	Unblocks all tasks waiting on a semaphore

- To clear all tasks waiting on a semaphore task-waiting list, some kernels support a *flush* operation.
- The flush operation is useful for broadcast signaling to a group of tasks.
- For example, a developer might design multiple tasks to complete certain activities first and then block while trying to acquire a common semaphore that is made unavailable.
- After the last task finishes doing what it needs to, the task can execute a semaphore flush operation on the common semaphore.
- This operation frees all tasks waiting in the semaphore's task waiting list.
- The synchronization scenario just described is also called *thread rendezvous*, when multiple tasks' executions need to meet at some point in time to synchronize execution control.

Summary of Key Points

Concept	Explanation
Flush Operation	Unblocks all tasks waiting on a semaphore at once
Use Case	Group/task-wide signaling, system-wide “go” condition
Typical Design Flow	Tasks block on unavailable semaphore → one task finishes → issues a flush → others resume
Related Concept	**Thread Rendezvous** – where multiple tasks wait until a common point in time to continue

Getting Semaphore Information

Operation	Description
Show info	Show general information about semaphore
Show blocked tasks	Get a list of IDs of tasks that are blocked on a semaphore

- At some point in the application design, developers need to obtain semaphore information to perform monitoring or debugging.
- These operations are relatively straightforward but should be used judiciously, as the semaphore information might be dynamic at the time it is requested.

Some Explanations

✅ Show Info

- Returns data such as:
  - Semaphore type (binary/counting)
  - Current value (number of tokens)
  - Whether it's locked or available

📘 Useful when:

- Verifying if a semaphore is being consumed/released properly.
- Checking whether a semaphore is stuck or never being released.

✅ Show Blocked Tasks

- Shows which tasks are **waiting** on this semaphore.
- Often includes:
  - Task names or IDs
  - Priority levels
  - Time blocked

📘 Useful when:

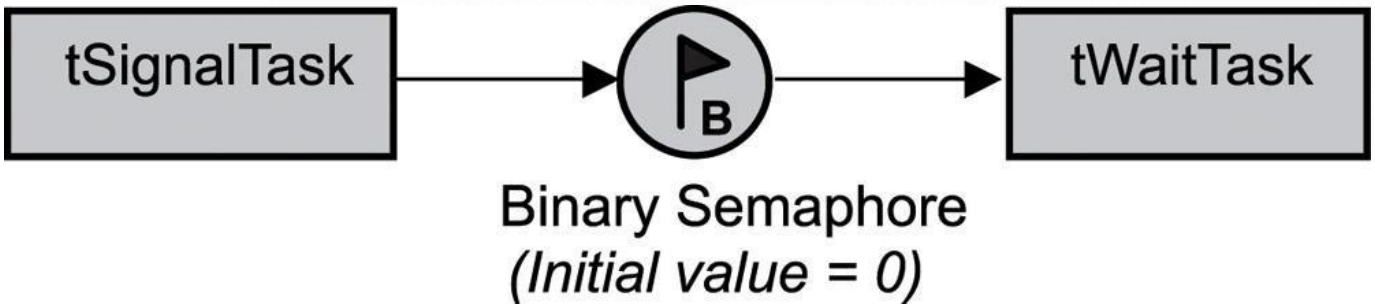
- Diagnosing **deadlocks**
- Detecting **priority inversion**
- Understanding **task dependencies**

Typical Semaphore Use

- Semaphores are useful either for synchronizing execution of multiple tasks or for coordinating access to a shared resource.
- The following examples illustrate using different types of semaphores to address common synchronization design requirements effectively, as listed:
  - wait-and-signal synchronization,
  - multiple-task wait-and-signal synchronization,
  - credit-tracking synchronization,
  - single shared-resource-access synchronization,
  - recursive shared-resource-access synchronization, and
  - multiple shared-resource-access synchronization.

Wait-and-Signal Synchronization

- Two tasks can communicate for the purpose of synchronization without exchanging data.
- For example, a binary semaphore can be used between two tasks to coordinate the transfer of execution control.



Wait-and-Signal Synchronization

Listing 6.1: Pseudo code for wait-and-signal synchronization

```
tWaitTask ( )
{
                :
                Acquire binary semaphore token
                :
}

tSignalTask ( )
{
                :
                Release binary semaphore token
                :
}
```

Because `tWaitTask`'s priority is higher than `tSignalTask`'s priority, as soon as the semaphore is released, `tWaitTask` preempts `tSignalTask` and starts to execute.

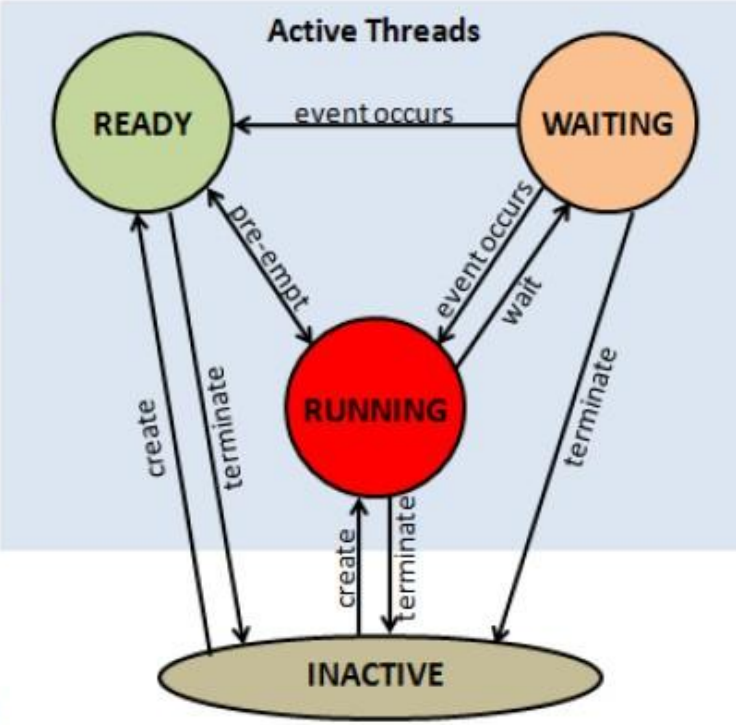
Task or Thread States

- Tasks/Threads can be in one of four states:
  1. RUNNING (or EXECUTING)
  2. READY to RUN (but not running)
  3. WAITING (for something *other* than the CPU → e.g., event); variations across different RTOS exist <SUSPENDED; BLOCKED>
  4. INACTIVE (or TERMINATED, or DORMANT, or SLEEPING)

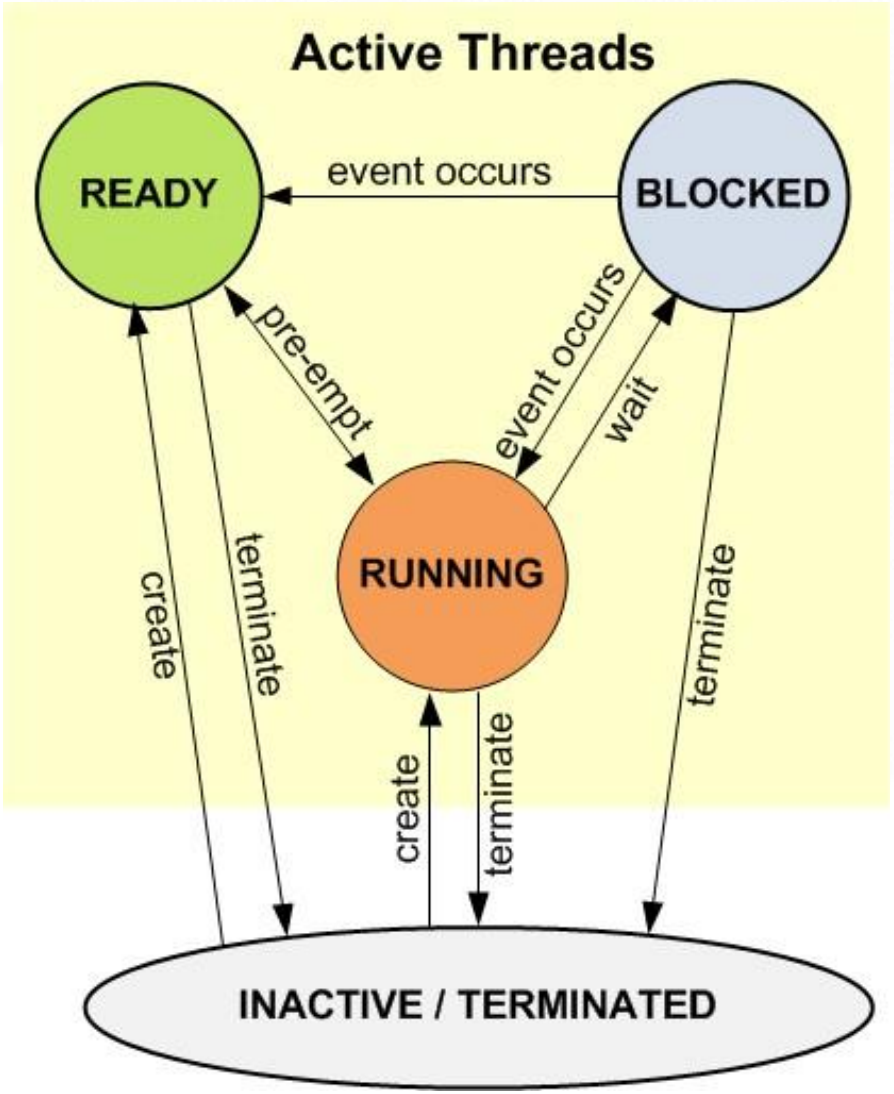
- Only one task can be RUNNING at a time (unless we are using a "multicore" CPU).
- A task which is waiting for the CPU is READY.
- When a task has requested I/O or put itself to sleep, it is WAITING.
- An INACTIVE task is waiting to be allowed into the schedule.

Thread's States

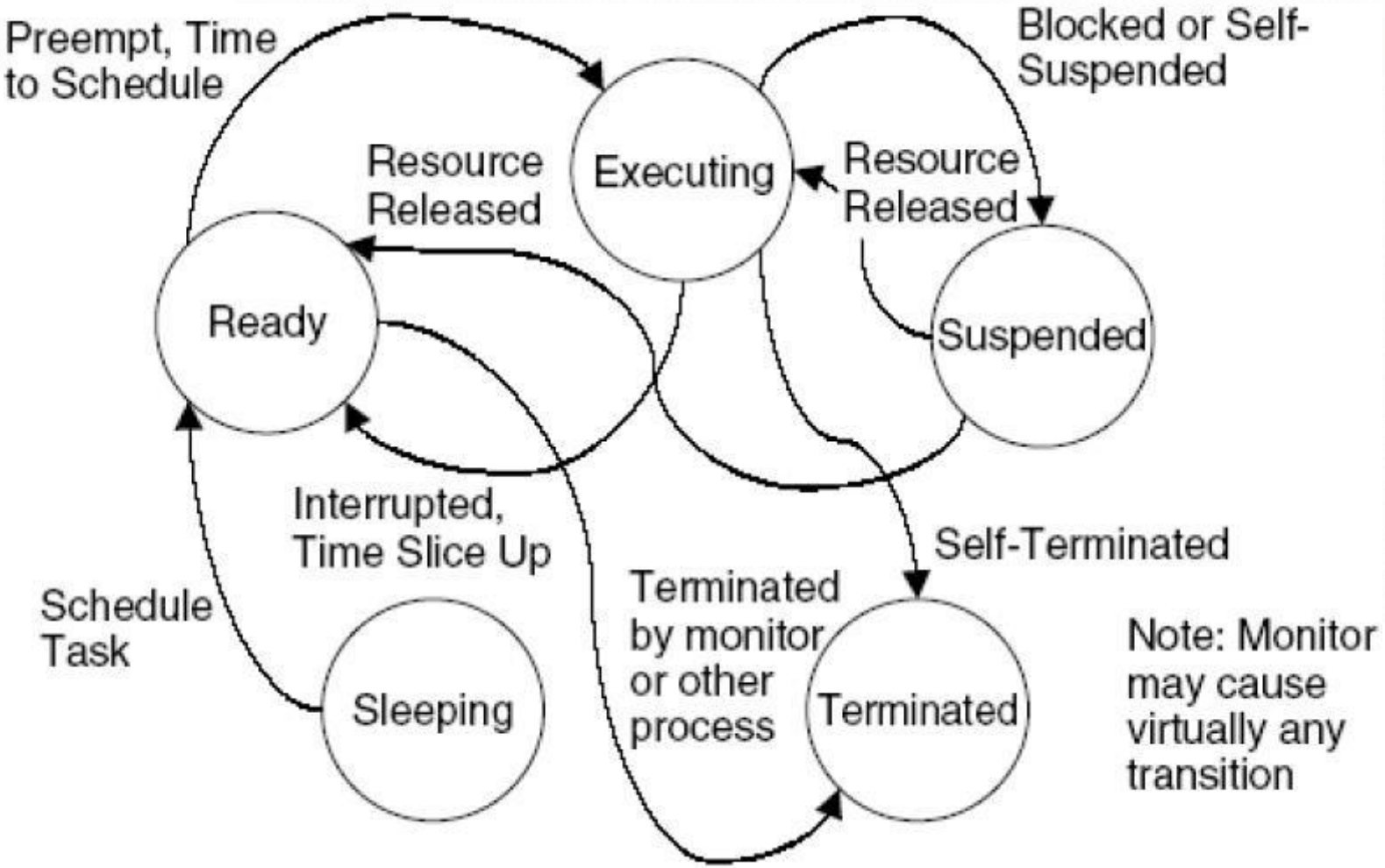
- **RUNNING:** Currently running thread is in the RUNNING state. Only one thread at a time can be in this state.
- **READY:** Threads which are ready to run are in the READY state. Once the RUNNING thread has terminated or is WAITING the next READY thread with the highest priority becomes the RUNNING thread.
- **WAITING:** Threads that are waiting for an event to occur are in the WAITING state.
- **INACTIVE:** Threads that are not created or terminated are in the INACTIVE state. These threads typically consume no system resources.



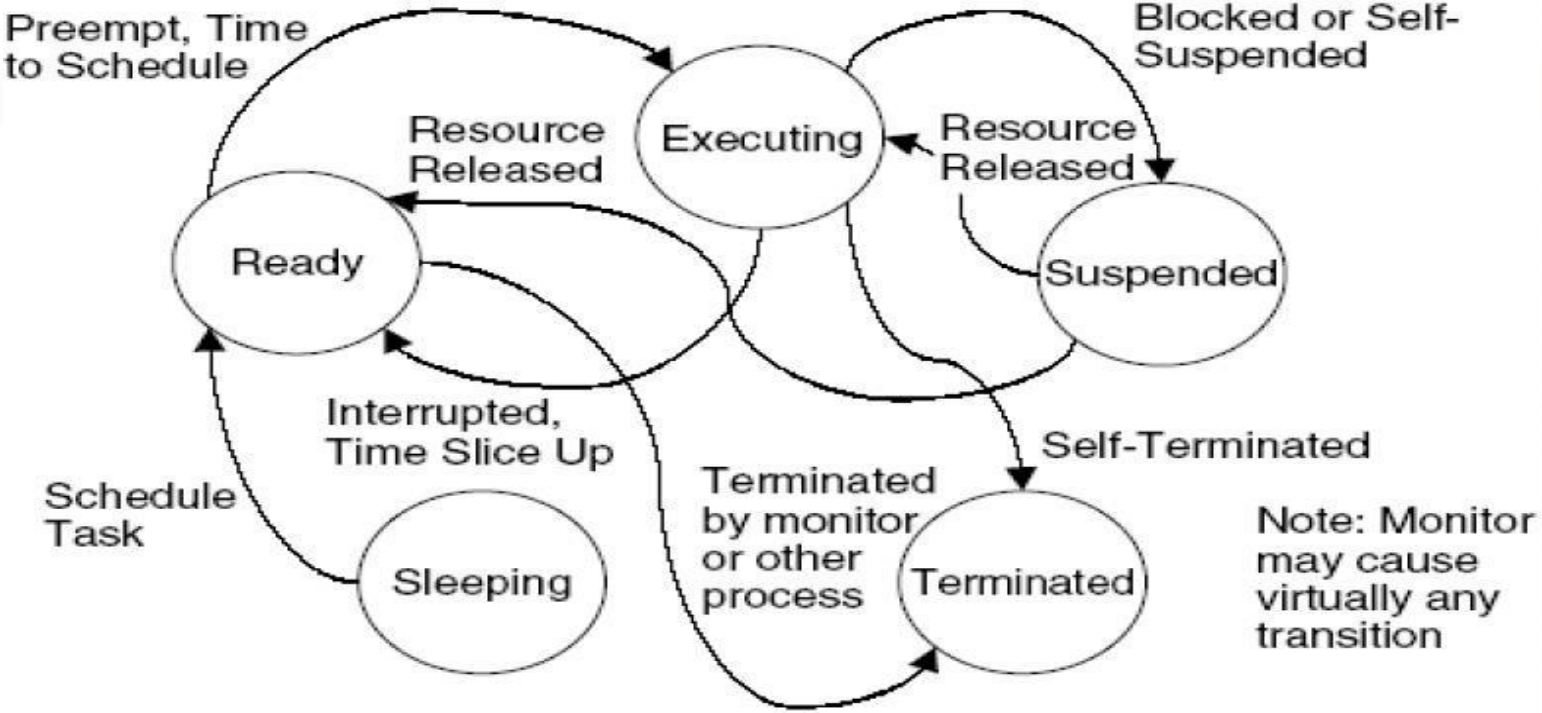
Task/Thread States: Variations



Task/Thread States: Variations

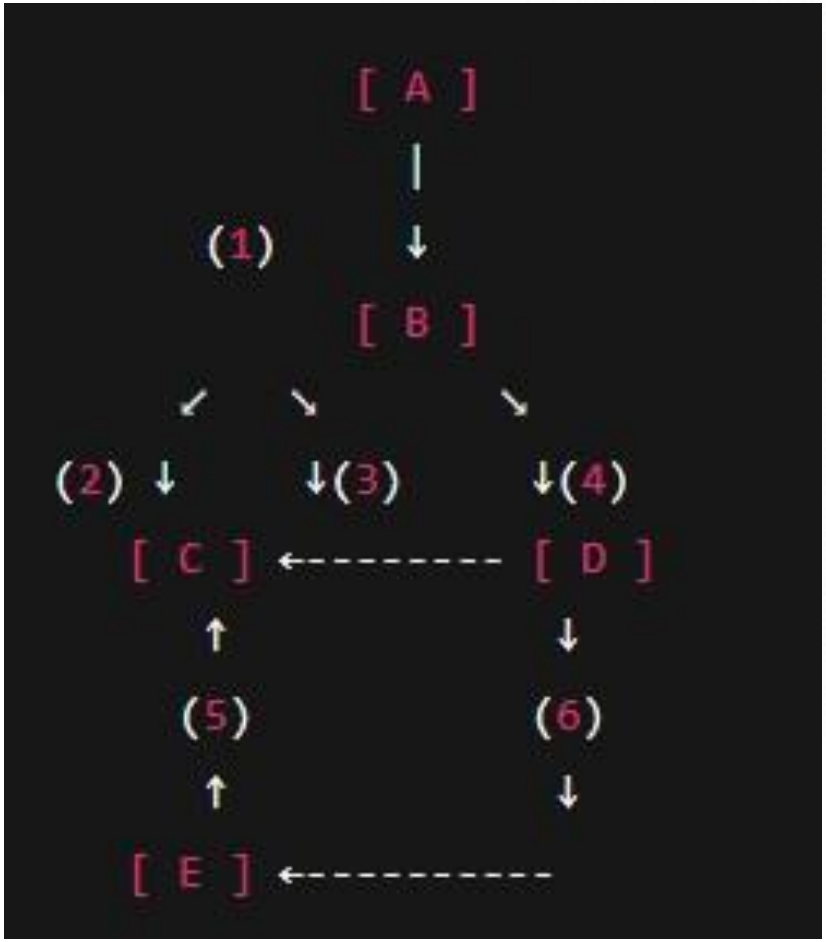


Questions



1. Which state will the task enter?
2. Once the resource becomes available again, what is the next likely state?

Exercise (3)



1. Scheduler assigns CPU <-> **Schedule Task**
2. Preemption or time slice ends <-> **Preempted or Time slice Up**
3. Task blocks (e.g., on resource) <-> **Blocked or Self-Suspended**
4. Voluntary sleep call <-> **Voluntary Sleep Call**
5. Wakes up from sleep <-> **Sleep Timeout Expires**
6. Task completes or is killed <-> **Self-Terminated or Killed**

Types of Tasks

- Periodic Tasks
- Intermittent Tasks
- Background Tasks
- Complex Tasks

Types of Tasks

Periodic Tasks

- Periodic tasks are started at **regular intervals** (arriving **at a fixed frequency**) and have to be completed before some deadline. Also called **time-driven tasks**; their activations are generated by **timers**.
- Found in Hard-Real time applications.
- Examples:

  1) Control: Every 10 ms, read sensors → compute control → output command.

  2) Multimedia: Every 22.727μsec, get music sample → compute DSP filter → output sample to DAC.

- Characterized by three attributes:

  1. *P* (Period): the regular time interval between runs of this task.
  2. *C* (Computing Resources): How much CPU time does it require each time.
  3. *D* (Deadline): How quickly must the task be completed after it is started each time tick. *C < D < P (often, D = P, but it can be D < P or D> P)*

- Obviously,

  ✓ *C <= P* (*C* may not be the same each time).

  ✓ Different code branches in the task may take different amounts of time. Use *C = Cmax*.

Types of Tasks

Intermittent Tasks

- Found in all types of applications.

- Examples:
  1) Send an email every night at 4:00 AM.
  2) Save all data when power is going down.
  3) Send a message to the plant operator when the tank runs low.
  4) Calibrate a sensor on startup.

- Characterized by two attributes:

  1. *C* (Computing Resources): How much CPU time does it require each time.
  2. *D* (Deadline): How quickly must the task be completed after it is started. (whenever that happens to be).

Types of Tasks

Background Tasks

- A soft real time or non real time task.
- Lower Priority.
- Will be accomplished only as CPU time is available when no hard real time tasks are ready.
- Characterized by:
  - *C* (Computing Resources): How much CPU time does it require each time between scheduler accesses.

Types of Tasks

Complex Tasks

- Found in all types of applications.
- Examples:

  1) Microsoft Word
  2) Apache Web Server

- Characteristics:

  1. Continuous need for CPU time.
  2. Frequent requests for I/O which free up the CPU.
  3. Waits for user input which free up CPU.

General Purpose Operating System(GPOS)

Scheduling Criteria

- **Utilization/efficiency**: keep the CPU busy **100%** of the time with **useful** work.
- **Throughput**: **maximize** the number of **jobs processed per hour**.
- **Turnaround time**: from **the time of submission** to the time of completion.
- **Waiting time**: **Sum of times** spent (in Ready queue) **waiting** to be scheduled on the CPU.
- **Response Time**: time from **submission** till the **first response** is produced (mainly for interactive jobs).
- **Fairness**: make sure each process gets a fair share of the CPU.

GPOS Exercise (Optional)

Consider the following set of processes submitted at time = 0, running under **First Come First Serve (FCFS)**:

Process	Burst Time (ms)
P1	10
P2	5
P3	8

Assume no I/O or preemption.

1. Compute the Turnaround Time for each process.
2. Compute the Waiting Time for each process.
3. Compute the Response Time for each process (FCFS = response = waiting).
4. What is the CPU Utilization if the system runs these jobs over 25 ms and the CPU is idle for 3 ms?

Ans to GPOS Exercise (Optional)

1. Compute the Turnaround Time for each process.

   Turnaround Time = Completion Time - Arrival Time

   P1: 10 - 0 = 10

   P2: 15 - 0 = 15

   P3: 23 - 0 = 23

Process	Burst Time (ms)
P1	10
P2	5
P3	8

2. Compute the Waiting Time for each process.

   Waiting Time = Turnaround Time - Burst Time

   P1: 10 - 10 = 0

   P2: 15 - 5 = 10

   P3: 23 - 8 = 15

3. Compute the Response Time for each process (FCFS = response = waiting).

   P1: 0

   P2: 10

   P3: 15

4. What is the CPU Utilization if the system runs these jobs over 25 ms and the CPU is idle for 3 ms?

   Total time = 25 msCPU

   idle time = 3 msSo,

   CPU busy time = 25 ms - 3 ms = 22 ms

   CPU Utilization=(22/25)×100%=88%

Examples of GPOS Scheduling

1. First-Come, First-Served (FCFS) Scheduling

- Serve the jobs in the order they arrive.
- Non-preemptive.
- Simple and easy to implement: When a process is ready, add it to the tail of the ready queue, and serve the ready queue in FCFS order.
- Very fair: No process is starved out, and the service order is immune to job size, etc.

Examples of GPOS Scheduling Algorithms

1. First-Come, First-Served (FCFS) (continues)

Process	Burst Time
P1	24
P2	3
P3	3

* The duration for which a process gets control of the CPU, is the Burst time for a process.

- Suppose that the processes arrive in the order: *P1* , *P2* , *P3*
  The Gantt Chart for the schedule is:

- Waiting time for *P1* = 0; *P2* = 24; *P3* = 27
- Average waiting time: (0 + 24 + 27)/3 = 17

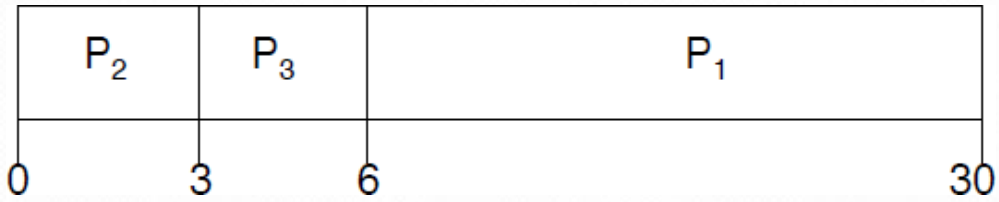
Examples of GPOS Scheduling Algorithms

1. First-Come, First-Served (FCFS) (continued)

Reducing Waiting Time

Suppose that the processes arrive in the order: *P2*, *P3*, *P1*

- The Gantt chart for the schedule is:



- Waiting time for *P1* = 6; *P2* = 0; *P3* = 3
- Average waiting time: (6 + 0 + 3)/3 = 3
- Much better than previous case
- *Convoy effect* short process behind long process

* Convoy Effect is a phenomenon associated with the First Come First Serve (FCFS) algorithm, in which the whole Operating System slows down due to a few slow processes

Examples of GPOS Scheduling Algorithms

2. Shortest-Job-First (SJF) Scheduling

- Associate with each process the length of its next CPU burst. Use these lengths to schedule the process in the shortest time.
  - SJF knows the burst time in advance.
  - Execute the tasks in ascending order of their burst time.

- Two schemes:
  - Non-preemptive: once CPU is given to the process, it cannot be preempted until it completes its CPU burst
  - Preemptive: if a new process arrives with CPU burst length less than the remaining time of the current executing process, then preempt. This scheme is known as the Shortest-Remaining-Time-First (SRTF)
- SJF is *optimal* – gives *minimum average waiting time* for a given set of processes

Examples of GPOS Scheduling Algorithms

2. Shortest-Job-First (SJF) Scheduling (continues)

Example (Non-Preemptive SJF)

SJF Process	Burst Time (in ms)	Arrival Time
P1	2	0
P2	8	0
P3	1	0
P4	4	0

Steps –

1. Check it with Arrival Time.
2. Since the Burst time for P3 (Burst = 1 sec) is the lowest, and therefore it is executed first.
3. Then Burst time for P1 is lowest in order thus it gets executed the 2nd time.
4. Do Similarly for P4 and then for P2.

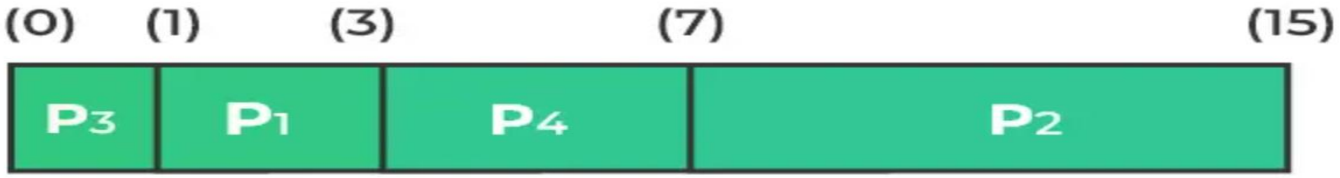
Examples of GPOS Scheduling Algorithms

2. Shortest-Job-First (SJF) Scheduling (continues)

Example (Non-Preemptive SJF)

Steps (Continuation) -

5. Waiting time and Gantt Chart.
6. Calculate the average waiting time.



P1 waiting time = 1
P2 waiting time = 7
P3 waiting time = 0
P4 waiting time = 3

The average waiting time is = (1+ 7 + 0 + 3)/ 4 = 2.75

Examples of GPOS Scheduling Algorithms

2. Shortest-Job-First (SJF) Scheduling (continues)

Example (Preemptive SJF)

Process Queue	Burst time	Arrival time
P1	6	2
P2	2	5
P3	8	1
P4	3	0
P5	4	4

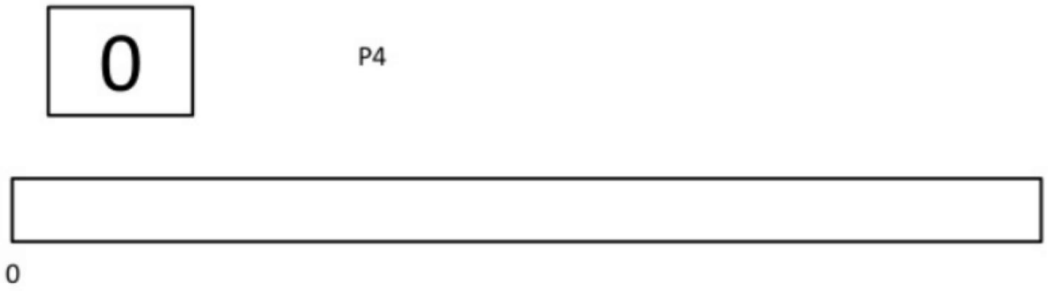
Examples of GPOS Scheduling Algorithms

2. Shortest-Job-First (SJF) Scheduling (continues)

Example (Preemptive SJF)

Step 0) At time=0, P4 arrives and starts execution.

Process Queue	Burst time	Arrival time
P1	6	2
P2	2	5
P3	8	1
P4	3	0
P5	4	4



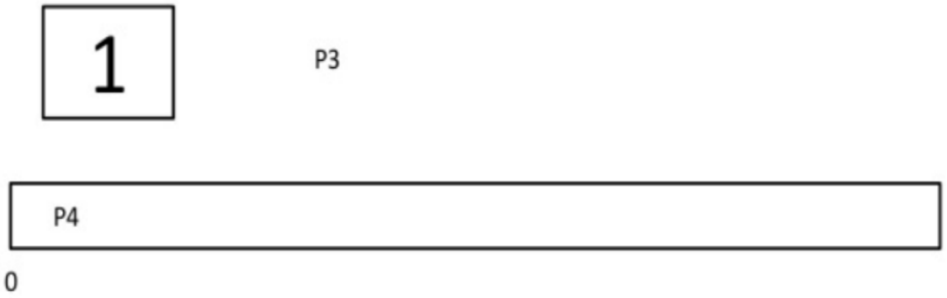
Examples of GPOS Scheduling Algorithms

2. Shortest-Job-First (SJF) Scheduling (continues)

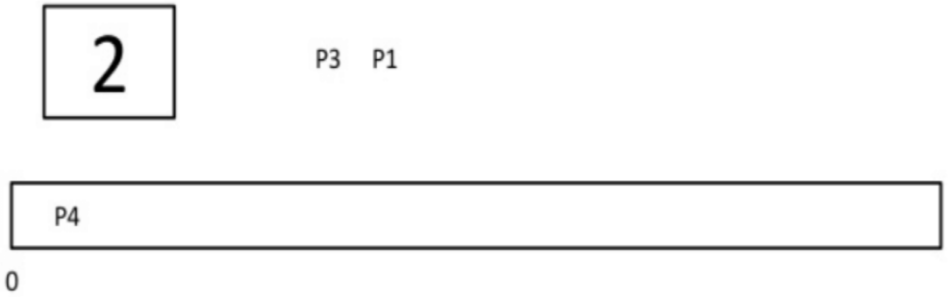
Example (Preemptive SJF)

Process Queue	Burst time	Arrival time
P1	6	2
P2	2	5
P3	8	1
P4	3	0
P5	4	4

Step 1) At time= 1, Process P3 arrives. But, P4 has a shorter burst time. It will continue execution.



Step 2) At time = 2, process P1 arrives with burst time = 6. The burst time is more than that of P4. Hence, P4 will continue execution.

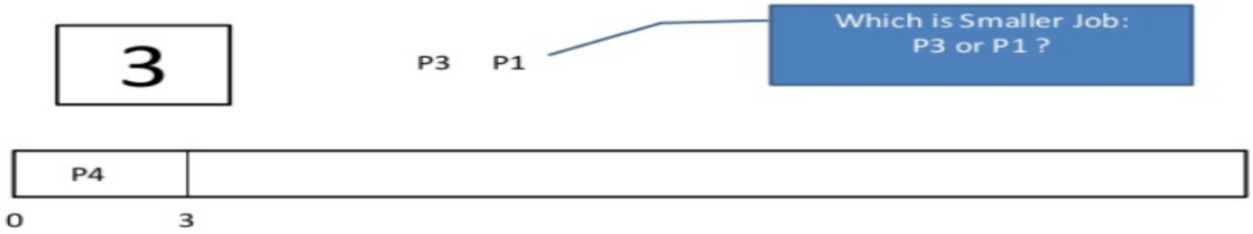


Examples of GPOS Scheduling Algorithms

2. Shortest-Job-First (SJF) Scheduling (continues)

Example (Preemptive SJF)

Step 3) At time = 3, process P4 will finish its execution. The burst time of P3 and P1 is compared. Process P1 is executed because its burst time is lower.



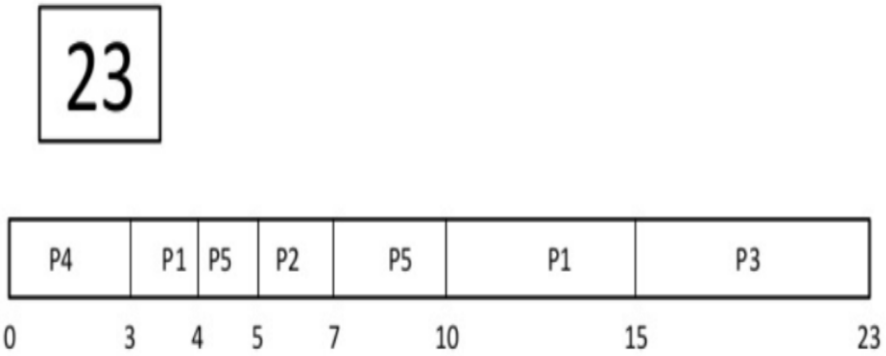
Step 4) At time = 4, process P5 will arrive. The burst time of P3, P5, and P1 is compared. Process P5 is executed because its burst time is lowest. Process P1 is preempted.

Process Queue	Burst time	Arrival time
P1	5 out of 6 is remaining	2
P2	2	5
P3	8	1
P4	3	0
P5	4	4

Examples of GPOS Scheduling Algorithms

2. Shortest-Job-First (SJF) Scheduling (continues)

Example (Preemptive SJF)



Process Queue	Burst time	Arrival time
P1	6	2
P2	2	5
P3	8	1
P4	3	0
P5	4	4

Waiting time P4= 0-0=0
P1= (3-2) + 6 =7
P2= 5-5 = 0
P5= 4-4+2 =2
P3= 15-1 = 14

Average Waiting Time = 0+7+0+2+14/5 = 23/5 =4.6

Features of SJF

1. It has a Minimum average waiting time among all scheduling algorithms.

2. The difficulty of SJF is knowing the length of the next CPU request.

3. It's used frequently in Long-term scheduling in Batch System as in this, the time limit is provided by the user specifying the process. We presume that the user provides an accurate time limit as a lower accurate value means a faster response.

4. SJF can't be implemented in Short-term scheduling as one can't know the exact length of the next CPU burst. It can be Approximated by doing an exponential average of the already measured length of the previous CPU burst. keeps the CPU until the process has finished its execution.

Exercise 4(1)

2. Shortest-Job-First (SJF) Scheduling (continues)

Non-Preemptive SJF

Process	Arrival Time	Burst Time
P1	0.0	7
P2	2.0	4
P3	4.0	1
P4	5.0	4

What the Average waiting time would be ?

Exercise 4(2)

2. Shortest-Job-First (SJF) Scheduling (continued)

(Preemptive SJF)

Process	Arrival Time	Burst Time
P1	0.0	7
P2	2.0	4
P3	4.0	1
P4	5.0	4

What the Average waiting time would be ?

Examples of GPOS Scheduling Algorithms

3. Round Robin (RR) Scheduling

- Each process gets a small unit of CPU time (***time quantum***): usually 10 to 100 milliseconds.
- After this time has elapsed, the process is preempted and added to the end of the ready queue.
- Approach:
  - If there are n processes in the ready queue and the time quantum is q, then each process gets $1/n$ of the CPU time.
  - In chunks of at most q time units at once.
  - No process waits for more than $(n\text{-}1)q$ time units.

Examples of GPOS Scheduling Algorithms

3. Round Robin (RR) Scheduling (continues)

In Round Robin Scheduling,

- CPU is assigned to the process on the basis of FCFS for a fixed amount of time.
- This fixed amount of time is called as time quantum or time slice.
- After the time quantum expires, the running process is preempted and sent to the ready queue.
- Then, the processor is assigned to the next arrived process.
- It is always preemptive in nature.

Example of RR Scheduling (1)

Question

	Arrival Time (s)	Job length (s)
P1	0	24
P2	0	3
P3	0	7

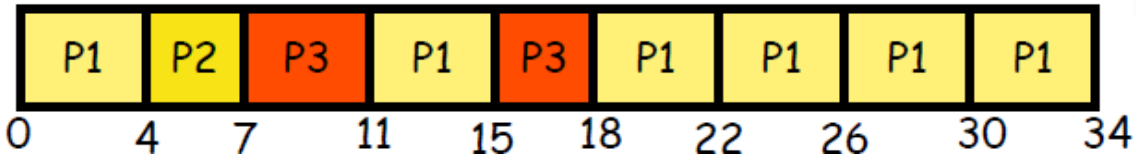
Time Quantum = 4 s

Draw a Gantt Chart

Example of RR Scheduling (2)

	Arrival Time (s)	Job length (s)
P1	0	24
P2	0	3
P3	0	7

Time Quantum = 4 s



Examples of GPOS Scheduling Algorithms

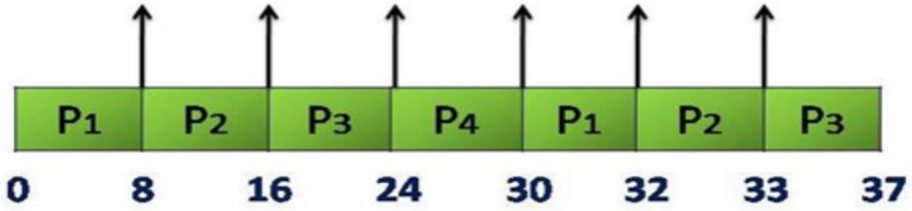
3. Round Robin (RR) Scheduling (continues)

Time quantum (TQ) = 8

Process No.	Arrival Time (AT)	Burst Time (BT)
P_1	0	10
P_2	1	9
P_3	2	12
P_4	3	6

Gantt Chart

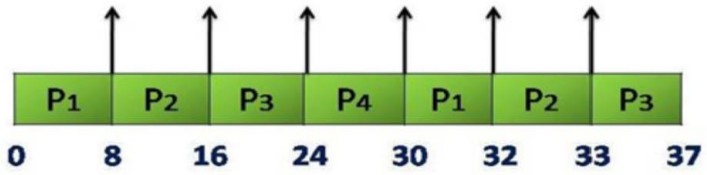
Context Switching



Examples of GPOS Scheduling Algorithms

3. Round Robin (RR) Scheduling (continues)

Context Switching



Process No.	Arrival Time (AT)	Burst Time (BT)
P1	0	10
P2	1	9
P3	2	12
P4	3	6

Number of Context Switching = **06**

Now we calculate **Turn Around Time (TAT)** and **Waiting Time (WT)** using the following formula:

TAT = CT - AT, WT = TAT – BT

And

Response Time (RT) = FR (First Response) - AR (Arrival Time)

P_No.	AT	BT	CT	TAT	WT	First Response (FR)	RT
P1	0	10	32	(32 – 0) = 32	(32 – 10) = 22	0	(0 – 0) = 0
P2	1	9	33	(33 – 1) = 32	(32 – 9) = 23	8	(8 – 1) = 7
P3	2	12	37	(37 – 2) = 35	(35 – 12) = 23	16	(16 – 2) = 14
P4	3	6	30	(30 – 3) = 27	(27 – 6) = 21	24	(24 – 3) = 21

3. Round Robin (RR) Scheduling (continues)

RR Time Quantum

- Round robin is virtually sharing the CPU between the processes giving each process the illusion that it is running in isolation (at *1/n*-th the CPU speed).
- Smaller the time quantum, the more realistic the illusion (note that when time quantum is of the order of job size, it degenerates to FCFS).
- But what is the drawback when time quantum gets smaller?

Examples of GPOS Scheduling Algorithms

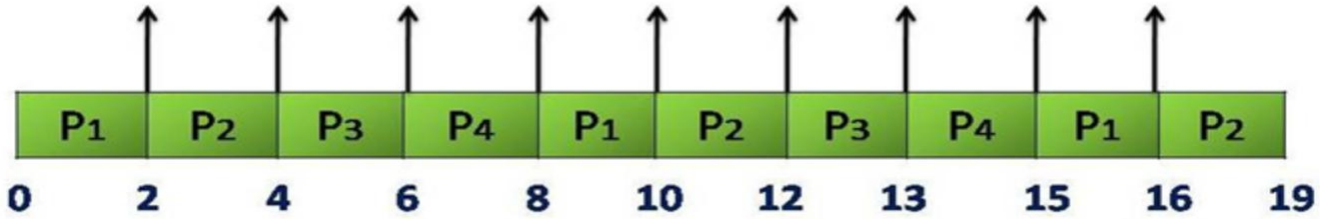
3. Round Robin (RR) Scheduling (continues)

Time quantum (TQ) = 2

Process No.	Arrival Time (AT)	Burst Time (BT)
P_1	0	5
P_2	1	7
P_3	2	3
P_4	3	4

Gantt Chart

Context Switching



Examples of GPOS Scheduling Algorithms

3. Round Robin (RR) Scheduling (continued)

RR Time Quantum

- For the considered example, if time quantum size drops to 2s from 8s, the number of context switches will increase, but response time will decreases.
- But context switches are not free! Each context switch adds CPU overhead, reducing overall CPU efficiency.
- Context switches involve
  - **Saving/restoring registers**
  - **Switching address spaces**
  - **Indirect costs (cache pollution)**

*** Cache pollution describes situations where an executing computer program loads data into CPU cache unnecessarily.**

Important Notes regarding Round Robin (RR) Scheduling

Note-01:

With decreasing value of time quantum,

- Number of context switch increases
- Response time decreases
- Chances of starvation decreases

* Starvation is the problem that occurs when high priority processes keep executing and low priority processes get blocked for indefinite time. In heavily loaded computer system, a steady stream of higher-priority processes can prevent a low-priority process from ever getting the CPU.

Thus, smaller value of time quantum is better in terms of response time.

Note-02:

With increasing value of time quantum,

- Number of context switch decreases
- Response time increases
- Chances of starvation increases

Thus, higher value of time quantum is better in terms of number of context switch.

Exercise 5(1)

In a non-preemptive priority-based system:

Process	Arrival	Burst	Priority (1 = Highest)
P1	0	10	1
P2	2	5	1
P3	4	4	2

Will P3 suffer from starvation if more priority-1 tasks keep arriving every 5 ms?

Exercise 5(2)

Two systems run Round Robin:

System A: 4 tasks, 20 ms CPU time each, quantum = 4 ms
System B: 4 tasks, 20 ms CPU time each, quantum = 10 ms
Context switch = 1 ms
Which system has more overhead from context switches? (Need statistical data to support your answer!!)

OS Scheduling Concepts

Time

- Time is measured by a piece of hardware which records actual clock time: a *real-time clock* → We use the terms "Time Quanta", "Time Slice", and "Ticks".

- Let us call the period of time ticks as *T*.

  Typically, ***T*** < ***Pmin***,

  where *Pmin* is the shortest period of **all tasks** in the system. In other words, the operating system measures time (i.e., time quanta/time slice/time ticks) in units, *T*, which are smaller than the shortest period of any task.

  Sometimes, ***T*** << ***Pmin***.

* If a system has three periodic tasks with periods 20 ms, 50 ms, and 10 ms, what is the maximum allowed tick interval *T*?

OS Scheduling Concepts

Scheduling

- At each opportunity, OS asks:

  If *N* tasks are READY, which one should I run?

- If we can show that a scheduler always can achieve deadlines, the system is "deterministically schedulable".

Pre-emptive Scheduler:
Pre-emptive Rate Monotonic Scheduling (RMS)

- **Rate-Monotonic Scheduling** (**RMS**) is a priority assignment algorithm used in (RTOS) with a static-priority scheduling class.
- The static priorities are assigned according to the cycle duration of the job, so a shorter cycle duration results in a higher job priority.
- In other words, priorities are assigned based on period, *P*.
- Short period = high priority.
- Highest priority task always gets the CPU, if it is READY.
- With RMS, we can prove optimality and schedulability.

By default, **Rate Monotonic Scheduling (RMS)** assigns higher priority to the task with the shorter period

RMS Schedulability:

C_i = computation (execution) time of task i
P_i = period of task i
n = number of tasks

Liu and Layland (1973) proved that for **a set of *n* periodic tasks** with unique periods, a feasible schedule that will always meet deadlines exists if the CPU utilization is:

$$U_{CPU\_RMS} = \sum_{i=1}^{n} \frac{C_i}{P_i} \leq U_{CPU(bound)} = n(\sqrt[n]{2} - 1)$$

Calculation of $U_{CPU(bound)}$: $U_{CPU(bound)} = n(\sqrt[n]{2} - 1)$

For n = 2: $U_{CPU(bound)} = 2(\sqrt[2]{2} - 1)$ = 0.8284
(or 0.8284 x 100% = 82.84%)

For n = 3: $U_{CPU(bound)} = 3(\sqrt[3]{2} - 1)$ = 0.77976
(or 0.77976 x 100% = 77.98%)

For n = 4: $U_{CPU(bound)} = 4(\sqrt[4]{2} - 1)$ = 0.75683
(or 0.75683 x 100% = 75.68%)

For n = 5: $U_{CPU(bound)} = 5(\sqrt[5]{2} - 1)$ = 0.74349
(or 0.74349 x 100% = 74.35%)

For n = 6: $U_{CPU(bound)} = 6(\sqrt[6]{2} - 1)$ = 0.73477
(or 0.73477 x 100% = 73.48%)

Calculation of $U_{CPU(bound)}$ for many many tasks: $U_{CPU(_{bound})} = n(\sqrt[n]{2} - 1)$

- When the number of processes tends towards infinity, this expression will tend towards:

$$\lim_{n \to \infty} n(\sqrt[n]{2} - 1) = ln2 \approx 0.693147$$

- Therefore, a rough estimate is that RMS can meet all of the deadlines if CPU utilization is less than 69.32%.
- The other 30.7% of the CPU can be dedicated to lower-priority non real-time tasks.
- It is known that a randomly generated periodic task system will meet all deadlines when the utilization is 85% or less; however, this fact depends on knowing the exact task statistics (periods, deadlines) which cannot be guaranteed for all tasks sets.

RMS → Example 1: Determine whether the system is schedulable.

Task	Execution Time	Period
A	1	8
B	2	5
C	2	10

Solution: Use $U_{CPU\_RMS} = \sigma_{i=1}^{n} \frac{C_i}{P_i}$ and compare it with $U_{CPU(bound)} = n(\sqrt[n]{2} - 1)$

$U_{CPU\_RMS} = \frac{1}{8} + \frac{2}{5} + \frac{2}{10} = 0.125 + 0.4 + 0.2 = 0.725$ → 0.725 x 100% = 72.5%

For n = 3: $U_{CPU\,(bound)} = 3(\sqrt[3]{2} - 1) = 0.77976$ → 0.77976 x 100% = 77.98%

Therefore, the system is schedulable since

$U_{CPU\_RMS} = 0.725$ (i.e., 72.5%) $\leq U_{CPU\,(bound)} = 0.77976$ (i.e., 77.98%)