COMPUTER ORGANIZATION AND OPERATING SYSTEMS BY

Ms. A Swapna
Assistant Professor

Ms. A Lakshmi
Assistant Professor

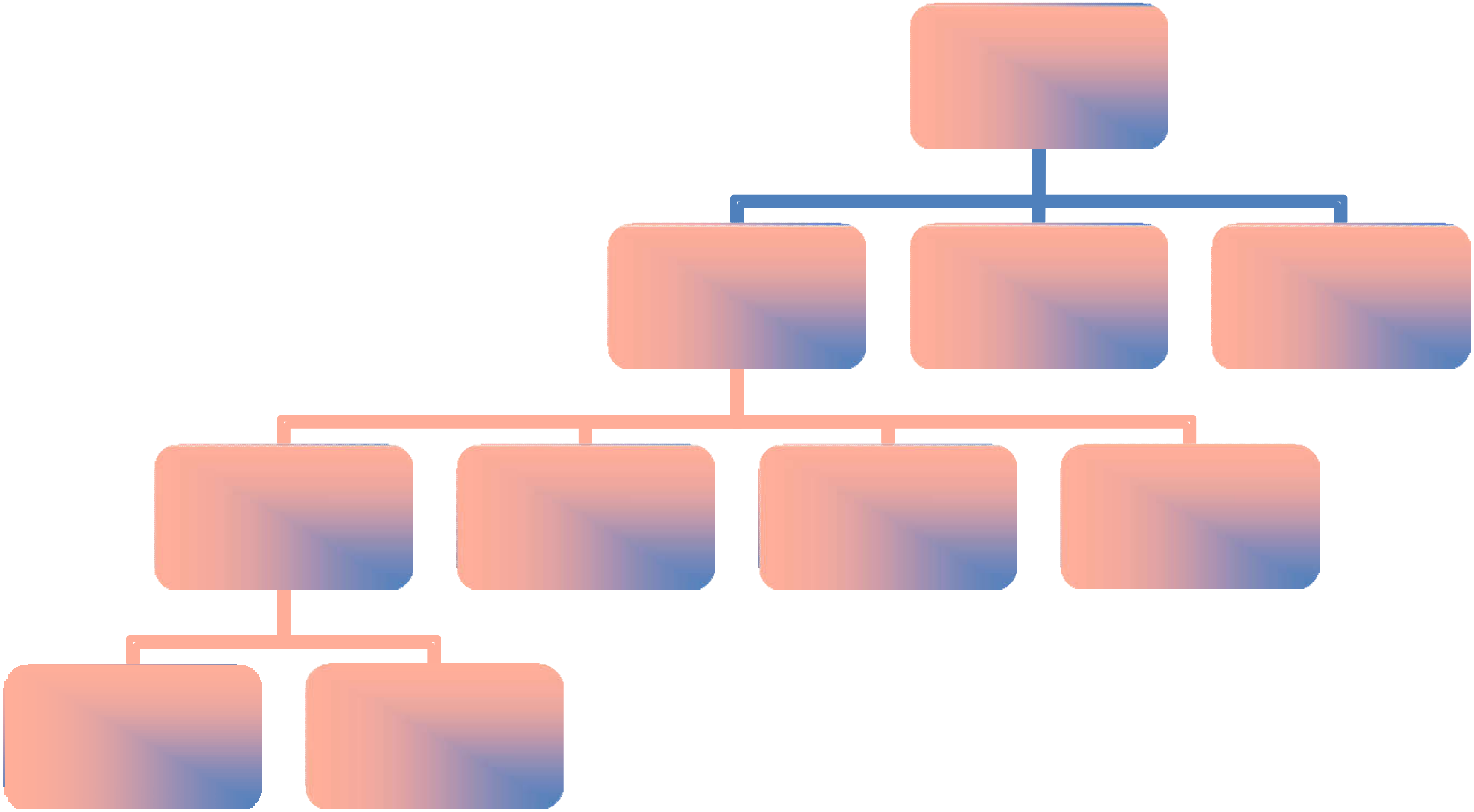
Mr.Ch.Srikanth
Assistant Professor

Mr.P.Sunil Kumar
Assistant Professor

UNIT-1

- **Computer Types**
- **Functional Units**
- **Basic Operational Concept**
- **Bus Structures**
- **Software**
- **Performance**
- **Multi processors and Multi Computers**
- **Data Representation**

Types of computers



Analog computer

Analog computer measures and answer the questions by the method of “HOW MUCH”. The input data is not a number infect a physical quantity like tem, pressure, speed, velocity.

- Signals are continuous of (0 to 10 V)
- Accuracy 1% Approximately
- High speed
- Output is continuous
- Time is wasted in transmission time

Digital Computers

Digital computer counts and answer the questions by the method of “HOW Many”. The input data is represented by a number. These are used for the logical and arithmetic operations.

- Signals are two level of (0 V or 5 V)
- Accuracy unlimited
- low speed sequential as well as parallel processing
- Output is continuous but obtain when computation is completed.

Micro Computer

Micro computer are the smallest computer system. There size range from calculator to desktop size. Its CPU is microprocessor. It also known as Grand child Computer.

- Application : - personal computer, Multi user system, offices.

Mini Computer

These are also small general purpose system. They are generally more powerful and most useful as compared to micro computer. Mini computer are also known as mid range computer or Child computer.

- Application :- Departmental systems, Network Servers, work group system.

Main Frame Computer

Mainframe computers are those computers that offer faster processing and grater storage area. The word “main frame” comes from the metal frames. It is also known as Father computer.

- Application – Host computer, Central data base server.

Super Computer

- Super computer are those computer which are designed for scientific job like whether forecasting and artificial intelligence etc. They are fastest and expensive. A super computer contains a number of CPU which operate in parallel to make it faster. It also known as grand father computer.
- Application – whether forecasting, weapons research and development.

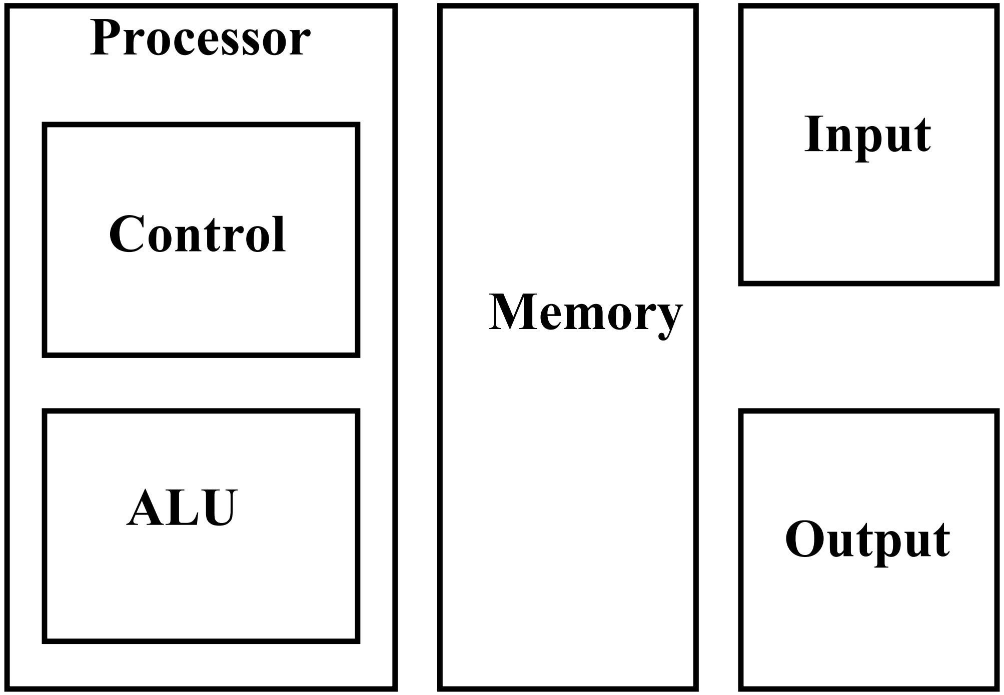
Classification of Digital computer

- Desktop
- Workstation
- Notebook
- Tablet PC
- Handheld computer
- Smart Phone

FUNCTIONAL UNITS OF COMPUTER

- Input Unit
- Output Unit
- Central processing Unit (ALU and Control Units)
- Memory
- Bus Structure

The Big Picture



Since 1946 all computers have had 5 components!!!

Function

- ALL computer functions are:
  - Data PROCESSING
  - Data STORAGE
  - Data MOVEMENT
  - CONTROL → Coordinates How Information is Used

Data = Information

- NOTHING ELSE!

INPUT UNIT:

- **Converts the external world data to a binary format, which can be understood by CPU.**

- **Eg: Keyboard, Mouse, Joystick etc**

OUTPUT UNIT:

- **Converts the binary format data to a format that a common man can understand.**

- **Eg: Monitor, Printer, LCD, LED etc**

Central Processing Unit

- The “brain” of the machine

- Responsible for carrying out computational task

- Contains ALU, CU, Registers

- ALU Performs Arithmetic and logical operations

- CU Provides control signals in accordance with some timings which in turn controls the execution process

- Register Stores data and result and speeds up the operation

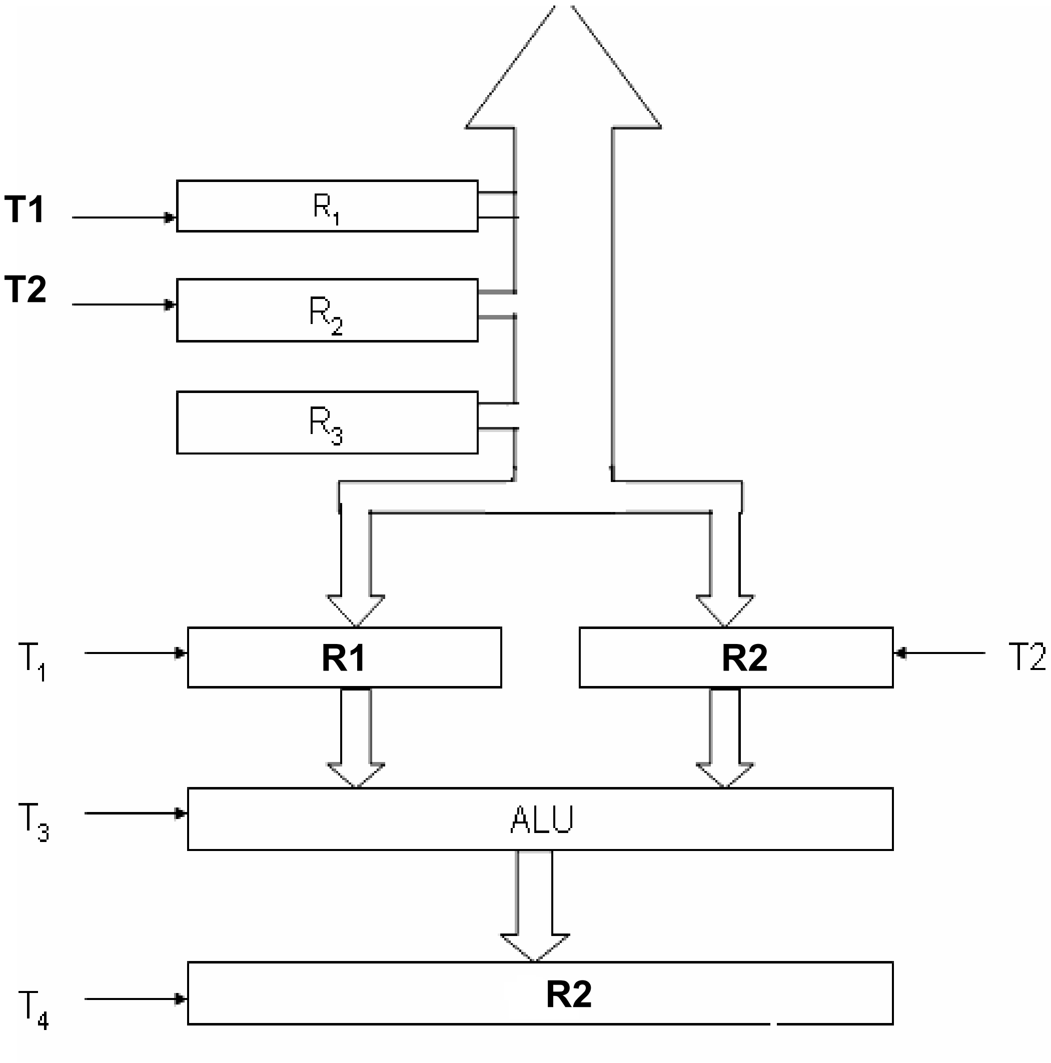
Example
Add R1, R2

T1 → **Enable R1**

T2 → **Enable R2**

T3 → **Enable ALU for addition operation**

T4 → **Enable out put of ALU to store result of the operation**



- Control unit works with a reference signal called processor clock

- Processor divides the operations into basic steps

- Each basic step is executed in one clock cycle

MEMORY

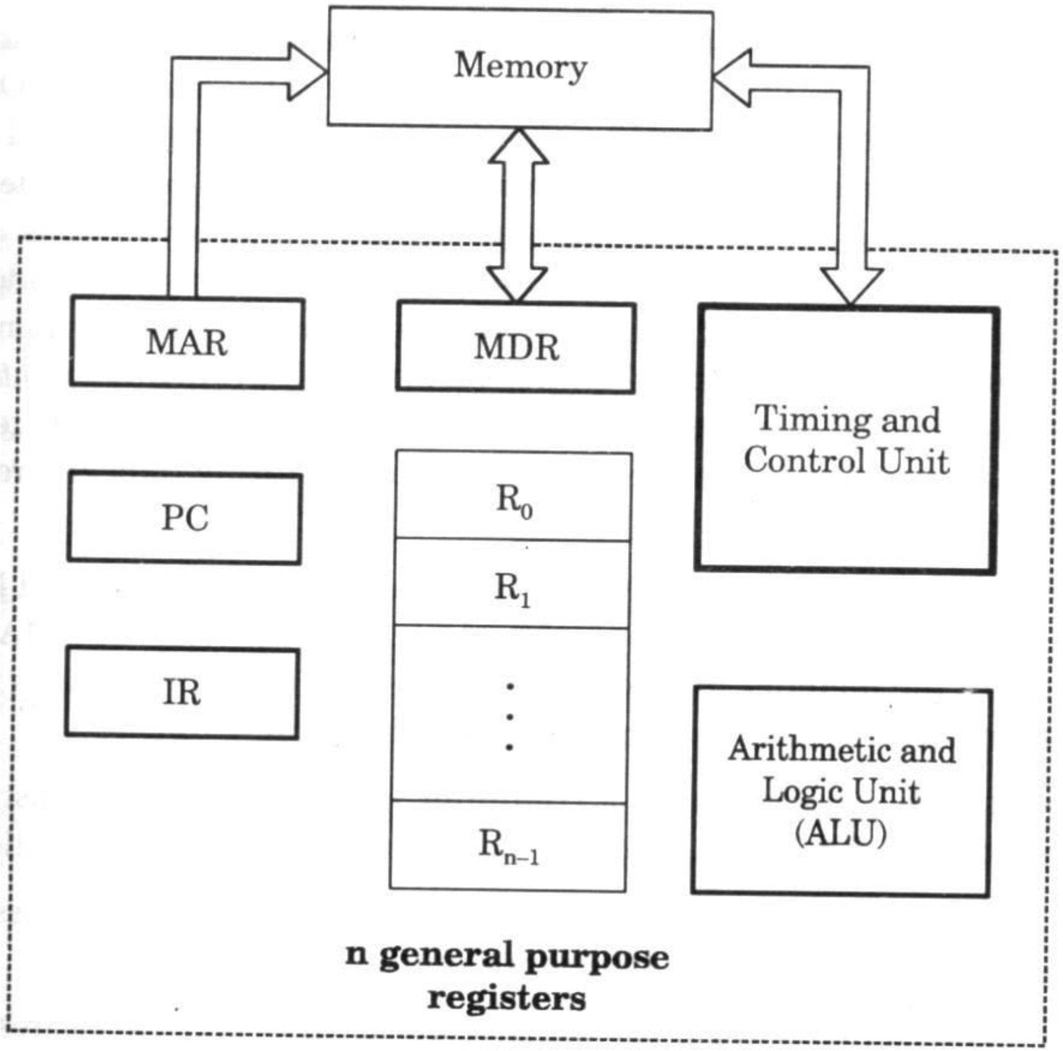
- Stores data, results, programs
- Two class of storage

  Primary (ii) Secondary
- Two types are RAM or R/W memory and ROM read only memory
- ROM is used to store data and program which is not going to change.
- Secondary storage is used for bulk storage or mass storage

Basic Operational Concepts

Basic Function of Computer

- To Execute a given task as per the appropriate program
- Program consists of list of instructions stored in memory



Interconnection between Processor and Memory

Registers

Registers are fast stand-alone storage locations that hold data temporarily. Multiple registers are needed to facilitate the operation of the CPU. Some of these registers are

❑ Two registers-MAR (Memory Address Register) and MDR (Memory Data Register) : To handle the data transfer between main memory and processor. MAR-Holds addresses, MDR-Holds data

❑ Instruction register (IR) : Hold the Instructions that is currently being executed

❑ Program counter: Points to the next instructions that is to be fetched from memory

- (PC) the contents of PC transferred to MAR)
- (MAR) (Address bus) Select a particular memory location
- Issues RD control signals
- Reads instruction present in memory and loaded into MDR
- Will be placed in IR (Contents transferred from MDR to IR)

- Instruction present in IR will be decoded by which processor understand what operation it has to perform.

- Increments the contents of PC by 1, so that it points to the next instruction address.

- If data required for operation is available in register, it performs the operation.

- If data is present in memory following sequence is performed

- Address of the data → MAR

- MAR → Address bus → select memory location where is issued RD signal

- Reads data via data bus → MDR

- From MDR data can be directly routed to ALU or it can be placed in register and then operation can be performed

- Results of the operation can be directed towards output device, memory or register

- Normal execution preempted (interrupt)

DATA REPRESENTATION

- **Data Types**
- **Complements**
- **Fixed Point Representations**
- **Floating Point Representations**
- **Other Binary Codes**
- **Error Detection Codes**

DATA REPRESENTATION

Information that a Computer is dealing with

* Data
- Numeric Data
  - Numbers(Integer, real)
- Non-numeric Data
  - Letters, Symbols

* Relationship between data elements
- Data Structures
  - Linear Lists, Trees, Rings, etc

* Program(Instruction)

NUMERIC DATA REPRESENTATION

Data

- Numeric data - numbers(integer, real)
- Non-numeric data - symbols, letters

Number System

- Nonpositional number system
  - Roman number system
- Positional number system
  - Each digit position has a value called a *weight* associated with it
  - Decimal, Octal, Hexadecimal, Binary

Base (or radix) R number

- Uses R distinct symbols for each digit
- Example $A_R = a_{n-1} a_{n-2} \dots a_1 a_0 . a_{-1} \dots a_{-m}$

Radix point(.) separates the integer portion and the fractional portion

- $V(A_R) = \displaystyle\sum_{i=-m}^{n-1} a_i R^i$

R = 10	Decimal number system,	R = 2	Binary
R = 8	Octal,	R = 16	Hexadecimal

WHY POSITIONAL NUMBER SYSTEM IN DIGITAL COMPUTERS ?

Major Consideration is the *COST* and *TIME*

- Cost of building *hardware*
  Arithmetic and Logic Unit, CPU, Communications
- Time to processing

Arithmetic - Addition of Numbers - *Table for Addition*

* Non-positional Number System
- Table for addition is infinite
  --> Impossible to build, very expensive even
  if it can be built

* Positional Number System
- Table for Addition is finite
  --> Physically realizable, but cost wise
  the smaller the table size, the less
  expensive --> Binary is favorable to Decimal

Binary Addition Table

	0	1
0	0	1
1	1	10

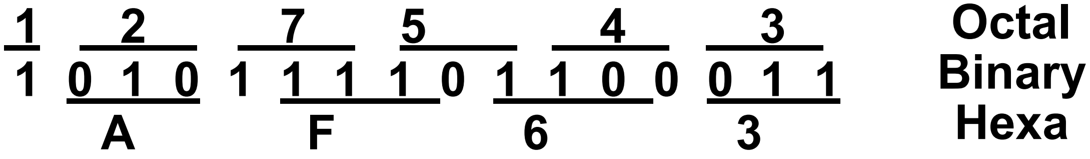
Decimal Addition Table

	0	1	2	3	4	5	6	7	8	9
0	0	1	2	3	4	5	6	7	8	9
1	1	2	3	4	5	6	7	8	9	10
2	2	3	4	5	6	7	8	9	10	11
3	3	4	5	6	7	8	9	10	11	12
4	4	5	6	7	8	9	10	11	12	13
5	5	6	7	8	9	10	11	12	13	14
6	6	7	8	9	10	11	12	13	14	15
7	7	8	9	10	11	12	13	14	15	16
8	8	9	10	11	12	13	14	15	16	17
9	9	10	11	12	13	14	15	16	17	18

REPRESENTATION OF NUMBERS - POSITIONAL NUMBERS

Decimal	Binary	Octal	Hexadecimal
00	0000	00	0
01	0001	01	1
02	0010	02	2
03	0011	03	3
04	0100	04	4
05	0101	05	5
06	0110	06	6
07	0111	07	7
08	1000	10	8
09	1001	11	9
10	1010	12	A
11	1011	13	B
12	1100	14	C
13	1101	15	D
14	1110	16	E
15	1111	17	F

Binary, octal, and hexadecimal conversion


1 2 7 5 4 3 Octal

1 0 1 0 1 1 1 1 0 1 1 0 0 0 1 1 Binary

A F 6 3 Hexa

CONVERSION OF BASES

Base R to Decimal Conversion

$$A = a_{n-1}\ a_{n-2}\ a_{n-3} \ldots a_0 . a_{-1} \ldots a_{-m}$$

$$V(A) = \Sigma\ a_k R^k$$

$$(736.4)_8 = 7 \text{ x } 8^2 + 3 \text{ x } 8^1 + 6 \text{ x } 8^0 + 4 \text{ x } 8^{-1}$$
$$= 7 \text{ x } 64 + 3 \text{ x } 8 + 6 \text{ x } 1 + 4/8 = (478.5)_{10}$$

$(110110)_2 = \ldots = (54)_{10}$

$(110.111)_2 = \ldots = (6.785)_{10}$

$(F3)_{16} = \ldots = (243)_{10}$

$(0.325)_6 = \ldots = (0.578703703 \ldots\ldots\ldots\ldots\ldots)_{10}$

Decimal to Base R number

- Separate the number into its *integer* and *fraction* parts and convert each part separately.

- Convert *integer part* into the base R number

→ successive divisions by R and accumulation of the remainders.

- Convert *fraction part* into the base R number

→ successive multiplications by R and accumulation of integer digits

EXAMPLE

Convert 41.6875_{10} to base 2.

Integer = 41

41	
20	1
10	0
5	0
2	1
1	0
0	1

$(41)_{10} = (101001)_2$

Fraction = 0.6875

$$\begin{array}{r} 0.6875 \\ \underline{\times \quad 2} \\ 1.3750 \\ \underline{\times \quad 2} \\ 0.7500 \\ \underline{\times \quad 2} \\ 1.5000 \\ \underline{\times \quad 2} \\ 1.0000 \end{array}$$

$(0.6875)_{10} = (0.1011)_2$

$(41.6875)_{10} = (101001.1011)_2$

Exercise

Convert $(63)_{10}$ to base 5: $(223)_5$

Convert $(1863)_{10}$ to base 8: $(3507)_8$

Convert $(0.63671875)_{10}$ to hexadecimal: $(0.A3)_{16}$

COMPLEMENT OF NUMBERS

Two types of complements for base R number system:

- R's complement and (R-1)'s complement

The (R-1)'s Complement

Subtract each digit of a number from (R-1)

Example

- 9's complement of 835_{10} is 164_{10}

- 1's complement of 1010_2 is 0101_2(bit by bit complement operation)

The R's Complement

Add 1 to the low-order digit of its (R-1)'s complement

Example

- 10's complement of 835_{10} is $164_{10} + 1 = 165_{10}$

- 2's complement of 1010_2 is $0101_2 + 1 = 0110_2$

FIXED POINT NUMBERS

Numbers: Fixed Point Numbers and Floating Point Numbers

Binary Fixed-Point Representation

$X = x_n x_{n-1} x_{n-2} \ldots x_1 x_0 . x_{-1} x_{-2} \ldots x_{-m}$

Sign Bit(x_n): 0 for positive - 1 for negative

Remaining Bits($x_{n-1} x_{n-2} \ldots x_1 x_0 . x_{-1} x_{-2} \ldots x_{-m}$)

SIGNED NUMBERS

Need to be able to represent both *positive* and *negative* numbers

- Following 3 representations

Signed magnitude representation
Signed 1's complement representation
Signed 2's complement representation

Example: Represent +9 and -9 in 7 bit-binary number

Only one way to represent +9 ==> 0 001001
Three different ways to represent -9:

In signed-magnitude:	**1 001001**
In signed-1's complement:	**1 110110**
In signed-2's complement:	**1 110111**

In general, in computers, fixed point numbers are represented either integer part only or fractional part only.

CHARACTERISTICS OF 3 DIFFERENT REPRESENTATIONS

Complement

Signed magnitude: Complement *only* the sign bit
Signed 1's complement: Complement *all* the bits including sign bit
Signed 2's complement: Take the 2's complement of the number, *including* its sign bit.

Maximum and Minimum Representable Numbers and Representation of Zero

$$X = x_n\, x_{n-1}\ \ldots\ x_0\,.\, x_{-1}\ \ldots\ x_{-m}$$

Signed Magnitude

Max:	$2^n - 2^{-m}$	011 ... 11.11 ... 1
Min:	$-(2^n - 2^{-m})$	111 ... 11.11 ... 1
Zero:	+0	000 ... 00.00 ... 0
	-0	100 ... 00.00 ... 0

Signed 1's Complement

Max:	$2^n - 2^{-m}$	011 ... 11.11 ... 1
Min:	$-(2^n - 2^{-m})$	100 ... 00.00 ... 0
Zero:	+0	000 ... 00.00 ... 0
	-0	111 ... 11.11 ... 1

Signed 2's Complement

Max:	$2^n - 2^{-m}$	011 ... 11.11 ... 1
Min:	-2^n	100 ... 00.00 ... 0
Zero:	0	000 ... 00.00 ... 0

2’s COMPLEMENT REPRE“ENTATION WEIGHT“

- “igned 2’s complement representation follows a “weight” scheme similar to that of unsigned numbers
  - Sign bit has negative weight
  - Other bits have regular weights

$$X = x_n\, x_{n-1}\ \ldots\ x_0$$

$$\Rightarrow\ V(X) = -\,x_n \times 2^n + \sum_{i=0}^{n-1} x_i \times 2^i$$

ARITHMETIC ADDITION: SIGNED MAGNITUDE

[1] Compare their signs

[2] If two signs are the *same* ,
ADD* the two magnitudes - Look out for an *overflow

[3] If *not the same* , compare the relative magnitudes of the numbers and
then *SUBTRACT* the smaller from the larger --> need a subtractor to add

[4] Determine the sign of the result

6 + 9

```
    6    0110
 +) 9    1001
 ______________
   15    1111 -> 01111
```

-6 + 9

```
   9    1001
- ) 6    0110
 ______________
   3    0011 -> 00011
```

6 + (- 9)

```
    9   1001
  -) 6   0110
 ______________
  - 3   0011 -> 10011
```

-6 + (-9)

```
   6   0110
 +) 9   1001
 ______________
  -15   1111 -> 11111
```

Overflow 9 + 9 or (-9) + (-9)

```
        9     1001
    +)  9     1001
 ______________
 overflow (1)0010
```

ARITHMETIC ADDITION: “IGNED 2’s COMPLEMENT

Add the two numbers, including their sign bit, and discard any carry out of leftmost (sign) bit - Look out for an *overflow*

Example

```
    6   0  0110
+)  9   0  1001
   15   0  1111
```

```
   -6   1  1010
+)  9   0  1001
    3   0  0011
```

```
    6   0  0110
+) -9   1  0111
   -3   1  1101
```

```
   -9      1  0111
+) -9      1  0111
  -18   (1)0  1110
```

$x'_{n-1}y'_{n-1}s_{n-1}$

$(c_{n-1} \oplus c_n)$

overflow

```
    9   0  1001
+)  9   0  1001
   18   1  0010
```

$x_{n-1}y_n s'_{n-1}$

$(c_{n-1} \oplus c_n)$

2 operands have the same sign and the result sign changes

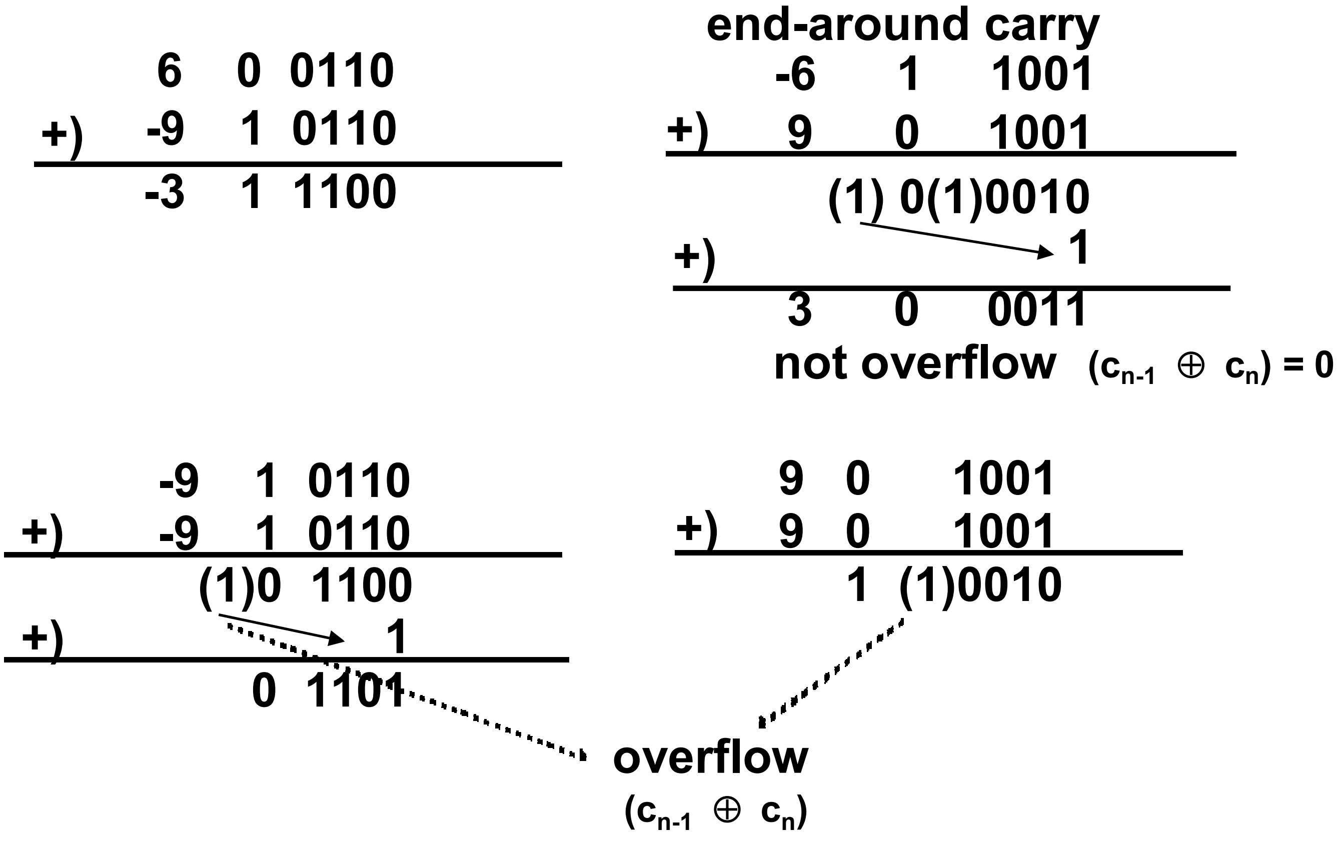
$x_{n-1}y_{n-1}s'_{n-1} + x'_{n-1}y'_{n-1}s_{n-1} = c_{n-1} \oplus c_n$

ARITHMETIC ADDITION: “IGNED 1’s COMPLEMENT

Add the two numbers, including their sign bits.

- If there is a carry out of the most significant (sign) bit, the result is incremented by 1 and the carry is discarded.

Example



```
        6    0  0110
+)     -9    1  0110
-----------------------
       -3    1  1100
```

end-around carry

```
       -6    1    1001
+)      9    0    1001
-----------------------
       (1) 0(1)0010
+)                 1
-----------------------
        3    0    0011
```

not overflow $(c_{n-1} \oplus c_n) = 0$

```
       -9    1  0110
+)     -9    1  0110
-----------------------
          (1)0  1100
+)                 1
-----------------------
             0  1101
```

```
        9   0     1001
+)      9   0     1001
-----------------------
            1  (1)0010
```

overflow

$(c_{n-1} \oplus c_n)$

COMPARISON OF REPRESENTATIONS

*** Easiness of negative conversion**

S + M > 1’s Complement > 2’s Complement

*** Hardware**

- S+M: Needs an adder and a subtractor for Addition
- 1’s and 2’s Complement: Need only an adder

*** Speed of Arithmetic**

2’s Complement > 1’s Complement(end-around C)

*** Recognition of Zero**

2’s Complement is fast

ARITHMETIC SUBTRACTION

Arithmetic Subtraction in 2's complement

Take the complement of the subtrahend (including the sign bit) and add it to the minuend including the sign bits.

$$(\pm A) - (-B) = (\pm A) + B$$

$$(\pm A) - B = (\pm A) + (-B)$$

FLOATING POINT NUMBER REPRESENTATION

*** The location of the fractional point is not fixed to a certain location**
*** The range of the representable numbers is wide**

$$F = EM$$

m_n	$e_k e_{k-1} \ldots e_0$	$m_{n-1} m_{n-2} \cdots m_0 . m_{-1} \cdots m_{-m}$
sign	exponent	mantissa

- Mantissa
Signed fixed point number, either an integer or a fractional number

- Exponent
Designates the position of the radix point

Decimal Value

$$V(F) = V(M) * R^{V(E)}$$

M: Mantissa
E: Exponent
R: Radix

FLOATING POINT NUMBERS

Example

sign		sign	
0	**.1234567**	**0**	**04**
mantissa		**exponent**	

==> +.1234567 x 10^{+04}

Note:

In Floating Point Number representation, only Mantissa(M) and Exponent(E) are explicitly represented. The Radix(R) and the position of the Radix Point are implied.

Example

A binary number +1001.11 in 16-bit floating point number representation (6-bit exponent and 10-bit fractional mantissa)

	Sign	**Exponent**	**Mantissa**
	0	**0 00100**	**100111000**
or	**0**	**0 00101** .	**010011100**

CHARACTERISTICS OF FLOATING POINT NUMBER REPRESENTATIONS

Normal Form

- There are many different floating point number representations of the same number
→ Need for a unified representation in a given computer

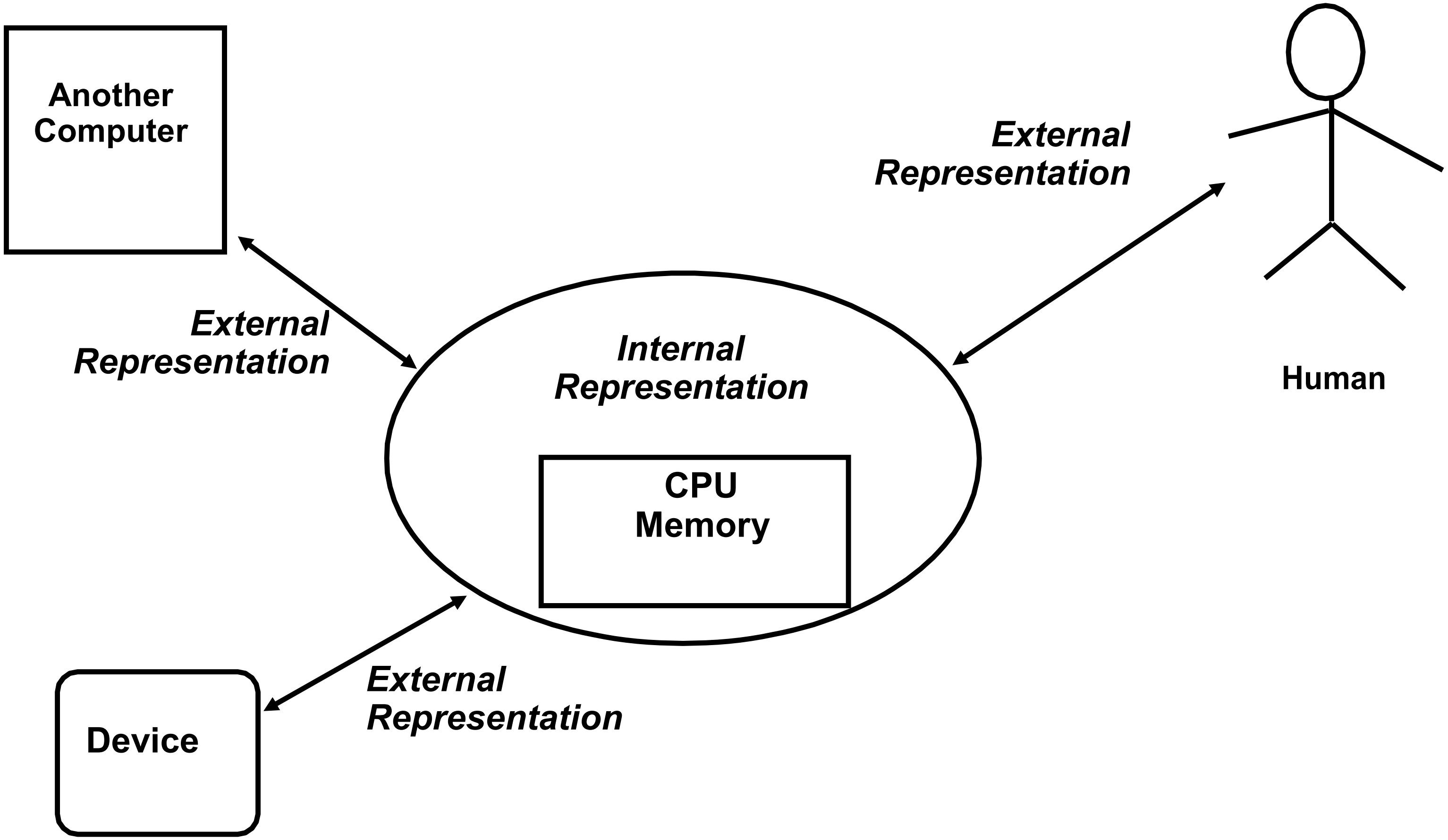
- *the most significant position of the mantissa contains a non-zero digit*

Representation of Zero

- Zero
Mantissa = 0

- Real Zero
Mantissa = 0
Exponent
= smallest representable number
which is represented as
00 ... 0
← Easily identified by the hardware

INTERNAL REPRESENTATION AND EXTERNAL REPRESENTATION



EXTERNAL REPRESENTATION

Numbers

Most of numbers stored in the computer are eventually changed by some kinds of calculations

→ *Internal Representation* for calculation efficiency

→ Final results need to be converted to as *External Representation* for presentability

Alphabets, Symbols, and some Numbers

Elements of these information do not change in the course of processing

→ No needs for Internal Representation since they are not used for calculations

→ External Representation for processing and presentability

Example

Decimal Number: 4-bit Binary Code
BCD(Binary Coded Decimal)

Decimal	BCD Code
0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111
8	1000
9	1001

OTHER DECIMAL CODES

Decimal	BCD(8421)	2421	84-2-1	Excess-3
0	0000	0000	0000	0011
1	0001	0001	0111	0100
2	0010	0010	0110	0101
3	0011	0011	0101	0110
4	0100	0100	0100	0111
5	0101	1011	1011	1000
6	0110	1100	1010	1001
7	0111	1101	1001	1010
8	1000	1110	1000	1011
9	1001	1111	1111	1100

Note: 8,4,2,-2,1,-1 in this table is the weight associated with each bit position.

$d_3\, d_2\, d_1\, d_0$: symbol in the codes

BCD: $d_3 \times 8 + d_2 \times 4 + d_1 \times 2 + d_0 \times 1$

$\Rightarrow$ 8421 code.

2421: $d_3 \times 2 + d_2 \times 4 + d_1 \times 2 + d_0 \times 1$

84-2-1: $d_3 \times 8 + d_2 \times 4 + d_1 \times (-2) + d_0 \times (-1)$

Excess-3: BCD + 3

BCD: It is difficult to obtain the 9's complement.

However, it is easily obtained with the other codes listed above.

→ Self-complementing codes

GRAY CODE

*** Characterized by having their representations of the binary integers differ in only one digit between consecutive integers**

*** Useful in some applications**

4-bit Gray codes

Decimal	Gray				Binary			
number	g_3	g_2	g_1	g_0	b_3	b_2	b_1	b_0
0	0	0	0	0	0	0	0	0
1	0	0	0	1	0	0	0	1
2	0	0	1	1	0	0	1	0
3	0	0	1	0	0	0	1	1
4	0	1	1	0	0	1	0	0
5	0	1	1	1	0	1	0	1
6	0	1	0	1	0	1	1	0
7	0	1	0	0	0	1	1	1
8	1	1	0	0	1	0	0	0
9	1	1	0	1	1	0	0	1
10	1	1	1	1	1	0	1	0
11	1	1	1	0	1	0	1	1
12	1	0	1	0	1	1	0	0
13	1	0	1	1	1	1	0	1
14	1	0	0	1	1	1	1	0
15	1	0	0	0	1	1	1	1

GRAY CODE - ANALYSIS

Letting $g_n g_{n-1} \ldots g_1 g_0$ be the (n+1)-bit Gray code for the binary number $b_n b_{n-1} \ldots b_1 b_0$

$$g_i = b_i \oplus b_{i+1} \quad , \quad 0 \leq i \leq n-1$$

$$g_n = b_n$$

and

$$b_{n-i} = g_n \oplus g_{n-1} \oplus \ldots \oplus g_{n-i}$$

$$b_n = g_n$$

Reflection of Gray codes

ε	0	0 0	0 00	0 000
	1	0 1	0 01	0 001
		1 1	0 11	0 011
		1 0	0 10	0 010
			1 10	0 110
			1 11	0 111
			1 01	0 101
			1 00	0 100
				1 100
				1 101
				1 111
				1 010
				1 011
				1 001
				1 101
				1 000

Note:

The Gray code has a reflection property

- **easy to construct a table without calculation,**
- **for any n: reflect case n-1 about a mirror at its bottom and prefix 0 and 1 to top and bottom halves, respectively**

CHARACTER REPRESENTATION ASCII

ASCII (American Standard Code for Information Interchange) Code

MSB (3 bits)

LSB (4 bits)

	0	1	2	3	4	5	6	7
0	NUL	DLE	SP	0	@	P	‘	P
1	SOH	DC1	!	1	A	Q	a	q
2	STX	DC2	“	2	B	R	b	r
3	ETX	DC3	#	3	C	S	c	s
4	EOT	DC4	$	4	D	T	d	t
5	ENQ	NAK	%	5	E	U	e	u
6	ACK	SYN	&	6	F	V	f	v
7	BEL	ETB	‘	7	G	W	g	w
8	BS	CAN	(	8	H	X	h	x
9	HT	EM	)	9	I	Y	I	y
A	LF	SUB	*	:	J	Z	j	z
B	VT	ESC	+	;	K	[	k	{
C	FF	FS	,	<	L	\	I	|
D	CR	GS	-	=	M	]	m	}
E	SO	RS	.	>	N	m	n	~
F	SI	US	/	?	O	n	o	DEL

CONTROL CHARACTER REPRESENTAION (ACSII)

NUL	Null	DC1	Device Control 1
SOH	Start of Heading (CC)	DC2	Device Control 2
STX	Start of Text (CC)	DC3	Device Control 3
ETX	End of Text (CC)	DC4	Device Control 4
EOT	End of Transmission (CC)	NAK	Negative Acknowledge (CC)
ENQ	Enquiry (CC)	SYN	Synchronous Idle (CC)
ACK	Acknowledge (CC)	ETB	End of Transmission Block (CC)
BEL	Bell	CAN	Cancel
BS	Backspace (FE)	EM	End of Medium
HT	Horizontal Tab. (FE)	SUB	Substitute
LF	Line Feed (FE)	ESC	Escape
VT	Vertical Tab. (FE)	FS	File Separator (IS)
FF	Form Feed (FE)	GS	Group Separator (IS)
CR	Carriage Return (FE)	RS	Record Separator (IS)
SO	Shift Out	US	Unit Separator (IS)
SI	Shift In	DEL	Delete
DLE	Data Link Escape (CC)		

(CC) Communication Control
(FE) Format Effector
(IS) Information Separator

ERROR DETECTING CODES

Parity System

- **Simplest method for error detection**
- **One *parity* bit attached to the information**
- ***Even Parity* and *Odd Parity***

Even Parity

- **One bit is attached to the information so that the total number of 1 bits is an even number**

$1011001\boxed{0}$

$1010010\boxed{1}$

Odd Parity

- **One bit is attached to the information so that the total number of 1 bits is an odd number**

$1011001\boxed{1}$

$1010010\boxed{0}$

PARITY BIT GENERATION

Parity Bit Generation

For $b_6 b_5 \ldots b_0$(7-bit information); even parity bit b_{even}

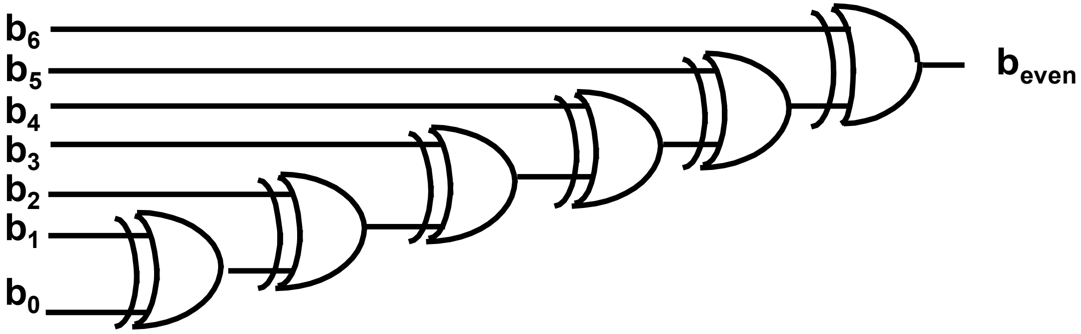
$$b_{even} = b_6 \oplus b_5 \oplus \ldots \oplus b_0$$

For odd parity bit

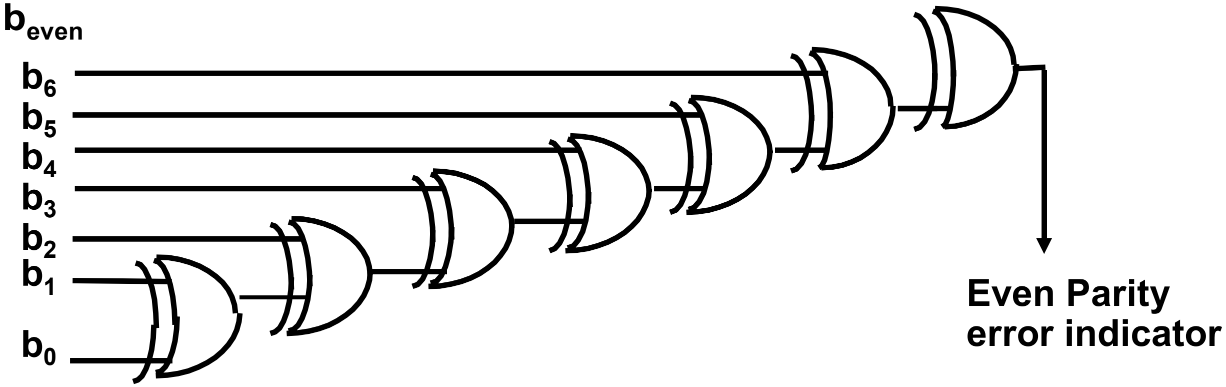
$$b_{odd} = b_{even} \oplus 1 = \overline{b}_{even}$$

PARITY GENERATOR AND PARITY CHECKER

Parity Generator Circuit (even parity)



Parity Checker



REGISTER TRANSFER AND MICROOPERATIONS

- Register Transfer Language
- Register Transfer
- Bus and Memory Transfers
- Arithmetic Micro-operations
- Logic Micro-operations
- Shift Micro-operations
- Arithmetic Logic Shift Unit

SIMPLE DIGITAL SYSTEMS

- Combinational and sequential circuits (learned in Chapters 1 and 2) can be used to create simple digital systems.

- These are the low-level building blocks of a digital computer.

- Simple digital systems are frequently characterized in terms of
  - the registers they contain, and
  - the operations that they perform.

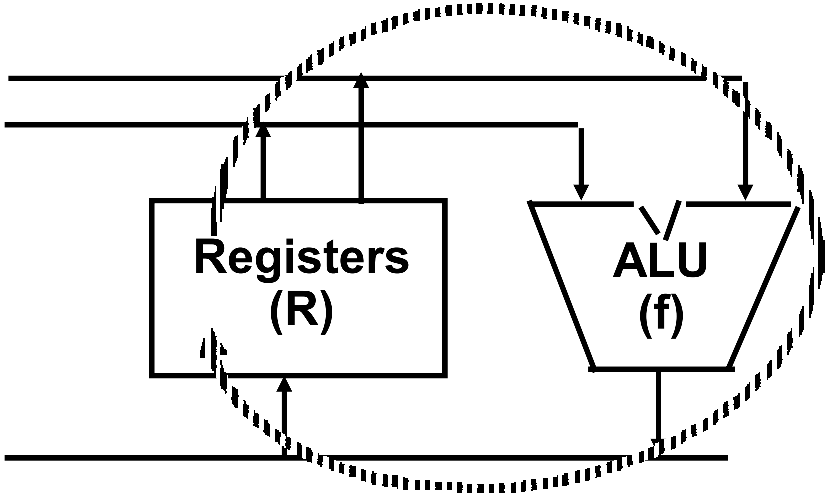
- Typically,
  - What operations are performed on the data in the registers
  - What information is passed between registers

MICROOPERATIONS (1)

- The operations on the data in registers are called micro-operations.
- The functions built into registers are examples of micro-operations
  - Shift
  - Load
  - Clear
  - Increment
  - …

MICROOPERATION (2)

An elementary operation performed (during one clock pulse), on the information stored in one or more registers



1 clock cycle

R ← f(R, R)

f: shift, load, clear, increment, add, subtract, complement, and, or, xor, …

ORGANIZATION OF A DIGITAL SYSTEM

- **Definition of the (internal) organization of a computer**

  - Set of registers and their functions

  - Micro-operations set

    Set of allowable micro-operations provided by the organization of the computer

  - Control signals that initiate the sequence of micro-operations (to perform the functions)

REGISTER TRANSFER LEVEL

- Viewing a computer, or any digital system, in this way is called the register transfer level

- This is because we're focusing on
  - The system's registers
  - The data transformations in them, and
  - The data transfers between them.

REGISTER TRANSFER LANGUAGE

- Rather than specifying a digital system in words, a specific notation is used, *register transfer language*

- For any function of the computer, the register transfer language can be used to describe the (sequence of) micro-operations

- Register transfer language
  - A symbolic language
  - A convenient tool for describing the internal organization of digital computers
  - Can also be used to facilitate the design process of digital systems.

DESIGNATION OF REGISTERS

- Registers are designated by capital letters, sometimes followed by numbers (e.g., A, R13, IR)
- Often the names indicate function:
  - MAR - memory address register
  - PC - program counter
  - IR - instruction register

- Registers and their contents can be viewed and represented in *various ways*
  - A register can be viewed as a single entity:

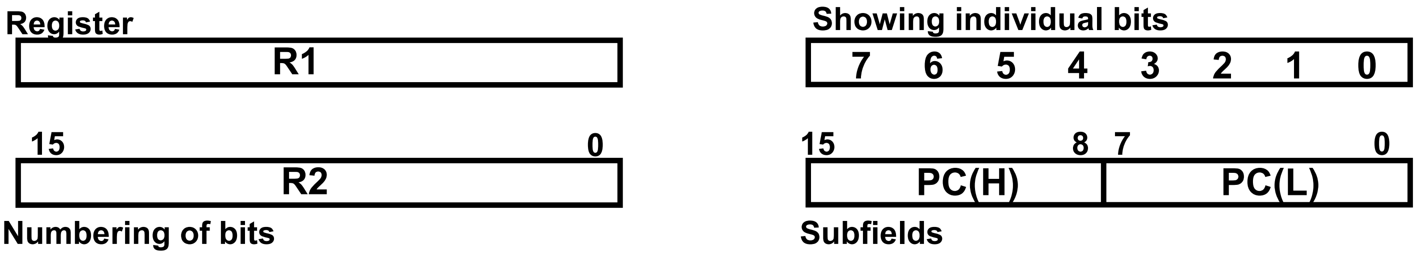
MAR

  - Registers may also be represented showing the bits of data they contain

DESIGNATION OF REGISTERS

- Designation of a register
  - a register
  - portion of a register
  - a bit of a register

- Common ways of drawing the block diagram of a register



Register

R1

Showing individual bits

7 6 5 4 3 2 1 0

15 0

R2

Numbering of bits

15 8 7 0

PC(H) PC(L)

Subfields

REGISTER TRANSFER

- Copying the contents of one register to another is a register transfer

- A register transfer is indicated as

R2 ← R1

  - In this case the contents of register R2 are copied (loaded) into register R1
  - A simultaneous transfer of all bits from the source R1 to the destination register R2, during one clock pulse
  - Note that this is a non-destructive; i.e. the contents of R1 are not altered by copying (loading) them to R2

REGISTER TRANSFER

- A register transfer such as

  R3 ← R5

  Implies that the digital system has

  - the data lines from the source register (R5) to the destination register (R3)
  - Parallel load in the destination register (R3)
  - Control lines to perform the action

CONTROL FUNCTIONS

- Often actions need to only occur if a certain condition is true
- This is similar to an “if” statement in a programming language
- In digital systems, this is often done via a *control signal*, called a *control function*
  - If the signal is 1, the action takes place
- This is represented as:

$R2 \leftarrow R1$

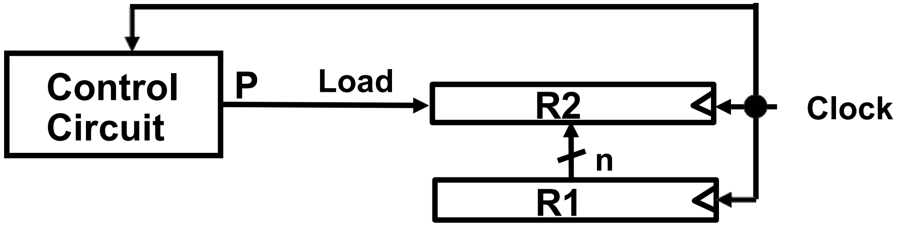
Which means “if P = 1, then load the contents of register R1 into register R2”, i.e., if $(P = 1)$ then $(R2 \leftarrow R1)$

HARDWARE IMPLEMENTATION OF CONTROLLED TRANSFERS

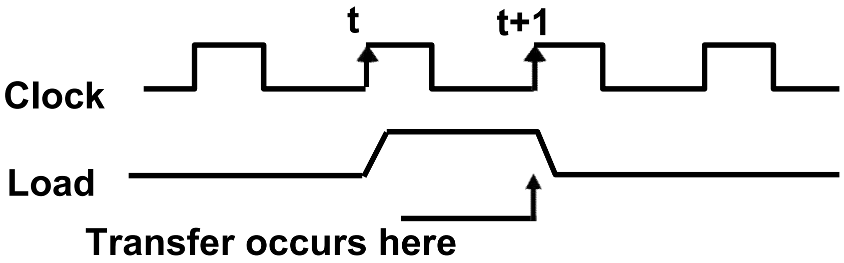
Implementation of controlled transfer

P: R2 ← R1

Block diagram



Timing diagram



- The same clock controls the circuits that generate the control function and the destination register
- Registers are assumed to use *positive-edge-triggered* flip-flops

SIMULTANEOUS OPERATIONS

- If two or more operations are to occur simultaneously, they are separated with commas

$$P:\ R3 \leftarrow R5,\ MAR \leftarrow IR$$

- Here, if the control function P = 1, load the contents of R5 into R3, and at the same time (clock), load the contents of register IR into register MAR

BASIC SYMBOLS FOR REGISTER TRANSFERS

Symbols	Description	Examples
Capital letters & numerals	Denotes a register	MAR, R2
Parentheses ()	Denotes a part of a register	R2(0-7), R2(L)
Arrow ←	Denotes transfer of information	R2 ← R1
Colon :	Denotes termination of control function	P:
Comma ,	Separates two micro-operations	A ← B, B ← A

CONNECTING REGISTRS

- In a digital system with many registers, it is impractical to have data and control lines to directly allow each register to be loaded with the contents of every possible other registers

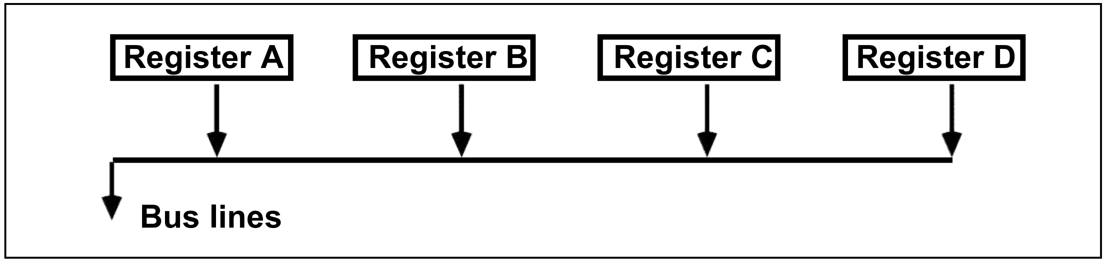
- To completely connect n registers → n(n-1) lines
- O(n^2) cost
  - This is not a realistic approach to use in a large digital system

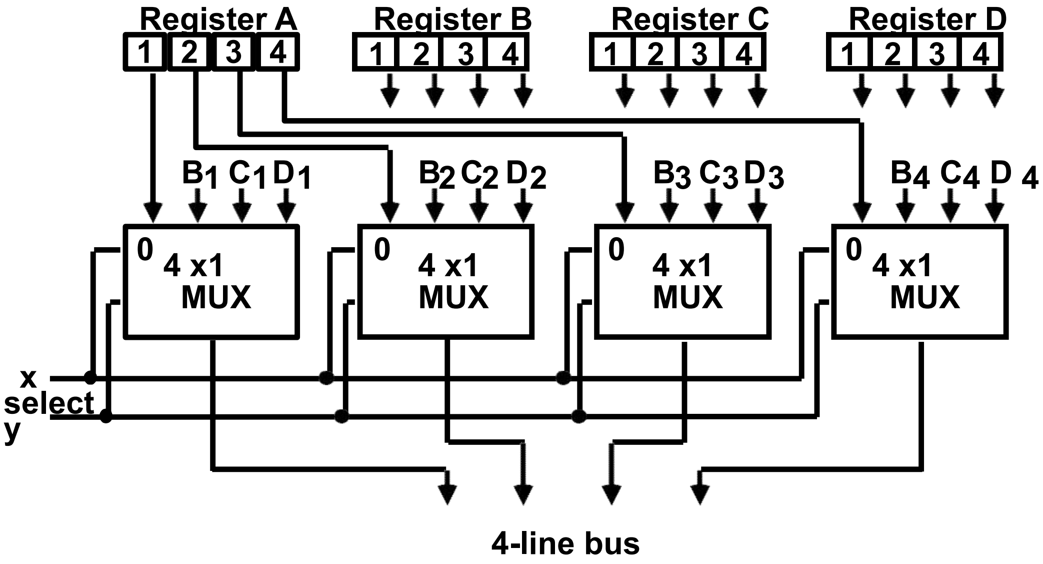
- Instead, take a different approach
- Have one centralized set of circuits for data transfer – the bus
- Have control circuits to select which register is the source, and which is the destination

BUS AND BUS TRANSFER

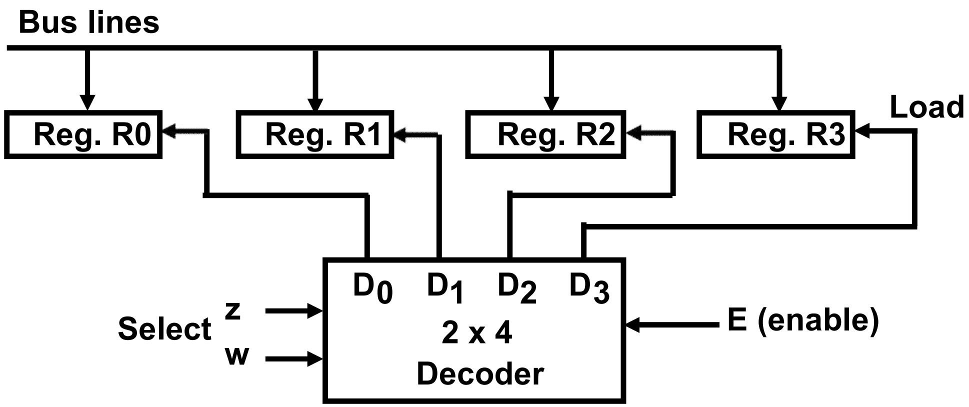
Bus is a path(of a group of wires) over which information is transferred, from any of several sources to any of several destinations.

From a register to bus: BUS ← R

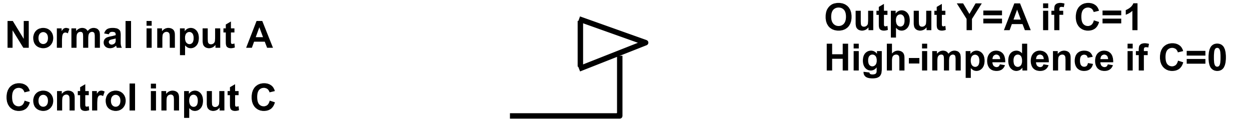




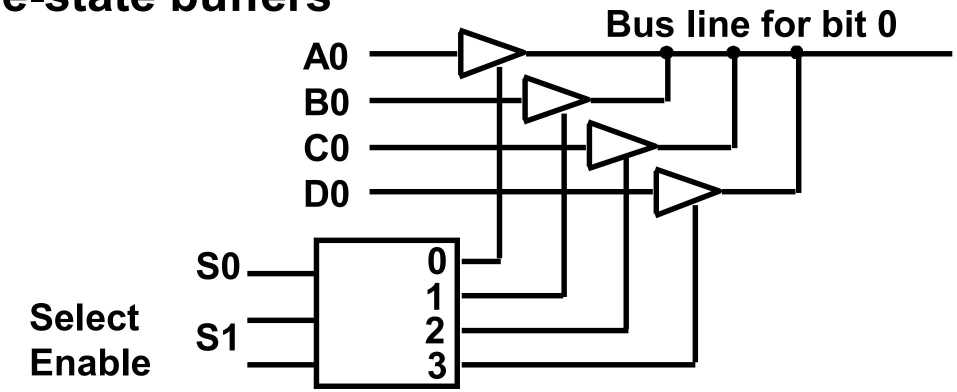
TRANSFER FROM BUS TO A DESTINATION REGISTER



Three-State Bus Buffers



Bus line with three-state buffers



BUS TRANSFER IN RTL

- Depending on whether the bus is to be mentioned explicitly or not, register transfer can be indicated as either

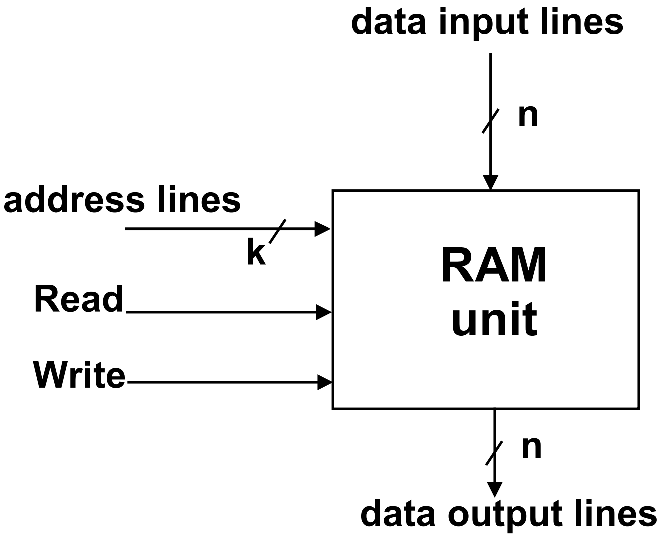
R2 ← R1

or **BUS ← R1, R2 ← BUS**

- In the former case the bus is implicit, but in the latter, it is explicitly indicated

MEMORY (RAM)

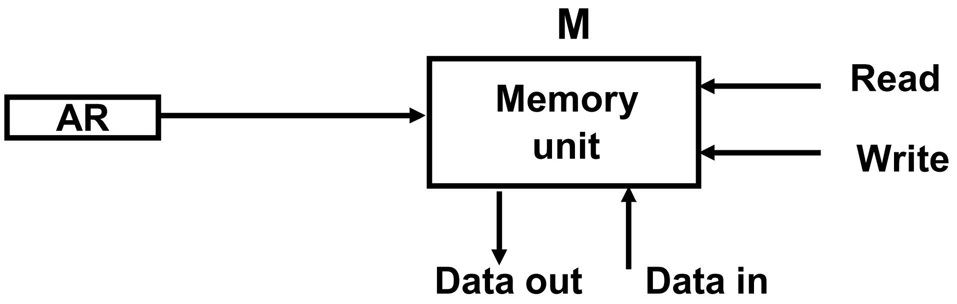
- Memory (RAM) can be thought as a sequential circuits containing some number of registers
- These registers hold the *words* of memory
- Each of the r registers is indicated by an *address*
- These addresses range from 0 to r-1
- Each register (word) can hold n bits of data
- Assume the RAM contains r = 2k words. It needs the following
  - n data input lines
  - n data output lines
  - k address lines
  - A Read control line
  - A Write control line



MEMORY TRANSFER

- Collectively, the memory is viewed at the register level as a device, M.
- Since it contains multiple locations, we must specify which address in memory we will be using
- This is done by indexing memory references

- Memory is usually accessed in computer systems by putting the desired address in a special register, the *Memory Address Register* (*MAR*, or *AR*)
- When memory is accessed, the contents of the MAR get sent to the memory unit's address lines



MEMORY READ

- To read a value from a location in memory and load it into a register, the register transfer language notation looks like this:

$$R1 \leftarrow M[MAR]$$

- This causes the following to occur
  - The contents of the MAR get sent to the memory address lines
  - A Read (= 1) gets sent to the memory unit
  - The contents of the specified address are put on the memory's output data lines
  - These get sent over the bus to be loaded into register R1

MEMORY WRITE

- To write a value from a register to a location in memory looks like this in register transfer language:

$$M[MAR] \leftarrow R1$$

- This causes the following to occur
  - The contents of the MAR get sent to the memory address lines
  - A Write (= 1) gets sent to the memory unit
  - The values in register R1 get sent over the bus to the data input lines of the memory
  - The values get loaded into the specified address in the memory

SUMMARY OF R. TRANSFER MICROOPERATIONS

A ← B	**Transfer content of reg. B into reg. A**
AR ← DR(AD)	**Transfer content of AD portion of reg. DR into reg. AR**
A ← constant	**Transfer a binary constant into reg. A**
ABUS ← R1,	**Transfer content of R1 into bus A and, at the same time,**
R2 ← ABUS	**transfer content of bus A into R2**
AR	**Address register**
DR	**Data register**
M[R]	**Memory word specified by reg. R**
M	**Equivalent to M[AR]**
DR ← M	**Memory *read* operation: transfers content of memory word specified by AR into DR**
M ← DR	**Memory *write* operation: transfers content of DR into memory word specified by AR**

MICROOPERATIONS

- **Computer system microoperations are of four types:**

  - Register transfer microoperations

  - Arithmetic microoperations

  - Logic microoperations

  - Shift microoperations

ARITHMETIC MICROOPERATIONS

- The basic arithmetic microoperations are
  - Addition
  - Subtraction
  - Increment
  - Decrement

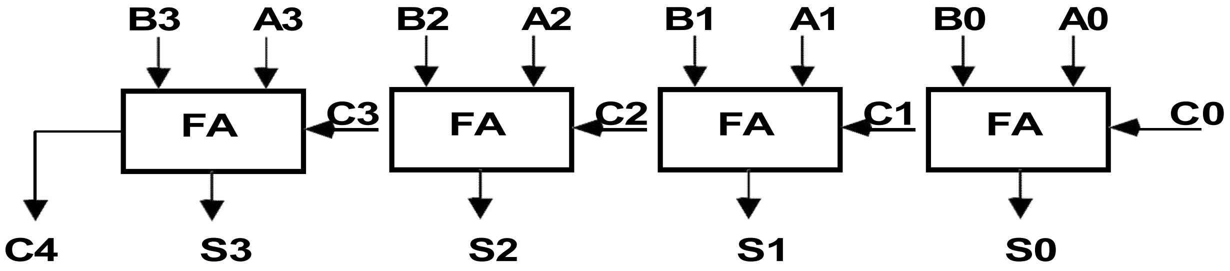
- The additional arithmetic microoperations are
  - Add with carry
  - Subtract with borrow
  - Transfer/Load
  - etc. …

Summary of Typical Arithmetic Micro-Operations

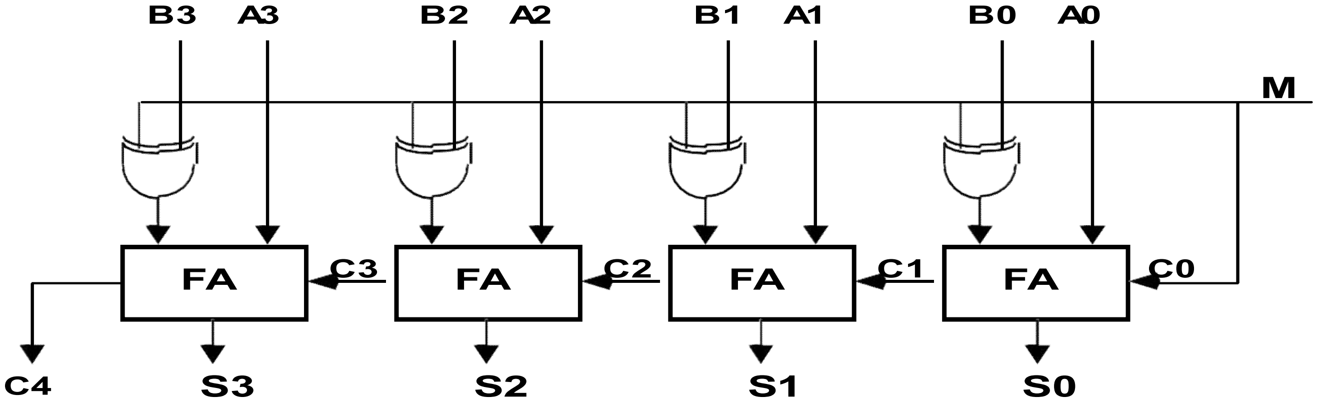
$R3 \leftarrow R1 + R2$	Contents of R1 plus R2 transferred to R3
$R3 \leftarrow R1 - R2$	Contents of R1 minus R2 transferred to R3
$R2 \leftarrow R2'$	Complement the contents of R2
$R2 \leftarrow R2' + 1$	2's complement the contents of R2 (negate)
$R3 \leftarrow R1 + R2' + 1$	subtraction
$R1 \leftarrow R1 + 1$	Increment
$R1 \leftarrow R1 - 1$	Decrement

BINARY ADDER / SUBTRACTOR / INCREMENTER

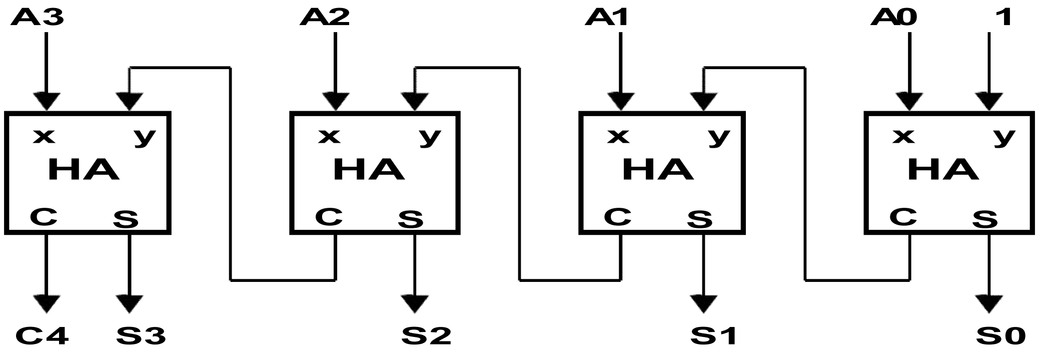
Binary Adder



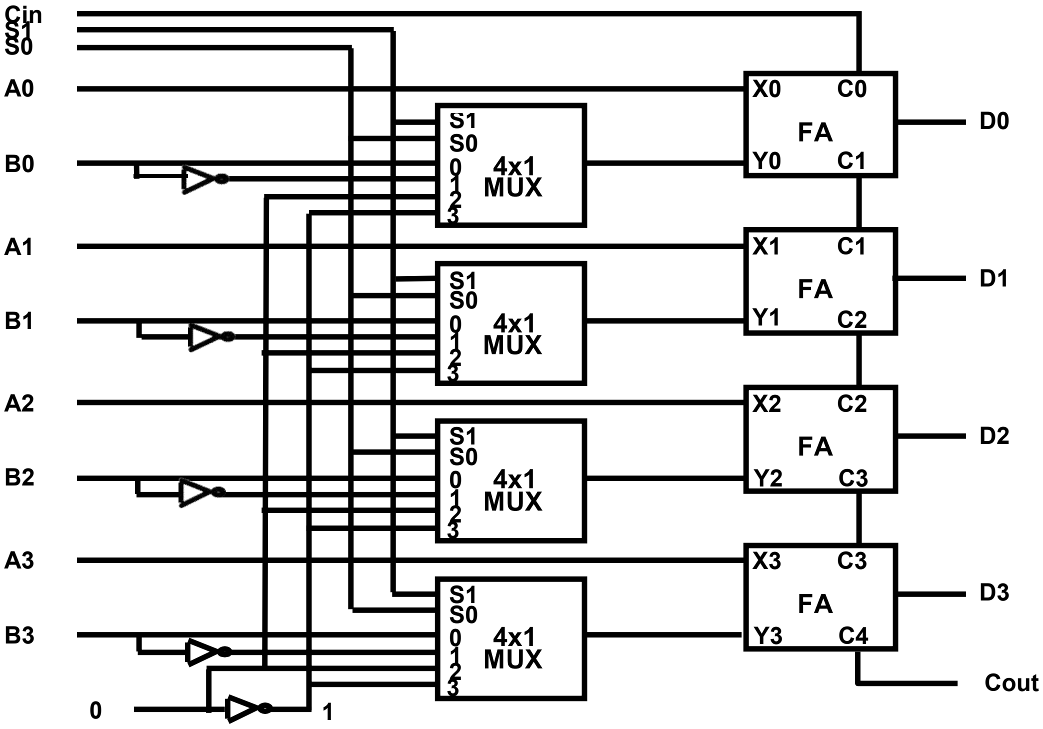
Binary Adder-Subtractor



Binary Incrementer



ARITHMETIC CIRCUIT



S1	S0	Cin	Y	Output	Microoperation
0	0	0	B	D = A + B	Add
0	0	1	B	D = A + B + 1	Add with carry
0	1	0	B’	D = A + B’	Subtract with borrow
0	1	1	B’	D = A + B’+ 1	Subtract
1	0	0	0	D = A	Transfer A
1	0	1	0	D = A + 1	Increment A
1	1	0	1	D = A - 1	Decrement A
1	1	1	1	D = A	Transfer A

LOGIC MICROOPERATIONS

- Specify binary operations on the strings of bits in registers
  - Logic microoperations are bit-wise operations, i.e., they work on the individual bits of data
  - useful for bit manipulations on binary data
  - useful for making logical decisions based on the bit value
- There are, in principle, 16 different logic functions that can be defined over two binary input variables

A	B	F_0	F_1	F_2	…	F_{13}	F_{14}	F_{15}
0	0	0	0	0	…	1	1	1
0	1	0	0	0	…	1	1	1
1	0	0	0	1	…	0	1	1
1	1	0	1	0	…	1	0	1

- However, most systems only implement four of these
  - AND ($\wedge$), OR ($\vee$), XOR ($\oplus$), Complement/NOT
- The others can be created from combination of these

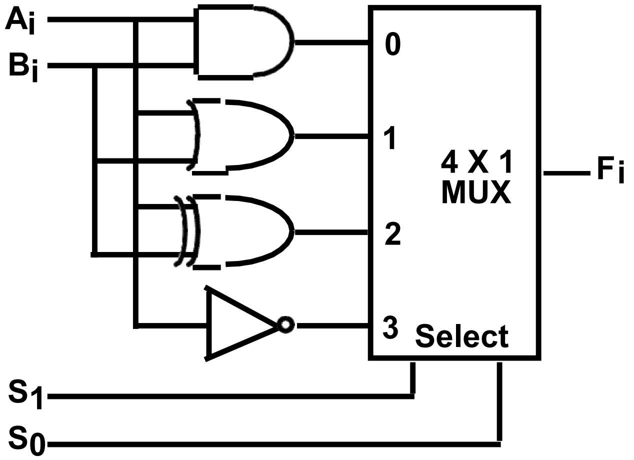
LIST OF LOGIC MICROOPERATIONS

- **List of Logic Microoperations**
  - **- 16 different logic operations with 2 binary vars.**
  - **- n binary vars → 2^{2^n} functions**

- **Truth tables for 16 functions of 2 variables and the corresponding 16 logic micro-operations**

x	0 0 1 1	*Boolean Function*	*Micro-Operations*	*Name*
y	0 1 0 1			
	0 0 0 0	$F0 = 0$	$F \leftarrow 0$	Clear
	0 0 0 1	$F1 = xy$	$F \leftarrow A \wedge B$	AND
	0 0 1 0	$F2 = xy'$	$F \leftarrow A \wedge B'$	
	0 0 1 1	$F3 = x$	$F \leftarrow A$	Transfer A
	0 1 0 0	$F4 = x'y$	$F \leftarrow A' \wedge B$	
	0 1 0 1	$F5 = y$	$F \leftarrow B$	Transfer B
	0 1 1 0	$F6 = x \oplus y$	$F \leftarrow A \oplus B$	Exclusive-OR
	0 1 1 1	$F7 = x + y$	$F \leftarrow A \vee B$	OR
	1 0 0 0	$F8 = (x + y)'$	$F \leftarrow (A \vee B)'$	NOR
	1 0 0 1	$F9 = (x \oplus y)'$	$F \leftarrow (A \oplus B)'$	Exclusive-NOR
	1 0 1 0	$F10 = y'$	$F \leftarrow B'$	Complement B
	1 0 1 1	$F11 = x + y'$	$F \leftarrow A \vee B$	
	1 1 0 0	$F12 = x'$	$F \leftarrow A'$	Complement A
	1 1 0 1	$F13 = x' + y$	$F \leftarrow A' \vee B$	
	1 1 1 0	$F14 = (xy)'$	$F \leftarrow (A \wedge B)'$	NAND
	1 1 1 1	$F15 = 1$	F ← all 1's	Set to all 1's

HARDWARE IMPLEMENTATION OF LOGIC MICROOPERATIONS



Function table

S_1	S_0	Output	μ-operation
0	0	$F = A \wedge B$	AND
0	1	$F = A \vee B$	OR
1	0	$F = A \oplus B$	XOR
1	1	$F = A'$	Complement

APPLICATIONS OF LOGIC MICROOPERATIONS

- Logic microoperations can be used to manipulate individual bits or a portions of a word in a register

- Consider the data in a register A. In another register, B, is bit data that will be used to modify the contents of A

– Selective-set	$A \leftarrow A + B$
– Selective-complement	$A \leftarrow A \oplus B$
– Selective-clear	$A \leftarrow A \bullet B'$
– Mask (Delete)	$A \leftarrow A \bullet B$
– Clear	$A \leftarrow A \oplus B$
– Insert	$A \leftarrow (A \bullet B) + C$
– Compare	$A \leftarrow A \oplus B$
– . . .	

SELECTIVE SET

- In a selective set operation, the bit pattern in B is used to *set* certain bits in A

$$\begin{array}{lll} 1\;1\;0\;0 & A_t & \\ 1\;0\;1\;0 & B & \\ \hline 1\;1\;1\;0 & A_{t+1} & (A \leftarrow A + B) \end{array}$$

- If a bit in B is set to 1, that same position in A gets set to 1, otherwise that bit in A keeps its previous value

SELECTIVE COMPLEMENT

- In a selective complement operation, the bit pattern in B is used to *complement* certain bits in A

$$\begin{array}{ll} 1\,1\,0\,0 & A_t \\ 1\,0\,1\,0 & B \\ \hline 0\,1\,1\,0 & A_{t+1} \qquad (A \leftarrow A \oplus B) \end{array}$$

- If a bit in B is set to 1, that same position in A gets complemented from its original value, otherwise it is unchanged

SELECTIVE CLEAR

- In a selective clear operation, the bit pattern in B is used to *clear* certain bits in A

$$\begin{array}{lll} 1\,1\,0\,0 & A_t & \\ \underline{1\,0\,1\,0} & \underline{B} & \\ 0\,1\,0\,0 & A_{t+1} & (A \leftarrow A \cdot B') \end{array}$$

- If a bit in B is set to 1, that same position in A gets set to 0, otherwise it is unchanged

MASK OPERATION

- In a mask operation, the bit pattern in B is used to *clear* certain bits in A

$$\begin{array}{lll} 1\,1\,0\,0 & A_t & \\ 1\,0\,1\,0 & B & \\ \hline 1\,0\,0\,0 & A_{t+1} & (A \leftarrow A \cdot B) \end{array}$$

- If a bit in B is set to 0, that same position in A gets set to 0, otherwise it is unchanged

CLEAR OPERATION

- In a clear operation, if the bits in the same position in A and B are the same, they are cleared in A, otherwise they are set in A

$$\begin{array}{ll} 1\;1\;0\;0 & A_t \\ 1\;0\;1\;0 & B \\ \hline 0\;1\;1\;0 & A_{t+1} \end{array} \qquad (A \leftarrow A \oplus B)$$

INSERT OPERATION

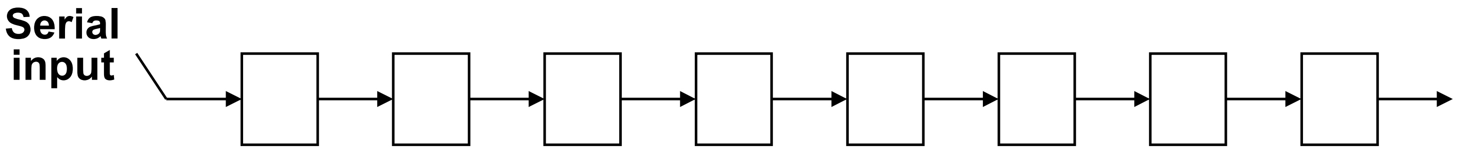
- An insert operation is used to introduce a specific bit pattern into A register, leaving the other bit positions unchanged
- This is done as
  - A mask operation to clear the desired bit positions, followed by
  - An OR operation to introduce the new bits into the desired positions
  - Example
    - Suppose you wanted to introduce 1010 into the low order four bits of A: 1101 1000 1011 0001 A (Original) 1101 1000 1011 1010 A (Desired)

    -
```
1101 1000 1011 0001             A (Original)
-------------------------------------------
1111 1111 1111 0000             Mask
1101 1000 1011 0000             A
-------------------------------------------
(Intermediate)
0000 0000 0000 1010             Added bits
1101 1000 1011 1010             A (Desired)
```

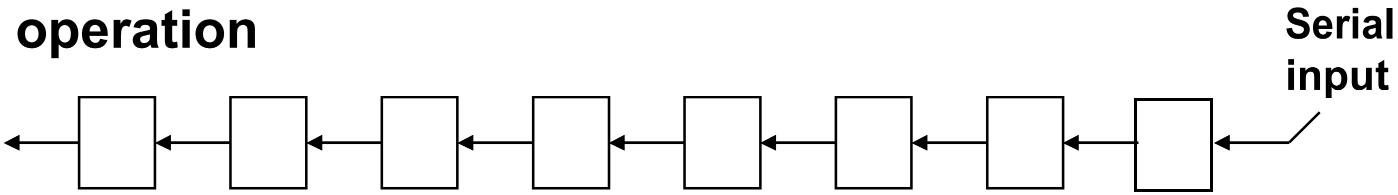
SHIFT MICROOPERATIONS

- There are three types of shifts
  - *Logical shift*
  - *Circular shift*
  - *Arithmetic shift*
- What differentiates them is the information that goes into the serial input

- **A right shift operation**



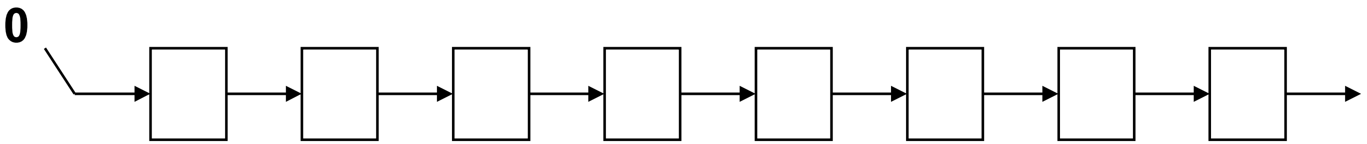
- **A left shift operation**



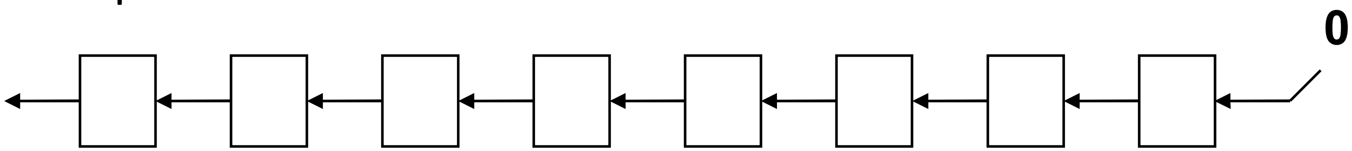
LOGICAL SHIFT

- In a logical shift the serial input to the shift is a 0.

- A right logical shift operation:



- A left logical shift operation:

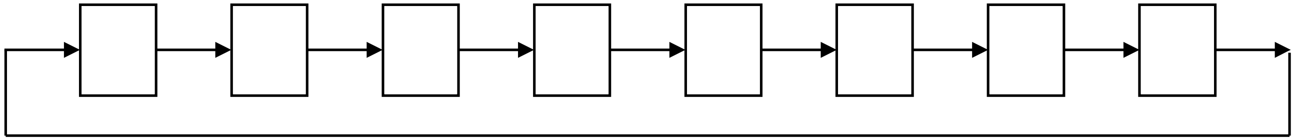


- In a Register Transfer Language, the following notation is used
  - *shl* for a logical shift left
  - *shr* for a logical shift right
  - Examples:
    - R2 ← *shr* R2
    - R3 ← *shl* R3

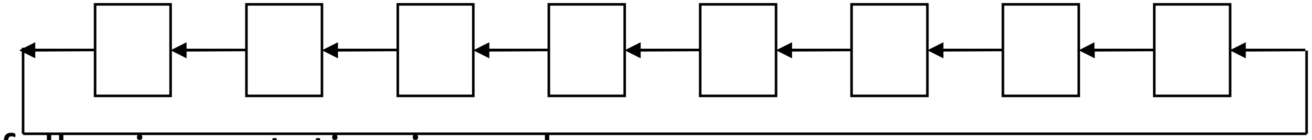
CIRCULAR SHIFT

- In a circular shift the serial input is the bit that is shifted out of the other end of the register.

- A right circular shift operation:



- A left circular shift operation:

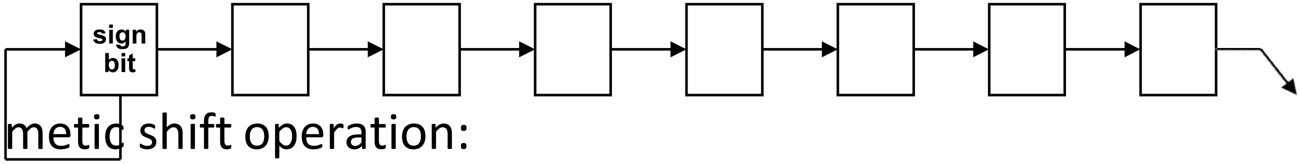


- In a RTL, the following notation is used
  - *cil* for a circular shift left
  - *cir* for a circular shift right
  - Examples:
    - R2 ← *cir* R2
    - R3 ← *cil* R3

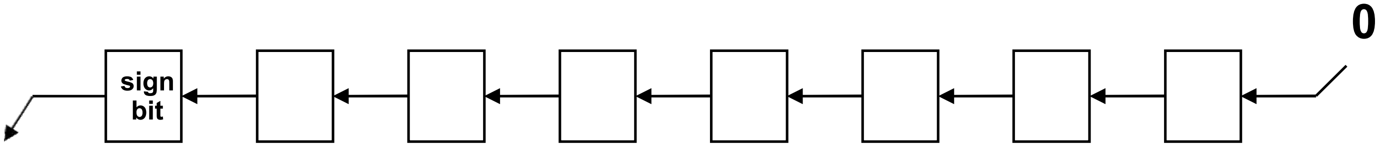
ARITHMETIC SHIFT

- An arithmetic shift is meant for signed binary numbers (integer)
- An arithmetic left shift multiplies a signed number by two
- An arithmetic right shift divides a signed number by two
- The main distinction of an arithmetic shift is that it must keep the sign of the number the same as it performs the multiplication or division

- A right arithmetic shift operation:

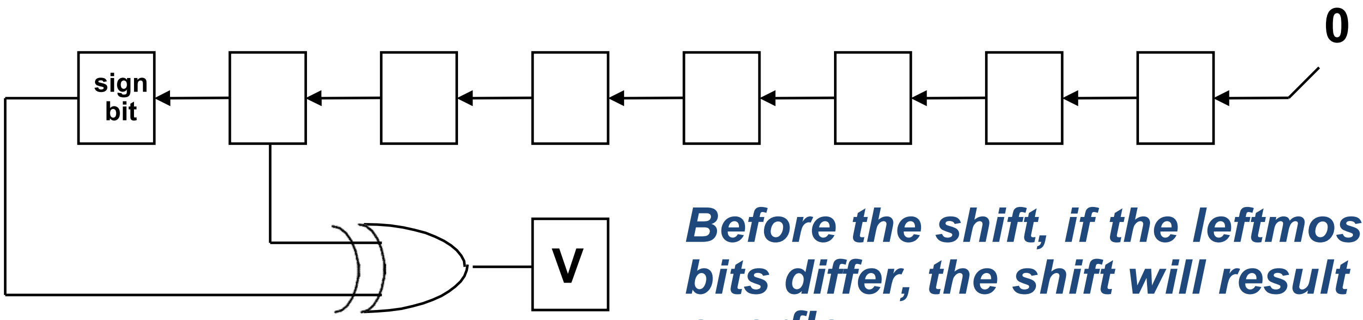


- A left arithmetic shift operation:



ARITHMETIC SHIFT

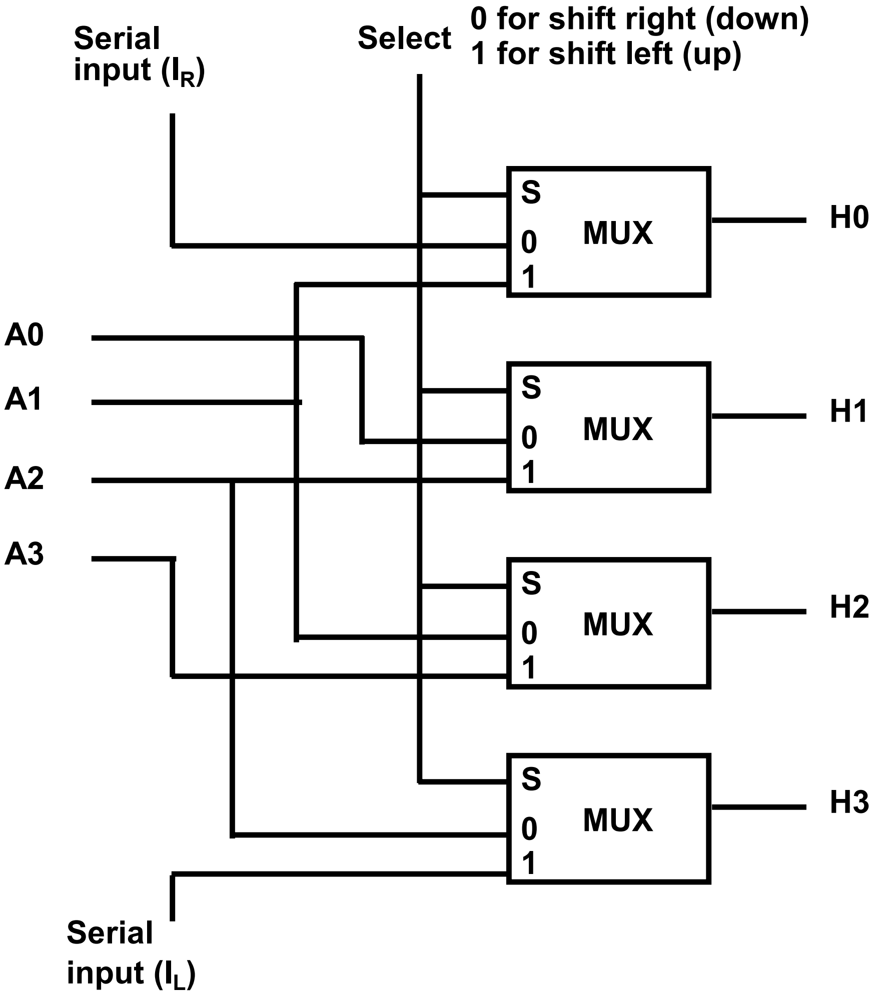
- An left arithmetic shift operation must be checked for the overflow



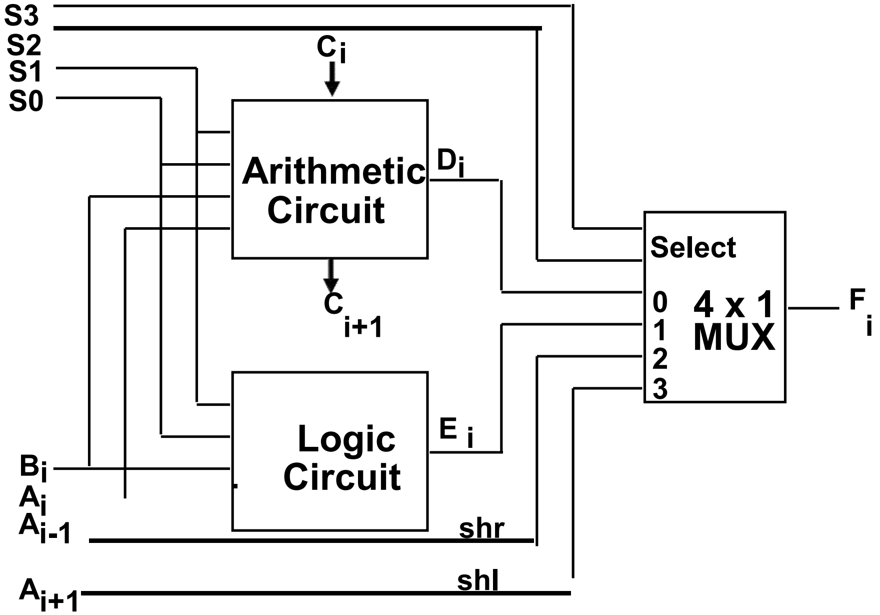
Before the shift, if the leftmost two bits differ, the shift will result in an overflow

- **In a RTL, the following notation is used**
  - ***ashl*** **for an arithmetic shift left**
  - ***ashr*** **for an arithmetic shift right**
  - **Examples:**
    - » **R2 ← *ashr* R2**
    - » **R3 ← *ashl* R3**

HARDWARE IMPLEMENTATION OF SHIFT MICROOPERATIONS



ARITHMETIC LOGIC SHIFT UNIT



S3	S2	S1	S0	Cin	Operation	Function
0	0	0	0	0	F = A	Transfer A
0	0	0	0	1	F = A + 1	Increment A
0	0	0	1	0	F = A + B	Addition
0	0	0	1	1	F = A + B + 1	Add with carry
0	0	1	0	0	F = A + B’	Subtract with borrow
0	0	1	0	1	F = A + B’+ 1	Subtraction
0	0	1	1	0	F = A - 1	Decrement A
0	0	1	1	1	F = A	TransferA
0	1	0	0	X	F = A $\wedge$ B	AND
0	1	0	1	X	F = A $\vee$ B	OR
0	1	1	0	X	F = A $\oplus$ B	XOR
0	1	1	1	X	F = A’	Complement A
1	0	X	X	X	F = shr A	Shift right A into F
1	1	X	X	X	F = shl A	Shift left A into F

BASIC COMPUTER ORGANIZATION AND DESIGN

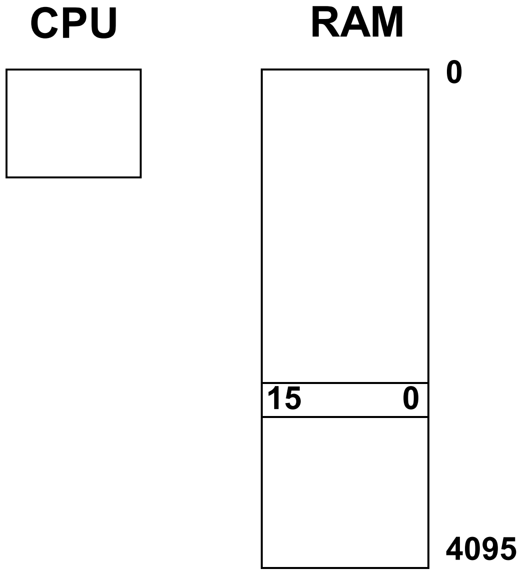
- Instruction Codes
- Computer Registers
- Computer Instructions
- Timing and Control
- Instruction Cycle
- Memory Reference Instructions
- Input-Output and Interrupt
- Complete Computer Description
- Design of Basic Computer
- Design of Accumulator Logic

INTRODUCTION

- Every different processor type has its own design (different registers, buses, micro-operations, machine instructions, etc)
- Modern processor is a very complex device
- It contains
  - Many registers
  - Multiple arithmetic units, for both integer and floating point calculations
  - The ability to pipeline several consecutive instructions to speed execution
  - Etc.
- However, to understand how processors work, we will start with a simplified processor model
- This is similar to what real processors were like ~25 years ago
- M. Morris Mano introduces a simple processor model he calls the *Basic Computer*
- We will use this to introduce processor organization and the relationship of the RTL model to the higher level computer processor

THE BASIC COMPUTER

- The Basic Computer has two components, a processor and memory
- The memory has 4096 words in it
  - 4096 = 2^{12}, so it takes 12 bits to select a word in memory
- Each word is 16 bits long



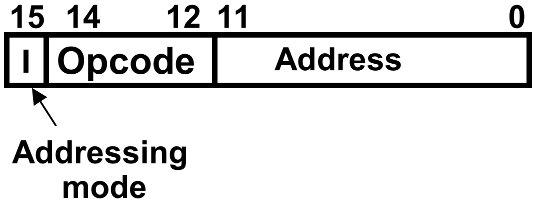
INSTRUCTIONS

- Program
  - A sequence of (machine) instructions
- (Machine) Instruction
  - A group of bits that tell the computer to *perform a specific operation* (a sequence of micro-operation)
- The instructions of a program, along with any needed data are stored in memory
- The CPU reads the next instruction from memory
- It is placed in an *Instruction Register* (IR)
- Control circuitry in control unit then translates the instruction into the sequence of microoperations necessary to implement it

INSTRUCTION FORMAT

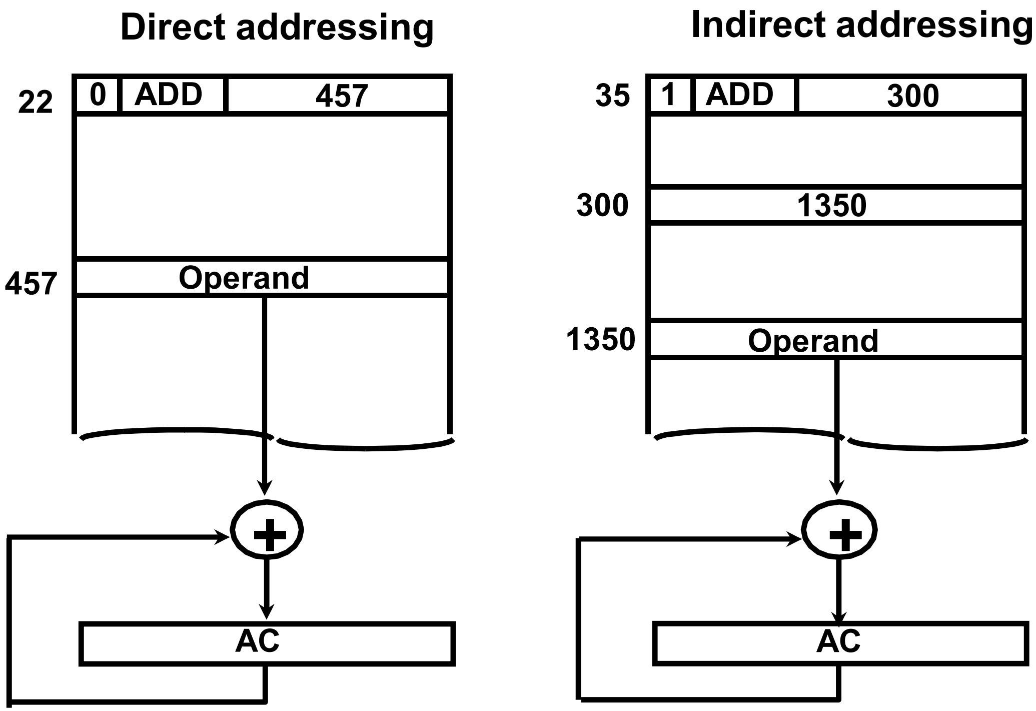
- A computer instruction is often divided into two parts
  - An *opcode* (Operation Code) that specifies the operation for that instruction
  - An *address* that specifies the registers and/or locations in memory to use for that operation
- In the Basic Computer, since the memory contains 4096 (= 2^{12}) words, we needs 12 bit to specify which memory address this instruction will use
- In the Basic Computer, bit 15 of the instruction specifies the *addressing mode* (0: direct addressing, 1: indirect addressing)
- Since the memory words, and hence the instructions, are 16 bits long, that leaves 3 bits for the instruction's opcode

Instruction Format



ADDRESSING MODES

- The address field of an instruction can represent either
  - Direct address: the address in memory of the data to use (the address of the operand), or
  - Indirect address: the address in memory of the address in memory of the data to use



- Effective Address (EA)
  - The address, that can be directly used without modification to access an operand for a computation-type instruction, or as the target address for a branch-type instruction

PROCESSOR REGISTERS

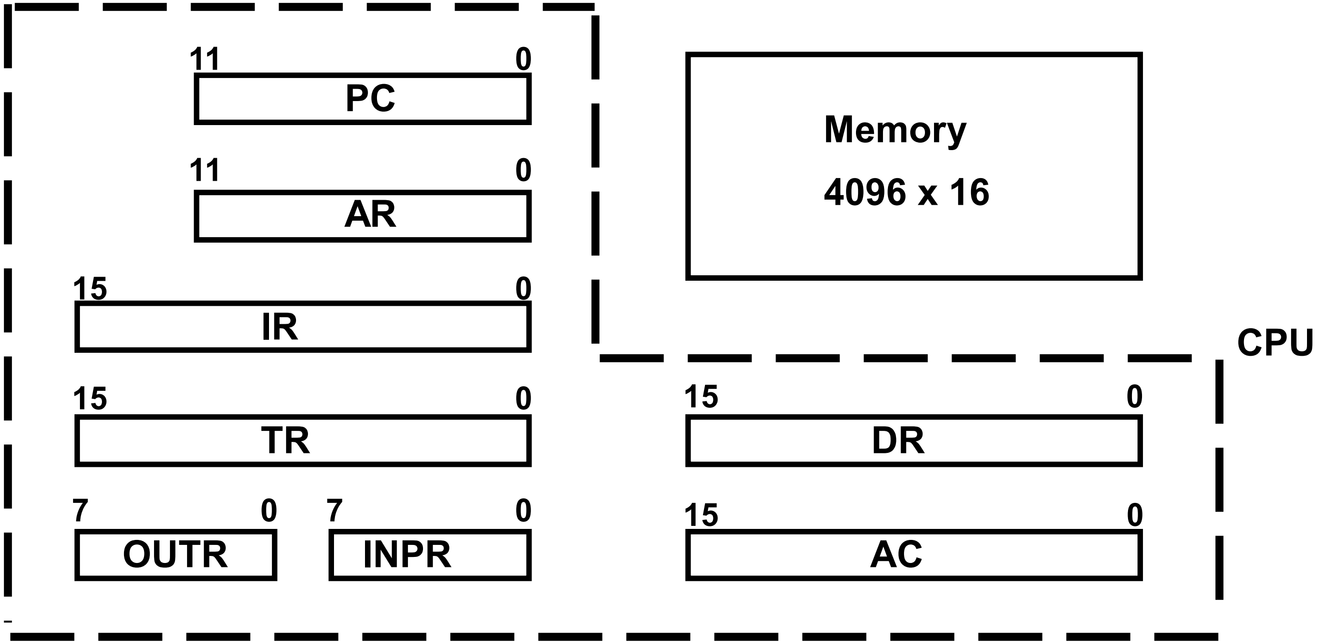
- A processor has many registers to hold instructions, addresses, data, etc
- The processor has a register, the *Program Counter* (PC) that holds the memory address of the next instruction to get
  - Since the memory in the Basic Computer only has 4096 locations, the PC only needs 12 bits
- In a direct or indirect addressing, the processor needs to keep track of what locations in memory it is addressing: The *Address Register* (AR) is used for this
  - The AR is a 12 bit register in the Basic Computer
- When an operand is found, using either direct or indirect addressing, it is placed in the *Data Register* (DR). The processor then uses this value as data for its operation
- The Basic Computer has a single *general purpose register* – the *Accumulator* (AC)

PROCESSOR REGISTERS

- The significance of a general purpose register is that it can be referred to in instructions
  - e.g. load AC with the contents of a specific memory location; store the contents of AC into a specified memory location
- Often a processor will need a scratch register to store intermediate results or other temporary data; in the Basic Computer this is the *Temporary Register* (TR)
- The Basic Computer uses a very simple model of input/output (I/O) operations
  - Input devices are considered to send 8 bits of character data to the processor
  - The processor can send 8 bits of character data to output devices
- The *Input Register* (INPR) holds an 8 bit character gotten from an input device
- The *Output Register* (OUTR) holds an 8 bit character to be send to an output device

BASIC COMPUTER REGISTERS

Registers in the Basic Computer



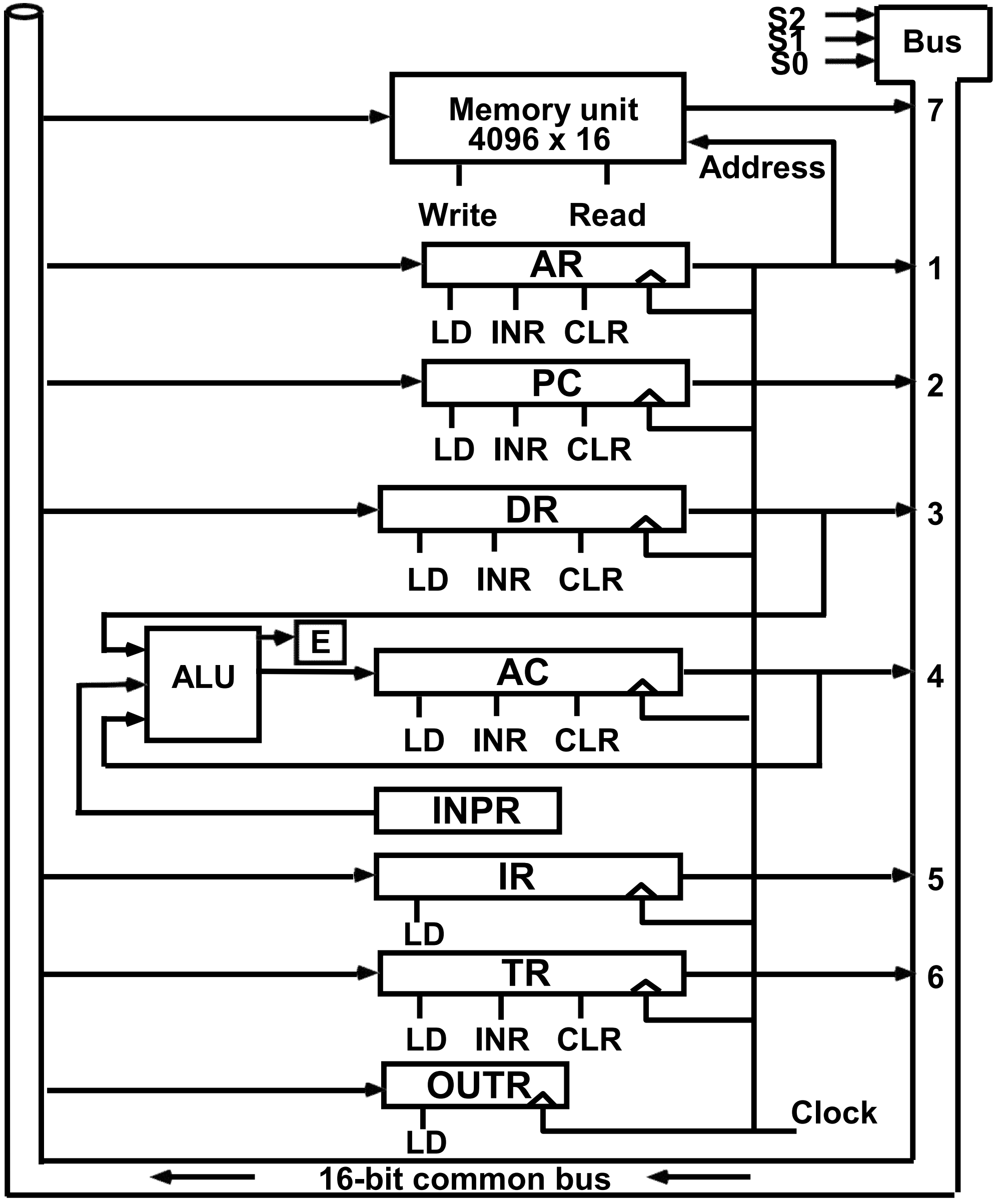
List of BC Registers

DR	16	Data Register	Holds memory operand
AR	12	Address Register	Holds address for memory
AC	16	Accumulator	Processor register
IR	16	Instruction Register	Holds instruction code
PC	12	Program Counter	Holds address of instruction
TR	16	Temporary Register	Holds temporary data
INPR	8	Input Register	Holds input character
OUTR	8	Output Register	Holds output character

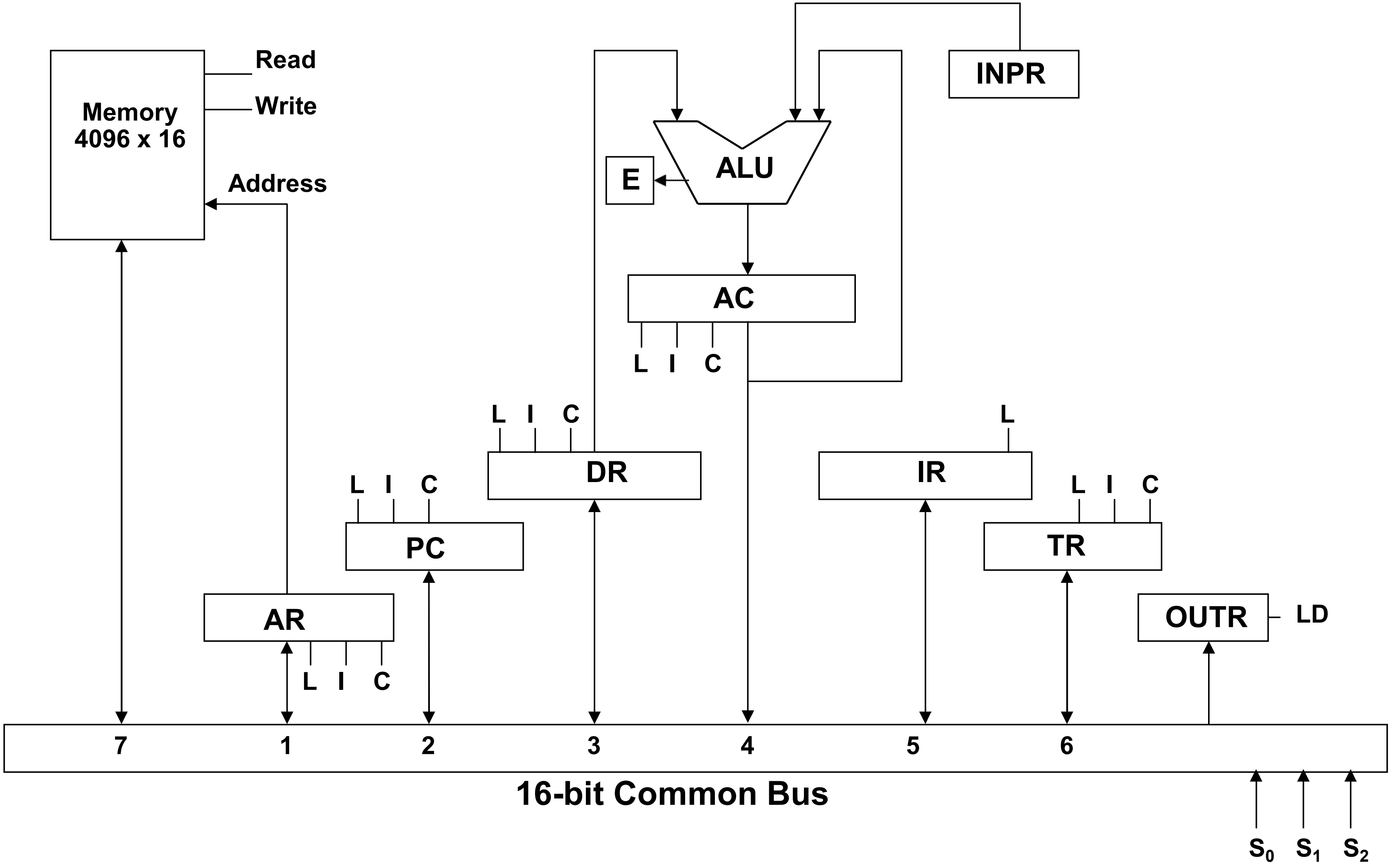
COMMON BUS SYSTEM

- The registers in the Basic Computer are connected using a bus
- This gives a savings in circuitry over complete connections between registers

COMMON BUS SYSTEM



COMMON BUS SYSTEM



COMMON BUS SYSTEM

- Three control lines, S_2, S_1, and S_0 control which register the bus selects as its input

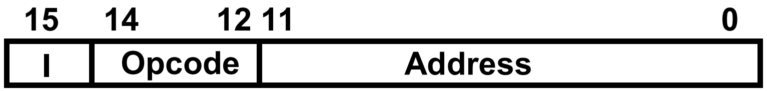
S_2 S_1 S_0	Register
0 0 0	x
0 0 1	AR
0 1 0	PC
0 1 1	DR
1 0 0	AC
1 0 1	IR
1 1 0	TR
1 1 1	Memory

- Either one of the registers will have its load signal activated, or the memory will have its read signal activated
  - Will determine where the data from the bus gets loaded
- The 12-bit registers, AR and PC, have 0's loaded onto the bus in the high order 4 bit positions
- When the 8-bit register OUTR is loaded from the bus, the data comes from the low order 8 bits on the bus

BASIC COMPUTER INSTRUCTIONS

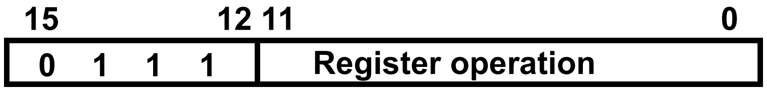
- **Basic Computer Instruction Format**

Memory-Reference Instructions (OP-code = 000 ~ 110)


15 14 12 11 0

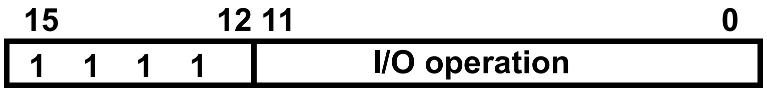
I | Opcode | Address

Register-Reference Instructions (OP-code = 111, I = 0)


15 12 11 0

0 1 1 1 | Register operation

Input-Output Instructions (OP-code =111, I = 1)


15 12 11 0

1 1 1 1 | I/O operation

BASIC COMPUTER INSTRUCTIONS

	Hex Code		
Symbol	***I = 0***	***I = 1***	***Description***
AND	0xxx	8xxx	AND memory word to AC
ADD	1xxx	9xxx	Add memory word to AC
LDA	2xxx	Axxx	Load AC from memory
STA	3xxx	Bxxx	Store content of AC into memory
BUN	4xxx	Cxxx	Branch unconditionally
BSA	5xxx	Dxxx	Branch and save return address
ISZ	6xxx	Exxx	Increment and skip if zero
CLA	7800		Clear AC
CLE	7400		Clear E
CMA	7200		Complement AC
CME	7100		Complement E
CIR	7080		Circulate right AC and E
CIL	7040		Circulate left AC and E
INC	7020		Increment AC
SPA	7010		Skip next instr. if AC is positive
SNA	7008		Skip next instr. if AC is negative
SZA	7004		Skip next instr. if AC is zero
SZE	7002		Skip next instr. if E is zero
HLT	7001		Halt computer
INP	F800		Input character to AC
OUT	F400		Output character from AC
SKI	F200		Skip on input flag
SKO	F100		Skip on output flag
ION	F080		Interrupt on
IOF	F040		Interrupt off

INSTRUCTION SET COMPLETENESS

A computer should have a set of instructions so that the user can construct machine language programs to evaluate any function that is known to be computable.

- **Instruction Types**

Functional Instructions
- Arithmetic, logic, and shift instructions
- ADD, CMA, INC, CIR, CIL, AND, CLA

Transfer Instructions
- Data transfers between the main memory and the processor registers
- LDA, STA

Control Instructions
- Program sequencing and control
- BUN, BSA, ISZ

Input/Output Instructions
- Input and output
- INP, OUT

CONTROL UNIT

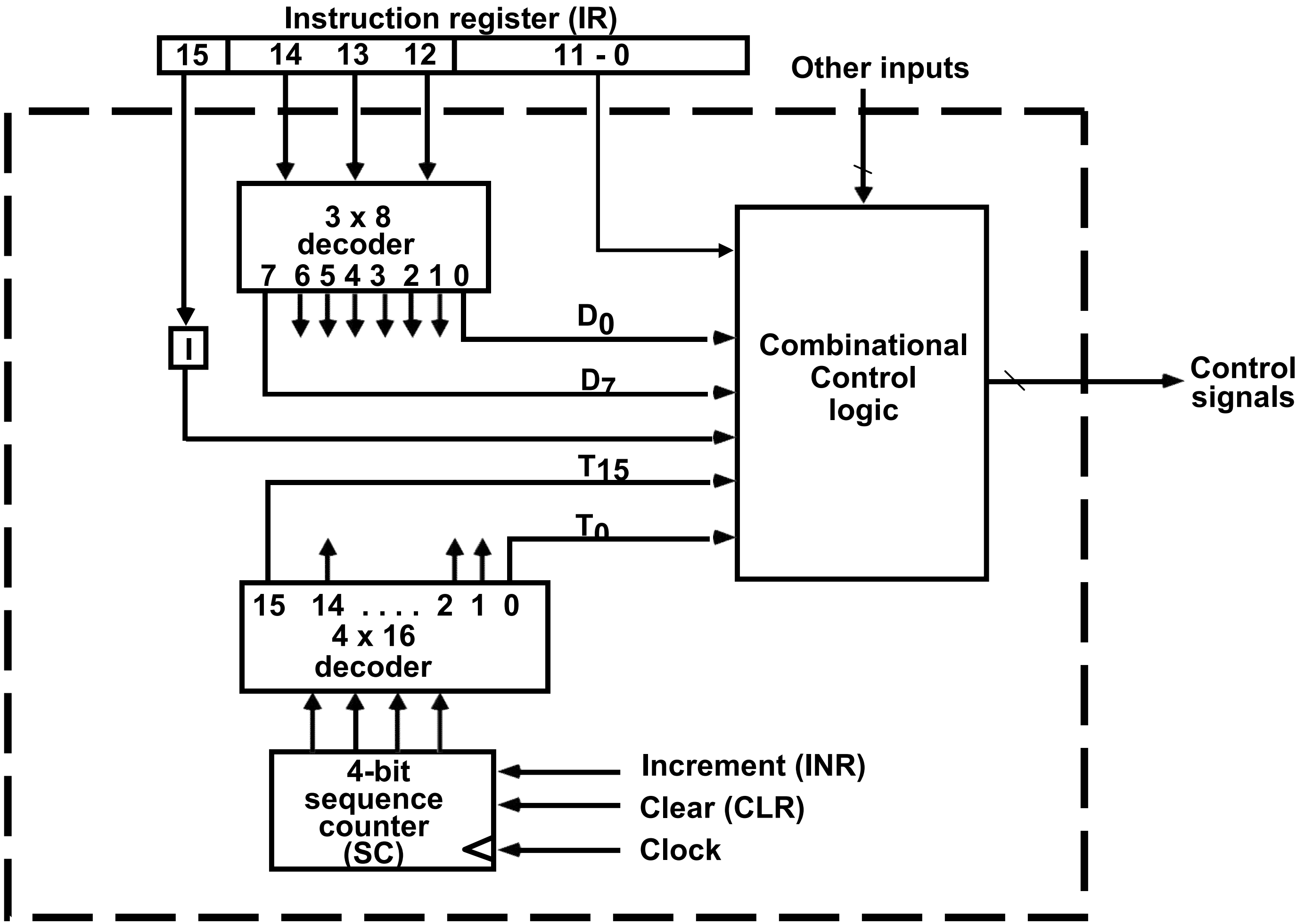
- Control unit (CU) of a processor translates from machine instructions to the control signals for the microoperations that implement them

- Control units are implemented in one of two ways
- ***Hardwired*** Control
  - CU is made up of sequential and combinational circuits to generate the control signals
- ***Microprogrammed*** Control
  - A control memory on the processor contains microprograms that activate the necessary control signals

- We will consider a hardwired implementation of the control unit for the Basic Computer

TIMING AND CONTROL

Control unit of Basic Computer

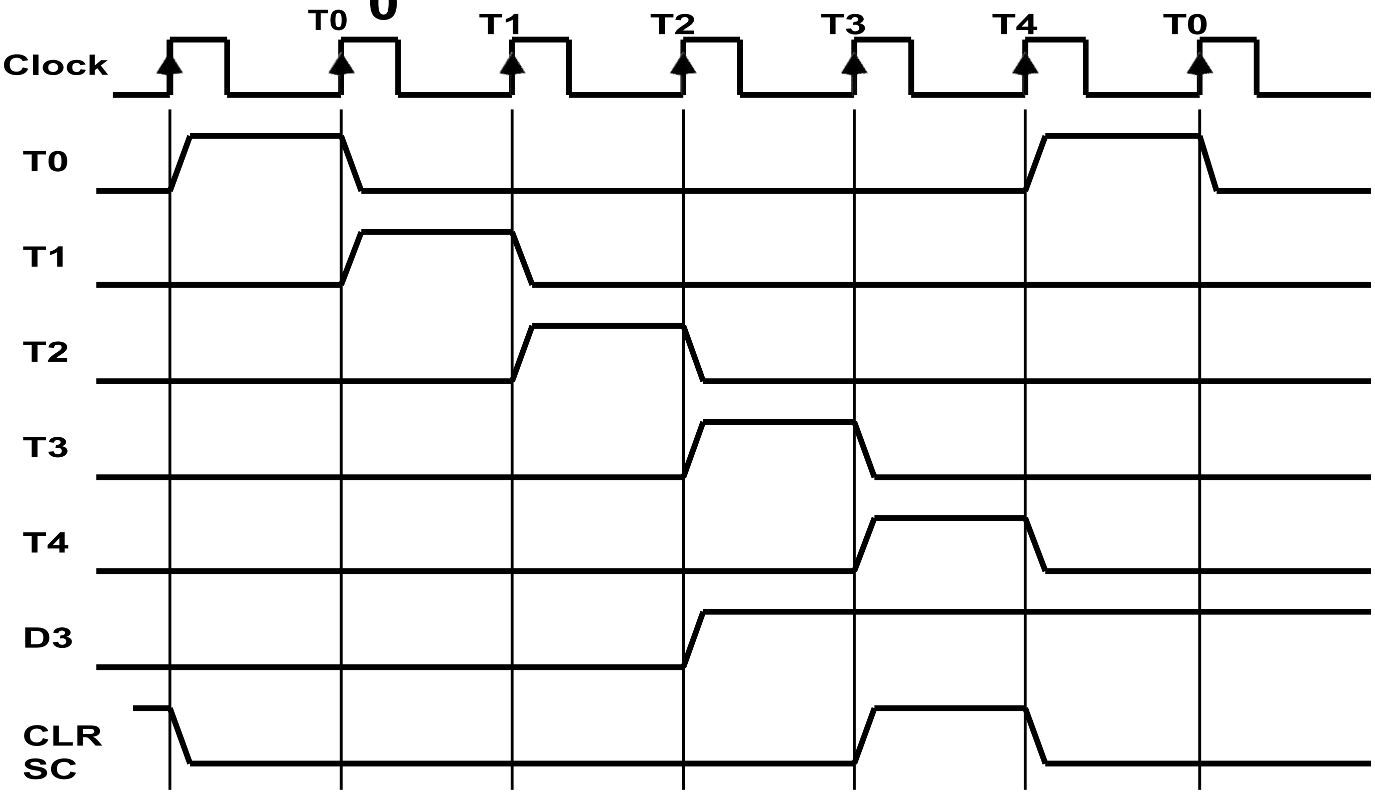


TIMING SIGNALS

- Generated by 4-bit sequence counter and 4×16 decoder
- The SC can be incremented or cleared.

- Example: $T_0, T_1, T_2, T_3, T_4, T_0, T_1, \ldots$
Assume: At time T_4, SC is cleared to 0 if decoder output D3 is active.

D_3T_4: SC ← 0



INSTRUCTION CYCLE

- In Basic Computer, a machine instruction is executed in the following cycle:
    1. Fetch an instruction from memory
    2. Decode the instruction
    3. Read the effective address from memory if the instruction has an indirect address
    4. Execute the instruction

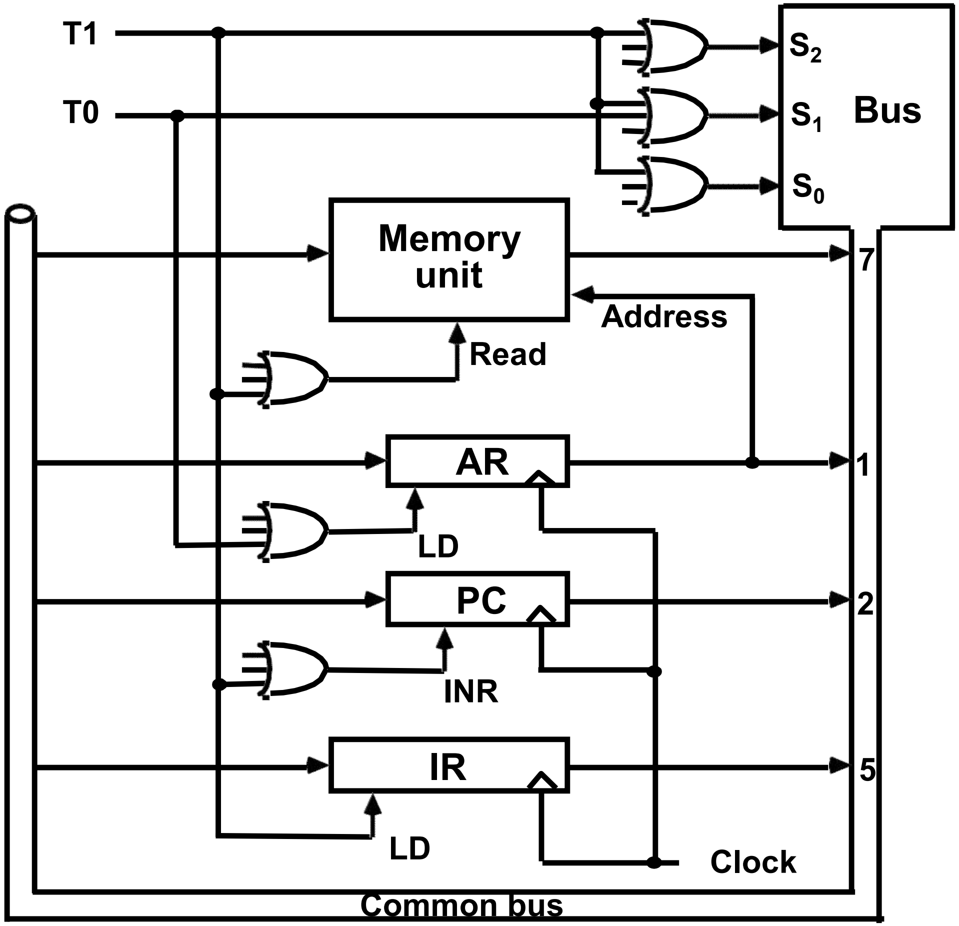
- After an instruction is executed, the cycle starts again at step 1, for the next instruction

- *Note*: Every different processor has its own (different) instruction cycle

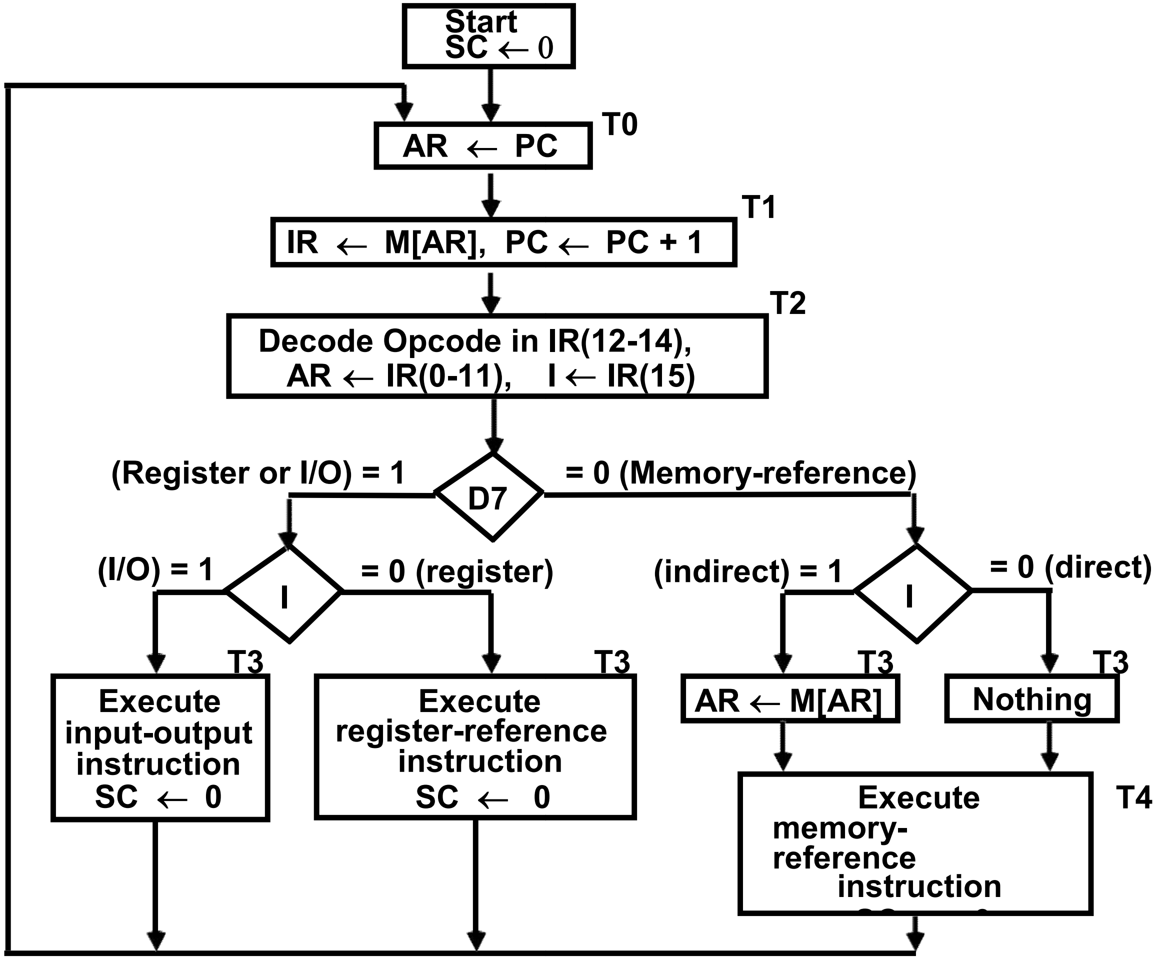
FETCH and DECODE

- **Fetch and Decode**

T0: AR ← PC ($S_0S_1S_2$=010, T0=1)
T1: IR ← M [AR], PC ← PC + 1 (S0S1S2=111, T1=1)
T2: D0, . . . , D7 ← Decode IR(12-14), AR ← IR(0-11), I ← IR(15)



DETERMINE THE TYPE OF INSTRUCTION


Start
SC ← 0
T0
AR ← PC
T1
IR ← M[AR], PC ← PC + 1
T2
Decode Opcode in IR(12-14),
AR ← IR(0-11), I ← IR(15)
(Register or I/O) = 1
D7
= 0 (Memory-reference)
(I/O) = 1
I
= 0 (register)
(indirect) = 1
I
= 0 (direct)
T3
Execute input-output instruction SC ← 0
T3
Execute register-reference instruction SC ← 0
T3
AR ← M[AR]
T3
Nothing
Execute memory-reference instruction
T4

D'_7IT_3: $AR \leftarrow M[AR]$

$D'_7I'T_3$: Nothing

$D_7I'T_3$: Execute a register-reference instr.

D_7IT_3: Execute an input-output instr.

REGISTER REFERENCE INSTRUCTIONS

Register Reference Instructions are identified when

- **$D_7 = 1$, $I = 0$**
- **Register Ref. Instr. is specified in $b_0 \sim b_{11}$ of IR**
- **Execution starts with timing signal T_3**

$r = D_7 I' T_3$ => Register Reference Instruction

$B_i = IR(i)$, i=0,1,2,...,11

	r:	$SC \leftarrow 0$
CLA	rB_{11}:	$AC \leftarrow 0$
CLE	rB_{10}:	$E \leftarrow 0$
CMA	rB_9:	$AC \leftarrow AC'$
CME	rB_8:	$E \leftarrow E'$
CIR	rB_7:	$AC \leftarrow \text{shr } AC,\ AC(15) \leftarrow E,\ E \leftarrow AC(0)$
CIL	rB_6:	$AC \leftarrow \text{shl } AC,\ AC(0) \leftarrow E,\ E \leftarrow AC(15)$
INC	rB_5:	$AC \leftarrow AC + 1$
SPA	rB_4:	if $(AC(15) = 0)$ then $(PC \leftarrow PC+1)$
SNA	rB_3:	if $(AC(15) = 1)$ then $(PC \leftarrow PC+1)$
SZA	rB_2:	if $(AC = 0)$ then $(PC \leftarrow PC+1)$
SZE	rB_1:	if $(E = 0)$ then $(PC \leftarrow PC+1)$
HLT	rB_0:	$S \leftarrow 0$ (S is a start-stop flip-flop)

MEMORY REFERENCE INSTRUCTIONS

Symbol	Operation Decoder	Symbolic Description
AND	D_0	$AC \leftarrow AC \wedge M[AR]$
ADD	D_1	$AC \leftarrow AC + M[AR], E \leftarrow C_{out}$
LDA	D_2	$AC \leftarrow M[AR]$
STA	D_3	$M[AR] \leftarrow AC$
BUN	D_4	$PC \leftarrow AR$
BSA	D_5	$M[AR] \leftarrow PC, PC \leftarrow AR + 1$
ISZ	D_6	$M[AR] \leftarrow M[AR] + 1$, if $M[AR] + 1 = 0$ then $PC \leftarrow PC+1$

- The effective address of the instruction is in AR and was placed there during timing signal T_2 when I = 0, or during timing signal T_3 when I = 1
- Memory cycle is assumed to be short enough to complete in a CPU cycle
- The execution of MR instruction starts with T_4

AND to AC

D_0T_4:	$DR \leftarrow M[AR]$	Read operand
D_0T_5:	$AC \leftarrow AC \wedge DR, SC \leftarrow 0$	AND with AC

ADD to AC

D_1T_4:	$DR \leftarrow M[AR]$	Read operand
D_1T_5:	$AC \leftarrow AC + DR, E \leftarrow C_{out}, SC \leftarrow 0$	Add to AC and store carry in E

MEMORY REFERENCE INSTRUCTIONS

LDA: Load to AC

D_2T_4: DR ← M[AR]

D_2T_5: AC ← DR, SC ← 0

STA: Store AC

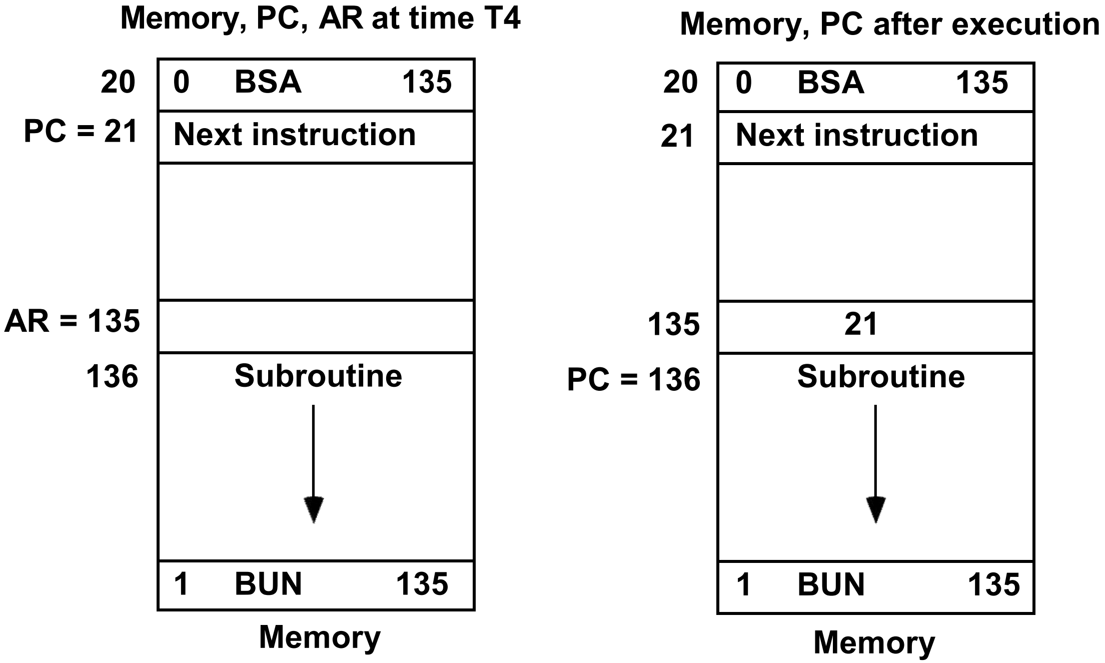
D_3T_4: M[AR] ← AC, SC ← 0

BUN: Branch Unconditionally

D_4T_4: PC ← AR, SC ← 0

BSA: Branch and Save Return Address

M[AR] ← PC, PC ← AR + 1



MEMORY REFERENCE INSTRUCTIONS

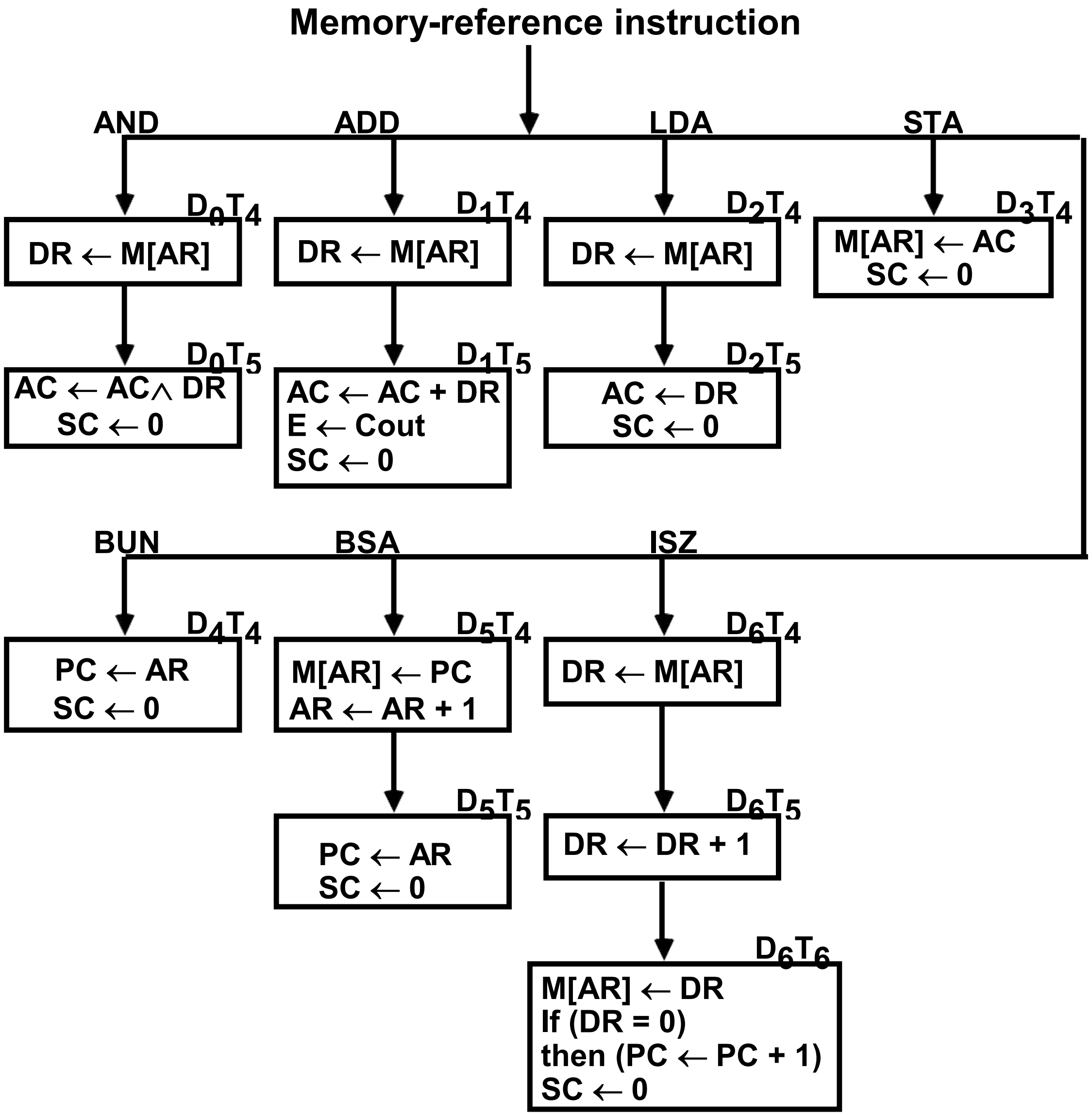
BSA:

D_5T_4: M[AR] $\leftarrow$ PC, AR $\leftarrow$ AR + 1
D_5T_5: PC $\leftarrow$ AR, SC $\leftarrow$ 0

ISZ: Increment and Skip-if-Zero

D_6T_4: DR $\leftarrow$ M[AR]
D_6T_5: DR $\leftarrow$ DR + 1
D_6T_4: M[AR] $\leftarrow$ DR, if (DR = 0) then (PC $\leftarrow$ PC + 1), SC $\leftarrow$ 0

FLOWCHART FOR MEMORY REFERENCE INSTRUCTIONS



Memory-reference instruction

AND

D_0T_4

DR ← M[AR]

D_0T_5

AC ← AC∧ DR
SC ← 0

ADD

D_1T_4

DR ← M[AR]

D_1T_5

AC ← AC + DR
E ← Cout
SC ← 0

LDA

D_2T_4

DR ← M[AR]

D_2T_5

AC ← DR
SC ← 0

STA

D_3T_4

M[AR] ← AC
SC ← 0

BUN

D_4T_4

PC ← AR
SC ← 0

BSA

D_5T_4

M[AR] ← PC
AR ← AR + 1

D_5T_5

PC ← AR
SC ← 0

ISZ

D_6T_4

DR ← M[AR]

D_6T_5

DR ← DR + 1

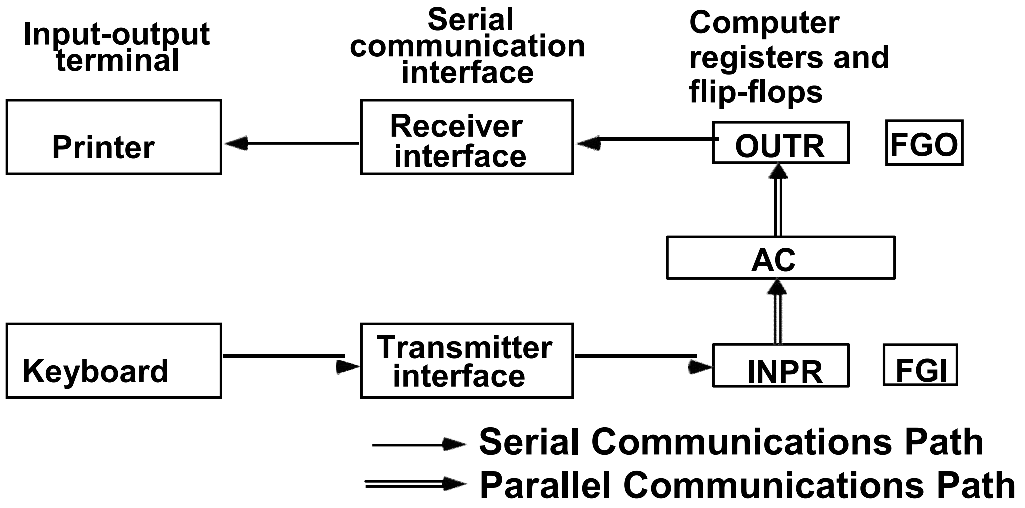
D_6T_6

M[AR] ← DR
If (DR = 0)
then (PC ← PC + 1)
SC ← 0

INPUT-OUTPUT AND INTERRUPT

A Terminal with a keyboard and a Printer

- **Input-Output Configuration**


Input-output terminal
Serial communication interface
Computer registers and flip-flops
Printer
Receiver interface
OUTR
FGO
AC
Keyboard
Transmitter interface
INPR
FGI
Serial Communications Path
Parallel Communications Path

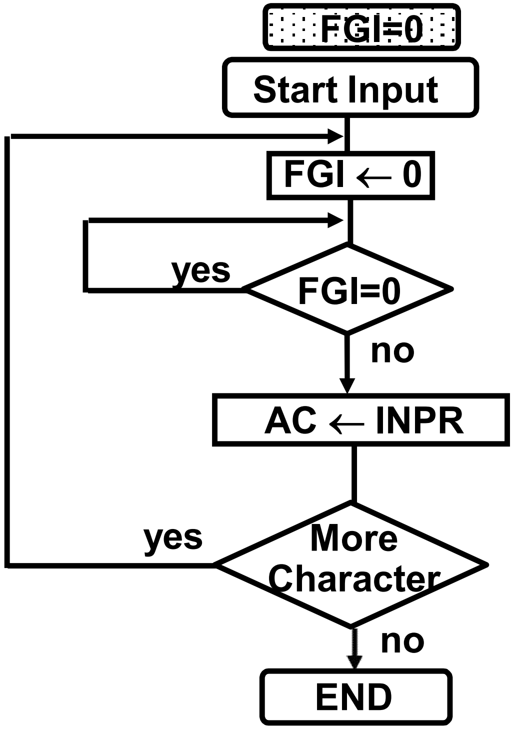
INPR	Input register - 8 bits
OUTR	Output register - 8 bits
FGI	Input flag - 1 bit
FGO	Output flag - 1 bit
IEN	Interrupt enable - 1 bit

- The terminal sends and receives serial information
- The serial info. from the keyboard is shifted into INPR
- The serial info. for the printer is stored in the OUTR
- INPR and OUTR communicate with the terminal serially and with the AC in parallel.
- The flags are needed to *synchronize* the timing difference between I/O device and the computer

PROGRAM CONTROLLED DATA TRANSFER

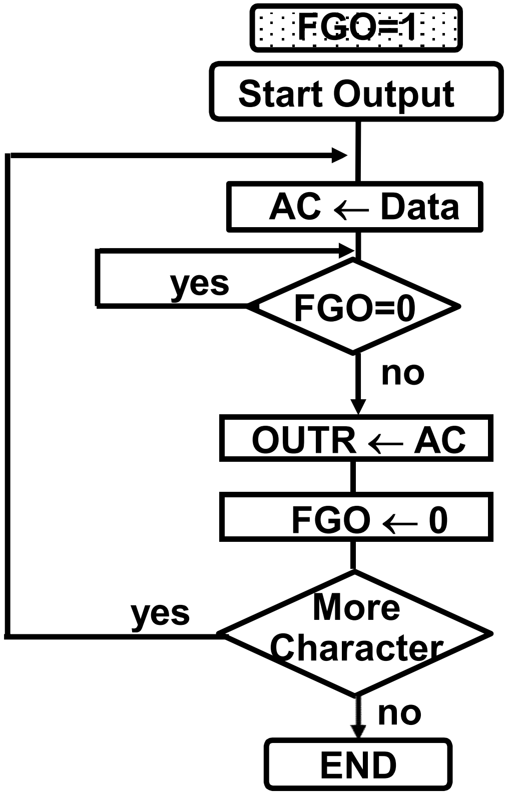
-- CPU --

```
/* Input */        /* Initially FGI = 0 */
   loop:  If FGI = 0 goto loop
              AC ← INPR,  FGI ← 0

/* Output */       /* Initially FGO = 1 */
   loop:  If FGO = 0 goto loop
              OUTR ← AC,  FGO ← 0
```



-- I/O Device --

```
loop: If FGI = 1 goto loop
        INPR ← new data, FGI ← 1

loop: If FGO = 1 goto loop
        consume OUTR, FGO ← 1
```



INPUT-OUTPUT INSTRUCTIONS

$D_7IT_3 = p$

$IR(i) = B_i$, $i = 6, \ldots, 11$

	p:	SC ← 0	Clear SC
INP	pB_{11}:	AC(0-7) ← INPR, FGI ← 0	Input char. to AC
OUT	pB_{10}:	OUTR ← AC(0-7), FGO ← 0	Output char. from AC
SKI	pB_9:	if(FGI = 1) then (PC ← PC + 1)	Skip on input flag
SKO	pB_8:	if(FGO = 1) then (PC ← PC + 1)	Skip on output flag
ION	pB_7:	IEN ← 1	Interrupt enable on
IOF	pB_6:	IEN ← 0	Interrupt enable off

PROGRAM-CONTROLLED INPUT/OUTPUT

- **Program-controlled I/O**
  - **- Continuous CPU involvement**
    I/O takes valuable CPU time
  - **- CPU slowed down to I/O speed**
  - **- Simple**
  - **- Least hardware**

Input

```
LOOP,     SKI    DEV
          BUN   LOOP
          INP    DEV
```

Output

```
LOOP,     LDA   DATA
LOP,      SKO   DEV
          BUN   LOP
          OUT   DEV
```

INTERRUPT INITIATED INPUT/OUTPUT

- Open communication only when some data has to be passed --> *interrupt*.

- The I/O interface, instead of the CPU, monitors the I/O device.

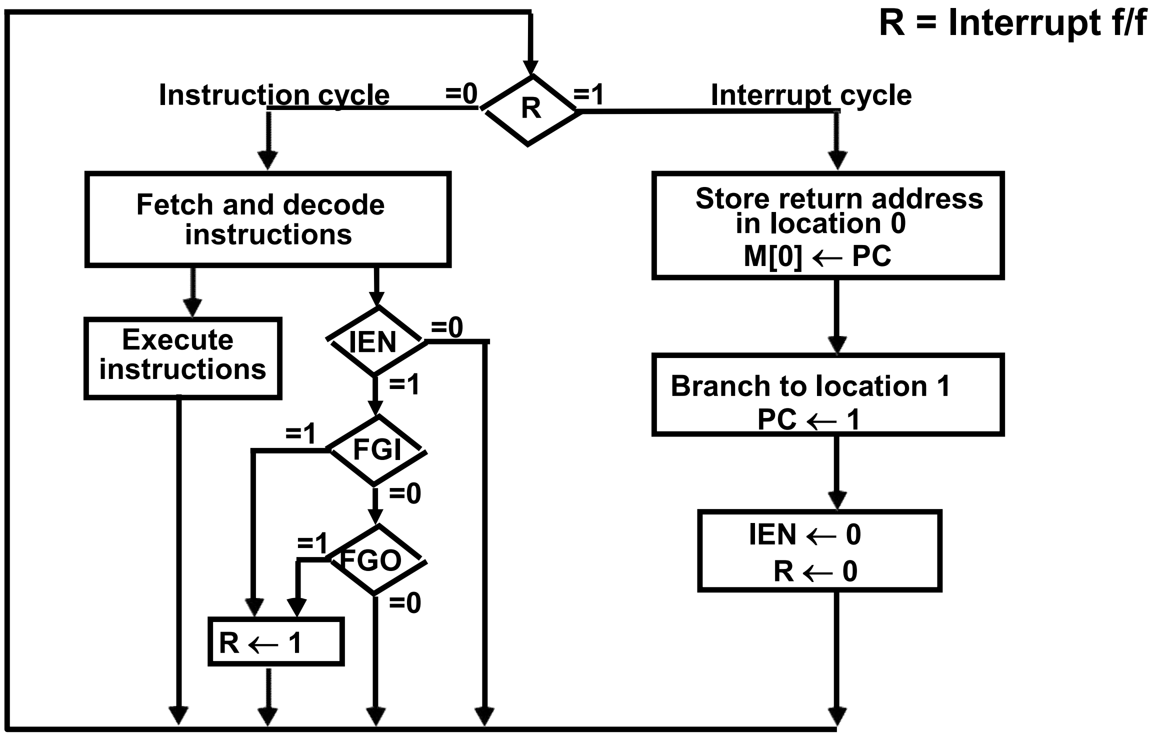
- When the interface founds that the I/O device is ready for data transfer, it generates an interrupt request to the CPU

- Upon detecting an interrupt, the CPU stops momentarily the task it is doing, branches to the service routine to process the data transfer, and then returns to the task it was performing.

*** IEN (Interrupt-enable flip-flop)**

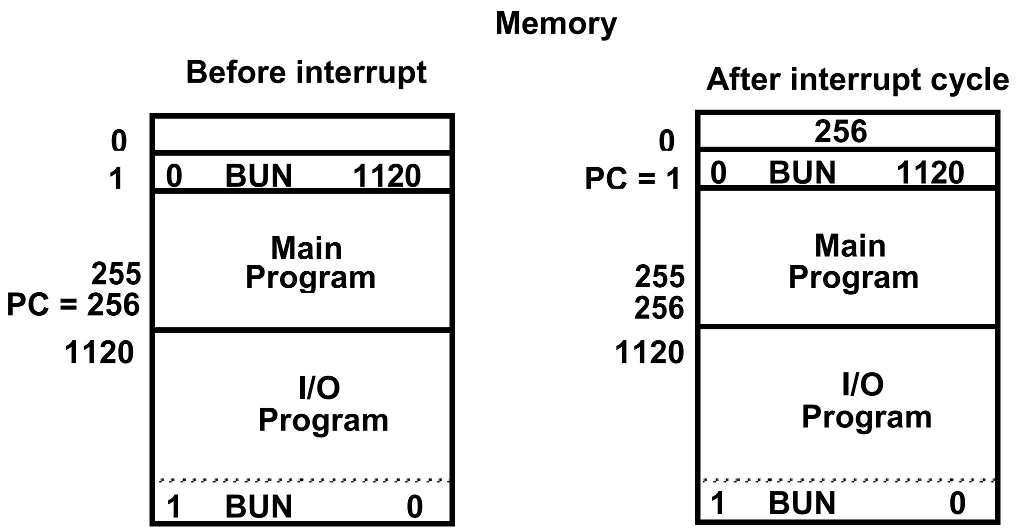
- can be set and cleared by instructions
- when cleared, the computer cannot be interrupted

FLOWCHART FOR INTERRUPT CYCLE



- The interrupt cycle is a HW implementation of a branch and save return address operation.
- At the beginning of the next instruction cycle, the instruction that is read from memory is in address 1.
- At memory address 1, the programmer must store a branch instruction that sends the control to an interrupt service routine
- The instruction that returns the control to the original program is "indirect BUN 0"

REGISTER TRANSFER OPERATIONS IN INTERRUPT CYCLE



Register Transfer Statements for Interrupt Cycle

- R F/F ← 1 if IEN (FGI + FGO)$T_0'T_1'T_2'$

$$\Leftrightarrow T_0'T_1'T_2' (IEN)(FGI + FGO): \quad R \leftarrow 1$$

- The fetch and decode phases of the instruction cycle must be modified ➔Replace T_0, T_1, T_2 with $R'T_0$, $R'T_1$, $R'T_2$
- The interrupt cycle :

RT_0: AR ← 0, TR ← PC

RT_1: M[AR] ← TR, PC ← 0

RT_2: PC ← PC + 1, IEN ← 0, R ← 0, SC ← 0

FURTHER QUESTIONS ON INTERRUPT

How can the CPU recognize the device requesting an interrupt ?

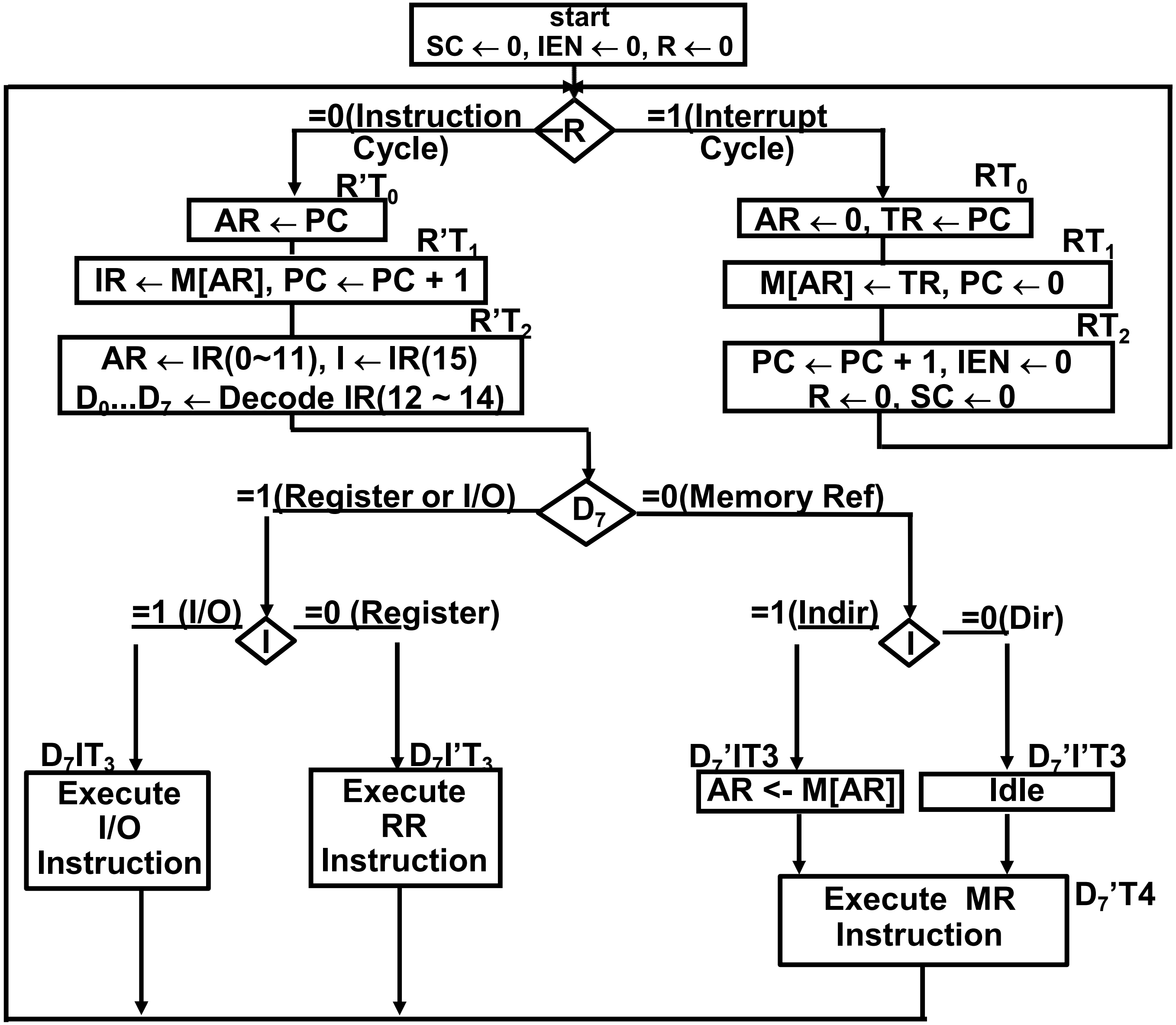
Since different devices are likely to require different interrupt service routines, how can the CPU obtain the starting address of the appropriate routine in each case ?

Should any device be allowed to interrupt the CPU while another interrupt is being serviced ?

How can the situation be handled when two or more interrupt requests occur simultaneously ?

COMPLETE COMPUTER DESCRIPTION

Flowchart of Operations



start
SC ← 0, IEN ← 0, R ← 0

R

=0(Instruction Cycle)

$R'T_0$
AR ← PC

$R'T_1$
IR ← M[AR], PC ← PC + 1

$R'T_2$
AR ← IR(0~11), I ← IR(15)
$D_0...D_7$ ← Decode IR(12 ~ 14)

=1(Interrupt Cycle)

RT_0
AR ← 0, TR ← PC

RT_1
M[AR] ← TR, PC ← 0

RT_2
PC ← PC + 1, IEN ← 0
R ← 0, SC ← 0

D_7

=1(Register or I/O)

I

=1 (I/O)

D_7IT_3
Execute I/O Instruction

=0 (Register)

$D_7I'T_3$
Execute RR Instruction

=0(Memory Ref)

I

=1(Indir)

D_7'IT3
AR <- M[AR]

=0(Dir)

D_7'I'T3
Idle

Execute MR Instruction
D_7'T4

COMPLETE COMPUTER DESCRIPTION

Microoperations

Fetch	**$R'T_0$:**	**AR ← PC**
	$R'T_1$:	**IR ← M[AR], PC ← PC + 1**
Decode	**$R'T_2$:**	**D0, ..., D7 ← Decode IR(12 ~ 14), AR ← IR(0 ~ 11), I ← IR(15)**
Indirect	**$D_7'IT_3$:**	**AR ← M[AR]**
Interrupt		
$T_0'T_1'T_2'$(IEN)(FGI + FGO):		**R ← 1**
	RT_0:	**AR ← 0, TR ← PC**
	RT_1:	**M[AR] ← TR, PC ← 0**
	RT_2:	**PC ← PC + 1, IEN ← 0, R ← 0, SC ← 0**
Memory-Reference		
AND	**D_0T_4:**	**DR ← M[AR]**
	D_0T_5:	**AC ← AC ∧ DR, SC ← 0**
ADD	**D_1T_4:**	**DR ← M[AR]**
	D_1T_5:	**AC ← AC + DR, E ← C_{out}, SC ← 0**
LDA	**D_2T_4:**	**DR ← M[AR]**
	D_2T_5:	**AC ← DR, SC ← 0**
STA	**D_3T_4:**	**M[AR] ← AC, SC ← 0**
BUN	**D_4T_4:**	**PC ← AR, SC ← 0**
BSA	**D_5T_4:**	**M[AR] ← PC, AR ← AR + 1**
	D_5T_5:	**PC ← AR, SC ← 0**
ISZ	**D_6T_4:**	**DR ← M[AR]**
	D_6T_5:	**DR ← DR + 1**
	D_6T_6:	**M[AR] ← DR, if(DR=0) then (PC ← PC + 1), SC ← 0**

COMPLETE COMPUTER DESCRIPTION

Microoperations

Register-Reference		
	$D_7I'T_3 = r$	(Common to all register-reference instr)
	$IR(i) = B_i$	(i = 0,1,2, ..., 11)
	r:	SC ← 0
CLA	rB_{11}:	AC ← 0
CLE	rB_{10}:	E ← 0
CMA	rB_9:	AC ← AC′
CME	rB_8:	E ← E′
CIR	rB_7:	AC ← shr AC, AC(15) ← E, E ← AC(0)
CIL	rB_6:	AC ← shl AC, AC(0) ← E, E ← AC(15)
INC	rB_5:	AC ← AC + 1
SPA	rB_4:	If(AC(15) =0) then (PC ← PC + 1)
SNA	rB_3:	If(AC(15) =1) then (PC ← PC + 1)
SZA	rB_2:	If(AC = 0) then (PC ← PC + 1)
SZE	rB_1:	If(E=0) then (PC ← PC + 1)
HLT	rB_0:	S ← 0
Input-Output	$D_7IT_3 = p$	(Common to all input-output instructions)
	$IR(i) = B_i$	(i = 6,7,8,9,10,11)
	p:	SC ← 0
INP	pB_{11}:	AC(0-7) ← INPR, FGI ← 0
OUT	pB_{10}:	OUTR ← AC(0-7), FGO ← 0
SKI	pB_9:	If(FGI=1) then (PC ← PC + 1)
SKO	pB_8:	If(FGO=1) then (PC ← PC + 1)
ION	pB_7:	IEN ← 1
IOF	pB_6:	IEN ← 0

DESIGN OF BASIC COMPUTER(BC)

Hardware Components of BC

A memory unit: 4096 x 16.
Registers:
AR, PC, DR, AC, IR, TR, OUTR, INPR, and SC
Flip-Flops(Status):
I, S, E, R, IEN, FGI, and FGO
Decoders: a 3x8 Opcode decoder
a 4x16 timing decoder
Common bus: 16 bits
Control logic gates:
Adder and Logic circuit: Connected to AC

Control Logic Gates

- **Input Controls of the nine registers**
- **Read and Write Controls of memory**
- **Set, Clear, or Complement Controls of the flip-flops**
- **S_2, S_1, S_0 Controls to select a register for the bus**
- **AC, and Adder and Logic circuit**

CONTROL OF REGISTERS AND MEMORY

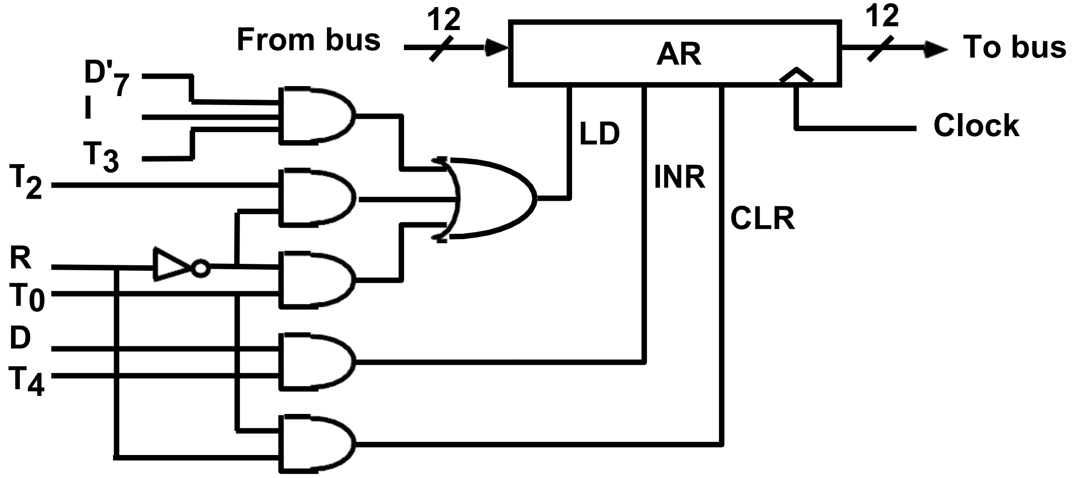
Address Register; AR

Scan all of the register transfer statements that change the content of AR:

$R'T_0$:	$AR \leftarrow PC$	LD(AR)
$R'T_2$:	$AR \leftarrow IR(0\text{-}11)$	LD(AR)
D'_7IT_3:	$AR \leftarrow M[AR]$	LD(AR)
RT_0:	$AR \leftarrow 0$	CLR(AR)
D_5T_4:	$AR \leftarrow AR + 1$	INR(AR)

⇩

$$LD(AR) = R'T_0 + R'T_2 + D'_7IT_3$$
$$CLR(AR) = RT_0$$
$$INR(AR) = D_5T_4$$

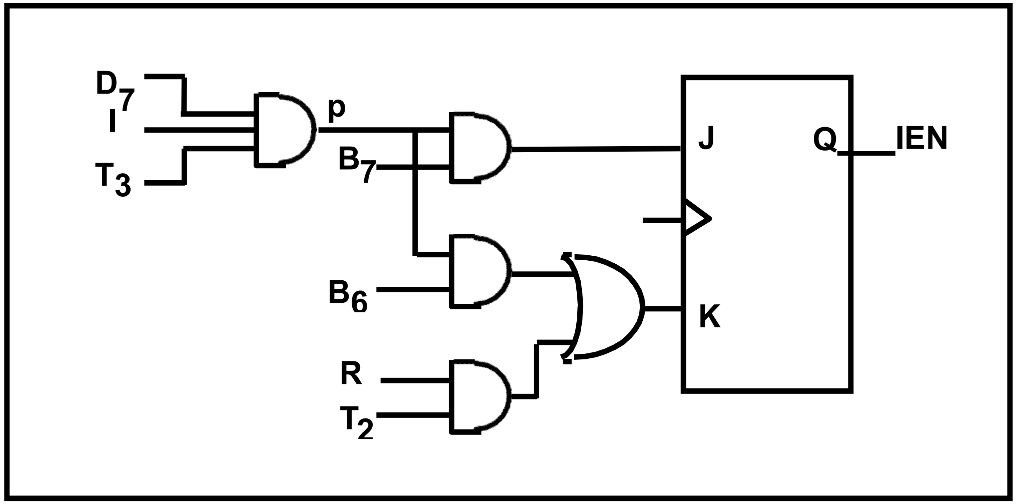


CONTROL OF FLAGS

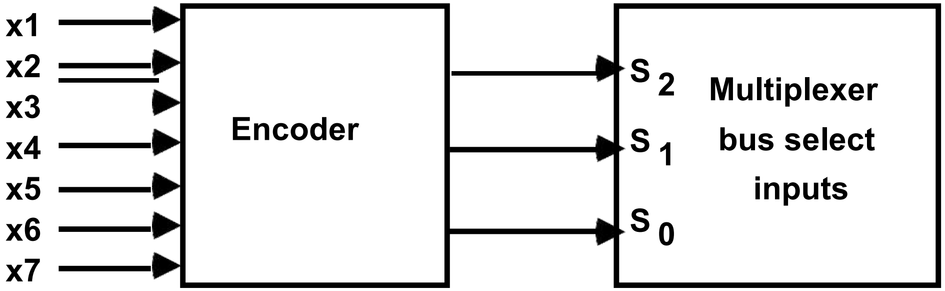
IEN: Interrupt Enable Flag

pB_7: IEN ← 1 (I/O Instruction)
pB_6: IEN ← 0 (I/O Instruction)
RT_2: IEN ← 0 (Interrupt)

$p = D_7IT_3$ (Input/Output Instruction)



CONTROL OF COMMON BUS



x1	x2	x3	x4	x5	x6	x7	S2	S1	S0	selected register
0	0	0	0	0	0	0	0	0	0	none
1	0	0	0	0	0	0	0	0	1	AR
0	1	0	0	0	0	0	0	1	0	PC
0	0	1	0	0	0	0	0	1	1	DR
0	0	0	1	0	0	0	1	0	0	AC
0	0	0	0	1	0	0	1	0	1	IR
0	0	0	0	0	1	0	1	1	0	TR
0	0	0	0	0	0	1	1	1	1	Memory

For AR

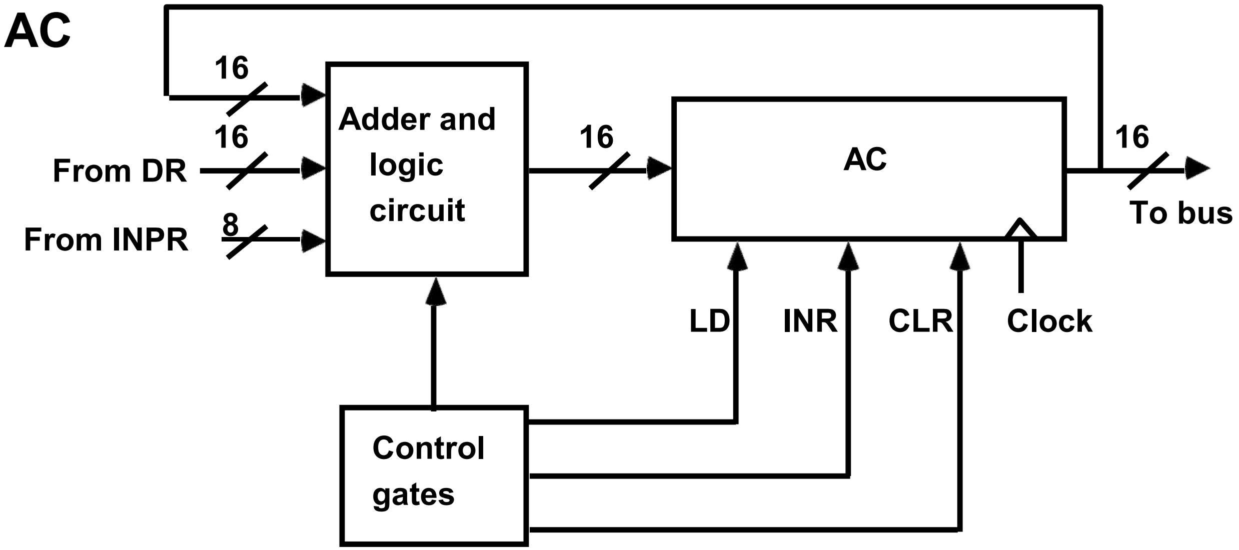
D_4T_4: PC ← AR
D_5T_5: PC ← AR

⇩

$x1 = D_4T_4 + D_5T_5$

DESIGN OF ACCUMULATOR LOGIC

Circuits associated with AC

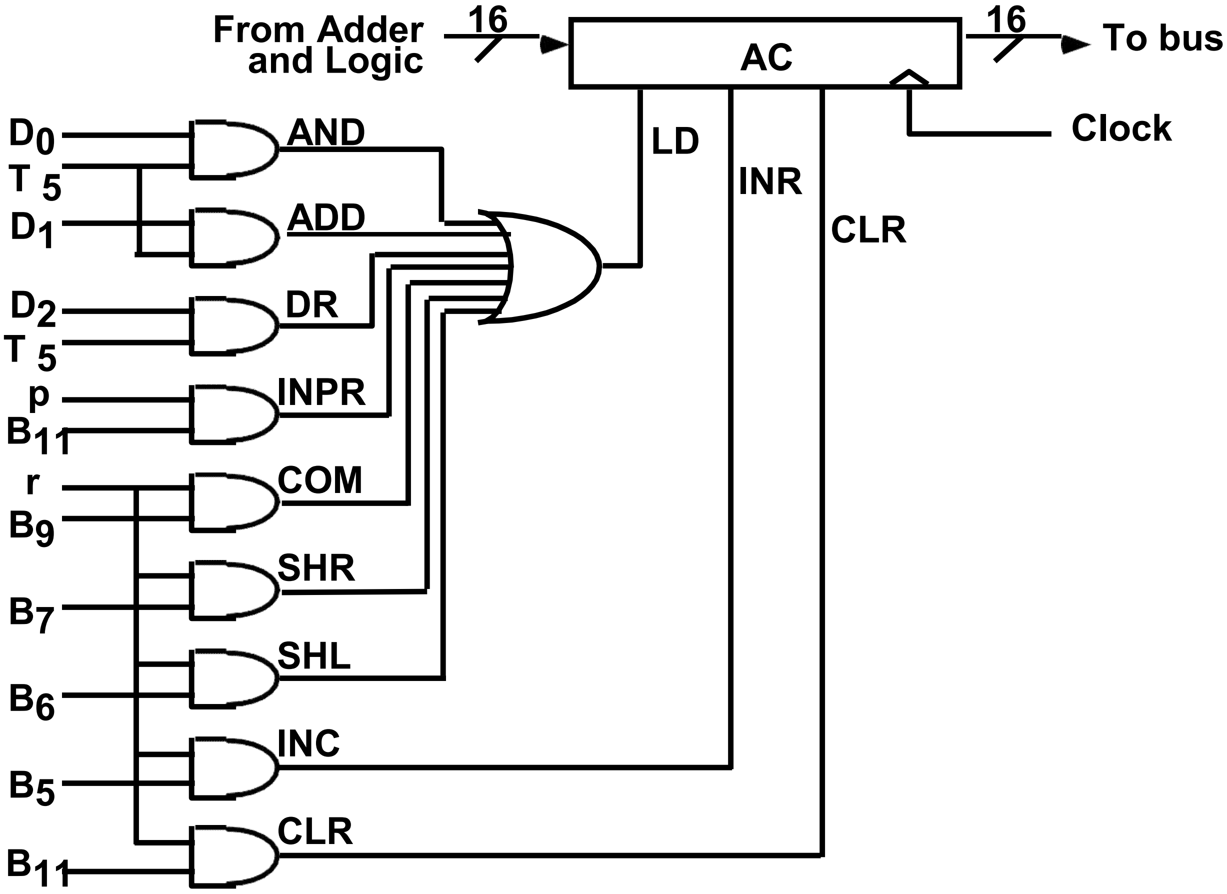


All the statements that change the content of AC

D_0T_5:	$AC \leftarrow AC \wedge DR$	AND with DR
D_1T_5:	$AC \leftarrow AC + DR$	Add with DR
D_2T_5:	$AC \leftarrow DR$	Transfer from DR
pB_{11}:	$AC(0\text{-}7) \leftarrow INPR$	Transfer from INPR
rB_9:	$AC \leftarrow AC'$	Complement
rB_7 :	$AC \leftarrow$ shr AC, $AC(15) \leftarrow E$	Shift right
rB_6 :	$AC \leftarrow$ shl AC, $AC(0) \leftarrow E$	Shift left
rB_{11} :	$AC \leftarrow 0$	Clear
rB_5 :	$AC \leftarrow AC + 1$	Increment

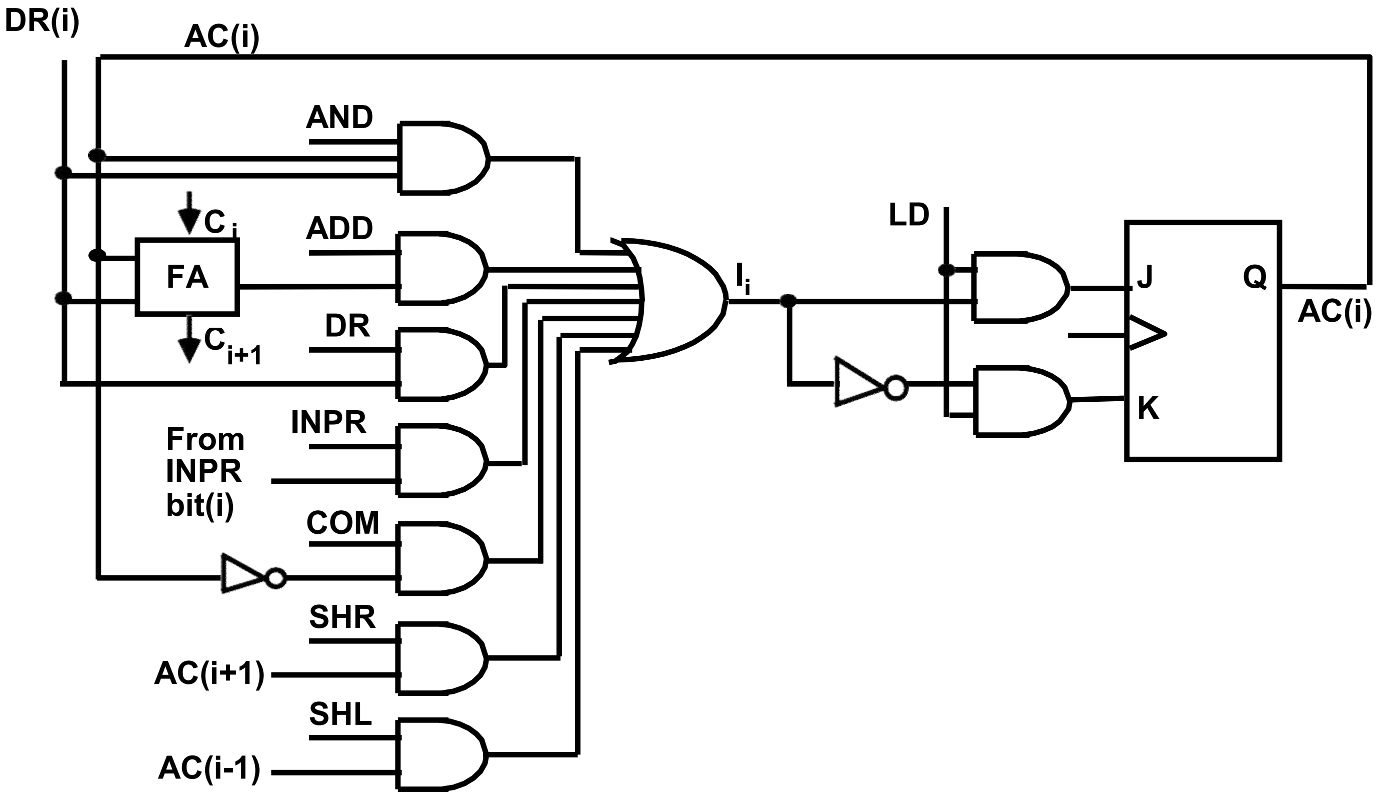
CONTROL OF AC REGISTER

Gate structures for controlling the LD, INR, and CLR of AC



ALU (ADDER AND LOGIC CIRCUIT)

One stage of Adder and Logic circuit



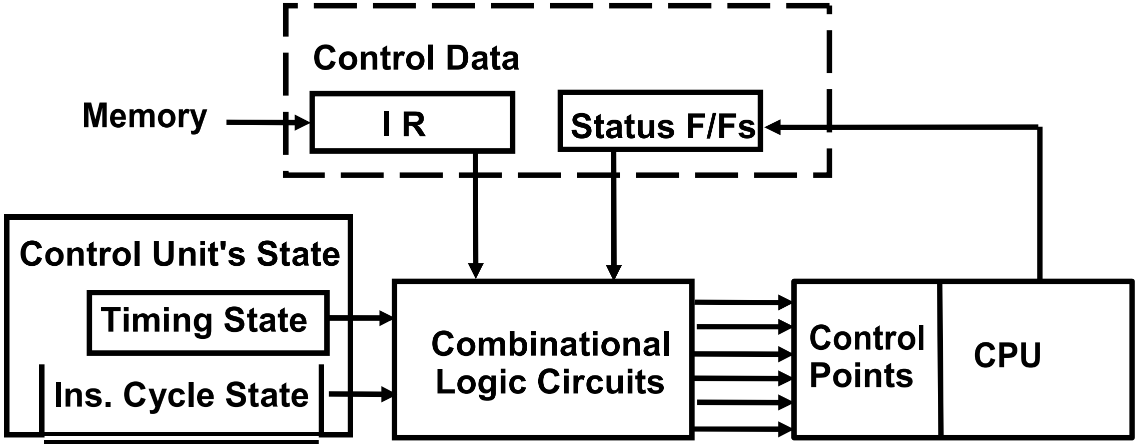
MICROPROGRAMMED CONTROL

- **Control Memory**
- **Sequencing Microinstructions**
- **Micro-program Example**
- **Design of Control Unit**
- **Microinstruction Format**
- **Nano storage and Nano program**

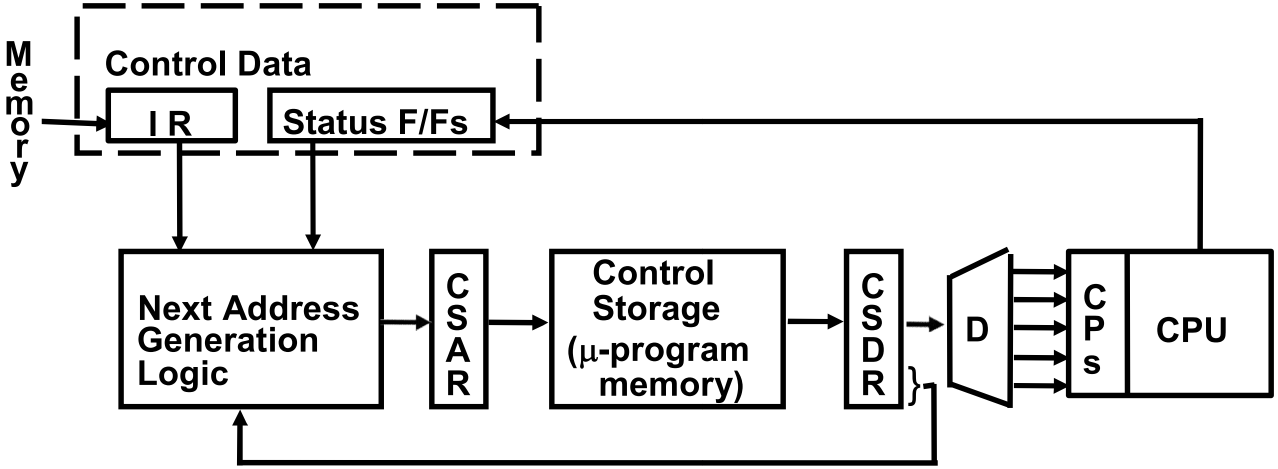
COMPARISON OF CONTROL UNIT IMPLEMENTATIONS

Control Unit Implementation

Combinational Logic Circuits (Hard-wired)



Microprogram


Memory
Control Data
I R
Status F/Fs
Next Address Generation Logic
CSAR
Control Storage (μ-program memory)
CSDR
D
CPs
CPU

TERMINOLOGY

Micro-program

- Program stored in memory that generates all the control signals required to execute the instruction set correctly
- Consists of microinstructions

Microinstruction

- Contains a control word and a sequencing word

  Control Word - All the control information required for one clock cycle
  Sequencing Word - Information needed to decide the next microinstruction address

- Vocabulary to write a micro-program

Control Memory(Control Storage: CS)

- Storage in the micro-programmed control unit to store the micro-program

Writeable Control Memory(Writeable Control Storage:WCS)

- CS whose contents can be modified

  -> Allows the micro program can be changed
  -> Instruction set can be changed or modified

Dynamic Microprogramming

- Computer system whose control unit is implemented with a micro-program in WCS
- Micro-program can be changed by a systems programmer or a user

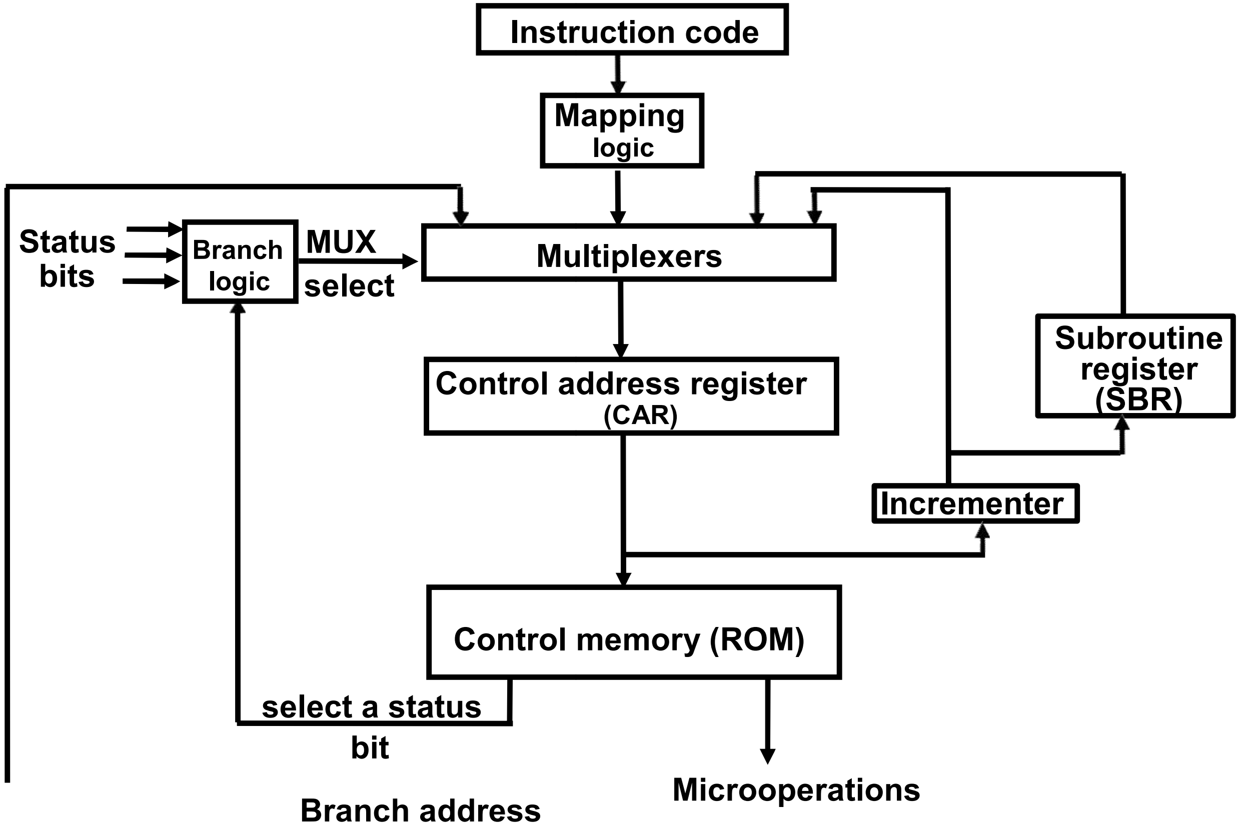
TERMINOLOGY

Sequencer (Micro-program Sequencer)

A Micro-program Control Unit that determines the Microinstruction Address to be executed in the next clock cycle

- In-line Sequencing
- Branch
- Conditional Branch
- Subroutine
- Loop
- Instruction OP-code mapping

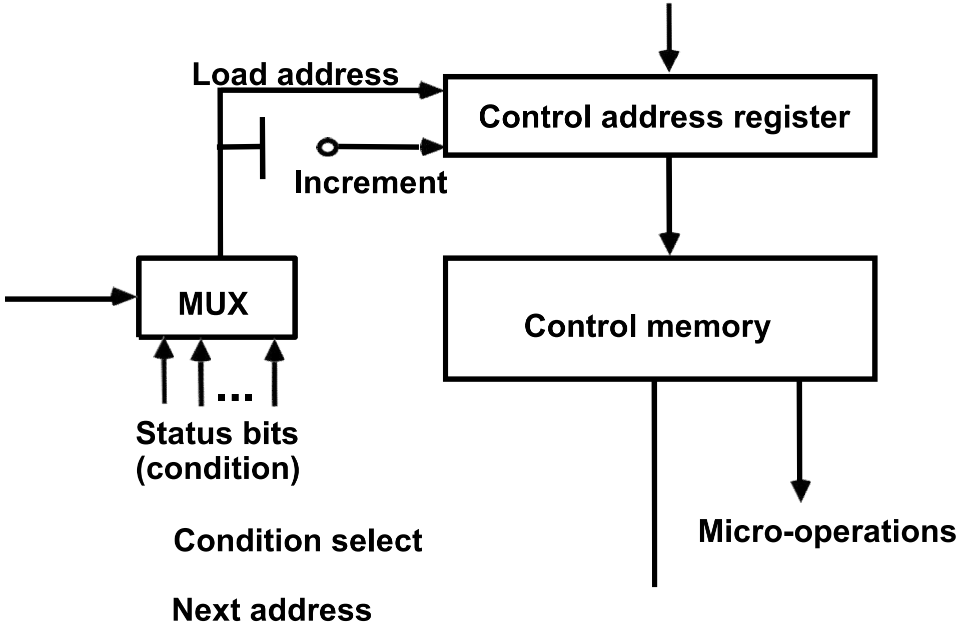
MICROINSTRUCTION SEQUENCING



Sequencing Capabilities Required in a Control Storage

- Incrementing of the control address register
- Unconditional and conditional branches
- A mapping process from the bits of the machine instruction to an address for control memory
- A facility for subroutine call and return

CONDITIONAL BRANCH



Conditional Branch

If *Condition* is true, then *Branch* (address from the next address field of the current microinstruction)
else *Fall Through*

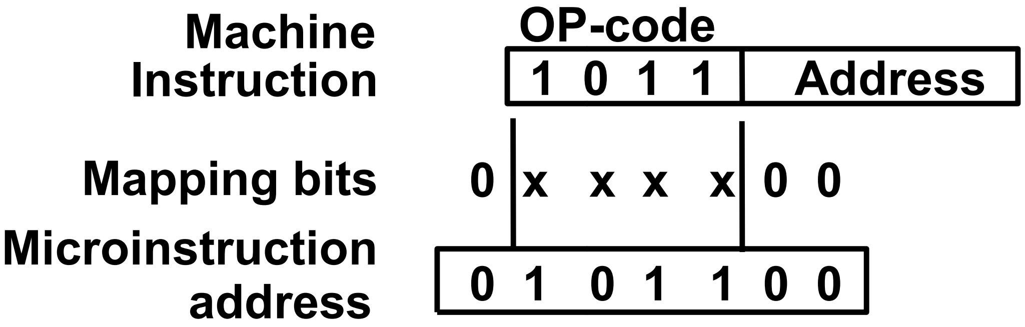
Conditions to Test: O(overflow), N(negative), Z(zero), C(carry), etc.

Unconditional Branch

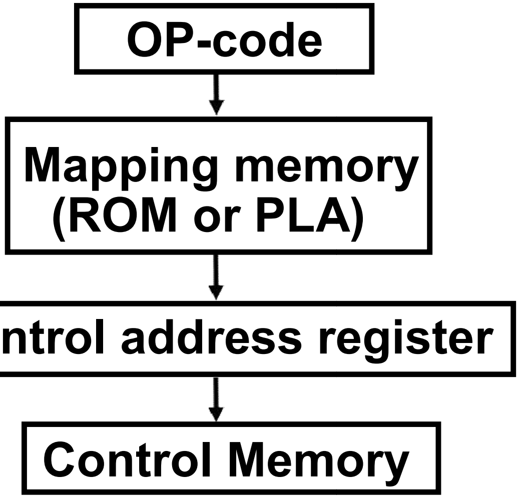
Fixing the value of one status bit at the input of the multiplexer to 1

MAPPING OF INSTRUCTIONS TO MICROROUTINES

Mapping from the OP-code of an instruction to the address of the Microinstruction which is the starting microinstruction of its execution micro-program.

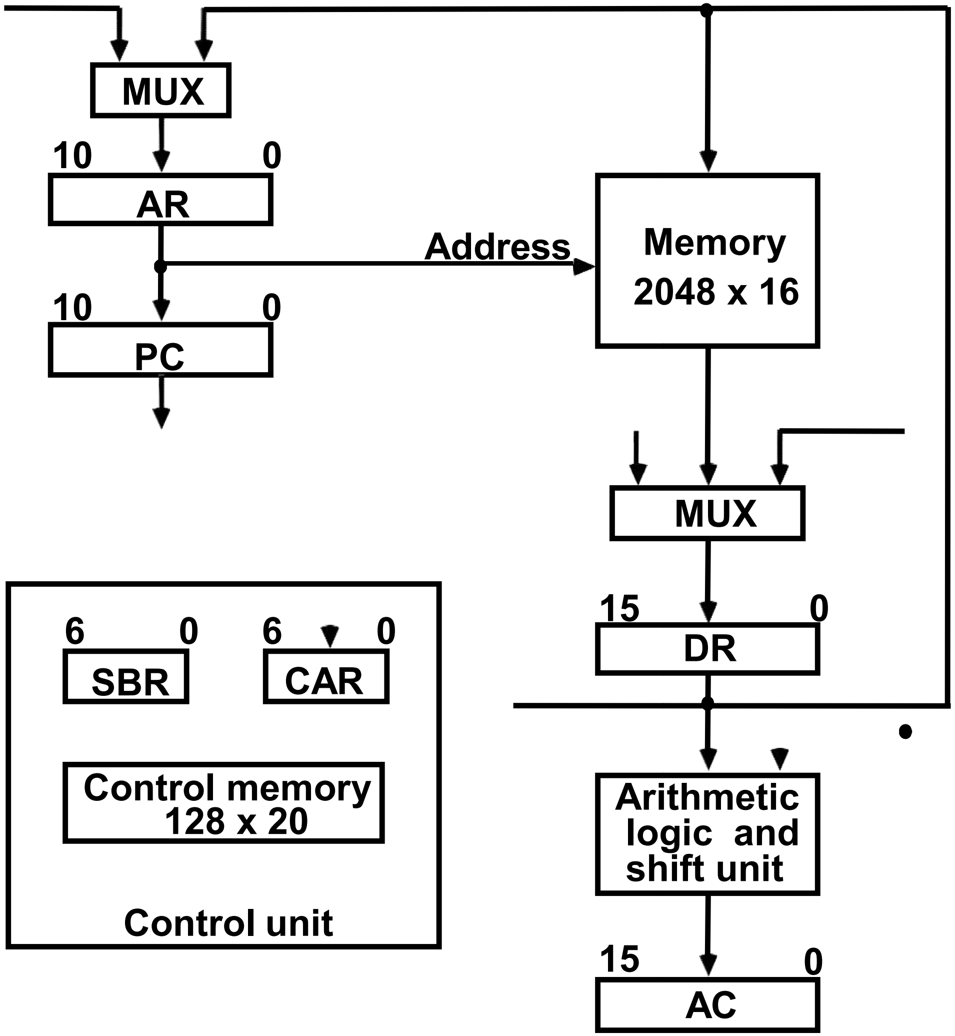


Mapping function implemented by ROM or PLA



MICROPROGRAM EXAMPLE

Computer Configuration



MACHINE INSTRUCTION FORMAT

Machine instruction format

15	14 11	10 0
I	Opcode	Address

Sample machine instructions

Symbol	OP-code	Description
ADD	0000	AC ← AC + M[EA]
BRANCH	0001	if (AC < 0) then (PC ← EA)
STORE	0010	M[EA] ← AC
EXCHANGE	0011	AC ← M[EA], M[EA] ← AC

EA is the effective address

Microinstruction Format

3	3	3	2	2	7
F1	F2	F3	CD	BR	AD

F1, F2, F3: Microoperation fields
CD: Condition for branching
BR: Branch field
AD: Address field

SYMBOLIC MICROINSTRUCTIONS

- Symbols are used in microinstructions as in assembly language
- A symbolic micro-program can be translated into its binary equivalent by a micro-program assembler**.**

Sample Format

five fields: label; micro-ops; CD; BR; AD

Label: may be empty or may specify a symbolic address terminated with a colon

Micro-ops: consists of one, two, or three symbols separated by commas

CD: one of {U, I, S, Z}, where

U: Unconditional Branch
I: Indirect address bit
S: Sign of AC
Z: Zero value in AC

BR: one of {JMP, CALL, RET, MAP}

AD: one of {Symbolic address, NEXT, empty}

SYMBOLIC MICROPROGRAM - FETCH ROUTINE

During FETCH, Read an instruction from memory and decode the instruction and update PC

Sequence of microoperations in the fetch cycle:

AR ← PC
DR ← M[AR], PC ← PC + 1
AR ← DR(0-10), CAR(2-5) ← DR(11-14), CAR(0,1,6) ← 0

Symbolic microprogram for the fetch cycle:

```
            ORG 64
FETCH:      PCTAR            U   JMP   NEXT
            READ, INCPC      U   JMP   NEXT
            DRTAR            U   MAP
```

Binary equivalents translated by an assembler

Binary address	F1	F2	F3	CD	BR	AD
1000000	110	000	000	00	00	1000001
1000001	000	100	101	00	00	1000010
1000010	101	000	000	00	11	0000000

SYMBOLIC MICROPROGRAM

- **Control Storage: 128 20-bit words**
- **The first 64 words: Routines for the 16 machine instructions**
- **The last 64 words: Used for other purpose (e.g., fetch routine and other subroutines)**
- **Mapping: OP-code XXXX into 0XXXX00, the first address for the 16 routines are 0(0 0000 00), 4(0 0001 00), 8, 12, 16, 20, ..., 60**

Partial Symbolic Microprogram

Label	Microops	CD	BR	AD
	ORG 0			
ADD:	NOP	I	CALL	INDRCT
	READ	U	JMP	NEXT
	ADD	U	JMP	FETCH
	ORG 4			
BRANCH:	NOP	S	JMP	OVER
	NOP	U	JMP	FETCH
OVER:	NOP	I	CALL	INDRCT
	ARTPC	U	JMP	FETCH
	ORG 8			
STORE:	NOP	I	CALL	INDRCT
	ACTDR	U	JMP	NEXT
	WRITE	U	JMP	FETCH
	ORG 12			
EXCHANGE:	NOP	I	CALL	INDRCT
	READ	U	JMP	NEXT
	ACTDR, DRTAC	U	JMP	NEXT
	WRITE	U	JMP	FETCH
	ORG 64			
FETCH:	PCTAR	U	JMP	NEXT
	READ, INCPC	U	JMP	NEXT
	DRTAR	U	MAP	
INDRCT:	READ	U	JMP	NEXT
	DRTAR	U	RET	

BINARY MICROPROGRAM

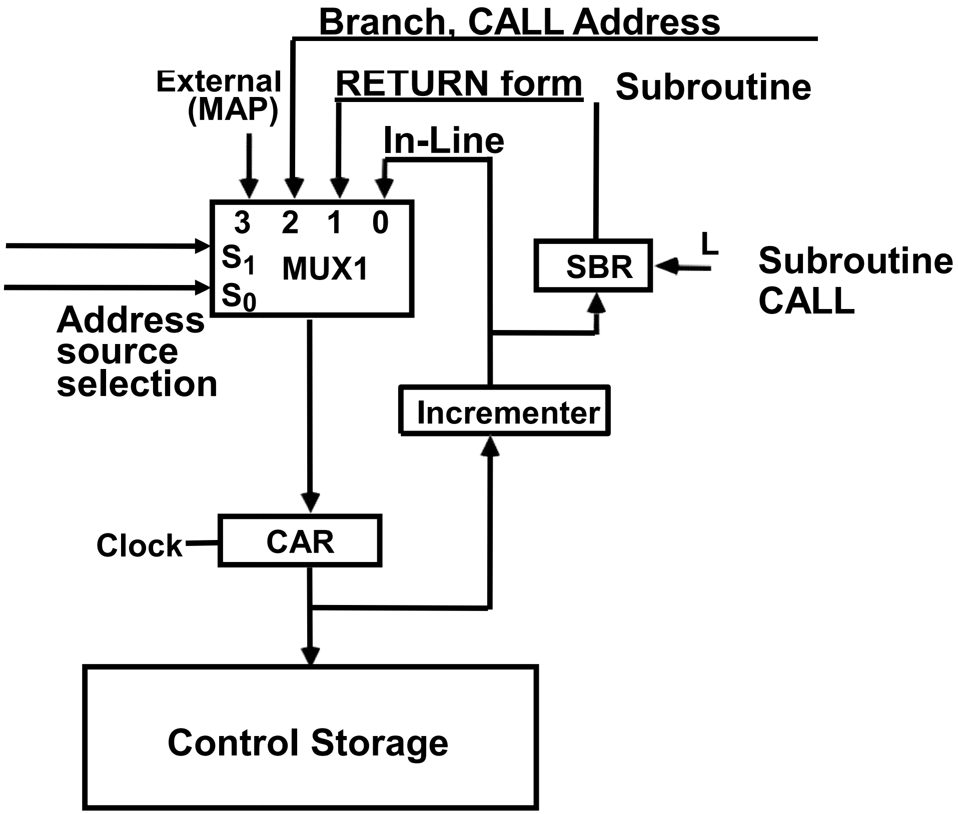
	Address		Binary Microinstruction					
Micro Routine	Decimal	Binary	F1	F2	F3	CD	BR	AD
ADD	0	0000000	000	000	000	01	01	1000011
	1	0000001	000	100	000	00	00	0000010
	2	0000010	001	000	000	00	00	1000000
	3	0000011	000	000	000	00	00	1000000
BRANCH	4	0000100	000	000	000	10	00	0000110
	5	0000101	000	000	000	00	00	1000000
	6	0000110	000	000	000	01	01	1000011
	7	0000111	000	000	110	00	00	1000000
STORE	8	0001000	000	000	000	01	01	1000011
	9	0001001	000	101	000	00	00	0001010
	10	0001010	111	000	000	00	00	1000000
	11	0001011	000	000	000	00	00	1000000
EXCHANGE	12	0001100	000	000	000	01	01	1000011
	13	0001101	001	000	000	00	00	0001110
	14	0001110	100	101	000	00	00	0001111
	15	0001111	111	000	000	00	00	1000000
FETCH	64	1000000	110	000	000	00	00	1000001
	65	1000001	000	100	101	00	00	1000010
	66	1000010	101	000	000	00	11	0000000
INDRCT	67	1000011	000	100	000	00	00	1000100
	68	1000100	101	000	000	00	10	0000000

This microprogram can be implemented using ROM

MICROPROGRAM SEQUENCER

- NEXT MICROINSTRUCTION ADDRESS LOGIC -

S_1S_0	Address Source
00	CAR + 1, In-Line
01	SBR RETURN
10	CS(AD), Branch or CALL
11	MAP

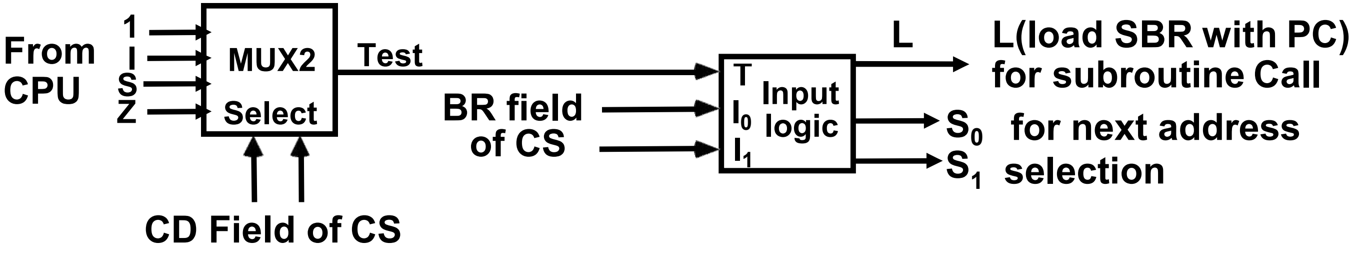


MUX-1 selects an address from one of four sources and routes it into a CAR

- In-Line Sequencing → CAR + 1
- Branch, Subroutine Call → CS(AD)
- Return from Subroutine → Output of SBR
- New Machine instruction → MAP

MICROPROGRAM SEQUENCER

- CONDITION AND BRANCH CONTROL -

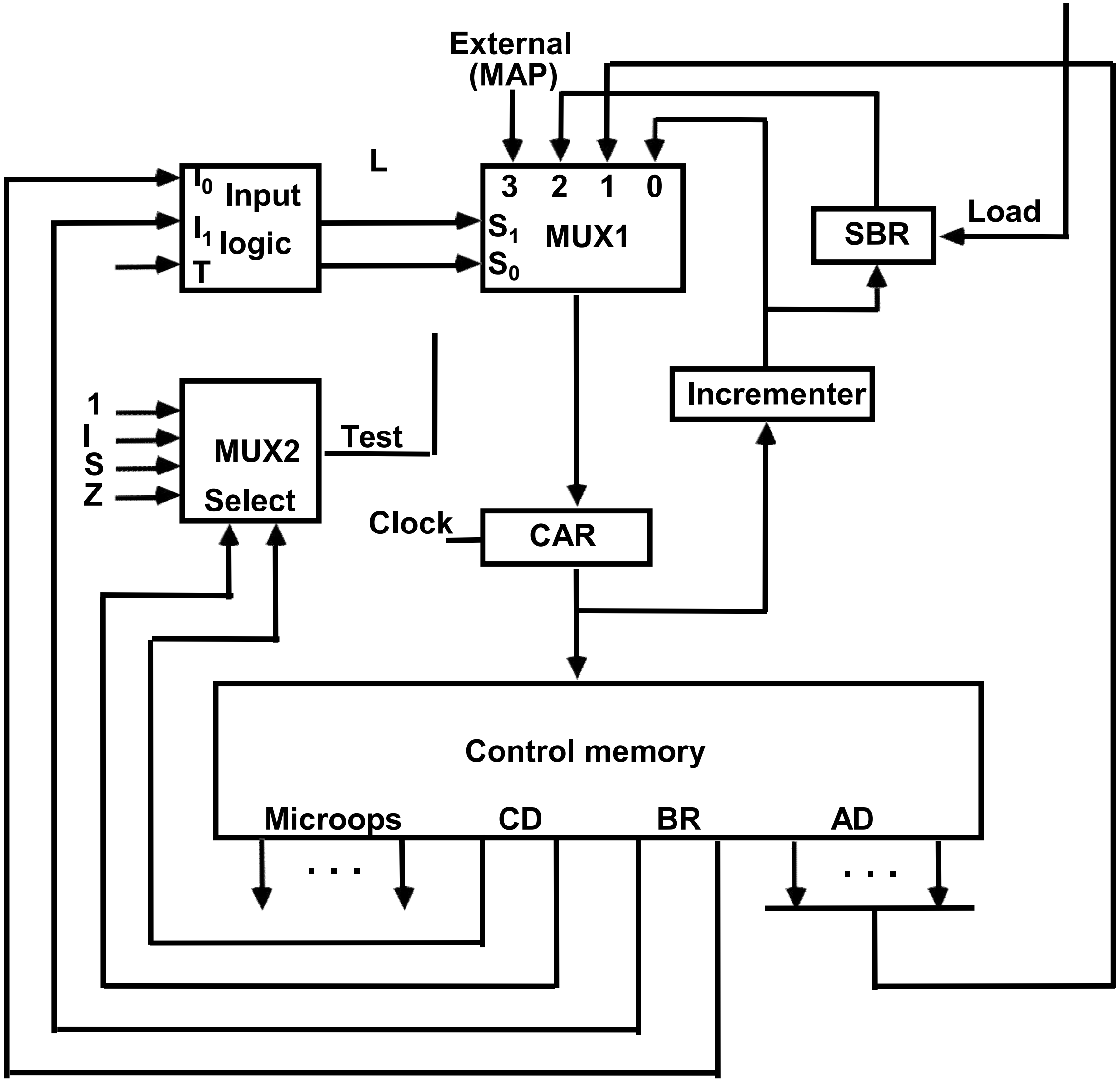


Input Logic

I_0I_1T	Meaning	Source of Address	S_1S_0	L
000	In-Line	CAR+1	00	0
001	JMP	CS(AD)	10	0
010	In-Line	CAR+1	00	0
011	CALL	CS(AD) and SBR <- CAR+1	10	1
10x	RET	SBR	01	0
11x	MAP	DR(11-14)	11	0

$$S_0 = I_0$$
$$S_1 = I_0I_1 + I_0'T$$
$$L = I_0'I_1T$$

MICROPROGRAM SEQUENCER



MICROINSTRUCTION FORMAT

Information in a Microinstruction

- Control Information
- Sequencing Information
- Constant

  Information which is useful when feeding into the system

These information needs to be organized in some way for

- Efficient use of the microinstruction bits
- Fast decoding

Field Encoding

- Encoding the microinstruction bits
- Encoding slows down the execution speed due to the decoding delay
- Encoding also reduces the flexibility due to the decoding hardware

HORIZONTAL AND VERTICAL MICROINSTRUCTION FORMAT

Horizontal Microinstructions

Each bit directly controls each micro-operation or each control point

Horizontal implies a long microinstruction word

Advantages: Can control a variety of components operating in parallel.

--> Advantage of efficient hardware utilization

Disadvantages: Control word bits are not fully utilized

--> CS becomes large --> Costly

Vertical Microinstructions

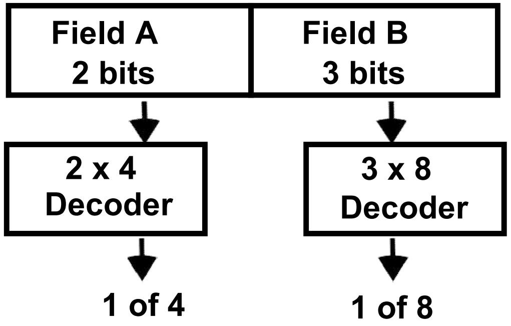
A microinstruction format that is not horizontal

Vertical implies a short microinstruction word

Encoded Microinstruction fields

--> Needs decoding circuits for one or two levels of decoding

One-level decoding



Field A 2 bits

Field B 3 bits

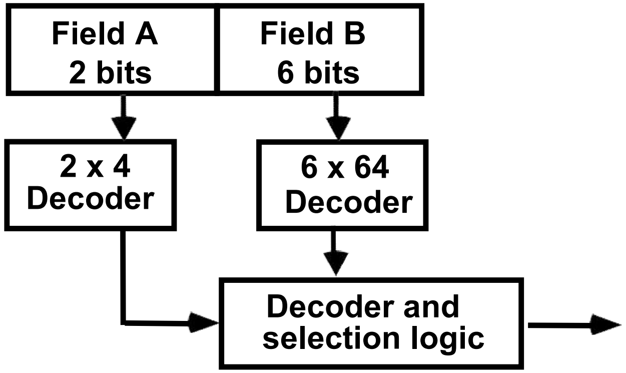
2 x 4 Decoder

3 x 8 Decoder

1 of 4

1 of 8

Two-level decoding



Field A 2 bits

Field B 6 bits

2 x 4 Decoder

6 x 64 Decoder

Decoder and selection logic

NANOSTORAGE AND NANOINSTRUCTION

The decoder circuits in a vertical microprogram
storage organization can be replaced by a ROM
=> Two levels of control storage
First level - *Control Storage*
Second level - *Nano Storage*

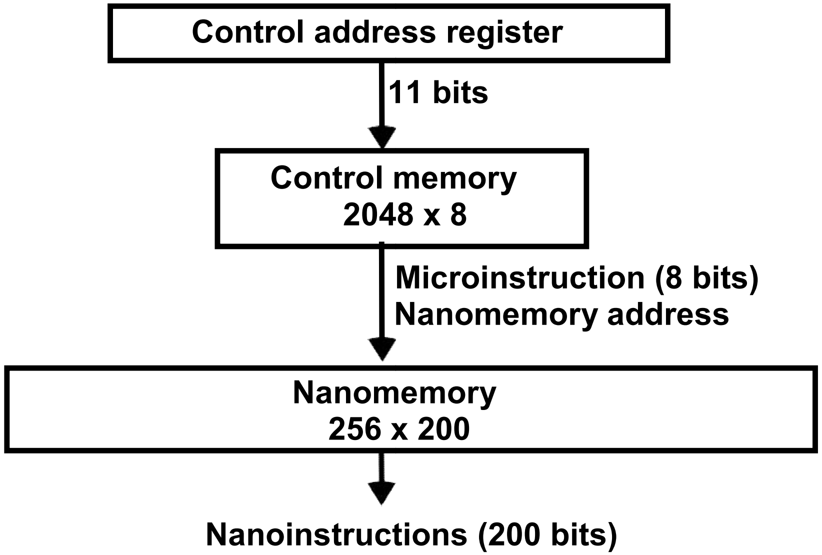
Two-level microprogram

First level
-*Vertical* format Microprogram
Second level
-*Horizontal* format Nanoprogram
- Interprets the microinstruction fields, thus converts a vertical microinstruction format into a horizontal nanoinstruction format.

Usually, the microprogram consists of a large number of short microinstructions, while the nanoprogram contains fewer words with longer nanoinstructions.

TWO-LEVEL MICROPROGRAMMING - EXAMPLE

* Micro-program: 2048 microinstructions of 200 bits each
* With 1-Level Control Storage: 2048 x 200 = 409,600 bits
* Assumption:
256 distinct microinstructions among 2048
* With 2-Level Control Storage:
Nano Storage: 256 x 200 bits to store 256 distinct nanoinstructions
Control storage: 2048 x 8 bits
To address 256 nano storage locations 8 bits are needed
* Total 1-Level control storage: 409,600 bits
Total 2-Level control storage: 67,584 bits (256 x 200 + 2048 x 8)



Memory Organization

- Memory hierarchy
- Main Memory
- RAM ,ROM Chips
- Memory Address map
- Memory Connection to CPU
- Associate memory
- Cache Memory
- Data cache ,Instruction cache, Miss and Hit ratio, Access time
- Associative ,Set associative ,mapping, waiting into cache
- Introduction to virtual memory

Learning Objectives

In this chapter, you will be able to know

❖Define memory system of the computer

❖State the need for memory system

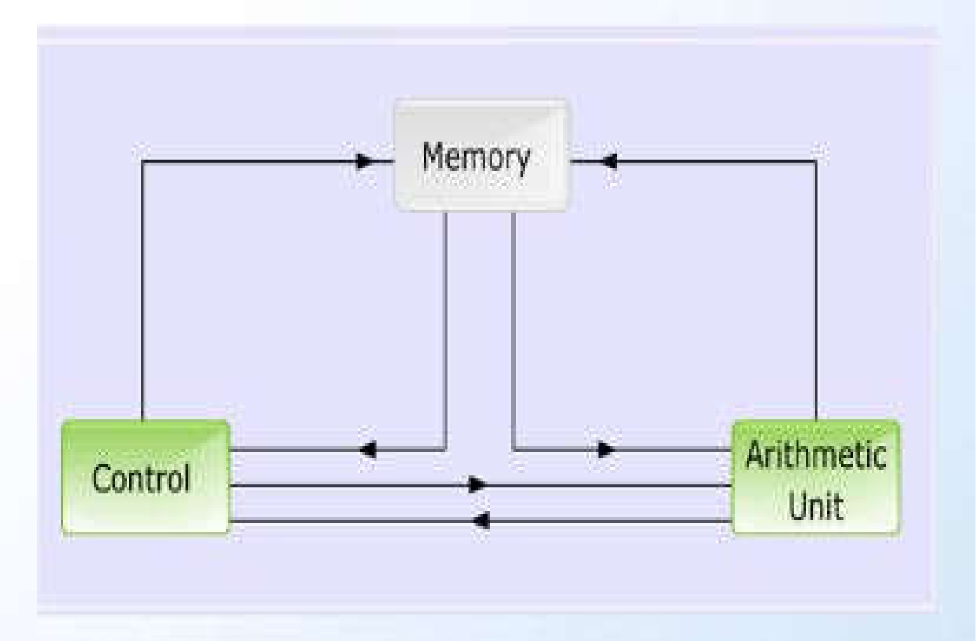
❖Explain memory hierarchy of the system

❖Describe the function of ROM and RAM chips

❖Demonstrate the connection between CPU and memory system

Memory System

- Memory refers to the physical devices used to store data on a temporary or permanent basis for the use in computer.
- Memory system is a collection of storage cells together with associated circuits needed to transfer information in and out of storage



Need for memory system

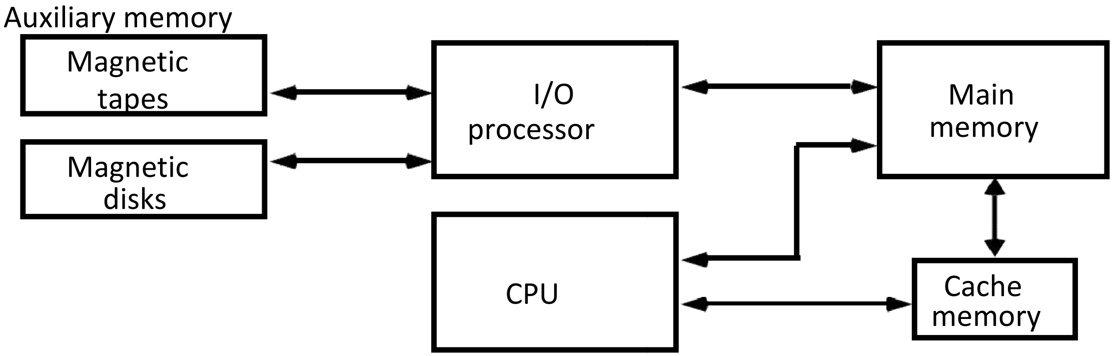
- Von Neumann's “stored program "concept demands for it.
- Computers would run more effectively if they were equipped with additional storage beyond the capacity of its own memory(registers) and main memory
- Memory system serves as a backup for storing the information that is not currently used by the CPU.

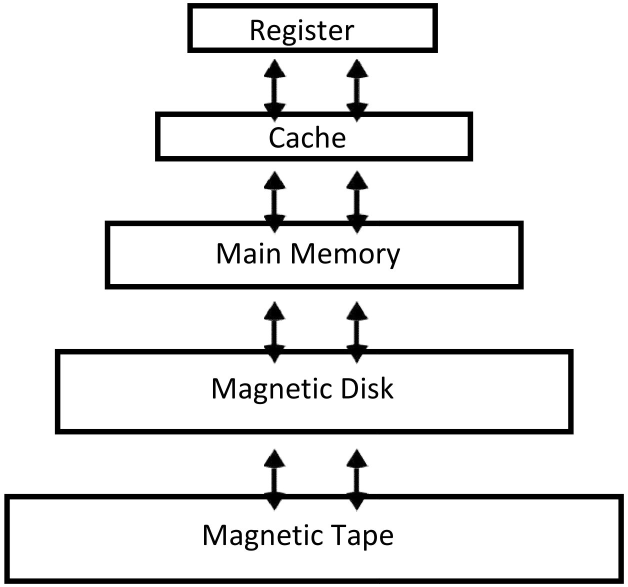
Memory Hierarchy

- The three key features which are a constraint to the design of a computer memory are capacity, access time and cost.
- The memory must be capable of storing huge amount of data.
- To achieve greatest performance, the memory must be able to keep up with the processor
- The cost of memory must be reasonable in relationship to other components.
- There is a trade-off among the three key characteristics of memory: namely , capacity , access time and cost.

MEMORY HIERARCHY

Memory Hierarchy is to obtain the highest possible access speed while minimizing the total cost of the memory system





MEMORY HIERARCHY

A variety of technologies are used to implement memory systems, and across this spectrum of technologies, the following relationship hold:

- Faster access time, grater cost per bit
- Greater capacity, smaller cost per bit
- Greater capacity. Slower access time

As one goes down the hierarchy, the following occur:

- Decreasing cost per bit
- Increasing capacity
- Increasing access time

Main Memory

- The main memory is the central storage unit in a computer system.
- It is relatively large and fast memory used to store programs and data during the computer operation.
- The principal technology used for the main memory is based on semiconductor integrated circuits.
- Integrated RAM chips are available in two possible operating modes , static and dynamic.

Main Memory

Static RAM(SRAM):

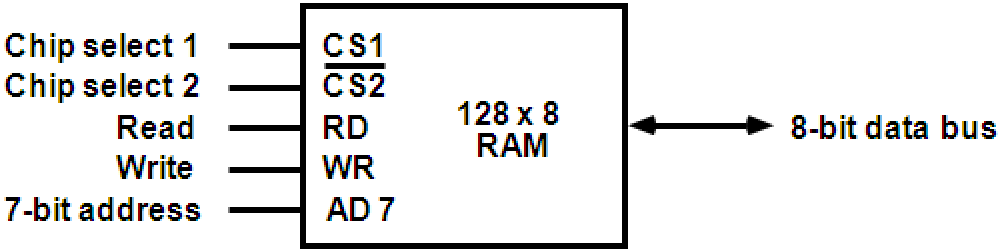
- The static RAM consists essentially of internal flip-flops that store the binary information.
- The stored information remains valid as long as power is applied to the unit.
- It is easier to use and has shorter read and write cycles.
- SRAM is used in implementing the cache memory.

Dynamic RAM(DRAM):

- the dynamic RAM stores the binary information in the form of electric charges that are applied to capacitors.
- It offers reduced power consumption.
- It has larger storage capacity
- DRAM is used in implementing the main memory.

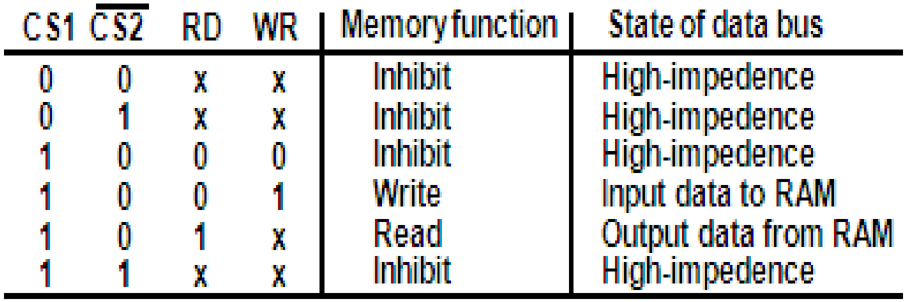
RAM CHIP

- The capacity of the memory is 128 words of eight bits(one byte)per word.
- This requires a 7-bit address and an 8-bit bidirectional data bus.
- The read and write inputs specify the memory operation and the two chips select(CS) control inputs are for enabling the chip only when it is selected by the microprocessor.
- The read and write inputs are sometimes combined in to one line labeled R/W.
- When the chip is selected , the two binary states in this line specify the two operations of read or write.



The Operation of the RAM chip

- The unit is in operation when CS1=1 and $\overline{\text{CS2}}$=0
- The bar on top of the second select variable indicates that this input is enabled when it is equal to 0.
- If the chip select inputs are not enabled, or if they are enabled but the read or write inputs are not enabled, the memory is inhibited and its data bus is in a high-impedance state.
- When CS1=1 and $\overline{\text{CS2}}$=0,the memory can be placed in **write or read mode.**
- When the WR input is enabled, the memory stores a byte from the data bus into a location specified by the address input lines.
- When the RD input is enabled, the content of the selected byte is placed into the data bus. the RD and WR signals control the memory operation as well as the bus buffers associated with the bidirectional data bus



CS1	$\overline{\text{CS2}}$	RD	WR	Memory function	State of data bus
0	0	x	x	Inhibit	High-impedence
0	1	x	x	Inhibit	High-impedence
1	0	0	0	Inhibit	High-impedence
1	0	0	1	Write	Input data to RAM
1	0	1	x	Read	Output data from RAM
1	1	x	x	Inhibit	High-impedence

Read Only Memory(ROM)

- ROM is a type of “built-in” memory that is capable of holding data.
- It is used for storing the bulk of the programs and data that are permanently reside in the computer.
- The ROM portion of main memory is needed for storing an initial program called a bootstrap loader, whose function is to start the computer software operating.
- The content of ROM remains unchanged when power is turned off.

Types of ROM

➢**PROM (Programmable ROM)**

- data is allowed to be loaded by user but this process is irreversible
- provide a faster and less expensive approach when only a small number of data are required

➢**EPROM (Erasable, Programmable ROM)**

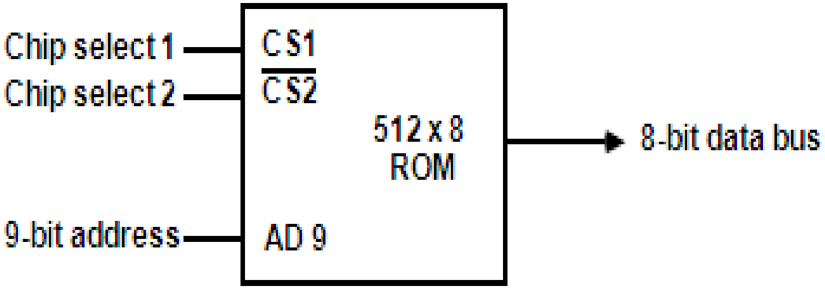
- stored data can be erased by exposing the chip to ultraviolet light and new data to be loaded

➢**EEPROM**

- stored data can be erased electrically and selectively
- different voltages are needed for erasing, writing, and reading the stored data

ROM Chip

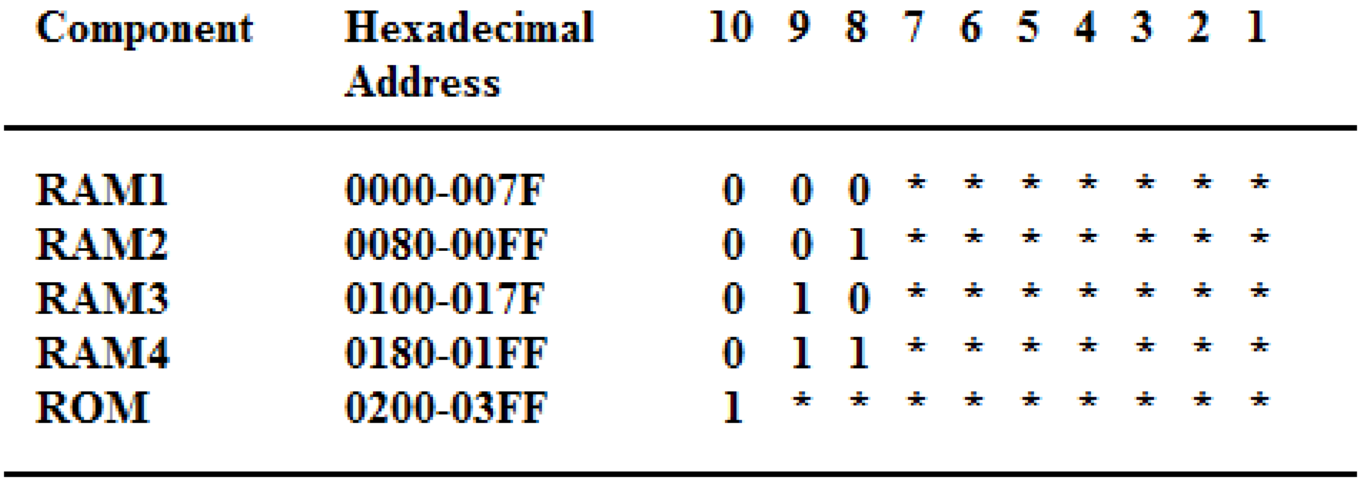
- The data bus can only be in an output mode since ROM can only read.
- The nine address lines in the ROM chip specify any one of the 512 bytes stored in it.
- The two chip select inputs must be CS1=1 and $\overline{\text{CS2}}$=0 for the unit to operate.
- There is no need for a read or write control because the unit can only read.



MEMORY ADDRESS MAP

- The addressing of memory can be established by means of a table that specifies the memory address assigned to each chip.
- Memory Address Map is a pictorial representation of assigned address space for each chip in the system.
- To demonstrate an example, assume that a computer system needs 512 bytes of RAM and 512 bytes of ROM.
- The RAM have 128 byte and need seven address lines, where the ROM have 512 bytes and need 9 address lines.

Memory Address Map

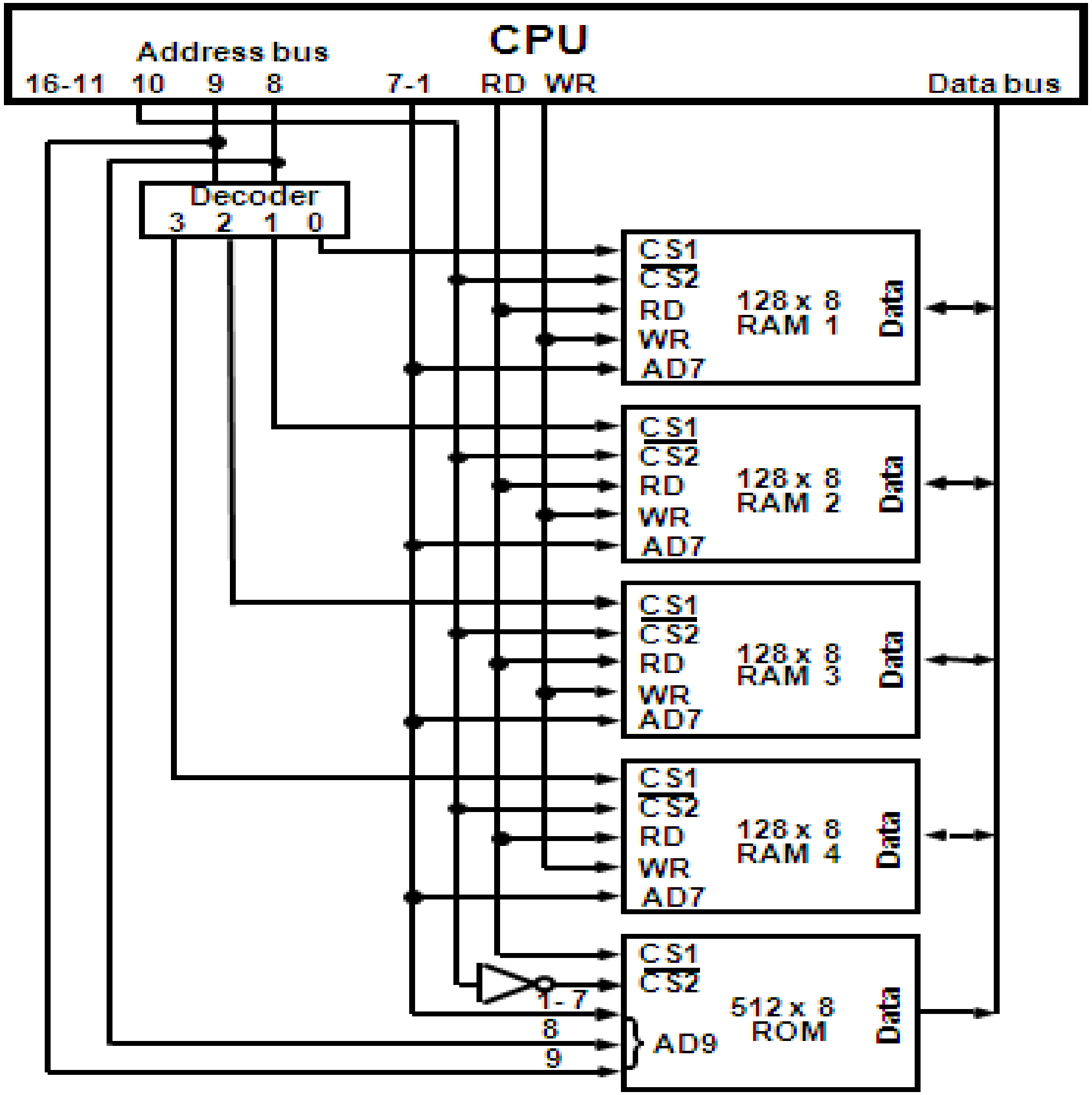


Component	Hexadecimal Address	10	9	8	7	6	5	4	3	2	1
RAM1	0000-007F	0	0	0	*	*	*	*	*	*	*
RAM2	0080-00FF	0	0	1	*	*	*	*	*	*	*
RAM3	0100-017F	0	1	0	*	*	*	*	*	*	*
RAM4	0180-01FF	0	1	1	*	*	*	*	*	*	*
ROM	0200-03FF	1	*	*	*	*	*	*	*	*	*

Memory Address Map

- The hexadecimal address assigns a range of hexadecimal equivalent address for each chip
- Line 8 and 9 represent four distinct binary combination to specify which RAM we chose
- When line 10 is 0, CPU selects a RAM. And when it’s 1, it selects the ROM

CONNECTION OF MEMORY TO CPU



CONNECTION OF MEMORY TO CPU

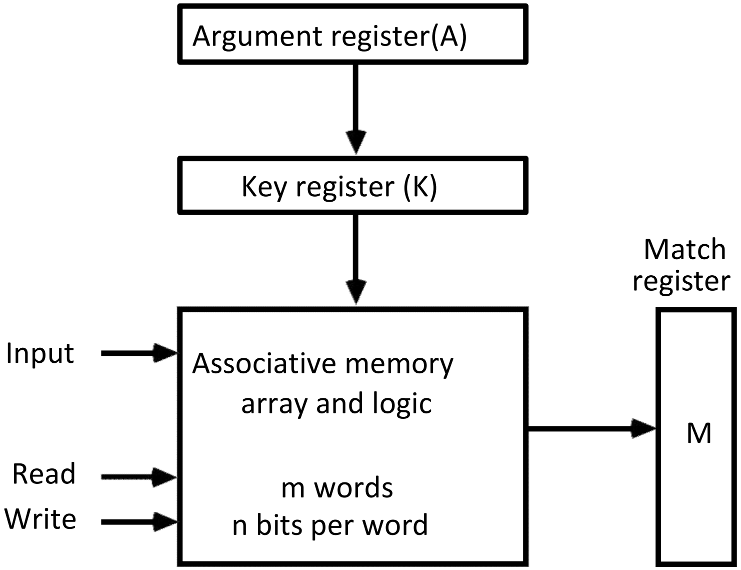
- The lower order lines in the address bus select the byte within the chips and other lines and select a particular chip through its chip select inputs.
- The selection between RAM and ROM is achieved through bus line 10.
- The RAMs are selected when the bit in this line is 0,and the ROM when the bit is 1.
- Address bus lines 1 to 9 are applied to the input address of ROM without going through the decoder. this assigns addresses 0 to 511 to RAM and 512 to 1023 to ROM.
- The data bus of the ROM has only an output capability, where as the data bus connected to the RAMS can transfer information in both directions

Associate memory

- The time required to find an item stored in memory can be reduced considerably if stored data can be identified for access by the content of the data itself rather than by an address.
- A memory unit accessed by content is called an **associative memory** or **content addressable memory(CAM)**
- This type of memory is accessed simultaneously and in parallel on the basis of data content rather than by specific address or location
- Associative memory is more expensive than a RAM because each cell must have storage capability as well as logic circuits
- Argument register – holds an external argument for content matching.
- Key register – mask for choosing a particular field or key in the argument word.

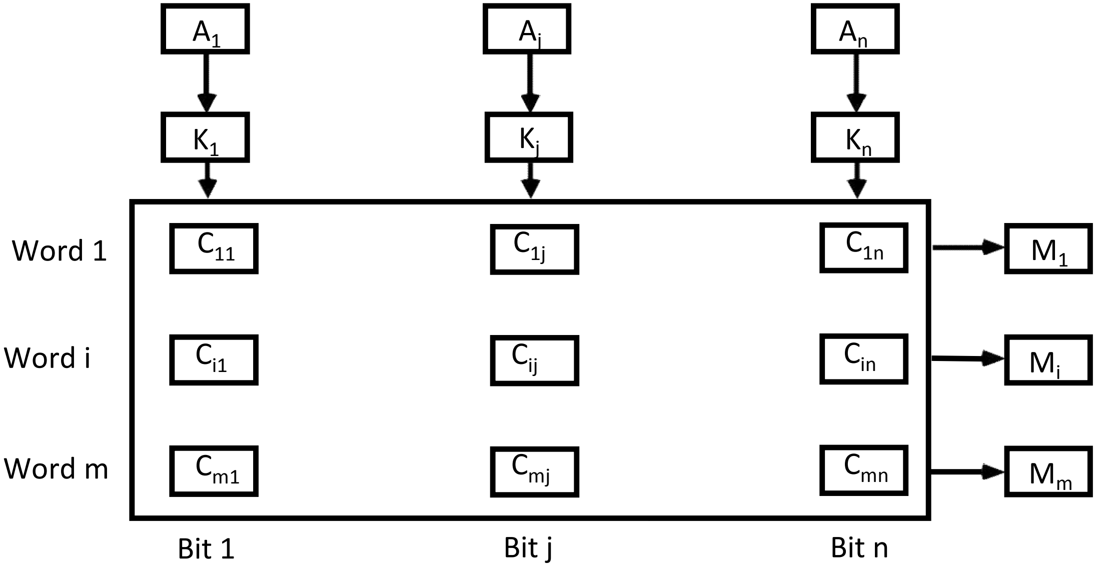
Block Diagram

-

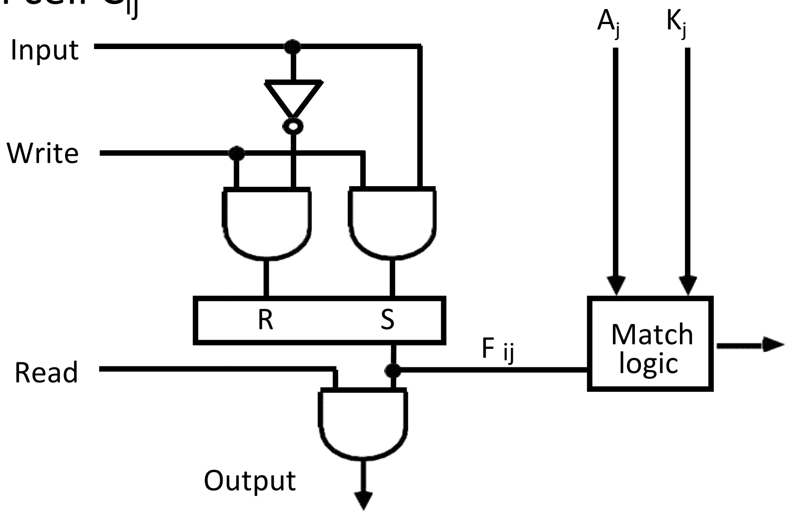


- Compare each word in CAM in parallel with the content of A(Argument Register)
- If CAM Word[i] = A, M(i) = 1
- Read sequentially accessing CAM for CAM Word(i) for M(i) = 1
- K(Key Register) provides a mask for choosing a particular field or key in the argument in A (only those bits in the argument that have 1's in their corresponding position of K are compared)

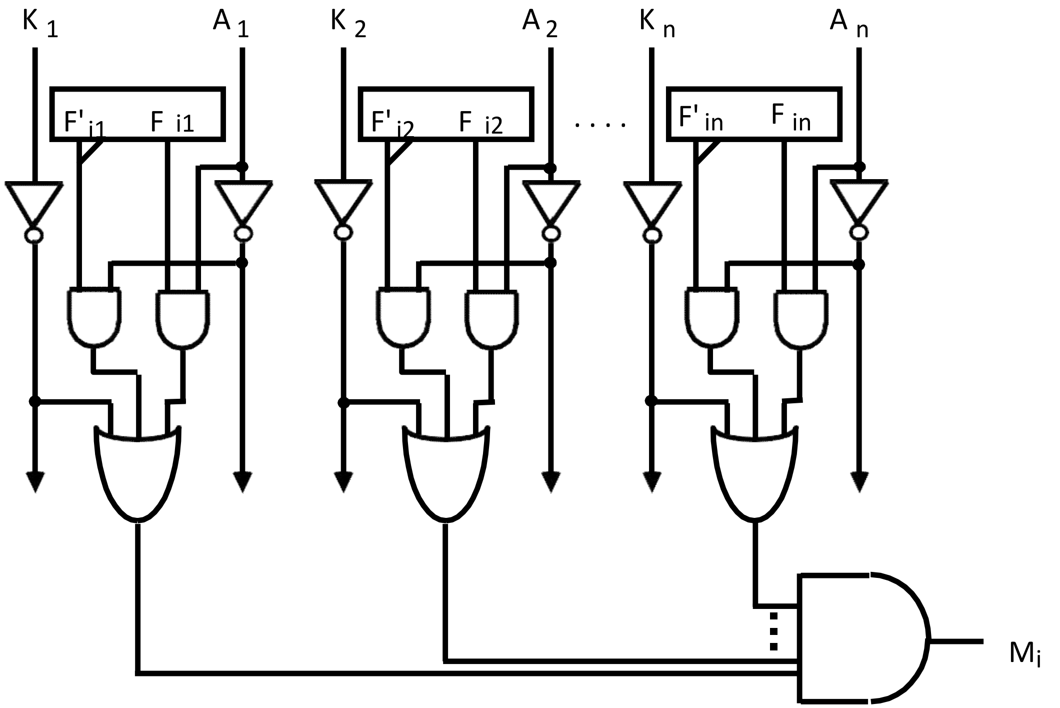
ORGANIZATION OF CAM



Internal organization of a typical cell C_{ij}



MATCH LOGIC



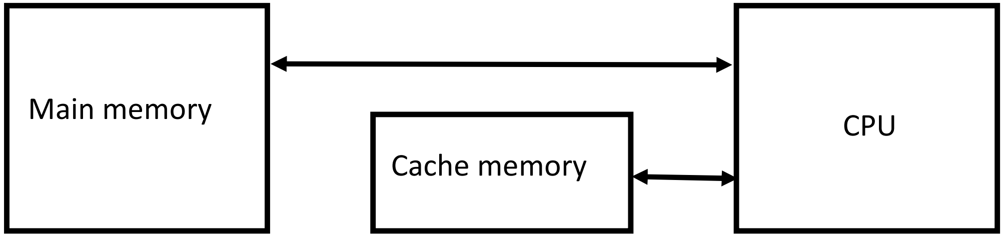
CACHE MEMORY

Locality of Reference

- The references to memory at any given time interval tend to be confined within a localized areas
- This area contains a set of information and the membership changes gradually as time goes by
- **Temporal Locality**
  The information which will be used in near future is likely to be in use already(e.g. Reuse of information in loops)
- **Spatial Locality**
  If a word is accessed, adjacent(near) words are likely accessed soon (e.g. Related data items (arrays) are usually stored together; instructions are executed sequentially)

Cache

- The property of Locality of Reference makes the Cache memory systems work
- Cache is a fast small capacity memory that should hold those information which are most likely to be accessed



CACHE MEMORY

- If the active portions of the program and data are placed in a fast small memory, the average memory access time can be reduced,
- Thus reducing the total execution time of the program
- Such a fast small memory is referred to as cache memory
- The cache is the fastest component in the memory hierarchy and approaches the speed of CPU component

Working

- When the CPU needs to access memory, the cache is examined.
- If the word is found in the cache, it is read from the fast memory.
- If the word addressed by the cpu is not found in the cache the main memory is accessed to read the word
- A block of words containing the one just accesses is then transferred from main memory to cache memory.
- The block size may vary from one word(the one just accessed) to about 16 words adjacent to the one just accessed.
- In this manner, some data are transferred to cache so that future references to memory find the required words in the fast cache memory.

Working

- Existence of a cache is transparent to the processor. The processor issues Read and
  Write requests in the same manner.

- If the data is in the cache it is called a Read or Write hit.

- Read hit:
  - □ The data is obtained from the cache.

- Write hit:
  - □ Cache has a replica of the contents of the main memory.
  - □ Contents of the cache and the main memory may be updated simultaneously. This is the write-through protocol.
  - □ Update the contents of the cache, and mark it as updated by setting a bit known as the dirty bit or modified bit. The contents of the main memory are updated when this block is replaced. This is write-back or copy-back protocol.

Working

- If the data is not present in the cache, then a Read miss or Write miss occurs.

- **Read miss:**
   Block of words containing this requested word is transferred from the memory.
   After the block is transferred, the desired word is forwarded to the processor.
   The desired word may also be forwarded to the processor as soon as it is transferred without waiting for the entire block to be transferred. This is called load-through or early-restart.

- **Write-miss:**
   Write-through protocol is used, then the contents of the main memory are updated directly.
   If write-back protocol is used, the block containing the addressed word is first brought into the cache. The desired word is overwritten with new information.

Working

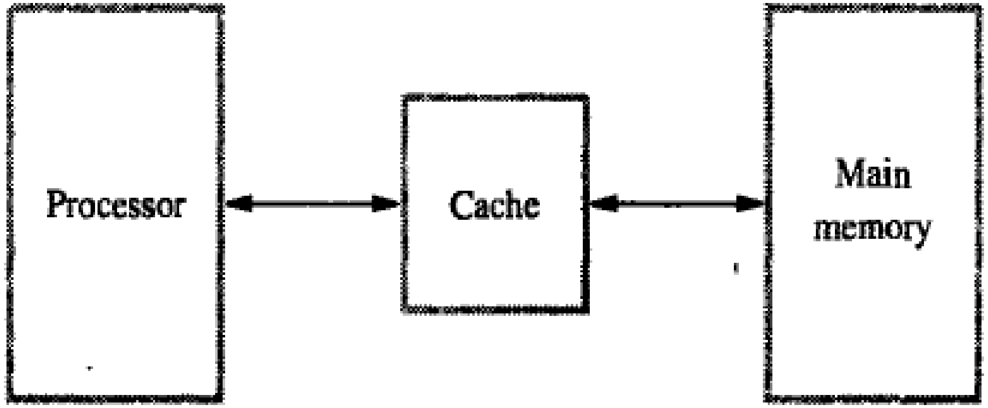
- In a read operation, the block containing the location specified is transferred in to the cache from the main memory, if it is not in the cache(a miss).otherwise(a hit),the block can be read from the cache directly
- The performance of cache memory is frequently measured in terms of hit ratio . High hit ratio verifies the validity of the local reference property.
- Hit ratio=Number of hits/total number of memory references

Working

Two different ways of write access for system with cache memory :

1)***Write-through*** method –the cache and the main memory locations are updated simultaneously.

2)***Write-back*** method -cache location updated during a write operation is marked with a dirty or modified bit. The main memory location is updated later when the block is to be removed from the cache.



- Processor issues a Read request, a block of words is transferred from the main memory to the cache, one word at a time.
- Subsequent references to the data in this block of words are found in the cache.
- At any given time, only some blocks in the main memory are held in the cache. Which blocks in the main memory are in the cache is determined by a **“mapping function”.**
- When the cache is full, and a block of words needs to be transferred from the main memory, some block of words in the cache must be replaced. This is determined by a **“replacement algorithm”.**

Mapping functions

The correspondence between the main memory blocks and those in the cache is specified by a mapping function.

- Associative Mapping
- Set-associative Mapping

To explain the mapping procedures, we consider

- a 2K cache consisting of 128 blocks of 16 words each, and
- a 64K main memory addressable by a 16-bit address, 4096 blocks of 16 words each.

Associative Mapped Cache

•Main memory block can be placed into any cache position.

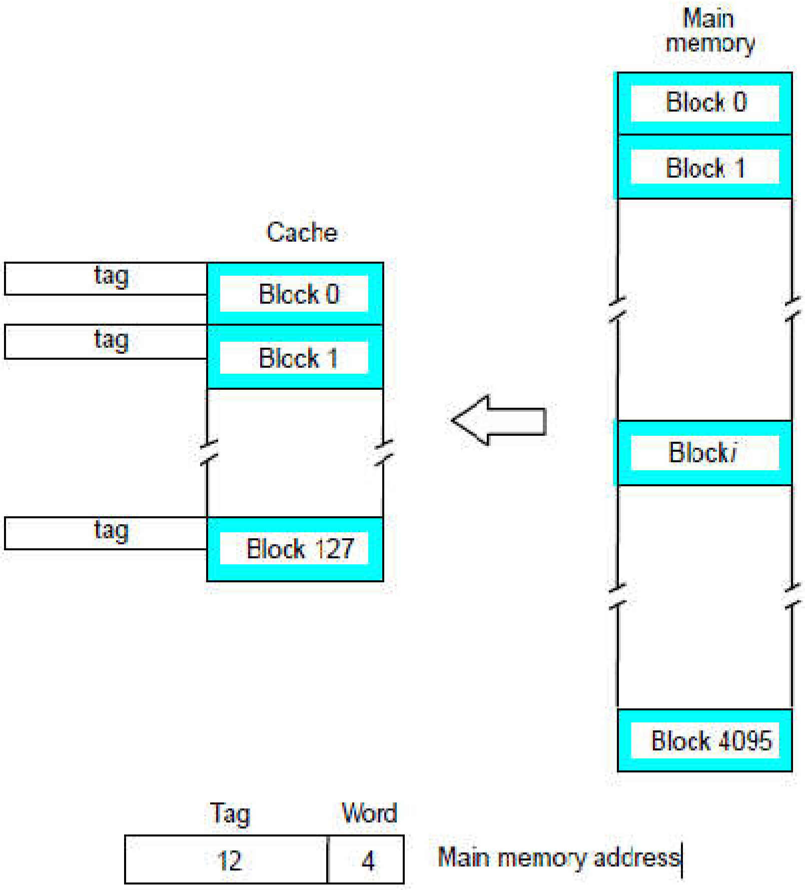
•Memory address is divided into two fields:

- Low order 4 bits identify the word within a block.
- High order 12 bits or tag bits identify a memory block when it is resident in the cache.

•Flexible, and uses cache space efficiently.

•Replacement algorithms can be used to replace an existing block in the cache when the cache is full.

•Cost is higher than direct-mapped cache because of the need to search all 128 patterns to determine whether a given block is in the cache.



- The cost of an associative cache is relatively high because of the need to search all 128 tags to determine whether a given block is in the cache.
- For performance reasons, ***associative search*** must be done in parallel.

Set-associative Mapping

- Blocks of the cache are grouped into sets, and the mapping allows a block of the main memory to reside in any block of a specific set.
- A cache that has *k* blocks per set is referred to as a *k*-way set-associative cache.
- The contention problem of the direct method is eased.
- The hardware cost of the associative method is reduced.

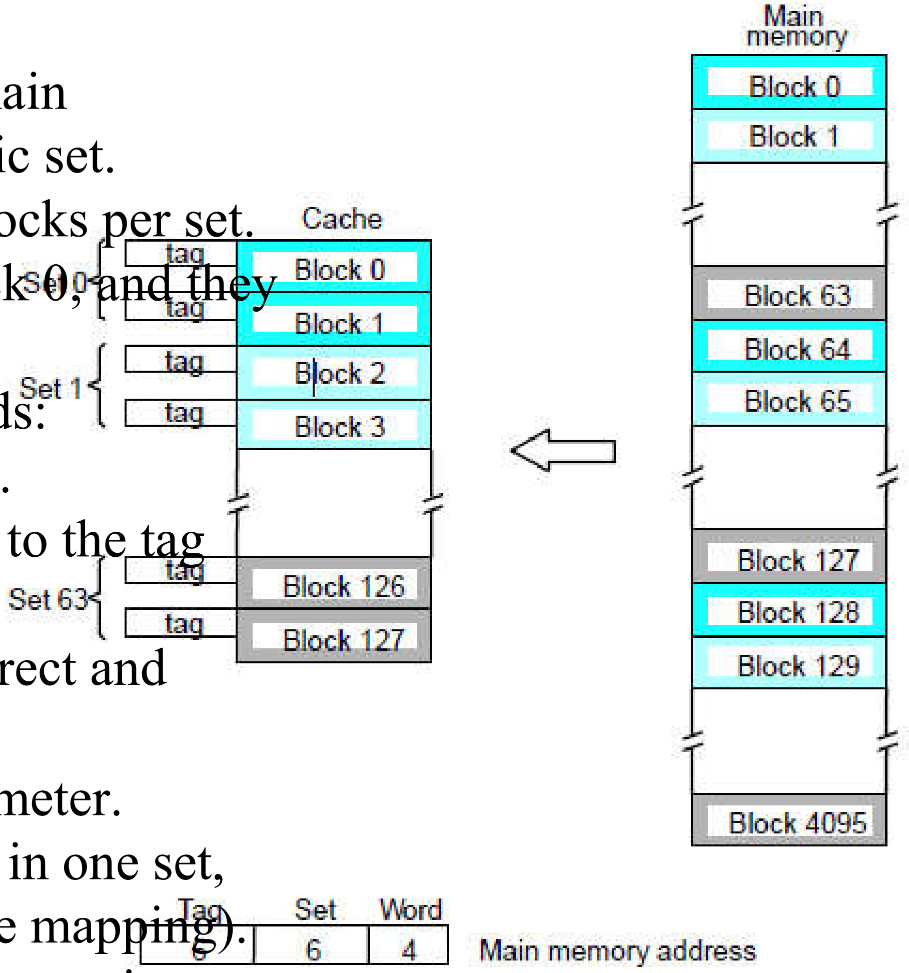
Set Associative Mapped Cache with two blocks per set

Blocks of cache are grouped into sets.
Mapping function allows a block of the main memory to reside in any block of a specific set.
Divide the cache into 64 sets, with two blocks per set.
Memory block 0, 64, 128 etc. map to block 0, and they can occupy either of the two positions.
Memory address is divided into three fields:

- 6 bit field determines the set number.
- High order 6 bit fields are compared to the tag fields of the two blocks in a set.

Set-associative mapping combination of direct and associative mapping.
Number of blocks per set is a design parameter.

- One extreme is to have all the blocks in one set, requiring no set bits (fully associ ative mapping).
- Other extreme is to have one block per set, is the same as direct mapping.

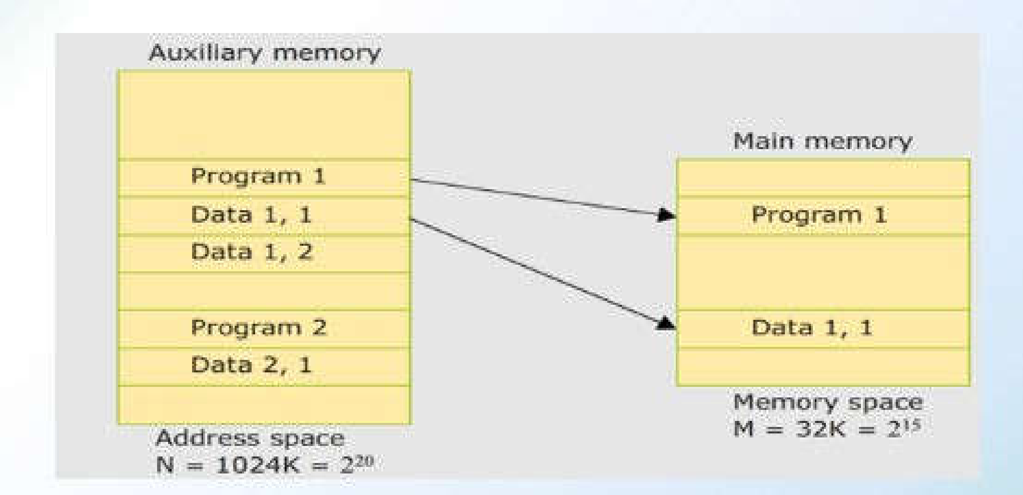


VIRTUAL MEMORY

- Virtual memory is a concept used in some large computer systems that permit the user to construct programs as though a large memory space were available, equal to the totality of auxiliary memory
- Each address that is referenced by the CPU goes through an address mapping from the so-called virtual address to a physical address in main memory.
- A virtual memory system provides a mechanism for translating program-generated addresses in to correct main memory locations
- The translation or mapping is handled automatically by the hardware by means of a mapping table.

Address space and memory space

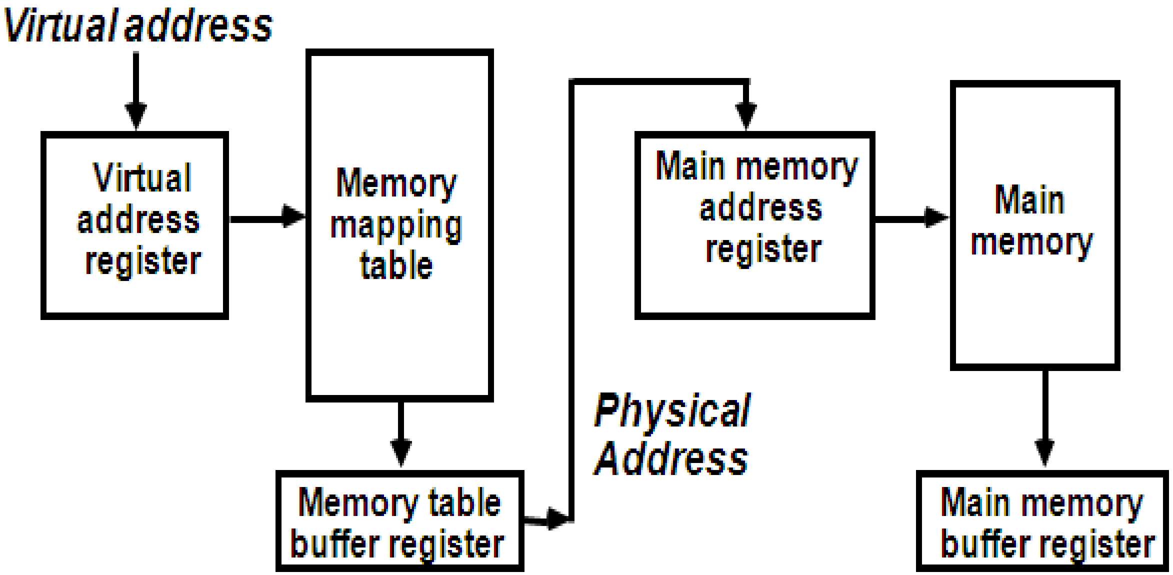
- An address used by a programmer will be called a virtual address, and the set of such addresses the address space.
- An address in main memory is called a location or physical address. the set of such location is called the memory space.



Relation between Address and memory space in a virtual memory system

Mapping using Memory table

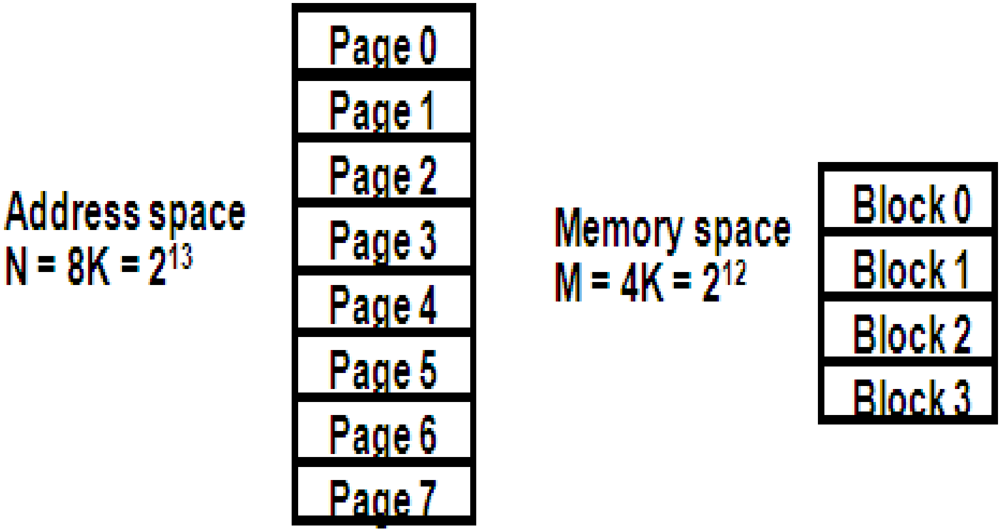
- The mapping table may be stored in a separate memory or in main memory



Memory table for mapping a virtual address

Mapping using paging or page table

- The physical memory is broken down in to groups of equal size blocks.
- The term page refers to groups to groups of address space of the same size as block.



Address space
N = 8K = 2^{13}

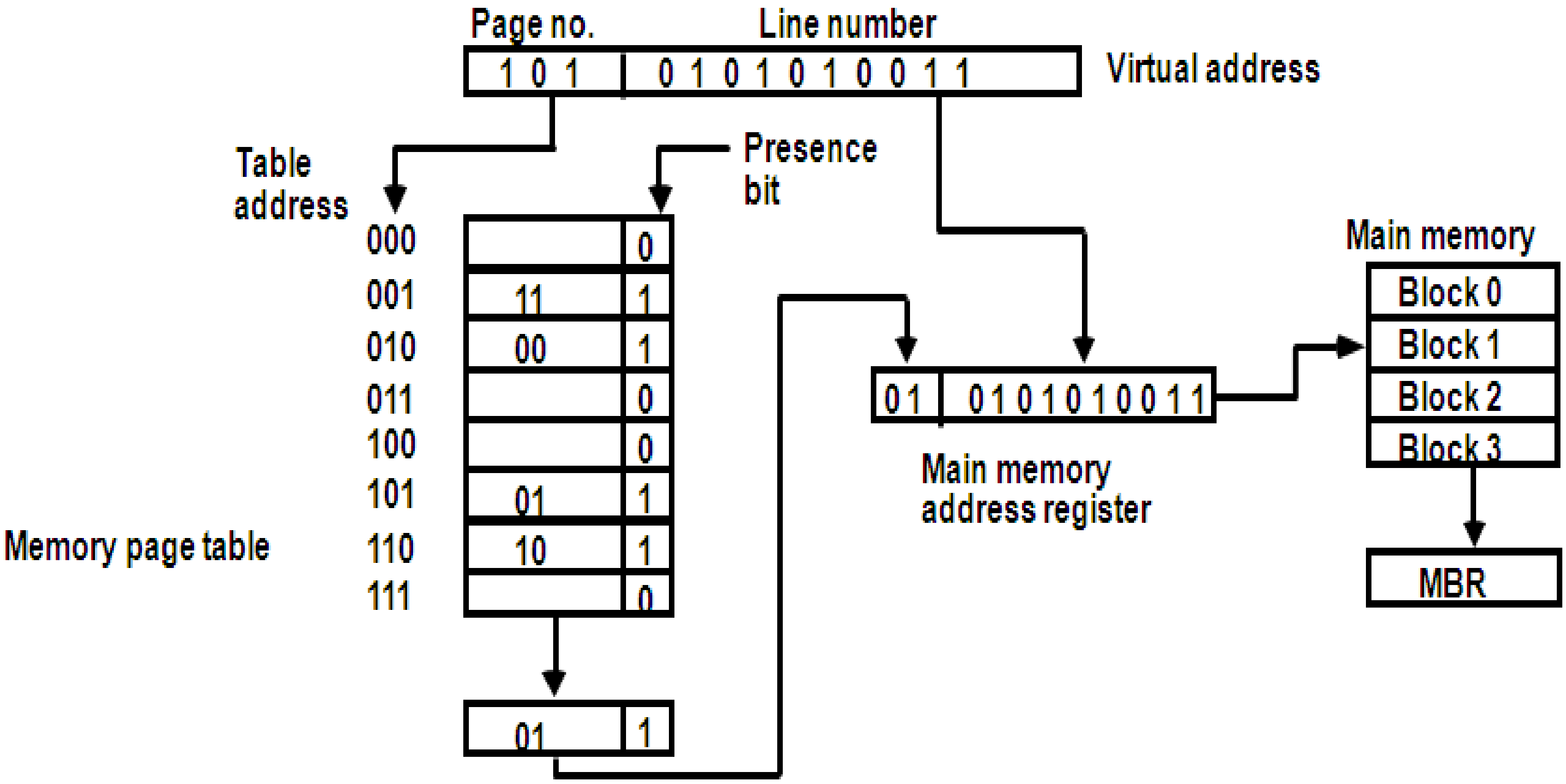
Page 0
Page 1
Page 2
Page 3
Page 4
Page 5
Page 6
Page 7

Memory space
M = 4K = 2^{12}

Block 0
Block 1
Block 2
Block 3

Address space and memory space split in to group of 1K words

Organization of memory Mapping Table in a paged system

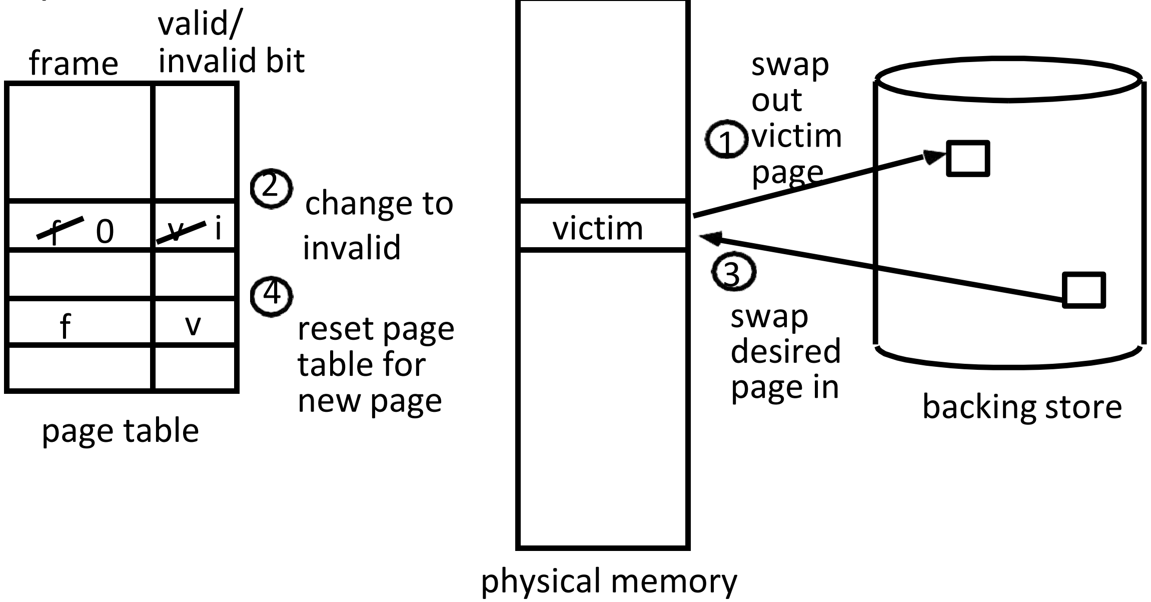


PAGE REPLACEMENT

Decision on which page to displace to make room for an incoming page when no free frame is available

Modified page fault service routine

1. Find the location of the desired page on the backing store
2. Find a free frame
   - If there is a free frame, use it
   - Otherwise, use a page-replacement algorithm to select a *victim* frame
   - Write the victim page to the backing store
3. Read the desired page into the (newly) free frame
4. Restart the user process



PAGE REPLACEMENT ALGORITHMS

FIFO

Reference string

7	0	1	2	0	3	0	4	2	3	0	3	2	1	2	0	1	7	0	1
7	7	7	2		2	2	4	4	4	0			0	0			7	7	7
	0	0	0		3	3	3	2	2	2			1	1			1	0	0
		1	1		1	0	0	0	3	3			3	2			2	2	1

Page frames

FIFO algorithm selects the page that has been in memory the longest time
Using a queue - every time a page is loaded, its
identification is inserted in the queue
Easy to implement
May result in a frequent page fault

Optimal Replacement (OPT) - Lowest page fault rate of all algorithms

Replace that page which will not be used for the longest period of time

Reference string

7	0	1	2	0	3	0	4	2	3	0	3	2	1	2	0	1	7	0	1
7	7	7	2		2		2			2			2				7		
	0	0	0		0		4			0			0				0		
		1	1		3		3			3			1				1		

Page frames

INPUT-OUTPUT ORGANIZATION

□ **Peripheral Devices**

□ **Input-Output Interface**

□ **Asynchronous Data Transfer**

□ **Modes of Transfer**

□ **Priority Interrupt**

□ **Direct Memory Access**

□ **Input-Output Processor**

□ **Serial Communication**

Learning objectives

In this chapter, you will be able to know

□ Peripheral devices and their use

□ I/O interface connection

□ Asynchronous data Transfer

□ Modes of Transfer

□ Define priority interrupt

□ Functioning of DMA Controller

PERIPHERAL DEVICES

- The input-output subsystem of a computer provides an efficient mode of communication between the central system and the outside environment.
- ➢ **Input System** – programs and data must be entered in to computer memory for processing.
- **Output System-** Results obtained from computations must be recorded or displayed for the user.
- Input and Output Devices that are under the direct control of the computer are designed to read information in to and out of the memory unit upon command from the CPU are called peripheral devices.
- There are three types of peripherals such as input , output , and input-output peripherals .

PERIPHERAL DEVICES

Input Devices

- Keyboard
- Optical input devices
    - Card Reader
    - Paper Tape Reader
    - Bar code reader
    - Digitizer
    - Optical Mark Reader
- Magnetic Input Devices
    - Magnetic Stripe Reader
- Screen Input Devices
    - Touch Screen
    - Light Pen
    - Mouse
- Analog Input Devices

Output Devices

- Card Puncher, Paper Tape Puncher
- CRT
- Printer (Impact, Ink Jet,
  Laser, Dot Matrix)
- Plotter
- Analog
- Voice

INPUT/OUTPUT INTERFACES

- Provides a method for transferring information between internal storage (such as memory and CPU registers) and external I/O devices
- Resolves the differences between the computer and peripheral device

The major differences are:

- **Peripherals** - Electromechanical Devices

  CPU or Memory - Electronic Device
- **Data Transfer Rate**

  Peripherals - Usually slower

  CPU or Memory - Usually faster than peripherals

  Some kinds of Synchronization mechanism may be needed
- **Unit of Information**

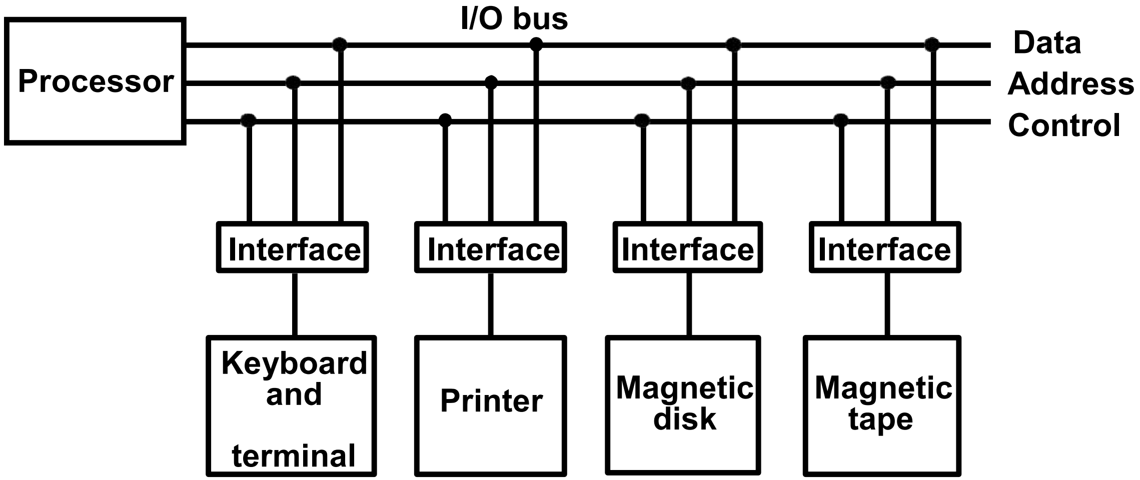
  Peripherals - Byte

  CPU or Memory - Word
- **Operating Modes**

  Peripherals - Autonomous, Asynchronous

  CPU or Memory - Synchronous

I/O BUS AND INTERFACE MODULES



- The I/O bus consists of data lines, address lines, and control lines.
- Each peripheral device has associated with it an interface unit.
- Each interface decodes the address and control received from the I/O bus, interprets them for the peripheral, and provides signals for the peripheral controller.
- To communicate with a particular device , the processor places a device address on the address lines.
- With the address in address lines, the processor provides a function code in the control lines.
- The function code is referred to as an **I/O command**.

I/O BUS AND INTERFACE MODULES

There are four types of commands that an interface may receive classified as control, status, data output , and data input

- A **control command** is used to activate the peripheral and to inform it what to do.
- A **status command** is used to test various status conditions in the interface and the peripheral.
- A **data output command** causes the interface to respond by transferring data from the bus into one of its registers.
- The interface receives an item of data from the peripheral and places it in its buffer register. the processor checks if data are available by means of a status command and then issues a data **input command**

I/O versus Memory Bus

In addition to communicating with I/O, the processor must communicate with the memory unit.

Like the I/O bus , the memory bus contains data , address , and read/write control lines.

There are three ways that computer buses can be used to communicate with memory and I/O.

1. Use two separate buses, one for memory and the other for I/O.
2. Use one common bus for both memory and I/O but have separate control lines for each.
3. Use one common bus for memory and I/O with common control lines.

ISOLATED vs. MEMORY MAPPED I/O

Isolated I/O

- Separate I/O read/write control lines in addition to memory read/write control lines
- Separate (isolated) memory and I/O address spaces
- Distinct input and output instructions

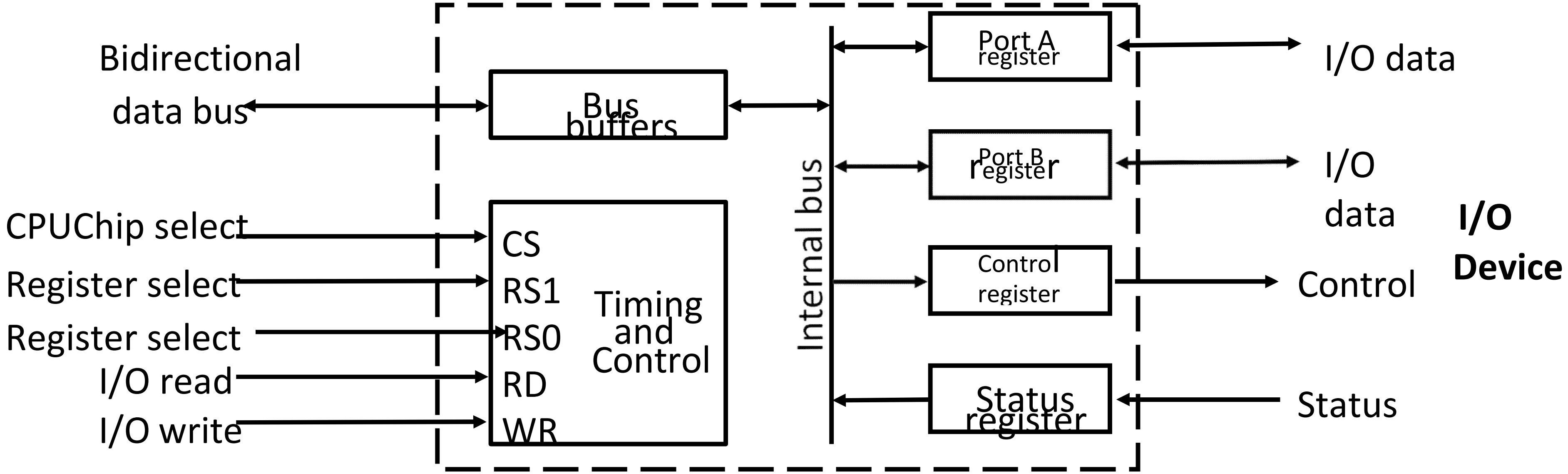
Memory-mapped I/O

- A single set of read/write control lines (no distinction between memory and I/O transfer)
- Memory and I/O addresses share the common address space

  -> reduces memory address range available
- No specific input or output instruction

  -> The same memory reference instructions can be used for I/O transfers
- Considerable flexibility in handling I/O operations

I/O INTERFACE-Example



Bidirectional data bus

CPUChip select

Register select

Register select

I/O read

I/O write

Bus buffers

CS

RS1

RS0

RD

WR

Timing and Control

Internal bus

Port A register

Port B register

Control register

Status register

I/O data

I/O data

Control

Status

I/O Device

CS	RS1	RS0	Register selected
0	x	x	None - data bus in high-imped
1	0	0	Port A register
1	0	1	Port B register
1	1	0	Control register
1	1	1	Status register

Programmable Interface

- Information in each port can be assigned a meaning depending on the mode of operation of the I/O device
  → Port A = Data; Port B = Command; Port C = Status

- CPU initializes(loads) each port by transferring a byte to the Control Register
  → Allows CPU can define the mode of operation of each port
  → Programmable Port: By changing the bits in the control register, it is possible to change the interface characteristics

ASYNCHRONOUS DATA TRANSFER

Synchronous and Asynchronous Operations

Synchronous - All devices derive the timing
information from common clock line

Asynchronous - No common clock

Asynchronous Data Transfer

Asynchronous data transfer between two independent units requires that control signals be transmitted between the communicating units to indicate the time at which data is being transmitted

Two Asynchronous Data Transfer Methods

Strobe pulse

- A strobe pulse is supplied by one unit to indicate the other unit when the transfer has to occur

Handshaking

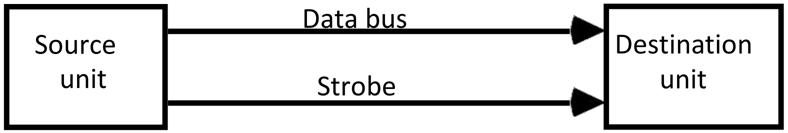
- A control signal is accompanied with each data being transmitted to indicate the presence of data
- The receiving unit responds with another control signal to acknowledge receipt of the data

STROBE CONTROL

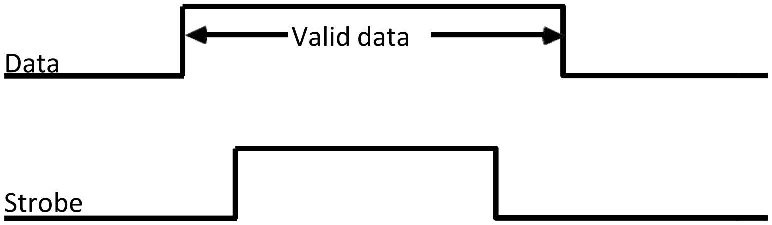
- The strobe control method of Asynchronous data transfer Employs a single control line to time each transfer
- The strobe may be activated by either the source or the destination unit

Source-Initiated Strobe for Data Transfer

Block Diagram

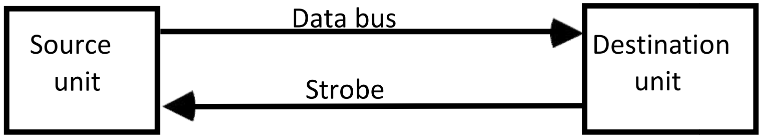


Timing Diagram

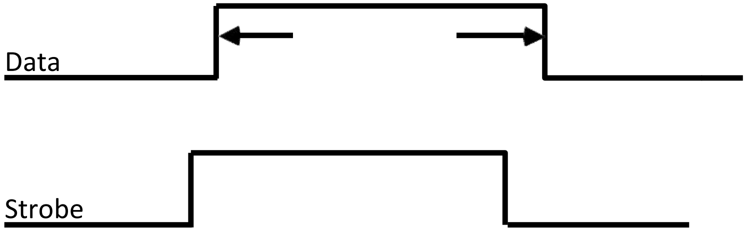


Destination-Initiated Strobe for Data Transfer

Block Diagram



Timing Diagram



Source-Initiated Strobe for Data Transfer

The block diagram shows a source initiated transfer in which

- The data bus carries the binary information from source unit to the destination unit.
- Typically, the bus has multiple lines to transfer an entire byte or word.
- The strobe is a single line that informs the destination unit when a valid data word is available in the bus.

The timing diagram shows

- Source unit first places the data on the data bus.
- After a brief delay to ensure that the data settle to a steady value, the source activates the strobe pulse.
- The information on the data bus and the strobe signal remain activate state for a sufficient time period to allow the destination unit to receive the data.

Destination-Initiated Strobe for Data Transfer

- The destination unit activates the strobe pulse, informing the source to provide the data.
- The source unit responds by placing the requested binary information on the data bus.
- The data must be valid and remain in the bus long enough for the destination unit to accept it.
- The falling edge of the strobe pulse can be used again to trigger a destination register.
- The destination unit then disables the strobe.
- The source removes the data from the bus after a predetermined time interval.

Disadvantages of Strobe methods

- **Source-Initiated**

  The source unit that initiates the transfer has no way of knowing whether the destination unit has actually received data

- **Destination-Initiated**

  The destination unit that initiates the transfer no way of knowing whether the source has actually placed the data on the bus

To solve this problem, the **HANDSHAKE** method introduces a second control signal to provide a Reply to the unit that initiates the transfer

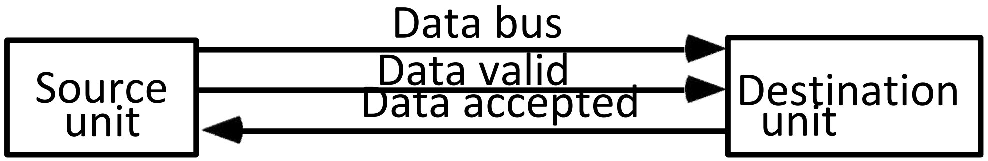
Hand Shaking

The basic principle of the two-wire handshaking method of data transfer is as follows

- One control line is in the same direction as the data flow in the bus from the source to destination.
- It is used by the source unit to inform the destination unit whether there are valid data in the bus .
- The other control line is in the other direction from the destination to source .
- It is used by the destination unit to inform the source whether it can accept data.
- The sequence of control during the transfer depends on the unit that initiates the transfer.

SOURCE-INITIATED TRANSFER USING HANDSHAKE

Block Diagram



Source unit

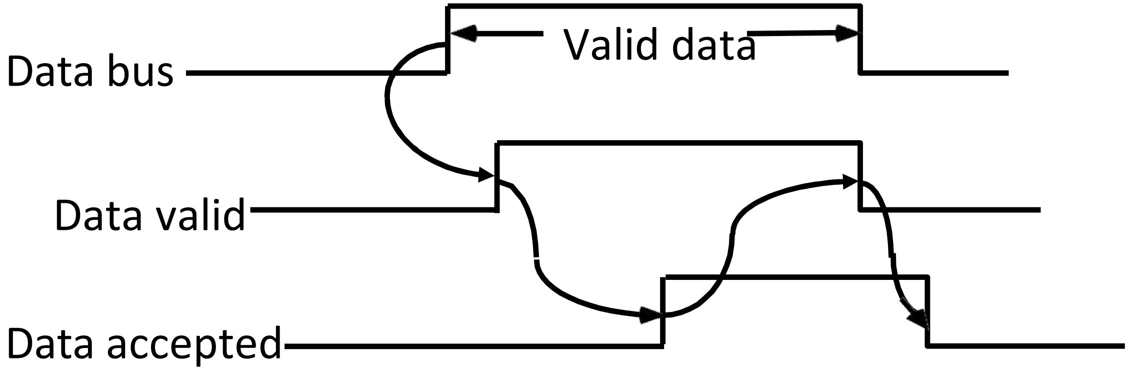
Data bus

Data valid

Data accepted

Destination unit

Timing Diagram



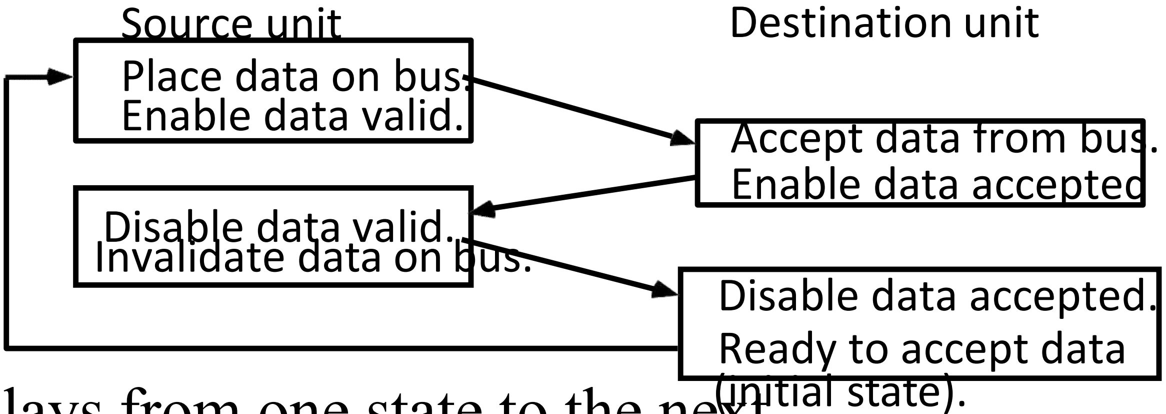
Data bus

Valid data

Data valid

Data accepted

Sequence of Events



Source unit

Destination unit

Place data on bus.
Enable data valid.

Accept data from bus.
Enable data accepted

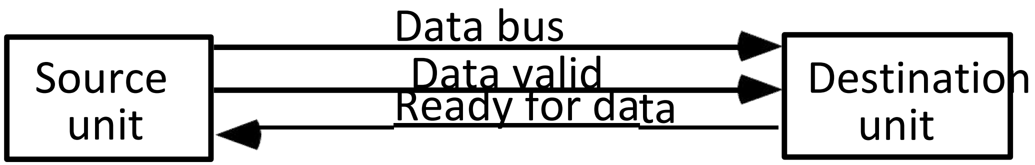
Disable data valid.
Invalidate data on bus.

Disable data accepted.
Ready to accept data (initial state).

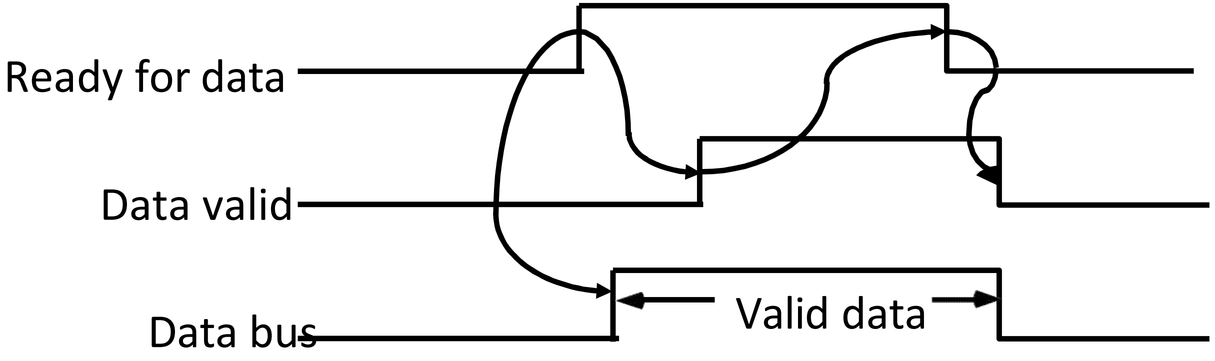
- Allows arbitrary delays from one state to the next
- Permits each unit to respond at its own data transfer rate
- The rate of transfer is determined by the slower unit

DESTINATION-INITIATED TRANSFER USING HANDSHAKE

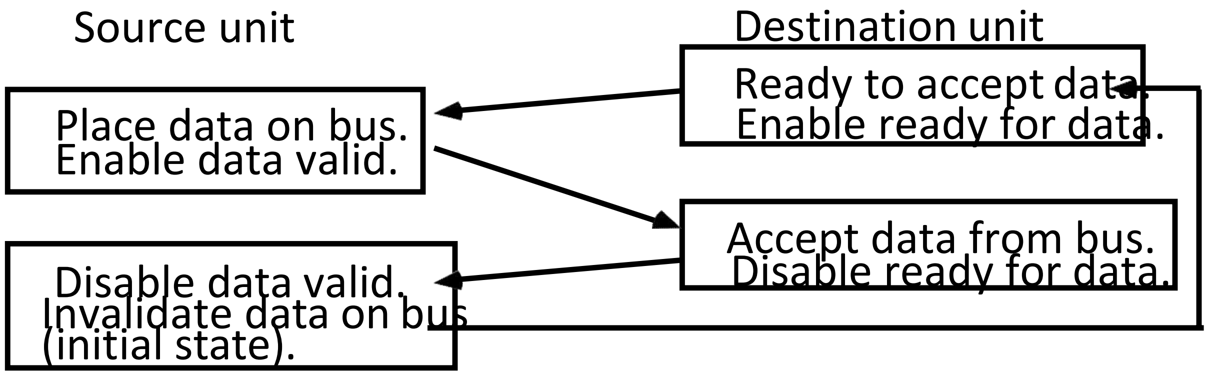
Block Diagram



Timing Diagram



Sequence of Events



Source unit

Destination unit

Ready to accept data.
Enable ready for data.

Place data on bus.
Enable data valid.

Accept data from bus.
Disable ready for data.

Disable data valid.
Invalidate data on bus
(initial state).

- Handshaking provides a high degree of flexibility and reliability because the successful completion of a data transfer relies on active participation by both units
- If one unit is faulty, data transfer will not be completed
  -> Can be detected by means of a *timeout* mechanism

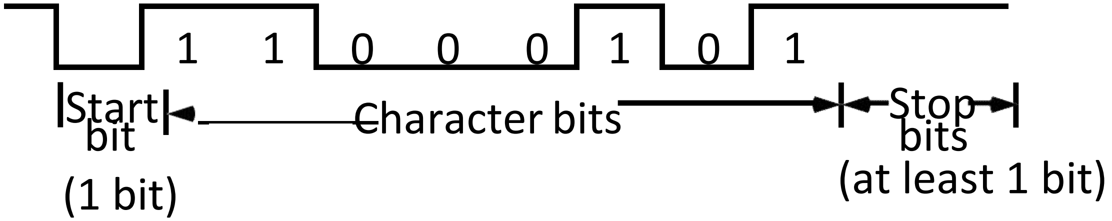
ASYNCHRONOUS SERIAL TRANSFER

Four Different Types of Transfer

Asynchronous serial transfer
Synchronous serial transfer
Asynchronous parallel transfer
Synchronous parallel transfer

Asynchronous Serial Transfer

- Employs special bits which are inserted at both ends of the character code
- Each character consists of three parts; Start bit; Data bits; Stop bits.



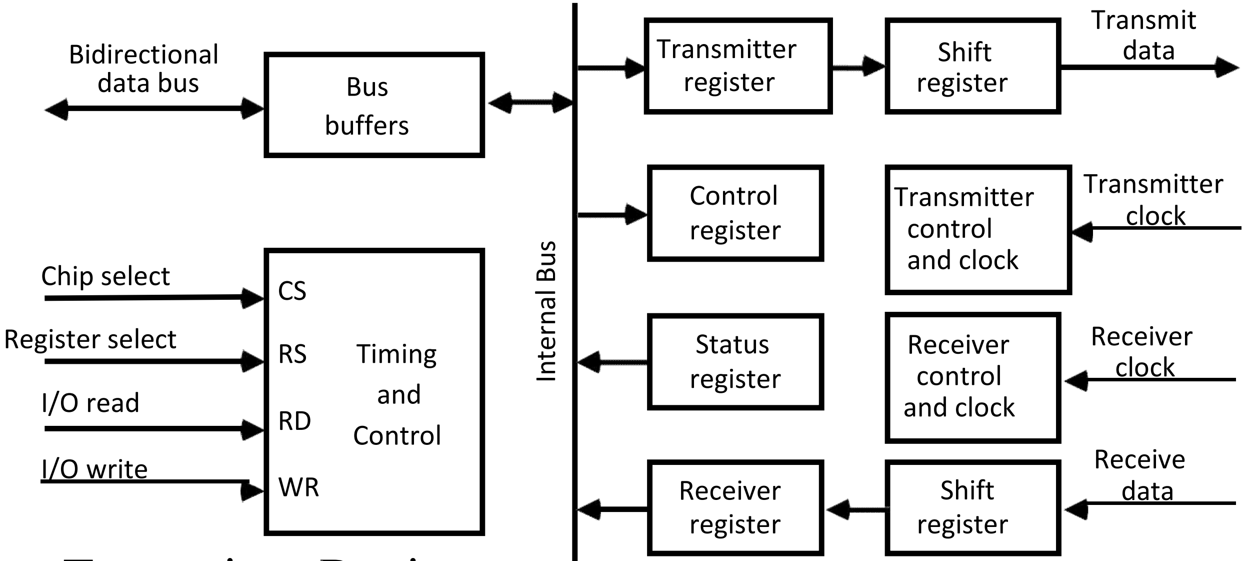
A character can be detected by the receiver from the knowledge of 4 rules;

- When data are not being sent, the line is kept in the 1-state (idle state)
- The initiation of a character transmission is detected by a Start Bit , which is always a 0
- The character bits always follow the *Start Bit*
- After the last character , a *Stop Bit* is detected when the line returns to the 1-state for at least 1 bit time

The receiver knows in advance the transfer rate of the bits and the number of information bits to expect

Asynchronous Communication Interface

A typical asynchronous communication interface available as an IC



CS	RS	Opera	Register selected
0	x	x	None
1	0	WR	Transmitter register
1	1	WR	Control register
1	0	RD	Receiver register
1	1	RD	Status register

Transmitter Register

- Accepts a data byte(from CPU) through the data bus
- Transferred to a shift register for serial transmission

Receiver

- Receives serial information into another shift register
- Complete data byte is sent to the receiver register

Status Register Bits

- Used for I/O flags and for recording errors

Control Register Bits

- Define baud rate, no. of bits in each character, whether to generate and check parity, and no. of stop bits

MODES OF TRANSFER

- Information transferred from the central computer into an external device originates in the memory unit.
- The CPU merely executes the I/O instructions and accepts the data temporarily, but the ultimate source or destination is the memory unit.
- Data transfer to and from peripherals may be handled in one of three possible modes.

1.Programmed I/O.

2.Interrupt-initiated I/O

3.Direct memory access(DMA)

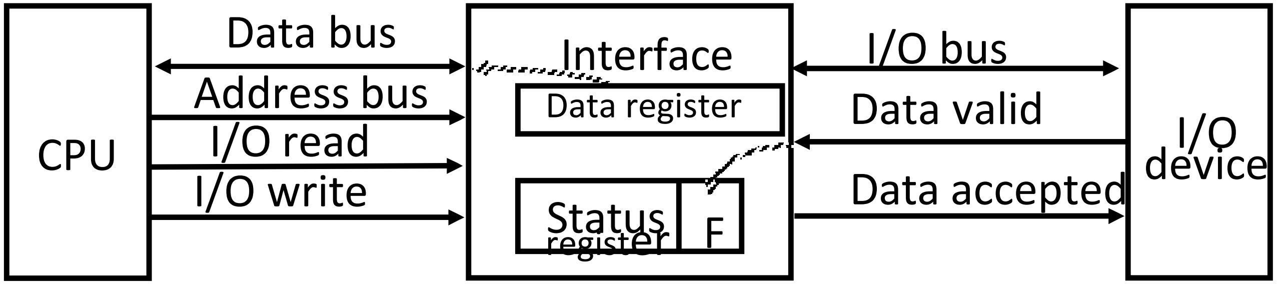
Programmed I/O

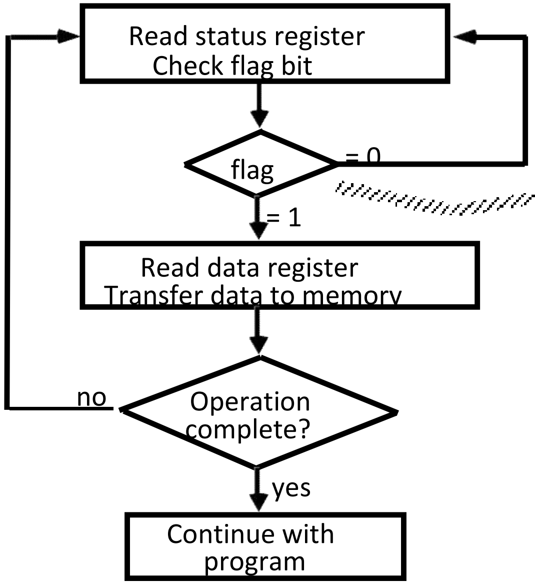
- Programmed I/O operations are the result of I/O instructions written in the computer program.
- Each data item transfer is initiated by an instruction in the program.
- Once a data transfer is initiated, the CPU is required to monitor the interface to see when a transfer can again be made.
- The CPU stays in a program loop until the I/O unit indicates that it is ready for data transfer. this is time consuming process which can be avoids by using an interrupt facility and special commands to inform the interface to issue an interrupt request signal when the data are available from the device.

Example of programmed I/O

A transfer from an I/O device to memory requires the execution of several instructions by the CPU ,including an input instruction to transfer the data from the device to the CPU and a store instruction to transfer the data from the CPU to memory

Data transfer from I/O device to CPU





Flowchart for CPU program to input data

Polling or Status Checking

- Continuous CPU involvement
- CPU slowed down to I/O speed
- Simple
- Least hardware

Interrupt-Initiated I/O

- In the programmed I/O method, the CPU stays in a program loop until the I/O unit indicates that it is ready for data transfer. This is a time-consuming process since it keeps the processor busy needlessly.
- It can be avoided by using an interrupt facility and special commands to inform the interface to issue an interrupt request signal when the data are available from the device.
- In the meantime the CPU can proceed to execute another program. The interface meanwhile keeps monitoring the device. When the interface determines that the device is ready for data transfer, it generates an interrupt request to the computer.
- Upon detecting the external interrupt signal, the CPU momentarily stops the task it is processing, branches to a service program to process the I/O transfer, and then returns to the task it was originally performing.

Example of Interrupt initiated I/O:

- Vectored interrupt
- Non vectored interrupt

Vectored interrupt :

In vectored interrupt, the source that interrupts supplies the branch information to the computer. This information is called the interrupt vector.

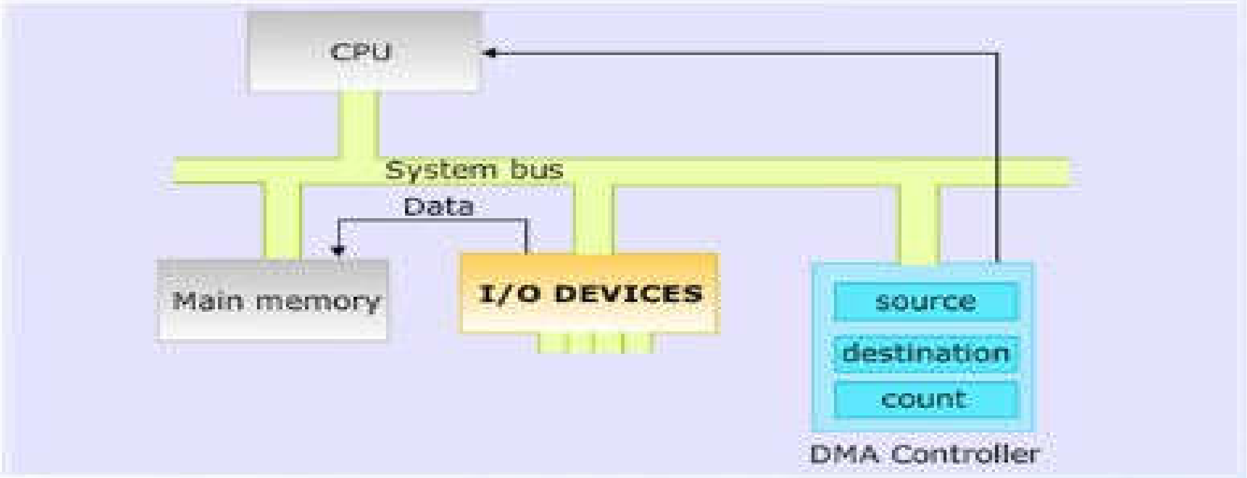
- **Non vectored interrupt**

  In a non vectored interrupt, the branch address is assigned to a fixed location in memory.

DIRECT MEMORY ACCESS

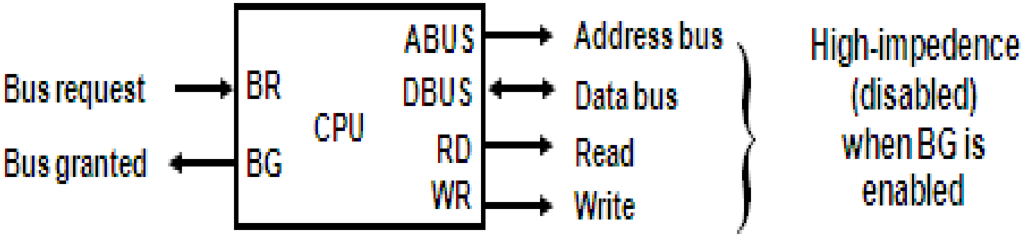
- The transfer of data between a fast storage device such as magnetic disk and memory is often limited by the speed of the CPU.
- Removing the CPU from the path and letting the peripheral device manage the memory buses directly would improve the speed of transfer.

This transfer technique is called direct memory access(DMA)



DMA Function

- During DMA transfer, the CPU is idle and has no control of the memory buses.
- A DMA controller takes over the buses to manage the transfer directly between the I/O devices and memory.
- The CPU may be placed in an idle state in variety of ways.
- One common method exclusively used in microprocessors is to disable the Buses through Special control signals.



DIRECT MEMORY ACCESS

- **BUS REQUEST**

  The bus request(BR) input is used by the DMA controller to request the CPU to relinquish control of the buses.

- **BUS GRANT**

  The CPU activates the bus grant(BG) output to inform the external DMA that the buses are in the high-impedance state

- **BUS TRANSFER**

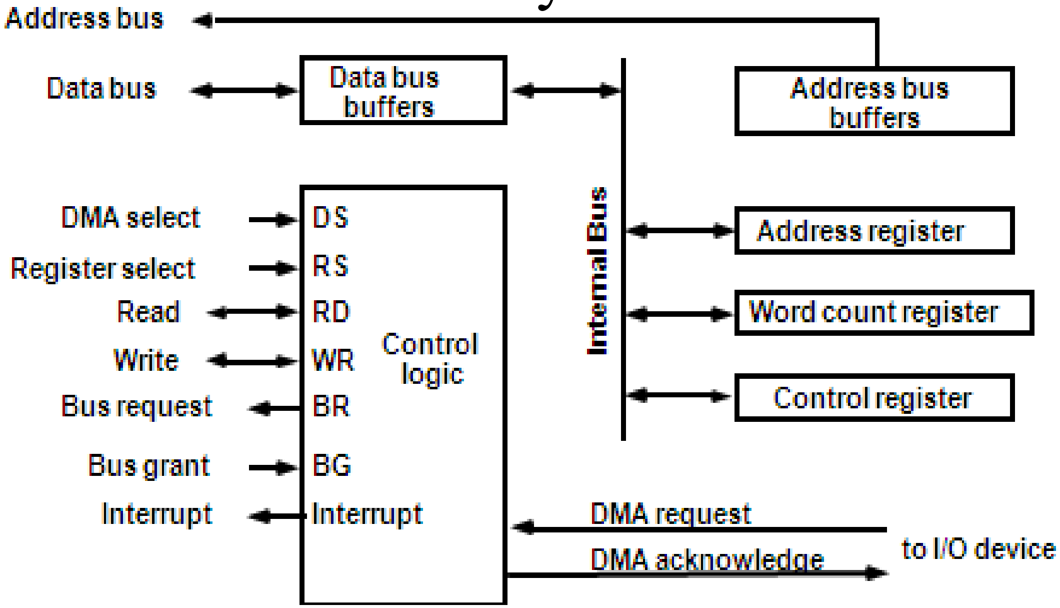
  In DMA burst transfer, a block sequence consisting of a number of memory words is transferred in a continuous burst while the DMA controller is mater of the memory buses.

- **CYCLE STEALING**

  An alternative technique called cycle stealing allows the DMA controller to transfer one data word at a time, after which it must return control of the buses to the CPU.

DMA Controller

- The DMA controller needs the usual circuits of an interface to communicate with the CPU and I/O device.
- The address register and lines are used for direct communication with the memory.
- The word count register specifies the number of words that must be transferred. The data transfer may be done directly between the device and memory under control of the DMA.



Working Of DMA Controller

- The unit communicate with the CPU via the data bus and control lines. The registers in the DMA are selected by the CPU through the address bus by enabling the DS(DMA select) and RS(register select) inputs. The RD(read) and WR(write) input are bidirectional.
- When the BG(bus grant) input is 0,the CPU can communicate with the DMA registers through the data bus to read from or write to the DMA registers. when BG=1,the CPU has relinquished the buses and the DMA can communicate directly with the memory by specifying an address in the address bus and activating the RD or WR control.
- The DMA communicate with the external peripheral through the request and acknowledge lines by using a prescribed handshaking procedure. The DMA controller has three registers: an address register, a word count register, and a control register.
- The address register contains an address to specify the desired location in memory. The address bits go through bus buffers in to the address bus.

DMA controller working

The CPU initializes the DMA by sending the following information through the data bus.

1.The starting address of the memory block where data are available(for read) or where data are to be stored(for write).

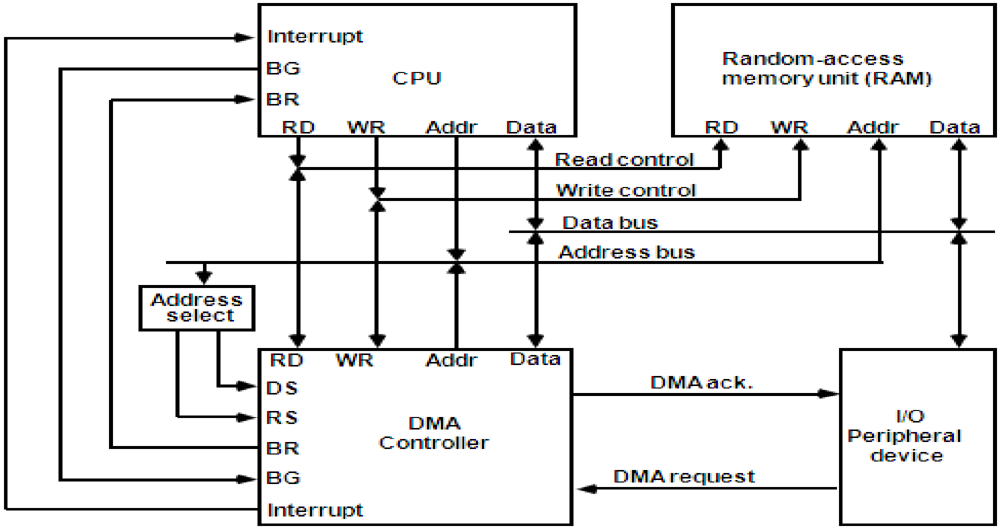
2.The word count, which is the number of words in the memory block.

3.Control to specify the mode of transfer such as read or write.

4.A control to start the DMA transfer.

DMA TRANSFER

- The CPU communicate with the DMA through the address and data buses as with any interface unit.
- The DMA has its own address, which activates the DS and RS lines
- The CPU initializes the DMA through the data bus. once the DMA receives the start control command, it can start the transfer between the peripheral device and the memory.



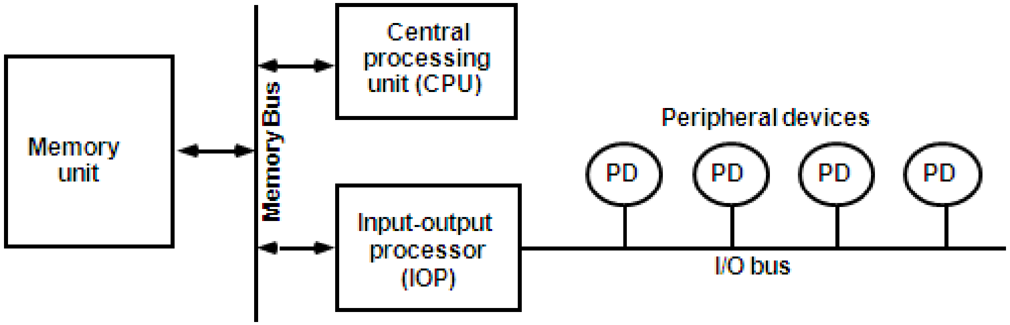
Input-Output Processor

IOP :

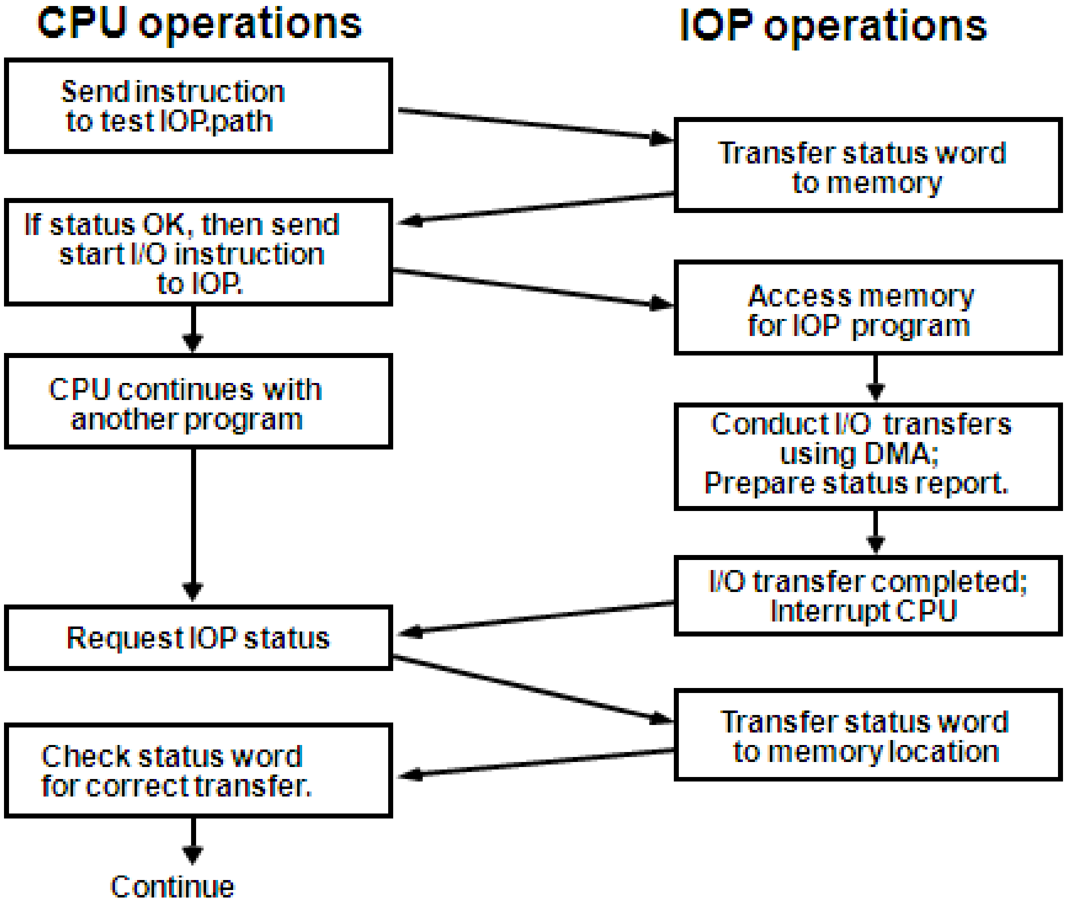
- Communicate directly with all I/O devices
- Fetch and execute its own instruction
  - IOP instructions are specifically designed to facilitate I/O transfer
  - DMAC must be set up entirely by the CPU
- Designed to handle the details of I/O processing

Command

- **Instruction** *that are read form memory by an* ***IOP***
  - Distinguish from instructions that are read by the CPU
  - Commands are prepared by experienced programmers and are stored in memory
  - Command word = IOP program



CPU-IOP Communication



CPU operations

IOP operations

Send instruction to test IOP.path

Transfer status word to memory

If status OK, then send start I/O instruction to IOP.

Access memory for IOP program

CPU continues with another program

Conduct I/O transfers using DMA; Prepare status report.

I/O transfer completed; Interrupt CPU

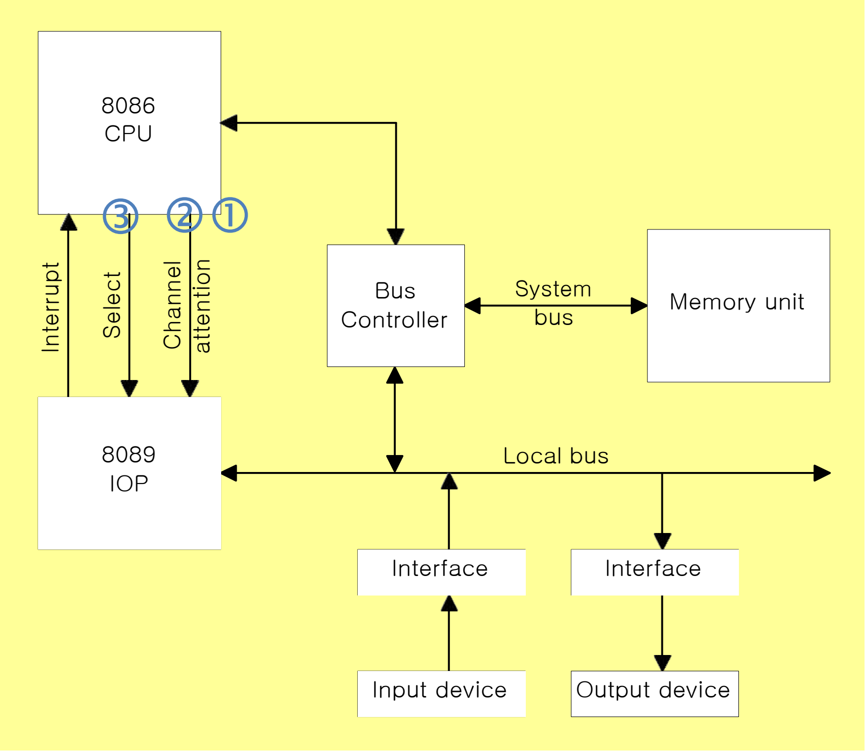
Request IOP status

Transfer status word to memory location

Check status word for correct transfer.

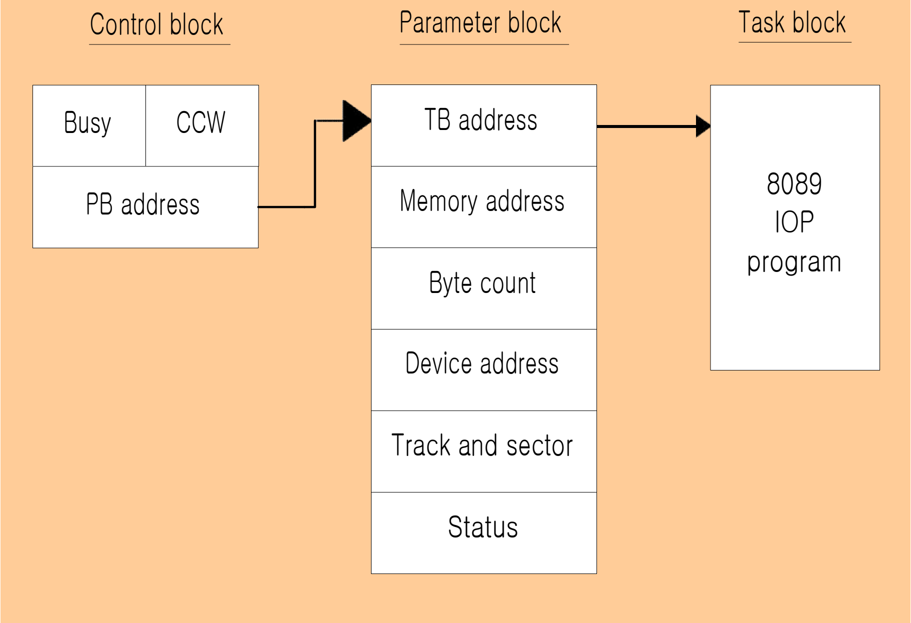
Continue

Intel 8089 IOP



① CPU enables channel attention
② Select one of two channels of 8089
③ 8089 gets attention of the CPU by sending an interrupt request

◆ Location of Information :



- Channel Command Word (CCW) : **message center**
  - » Start command
  - » Suspend command
  - » Resume command
  - » Halt command

UNTI-1

- **Operating systems overview-operating systems functions**
- **overview of computer operating systems**
- **protection and security**
- **distributed systems**
- **special purpose systems**
- **operating systems structures-operating system services , system calls, system programs, operating system structures, operating system generations**

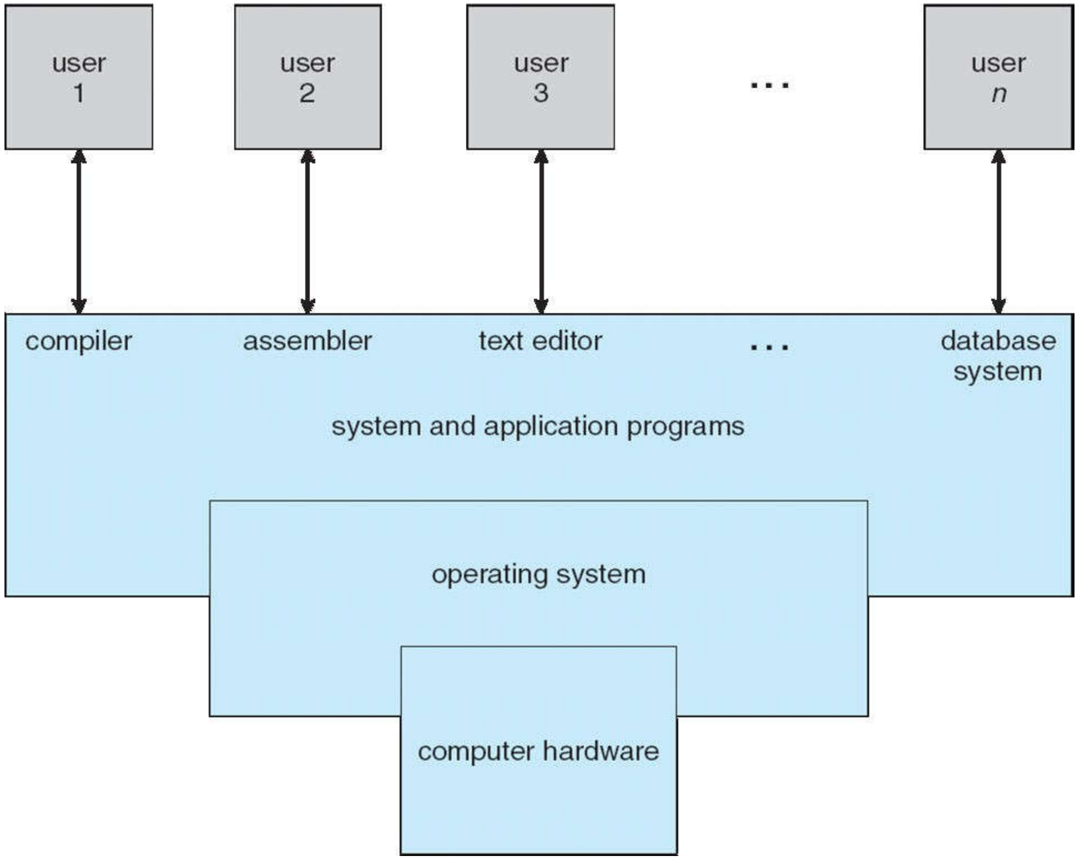
What is an Operating System?

- A program that acts as an intermediary between a user of a computer and the computer hardware
- Operating system goals:
  - Execute user programs and make solving user problems easier
  - Make the computer system convenient to use
  - Use the computer hardware in an efficient manner

What operating systems do

- computer system can be divided roughly into four components the hardware, the operating system, the application programs and the users.
- The hardware – which consists of CPU, memory and I/O devices, provides the basic computing resources for the system.
- The application programs define the ways in which these resources are used to solve users' computing problems. The operating system controls and co-ordinates the use of hardware among the various application programs for the various users.

Four Components of a Computer System



Operating system from the user view

- The user's view of the computer varies according to the interface being used.
- While designing a PC for one user, the goal is to maximize the work that the user is performing.
- Here OS is designed mostly for ease of use. In another case the user sits at a terminal connected to a main frame or minicomputer. Other users can access the same computer through other terminals.

Operating system from the user view

- OS here is designed to maximize resource utilization to assure that all available CPU time, memory and I/O are used efficiently.
- In other cases, users sit at workstations connected to networks of other workstations and servers.
- These users have dedicated resources but they also share resources such as networking and servers. Here OS is designed to compromise between individual usability and resource utilization.

Operating system from the system view

- From the computer’s point of view, OS is the program which is widely involved with hardware.

- Hence OS can be viewed as resource allocator where in resources are – CPU time, memory space, file storage space, I/O devices etc.

- OS must decide how to allocate these resources to specific programs and users so that it can operate the computer system efficiently.

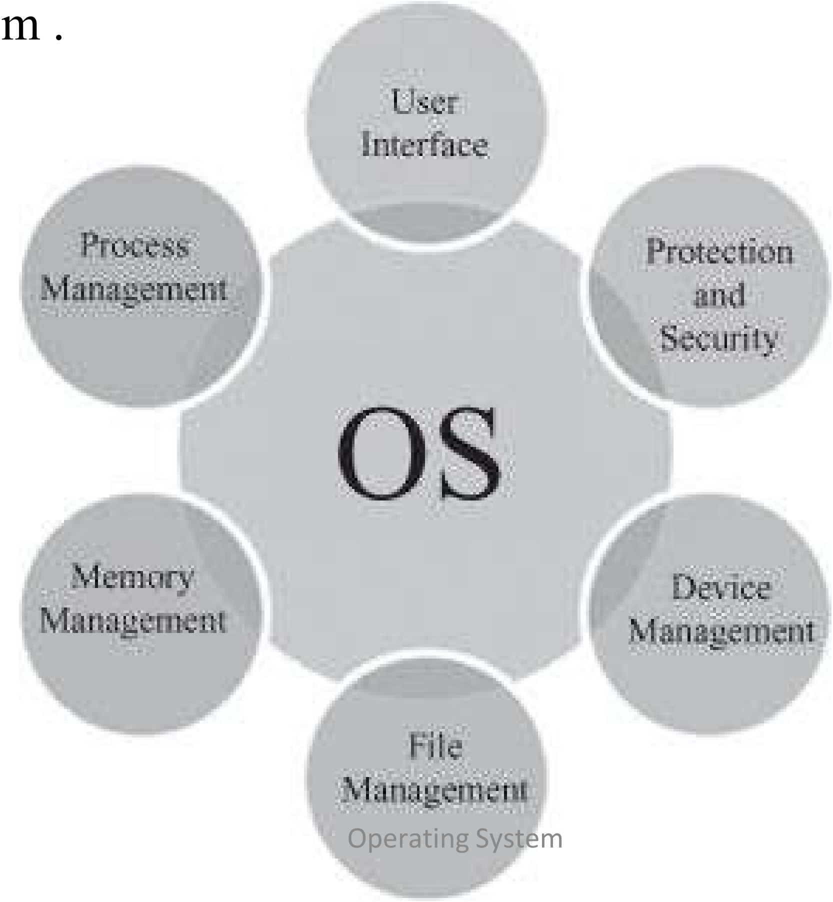
Operating system from the system view

- OS is also a control program. A control program manages the execution of user programs to prevent errors and improper use of computer.

- It is concerned with the operation and control of I/O devices. Defining operating systems- OS exists because they offer a reasonable way to solve the problem of creating a usable computing system. Goal of computer systems is to execute user program and to make solving user problems easier.
- Hence hardware is constructed. Since hardware alone is not easy to use, application programs are develope**d.**

Operating systems functions

- Operating system is a large and complex software consisting of several components.
- Each component of the operating system has its own set of defined inputs and outputs.
- Different components of OS perform specific tasks to provide the overall functionality of the operating system .



Operating systems functions

- **Process Management**— The process management activities handled by the OS are—

i. control access to shared resources like file, memory, I/O and CPU

ii. control execution of applications

iii. create, execute and delete a process (system process or user process)

iv. cancel or resume a process

v. schedule a process

vi. synchronization, communication and deadlock handling for processes.

Operating systems functions

- **memory Management**— The activities of memory management handled by OS are—(1) allocate memory, (2) free memory, (3) re-allocate memory to a program when a used block is freed, and (4) keep track of memory usage.

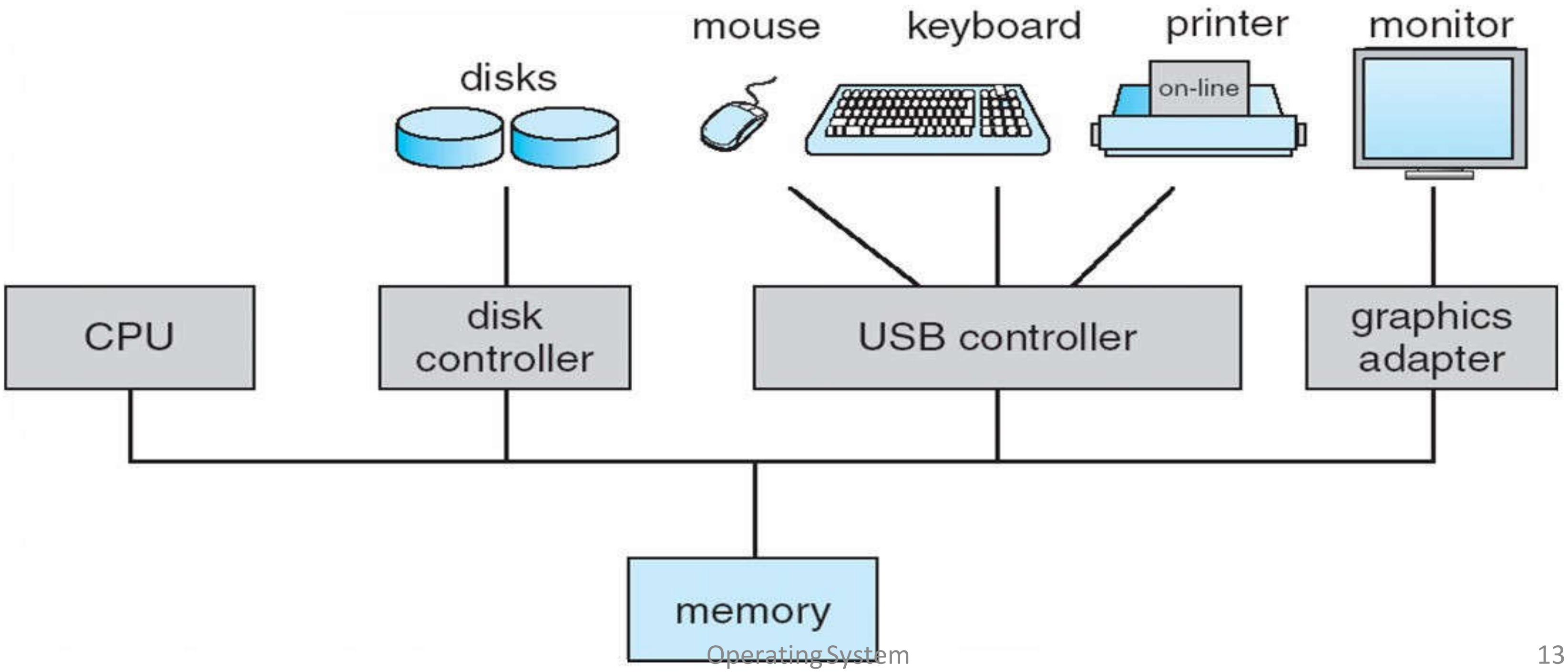
- **File Management**— The file management tasks include—(1) create and delete both files and directories, (2) provide access to files, (3) allocate space for files, (4) keep back-up of files, and (5) secure files.

Operating systems functions

- **Device Management**— The device management tasks handled by OS are—(1) open, close and write device drivers, and (2) communicate, control and monitor the device driver.
- **Protection and Security**— OS protects the resources of system. User authentication, file attributes like read, write, encryption, and back-up of data are used by OS to provide basic protection.
- **User Interface** or **Command Interpreter**— Operating system provides an interface between the computer user and the computer hardware. The user interface is a set of commands or a graphical user interface via which the user interacts with the applications and the hardware**.**

Overview of computer operating systems

- Computer-system operation
  - One or more CPUs, device controllers connect through common bus providing access to shared memory
  - Concurrent execution of CPUs and devices competing for memory cycles



Computer-System Operation

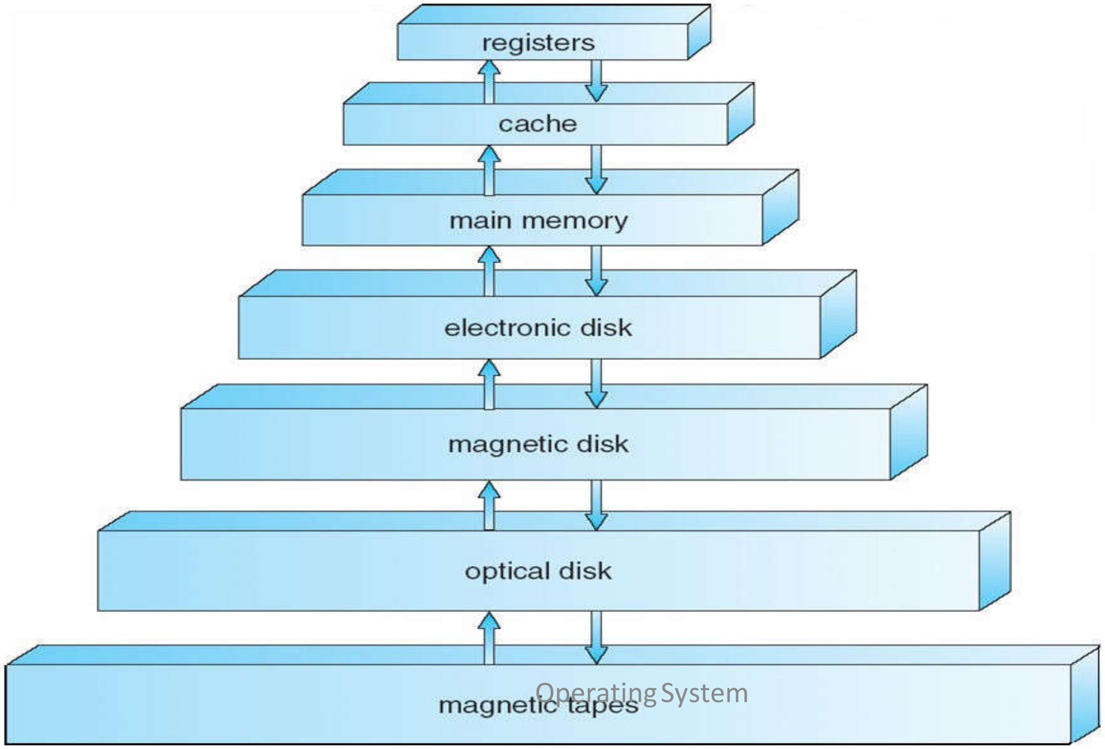
- I/O devices and the CPU can execute concurrently.
- Each device controller is in charge of a particular device type
- Each device controller has a local buffer
- CPU moves data from/to main memory to/from local buffers
- I/O is from the device to local buffer of controller
- Device controller informs CPU that it has finished its operation by causing an *interrupt*.

Storage Structure

- **Main memory** – only large storage media that the CPU can access directly
- **Secondary storage** – extension of main memory that provides large nonvolatile storage capacity
- **Magnetic disks** – rigid metal or glass platters covered with magnetic recording material
  - Disk surface is logically divided into **tracks**, which are subdivided into **sectors**
  - The **disk controller** determines the logical interaction between the device and the computer

Storage Hierarchy

- **Storage systems organized in hierarchy**
  - **Speed**
  - **Cost**
  - **Volatility**
- **Caching – copying information into faster storage system; main memory can be viewed as a last *cache* for secondary storage**



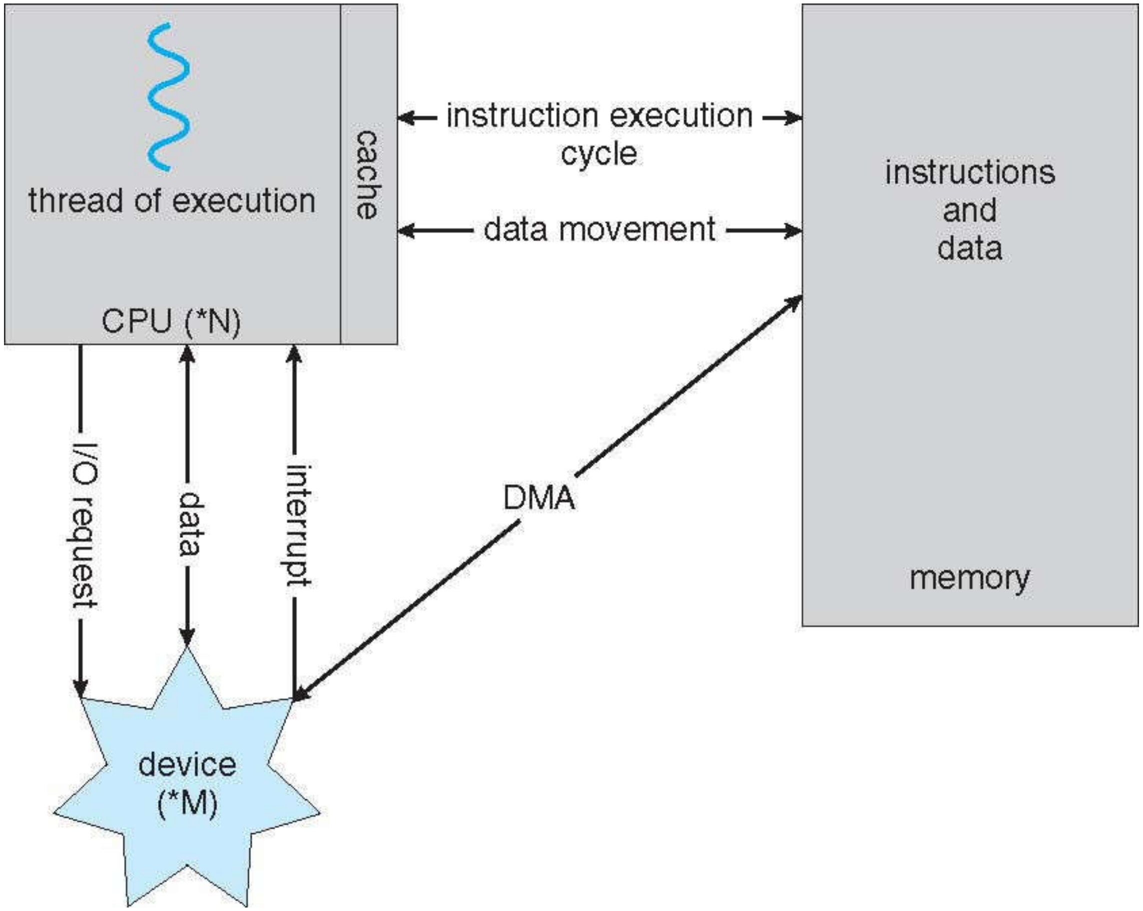
I/O Structure

- After I/O starts, control returns to user program only upon I/O completion
  - Wait instruction idles the CPU until the next interrupt
  - Wait loop (contention for memory access)
  - At most one I/O request is outstanding at a time, no simultaneous I/O processing
- After I/O starts, control returns to user program without waiting for I/O completion
  - **System call** – request to the operating system to allow user to wait for I/O completion
  - Device-status table contains entry for each I/O device indicating its type, address, and state
  - Operating system indexes into I/O device table to determine device status and to modify table entry to include interrupt.

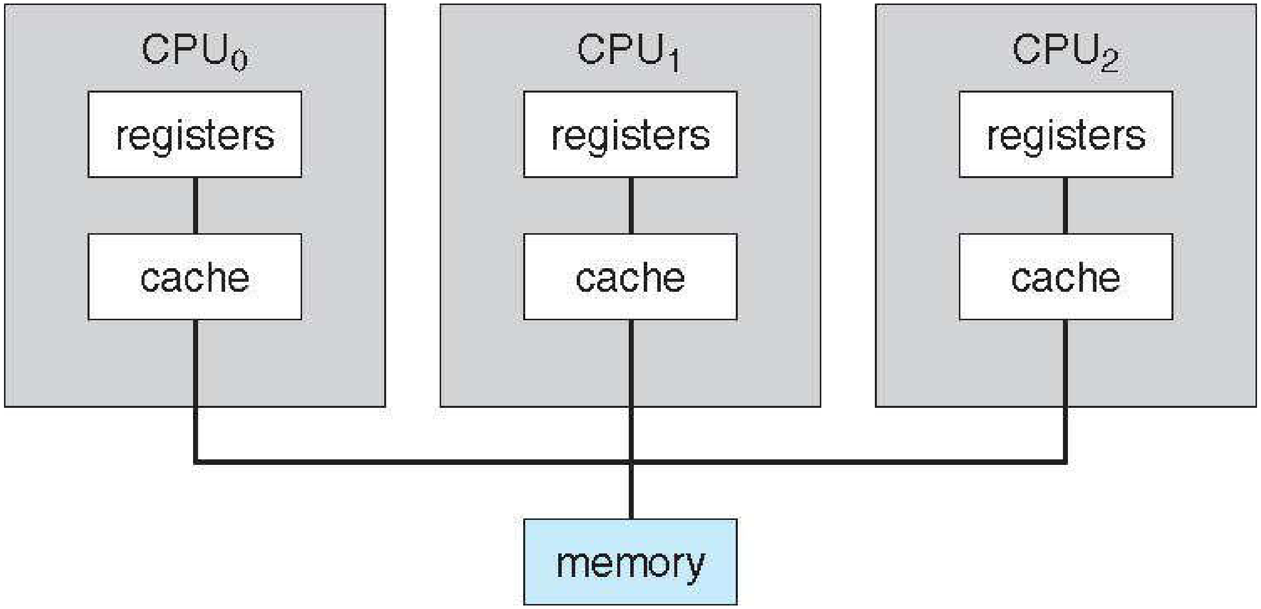
Computer-System Architecture

- Most systems use a single general-purpose processor (PDAs through mainframes)
  - Most systems have special-purpose processors as well
- **Multiprocessors** systems growing in use and importance
  - Also known as parallel systems, tightly-coupled systems
  - Advantages include
    1. Increased throughput
    2. Economy of scale
    3. Increased reliability – graceful degradation or fault tolerance
  - Two types
    1. Asymmetric Multiprocessing
    2. Symmetric Multiprocessing

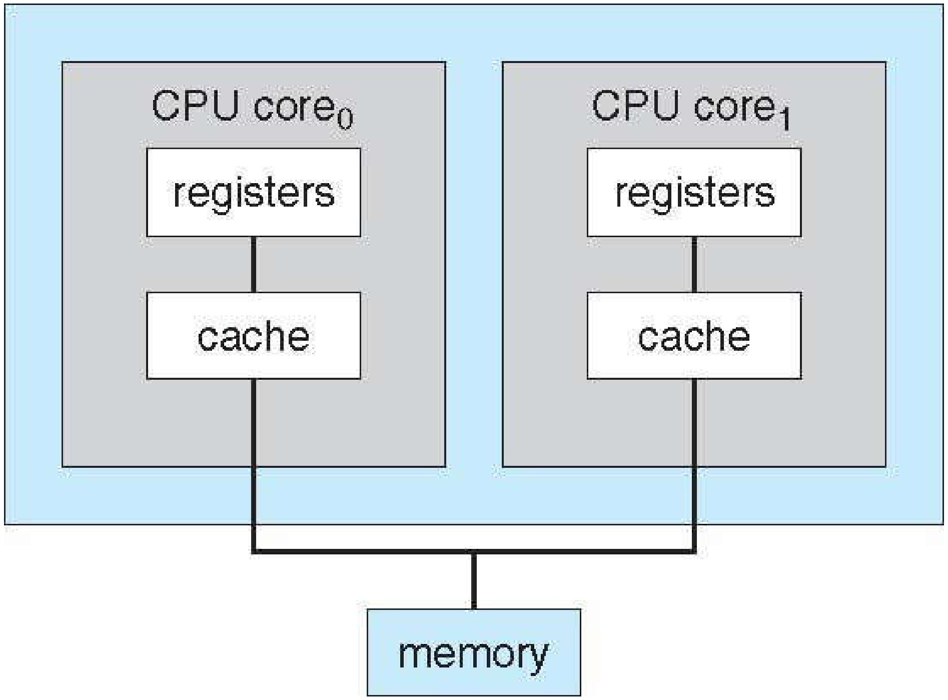
How a Modern Computer Works



Symmetric Multiprocessing Architecture



A Dual-Core Design



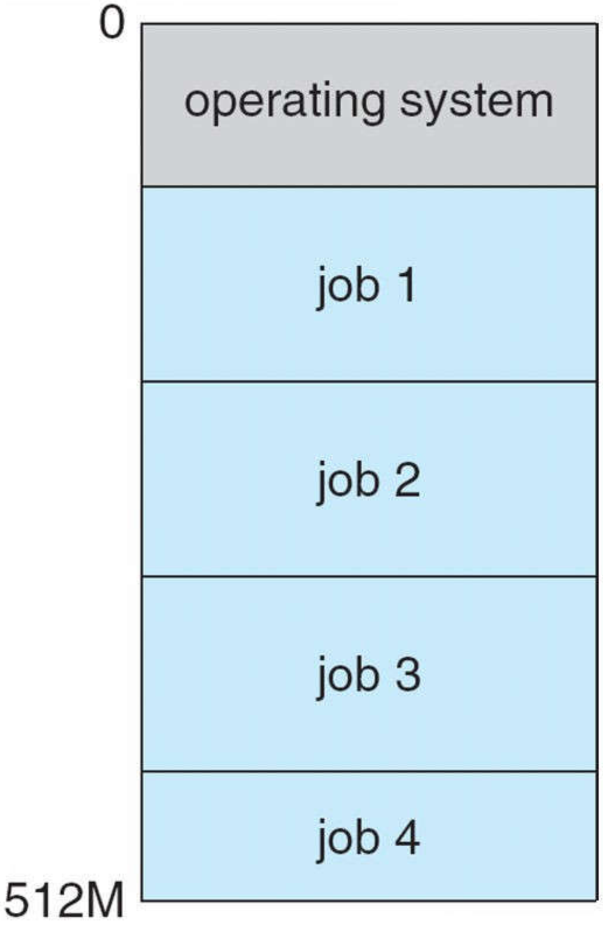
Clustered Systems

- Like multiprocessor systems, but multiple systems working together
  - Usually sharing storage via a storage-area network (SAN)
  - Provides a high-availability service which survives failures
    - Asymmetric clustering has one machine in hot-standby mode
    - Symmetric clustering has multiple nodes running applications, monitoring each other
  - Some clusters are for high-performance computing (HPC)
    - Applications must be written to use parallelization

Operating System Structure

- Multiprogramming needed for efficiency
  - Single user cannot keep CPU and I/O devices busy at all times
  - Multiprogramming organizes jobs (code and data) so CPU always has one to execute
  - A subset of total jobs in system is kept in memory
  - One job selected and run via job scheduling
  - When it has to wait (for I/O for example), OS switches to another job
- Timesharing (multitasking) is logical extension in which CPU switches jobs so frequently that users can interact with each job while it is running, creating interactive computing
  - Response time should be < 1 second
  - Each user has at least one program executing in memory ⇆process
  - If several jobs ready to run at the same time ⇆ CPU scheduling
  - If processes don't fit in memory, swapping moves them in and out to run
  - Virtual memory allows execution of processes not completely in memory

Memory Layout for Multiprogrammed System



Operating-System Operations

- Interrupt driven by hardware
- Software error or request creates exception or trap
  - Division by zero, request for operating system service
- Other process problems include infinite loop, processes modifying each other or the operating system
- Dual-mode operation allows OS to protect itself and other system components
  - User mode and kernel mode
  - Mode bit provided by hardware
    - Provides ability to distinguish when system is running user code or kernel code
    - Some instructions designated as privileged, only executable in kernel mode
    - System call changes mode to kernel, return from call resets it to user

Protection and security-

- **If a computer system has multiple users and allows the concurrent execution of multiple processes, then access to data must be regulated. Hence mechanisms ensure that files, memory segments, CPU and other resources can be operated on by only those processes that have gained proper authorization from the OS.**

- **Protection is a mechanism for controlling the access of processes or users to the resources defined by a computer system. This mechanism must provide means for specification of the controls to be imposed and means for enforcement. Protection improves reliability by detecting latent errors at the interfaces between component sub systems.**

- **It is the job of security to defend a system from external and internal attacks. Such attacks spread across a huge range and include viruses and worms, denial of service attacks, identity theft and theft of service.**

Distributed Systems

- **A distributed system is a collection of physically separate, possibly heterogeneous computer systems that are networked to provide the users with access to the various resources that the system maintains.**

- **Access to a shared resource increases computation speed, functionality, data availability and reliability.**

- **The protocols that create a distributed system can greatly affect that system's utility and popularity. A network is a communication path between two or more systems.**

Distributed Systems

- **Distributed systems depend on networking for their functionality. Networks are characterized based on the distances between their nodes.**

- **A local area network (LAN) connects computers within a room, a floor or a building.**

- **A wide area network (WAN) links buildings, cities or countries.**

- **A metropolitan area network (MAN) could link buildings within a city.**

Special Purpose Systems

- Classes of computers whose functions are limited and objective is to deal with limited computation domains.
- **Real Time Embedded Systems**: Embedded computers are devices found from car engines and manufacturing robots to VCR's and microwave ovens. These have specific tasks to accomplish. Embedded systems almost always run real time operating system.
- **Multimedia systems**: Most operating systems are designed to handle conventional data such as text files, programs, and word processing documents and spread sheets. A recent trend is incorporation of multimedia data into computer systems. Multimedia data consist of audio and video files as well as conventional files.
- **Handheld systems**: Handheld systems include personal digital assistants (PDA's), cellular telephones many of which use special purpose embedded operating systems.

Operating systems structures

OS services-

- **OS provides an environment for execution of programs. It provides certain services to programs and to the users of those programs.**
- **OS services are provided for the convenience of the programmer, to make the programming task easier.**

One set of SOS services provides functions that are helpful to the user –

a. User interface**: All OS have a user interface(UI).Interfaces are of three types-**

Command Line Interface**: uses text commands and a method for entering them Batch interface: commands and directives to control those commands are entered into files and those files are executed. Graphical user interface: This is a window system with a pointing device to direct I/O, choose from menus and make selections and a keyboard to enter text.**

- **b. Program execution**: System must be able to load a program into memory and run that program. The program must be able to end its execution either normally or abnormally.
- **c. I/O operations**: A running program may require I/O which may involve a file or an I/O device. For efficiency and protection, users cannot control I/O devices directly.
- **d. File system manipulation**: Programs need to read and write files and directories. They also need to create and delete them by name, search for a given file, and list file information.
- **e. Communications**: One process might need to exchange information with another process. Such communication may occur between processes that are executing on the same computer or between processes that are executing on different computer systems tied together by a computer network. Communications may be implemented via shared memory or through message passing.

System Calls

- **Programming interface to the services provided by the OS**

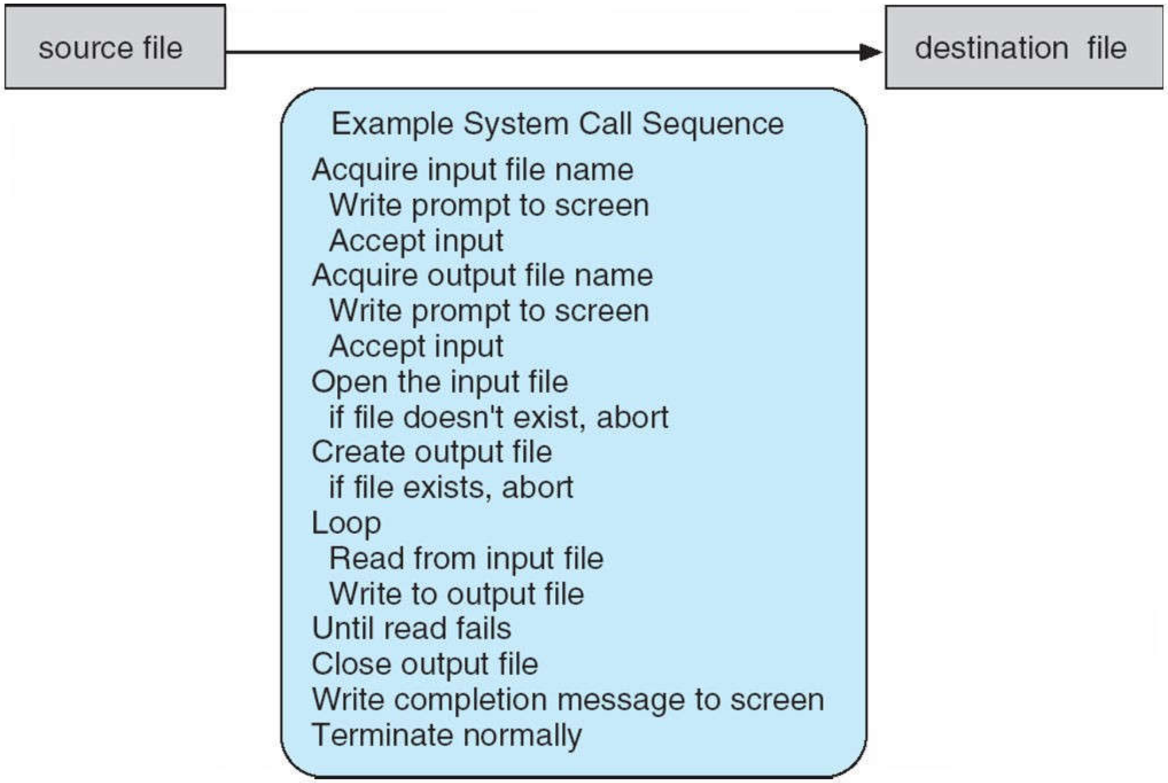
- **Typically written in a high-level language (C or C++)**

- **Mostly accessed by programs via a high-level Application Program Interface (API) rather than direct system call use**

- **Three most common APIs are Win32 API for Windows, POSIX API for POSIX-based systems (including virtually all versions of UNIX, Linux, and Mac OS X), and Java API for the Java virtual machine (JVM)**

Example of System Calls

- System call sequence to copy the contents of one file to another file



source file

destination file

Example System Call Sequence

```
Acquire input file name
  Write prompt to screen
  Accept input
Acquire output file name
  Write prompt to screen
  Accept input
Open the input file
  if file doesn't exist, abort
Create output file
  if file exists, abort
Loop
  Read from input file
  Write to output file
Until read fails
Close output file
Write completion message to screen
Terminate normally
```

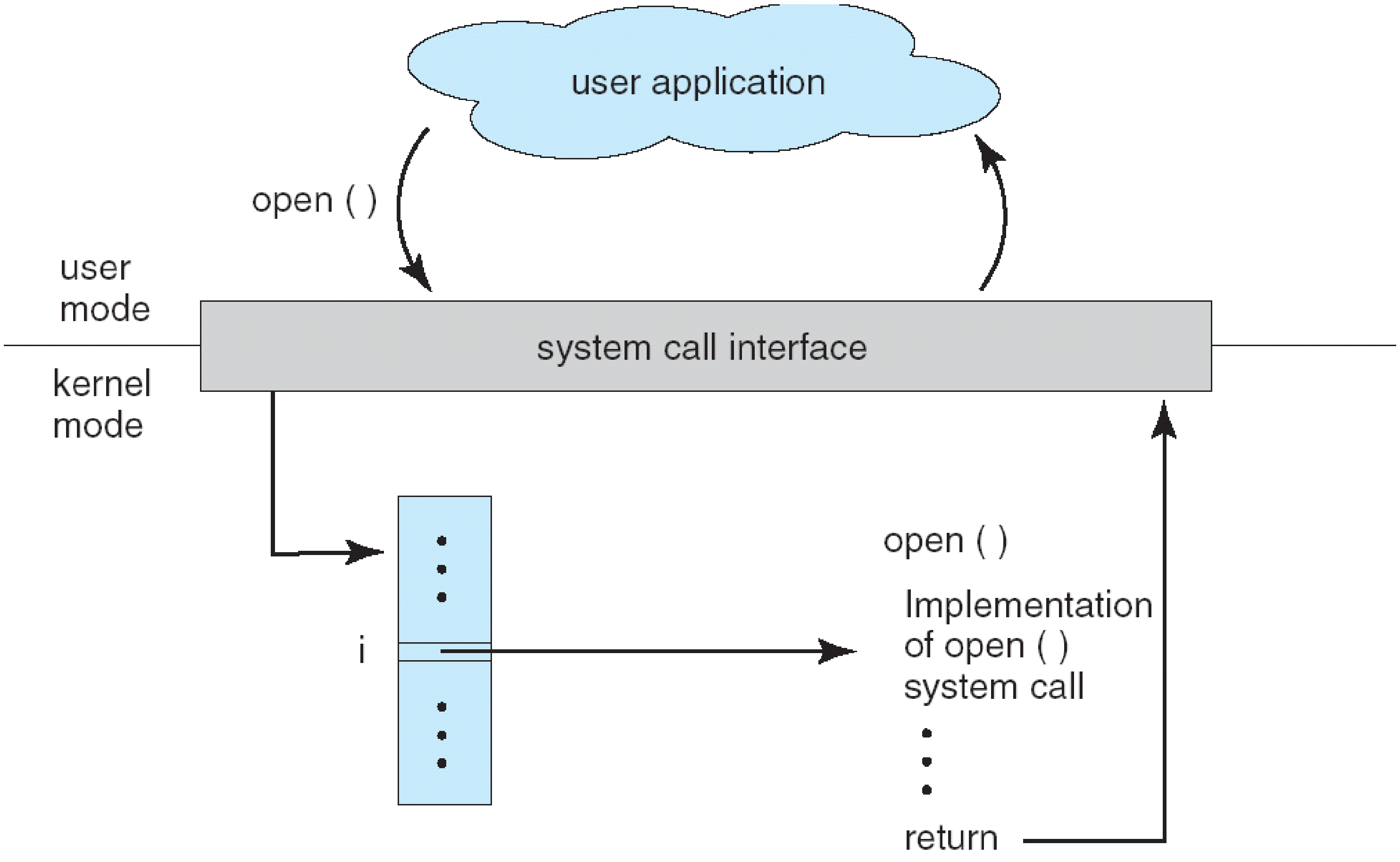
Example of Standard API

- Consider the ReadFile() function in the
- Win32 API—a function for reading from a file
- A description of the parameters passed to ReadFile()
  - HANDLE file—the file to be read
  - LPVOID buffer—a buffer where the data will be read into and written from
  - DWORD bytesToRead—the number of bytes to be read into the buffer
  - LPDWORD bytesRead—the number of bytes read during the last read
  - LPOVERLAPPED ovl—indicates if overlapped I/O is being used

System Call Implementation

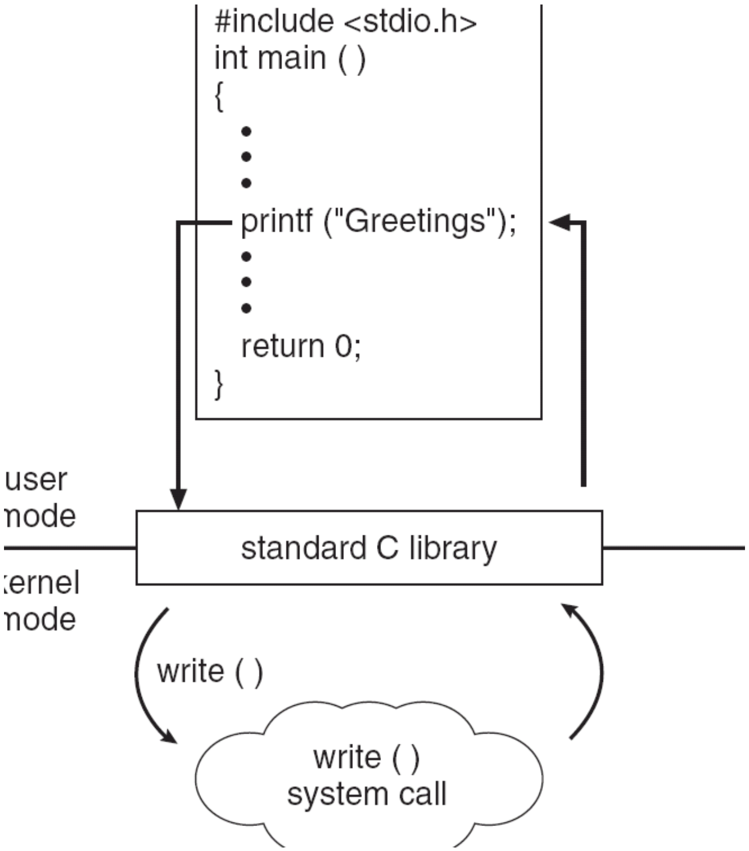
- **Typically, a number associated with each system call**
  - **System-call interface maintains a table indexed according to these numbers**
- **The system call interface invokes intended system call in OS kernel and returns status of the system call and any return values**
- **The caller need know nothing about how the system call is implemented**
  - **Just needs to obey API and understand what OS will do as a result call**
  - **Most details of OS interface hidden from programmer by API**
    - **Managed by run-time support library (set of functions built into libraries included with compiler)**

API – System Call – OS Relationship



Standard C Library Example

- C program invoking printf() library call, which calls write() system call

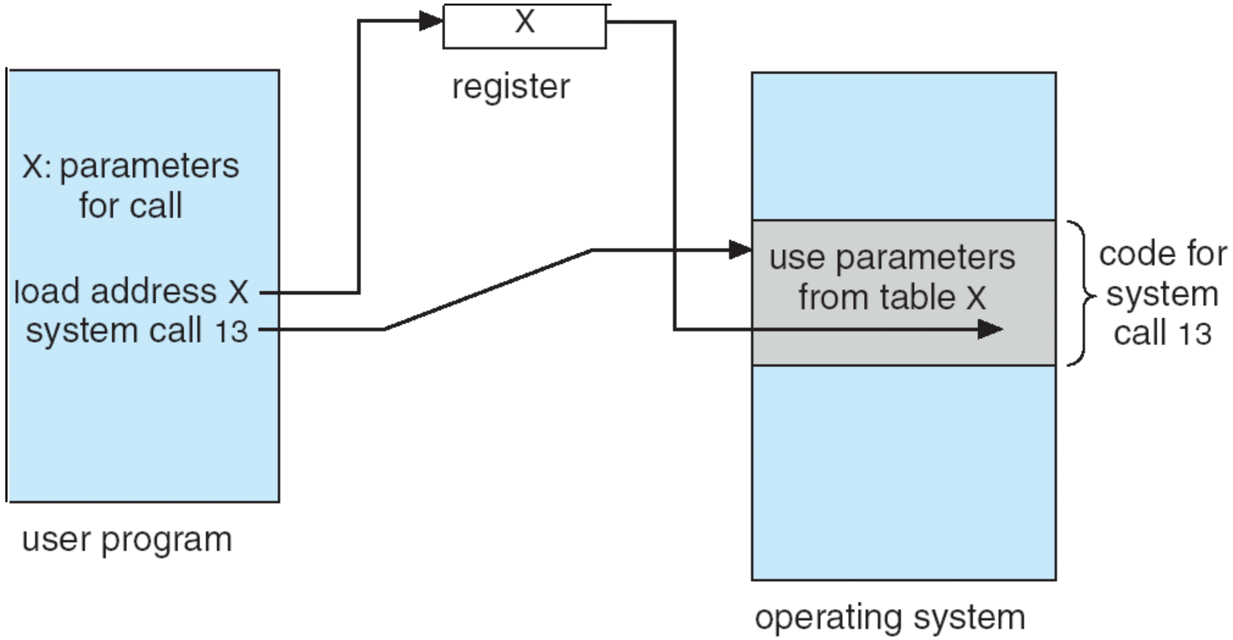


```
#include <stdio.h>
int main ( )
{
  •
  •
  •
  printf ("Greetings");
  •
  •
  •
  return 0;
}
```

System Call Parameter Passing

- **Often, more information is required than simply identity of desired system call**
  - **Exact type and amount of information vary according to OS and call**
- **Three general methods used to pass parameters to the OS**
  - **Simplest: pass the parameters in *registers***
    - **In some cases, may be more parameters than registers**
  - **Parameters stored in a *block,* or table, in memory, and address of block passed as a parameter in a register**
    - **This approach taken by Linux and Solaris**
  - **Parameters placed, or *pushed,* onto the *stack* by the program and *popped* off the stack by the operating system**
  - **Block and stack methods do not limit the number or length of parameters being passed**

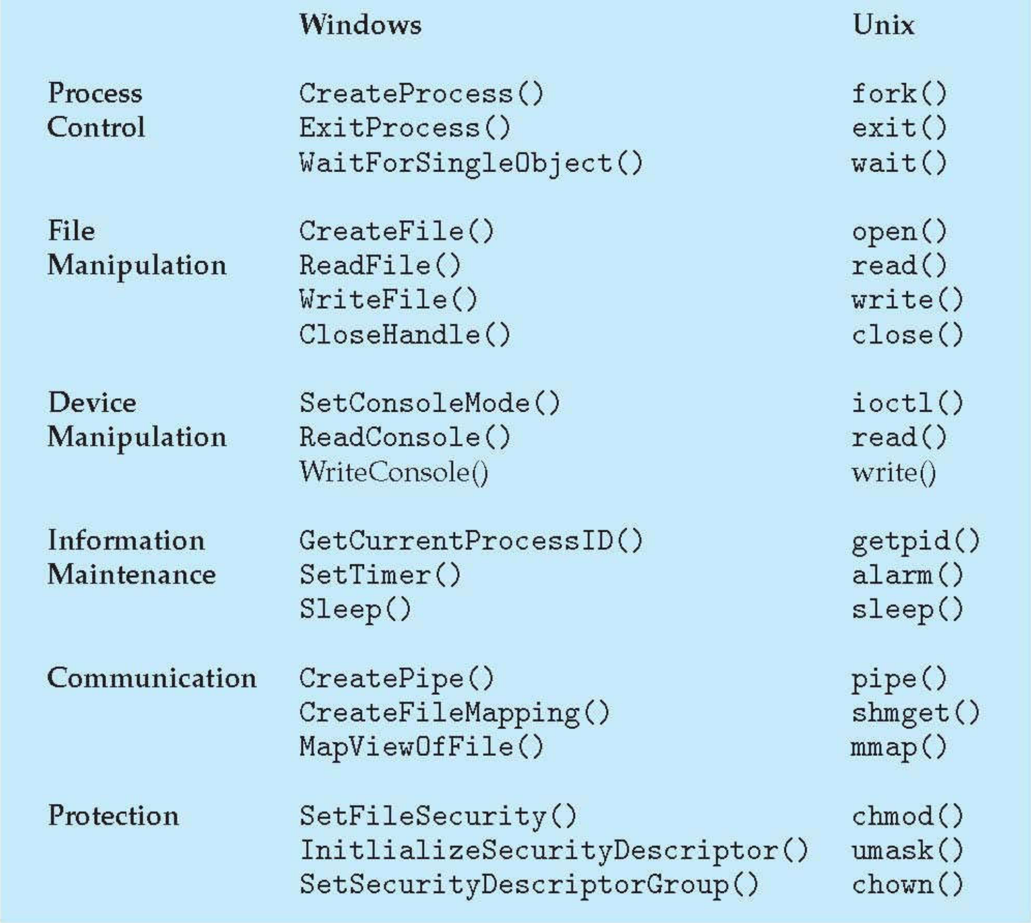
Parameter Passing via Table



Types of System Calls

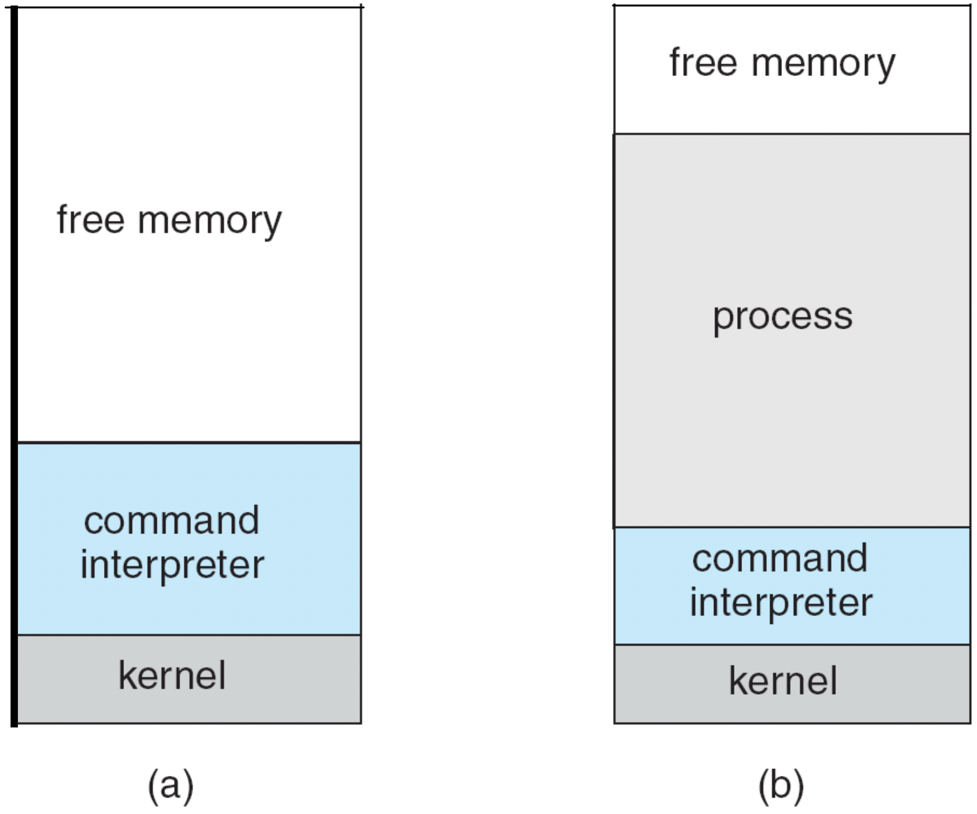
- **Process control**
- **File management**
- **Device management**
- **Information maintenance**
- **Communications**
- **Protection**

Examples of Windows and Unix System Calls



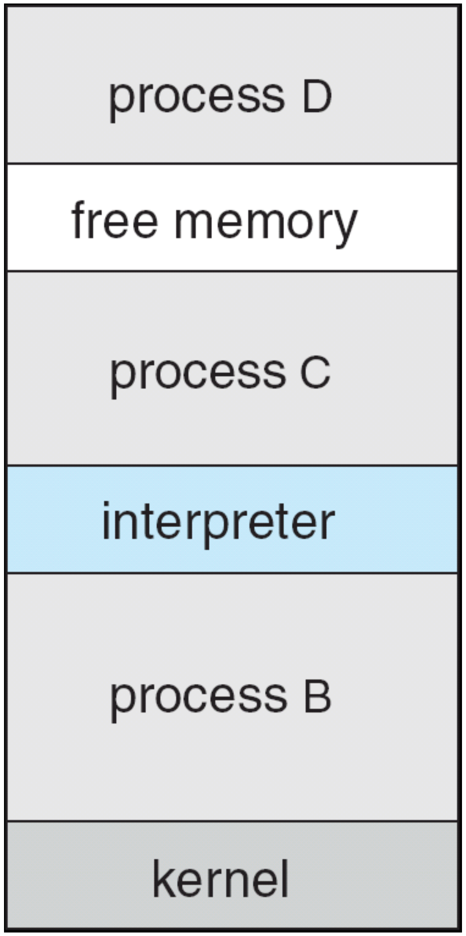
	Windows	Unix
Process Control	CreateProcess()	fork()
	ExitProcess()	exit()
	WaitForSingleObject()	wait()
File Manipulation	CreateFile()	open()
	ReadFile()	read()
	WriteFile()	write()
	CloseHandle()	close()
Device Manipulation	SetConsoleMode()	ioctl()
	ReadConsole()	read()
	WriteConsole()	write()
Information Maintenance	GetCurrentProcessID()	getpid()
	SetTimer()	alarm()
	Sleep()	sleep()
Communication	CreatePipe()	pipe()
	CreateFileMapping()	shmget()
	MapViewOfFile()	mmap()
Protection	SetFileSecurity()	chmod()
	InitlializeSecurityDescriptor()	umask()
	SetSecurityDescriptorGroup()	chown()

MS-DOS execution



(a) At system startup (b) running a program

FreeBSD Running Multiple Programs



System Programs

- System programs provide a convenient environment for program development and execution. They can be divided into these categories-

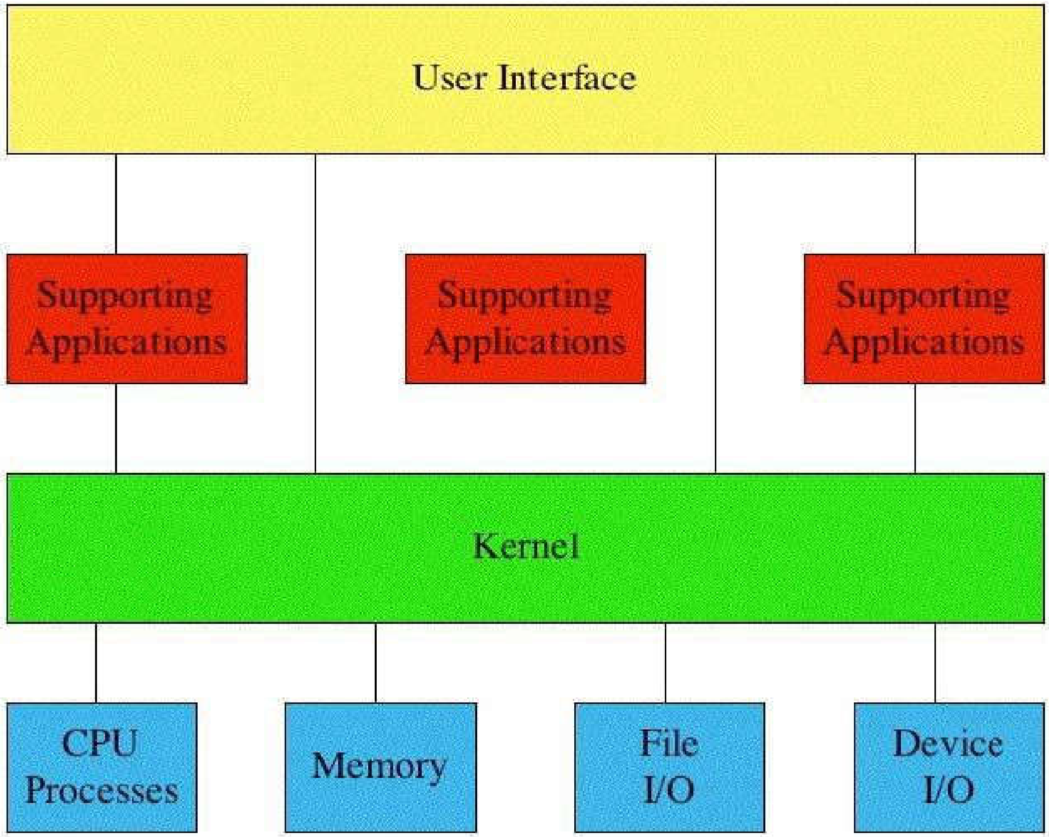
- **File management:** These programs create, delete, copy, rename, print, dump, list and manipulate files and directories.

- **Status information:** Some programs ask the system for the date, time, and amount of available memory or disk space, number of users.
-
- **File modification**: Text editors may be available to create and modify the content of files stored on disk or other storage devices.

System Programs

- **Programming language support:** Compilers, assemblers, debuggers and interpreters for common programming languages are often provided to the user with the OS.
- **Program loading and execution:** Once a program is assembled or compiled, it must be loaded into memory to be executed. System provides absolute loaders, relocatable loaders, linkage editors and overlay loaders.
- **Communications:** These programs provide the mechanism for creating virtual connections among processes, users and computer systems.

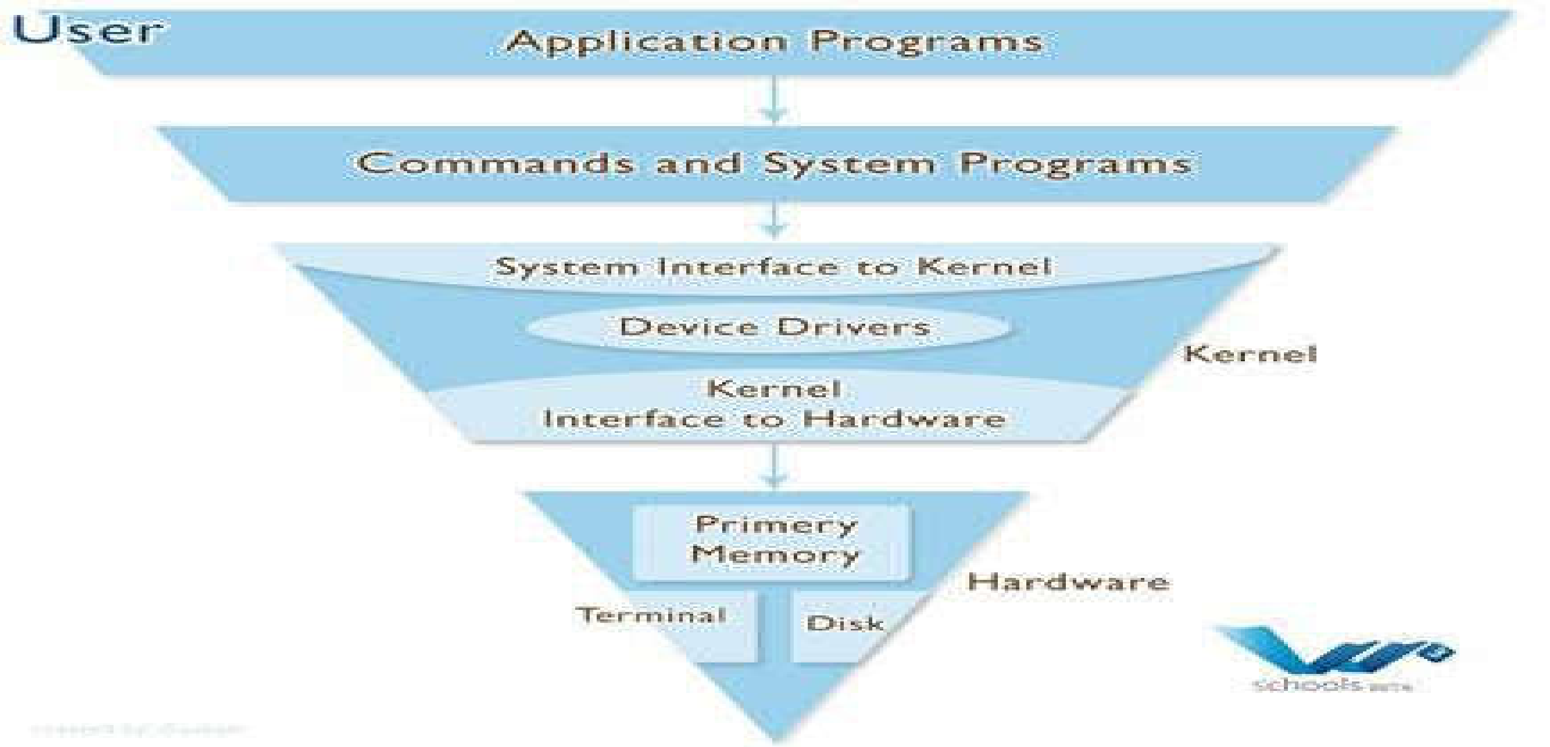
Operating System Structure



Layered approach:

- **With proper hardware support, OS can be broken into pieces that are smaller and more appropriate.**
- **OS can then retain much greater control over the computer and over the applications that make use of the computer.**
- **Under the top down approach, the overall functionality and features are determined and are separated into components.**
- **A system can be made modular in many ways – one method is the layered approach in which the OS is broken up into number of layers (levels).**
- **The bottom layer is the hardware and the highest layer is the user interface**

Layered approach



Micro kernels

- **This method structures the OS by removing all non essential components from the kernel and implementing them as system and user level programs which results in a smaller kernel.**

- **Micro kernels provide minimal process and memory management in addition to a communication facility.**

- **The main function of micro kernel is to provide a communication facility between the client program and the various services that are also running in user space.**

- **Communication is provided by message passing. Advantage of the micro kernel approach is ease of extending the operating system.**

Micro kernels

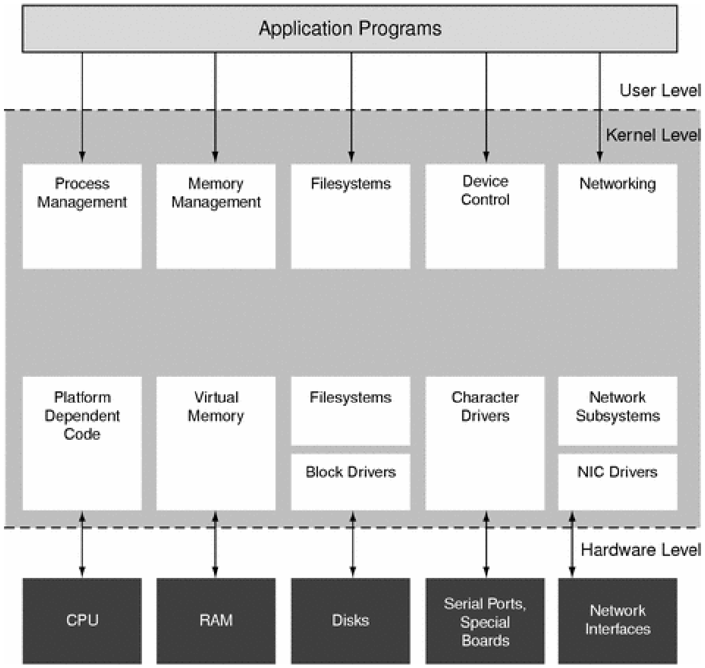
- **All new users are added to user space and hence do not require modification of the kernel.**

- **The resulting operating system is easier to port from one hardware design to another. Microkernel also provides more security and reliability since most services are running as user processes.**

- **But micro kernels can suffer from performance decreases due to increased system function over head.**

Modules:

The best current methodology for operating system design involves using object oriented programming techniques to create a modular kernel. The kernel has a set of core components and dynamically links in additional services either during boot time or run time.



Solaris loadable kernel modules

Solaris OS structure is organized around a core kernel with seven types of loadable kernels. Such a design allows the kernel to provide core services and also allows certain features to be implemented dynamically.

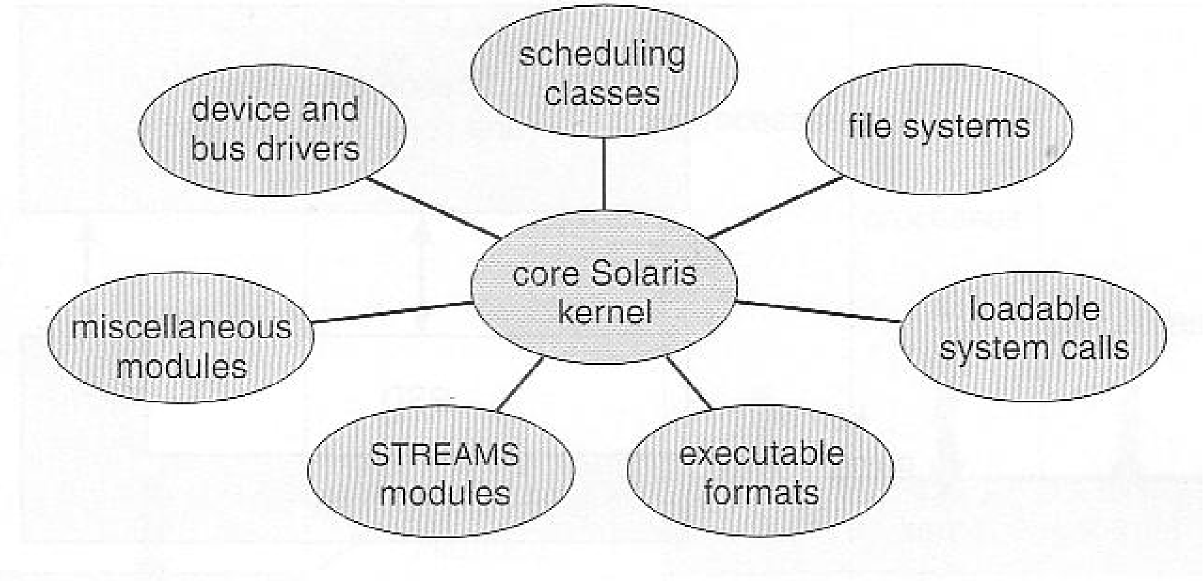


Figure 2.13 Solaris loadable modules.

Operating System Generation

- Operating systems are designed to run on any class of machines at a variety of sites with a variety of peripheral configurations. The system must then be configured or generated for
- each specific computer site, a process known as system generation (SYSGEN). This SYSGEN program reads from a given file or asks the operator of the system for information concerning the specific configuration of the hardware system or probes the hardware directly to determine what components are there. The following information must be determined:
- a) What CPU is to be used? What options are installed? For multiple CPU systems, each CPU system must be described.
- b) How much memory is available?
- c) What devices are available?
- d) What operating system options are desired or what parameter values are to be used?

Operating System Generation

- Once this information is determined, it can be used in several ways. It can be used by the system administrator to modify a copy of the source code of the OS. OS is then completely compiled.
- All the code is always part of the system and selection occurs at execution time rather than compile time or link time.
- The major differences among these approaches are the size and generality of the generated system and the ease of modification as the hardware configuration changes.

Unit 4

Memory Management

Memory Management

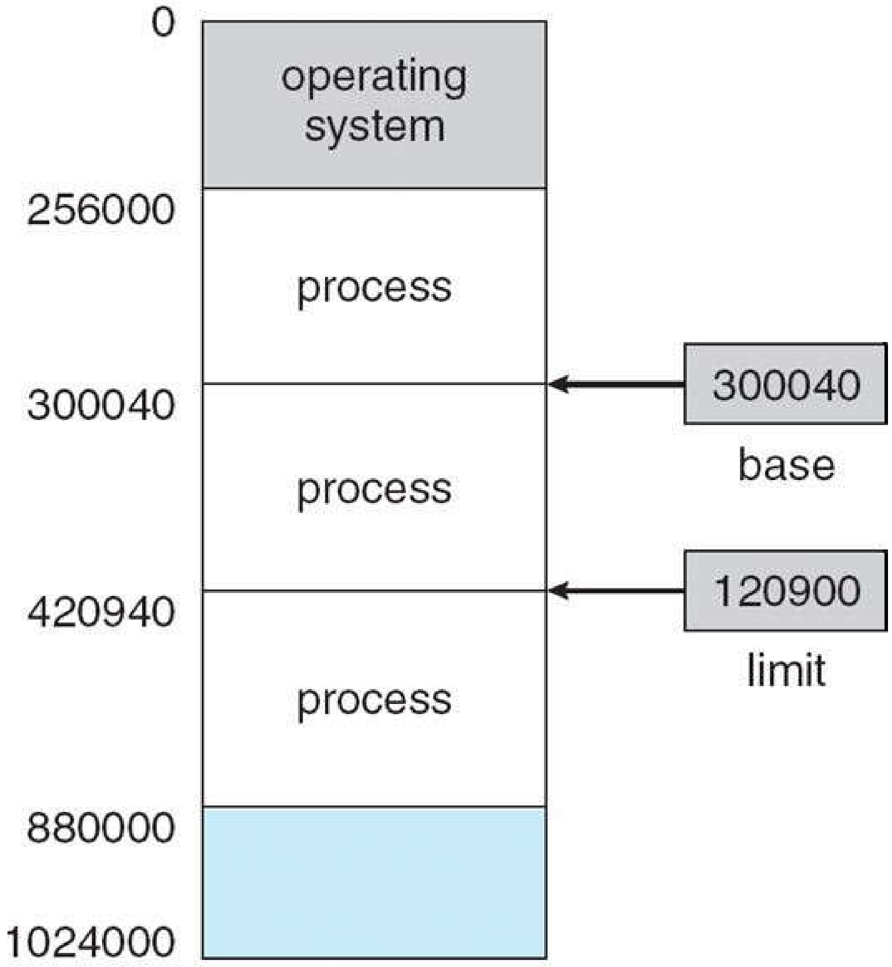
- **Background**
- **Swapping**
- **Contiguous memory allocation**
- **Paging**
- **Structure of the page table**
- **Segmentation**
- **Virtual memory**
- **Demand paging**
- **Page- replacement**
- **Algorithms**
- **Allocation of frames**
- **Thrashing**

Background

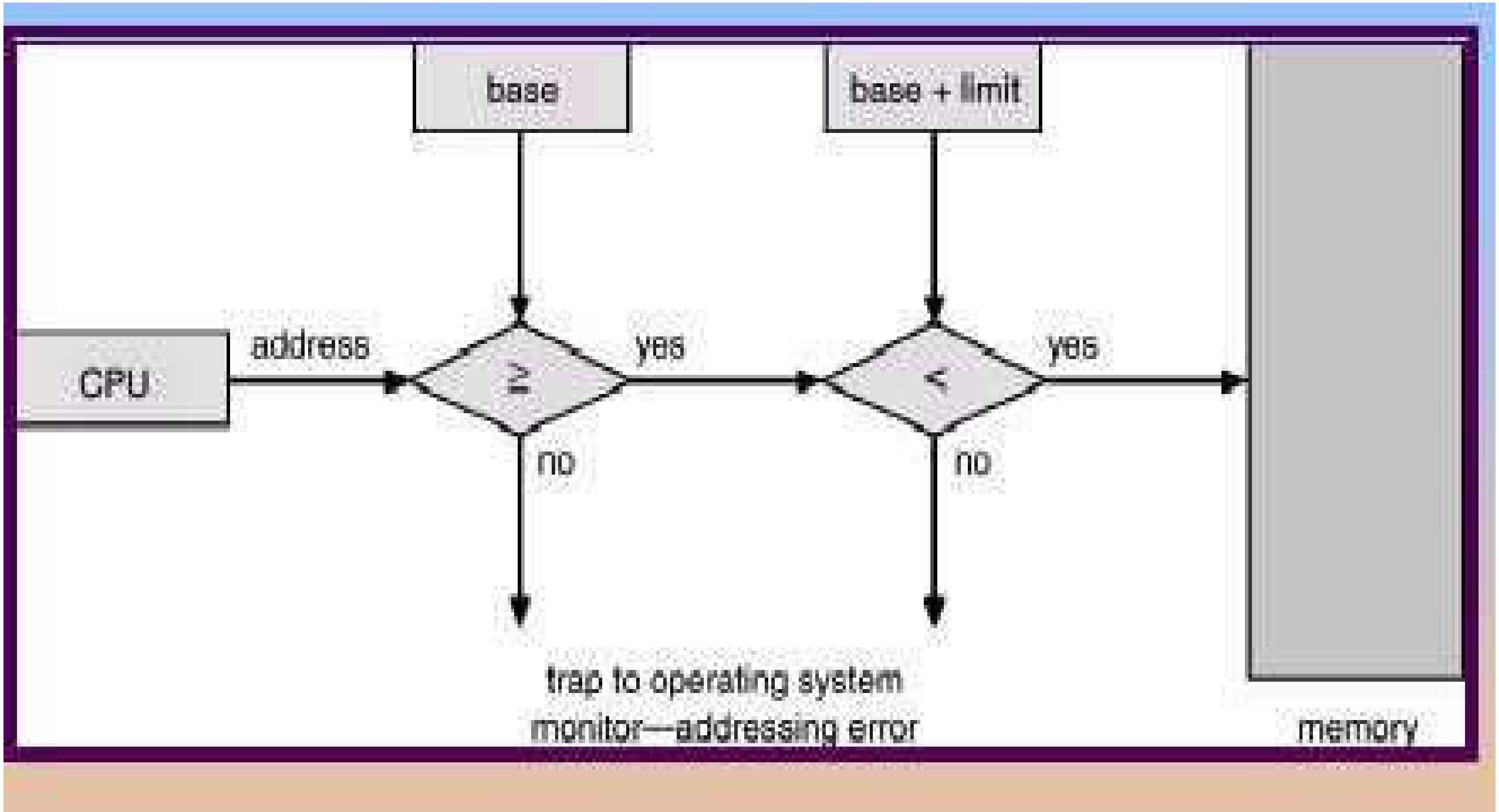
- Program must be brought (from disk) into memory and placed within a process for it to be run
- Main memory and registers are only storage CPU can access directly
- Register access in one CPU clock (or less)
- Main memory can take many cycles
- Cache sits between main memory and CPU registers
- Protection of memory required to ensure correct operation

Base and Limit Registers

- A pair of **base** and **limit** registers define the logical address space



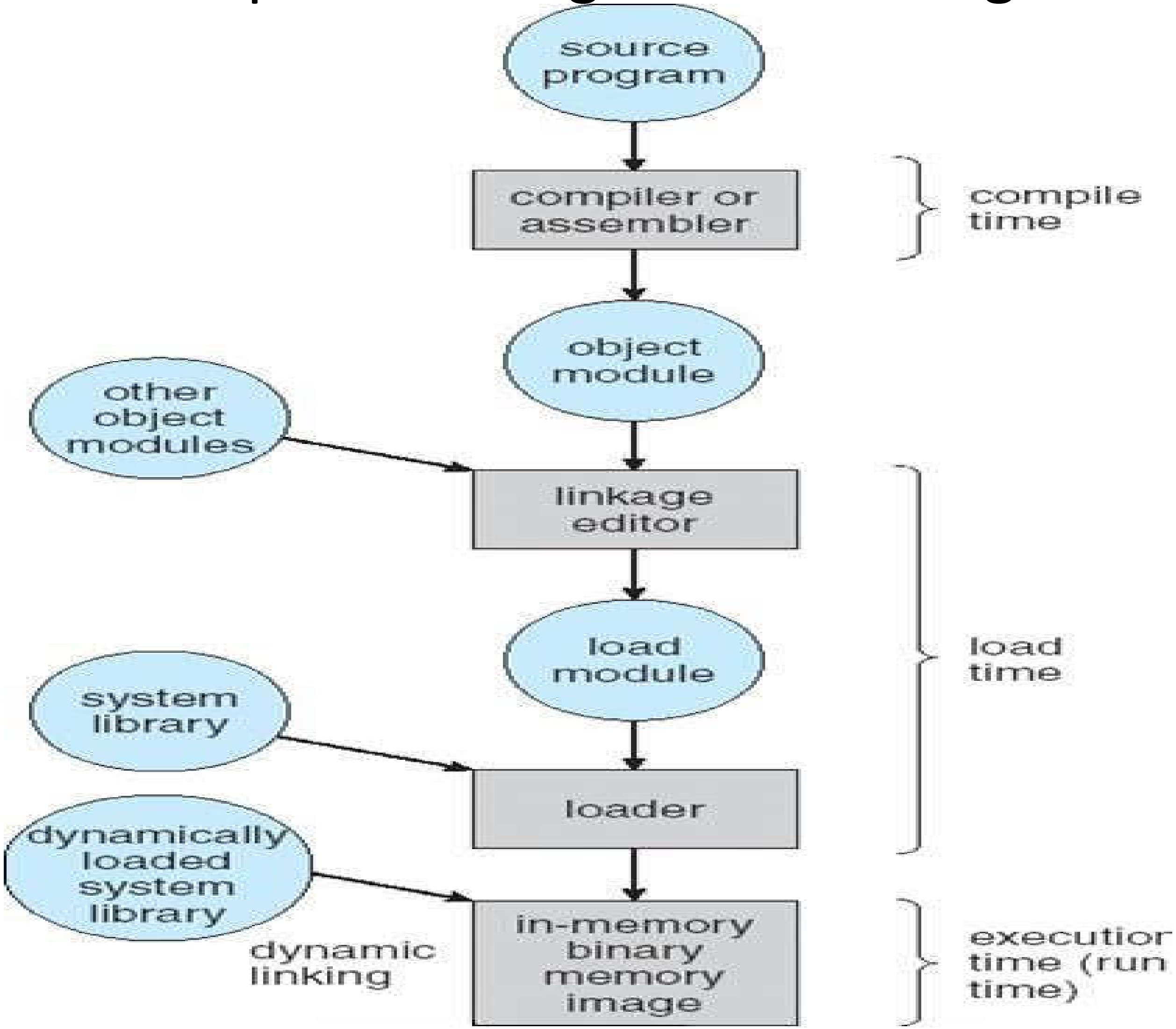
Base and Limit Registers



Binding of Instructions and Data to Memory

- Address binding of instructions and data to memory addresses can happen at three different stages
  - **Compile time**: If memory location known a priori, **absolute code** can be generated; must recompile code if starting location changes
  - **Load time**: Must generate **relocatable code** if memory location is not known at compile time
  - **Execution time**: Binding delayed until run time if the process can be moved during its execution from one memory segment to another. Need hardware support for address maps (e.g., base and limit registers)

Multistep Processing of a User Program



Logical vs. Physical Address Space

- The concept of a logical address space that is bound to a separate physical address space is central to proper memory management
  - **Logical address** – generated by the CPU; also referred to as **virtual address**
  - **Physical address** – address seen by the memory unit
- Logical and physical addresses are the same in compile-time and load-time address-binding schemes; logical (virtual) and physical addresses differ in execution-time address-binding scheme

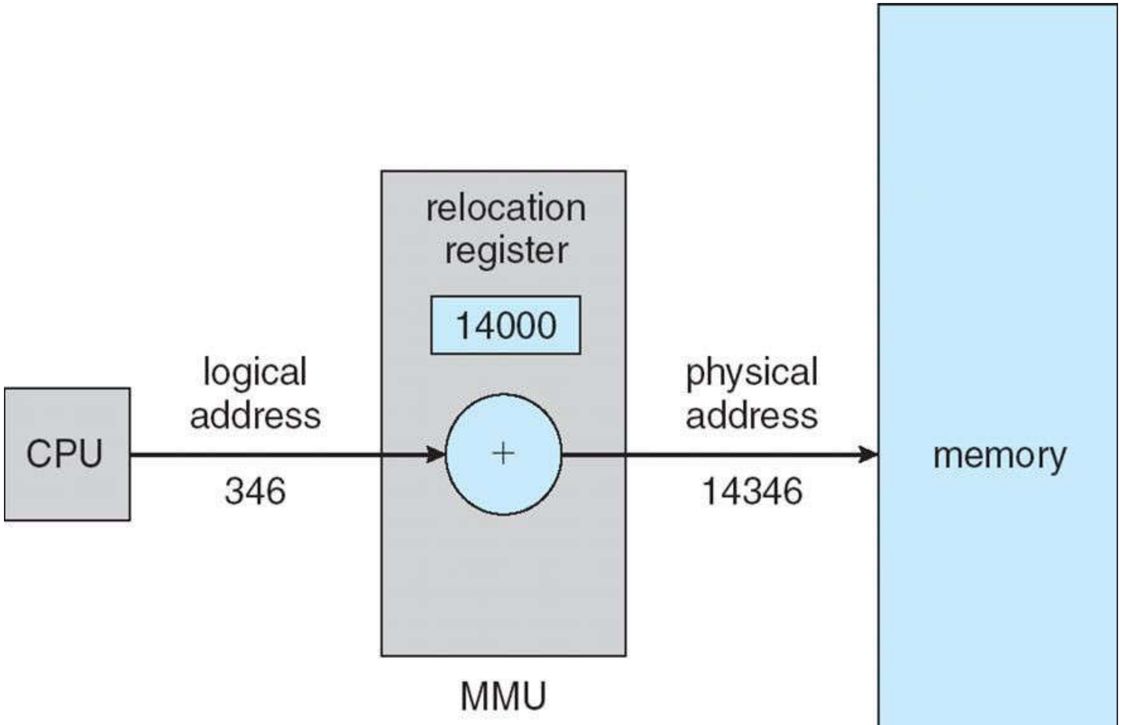
Memory-Management Unit (MMU)

- Hardware device that maps virtual to physical address

- In MMU scheme, the value in the relocation register is added to every address generated by a user process at the time it is sent to memory

- The user program deals with *logical* addresses; it never sees the *real* physical addresses

Dynamic relocation using a relocation register



Dynamic Loading

- Routine is not loaded until it is called
- Better memory-space utilization; unused routine is never loaded
- Useful when large amounts of code are needed to handle infrequently occurring cases
- No special support from the operating system is required implemented through program design

Dynamic Linking

- Linking postponed until execution time
- Small piece of code, *stub*, used to locate the appropriate memory-resident library routine
- Stub replaces itself with the address of the routine, and executes the routine
- Operating system needed to check if routine is in processes' memory address
- Dynamic linking is particularly useful for libraries
- System also known as shared libraries

Swapping

- A process can be swapped temporarily out of memory to a backing store, and then brought back into memory for continued execution

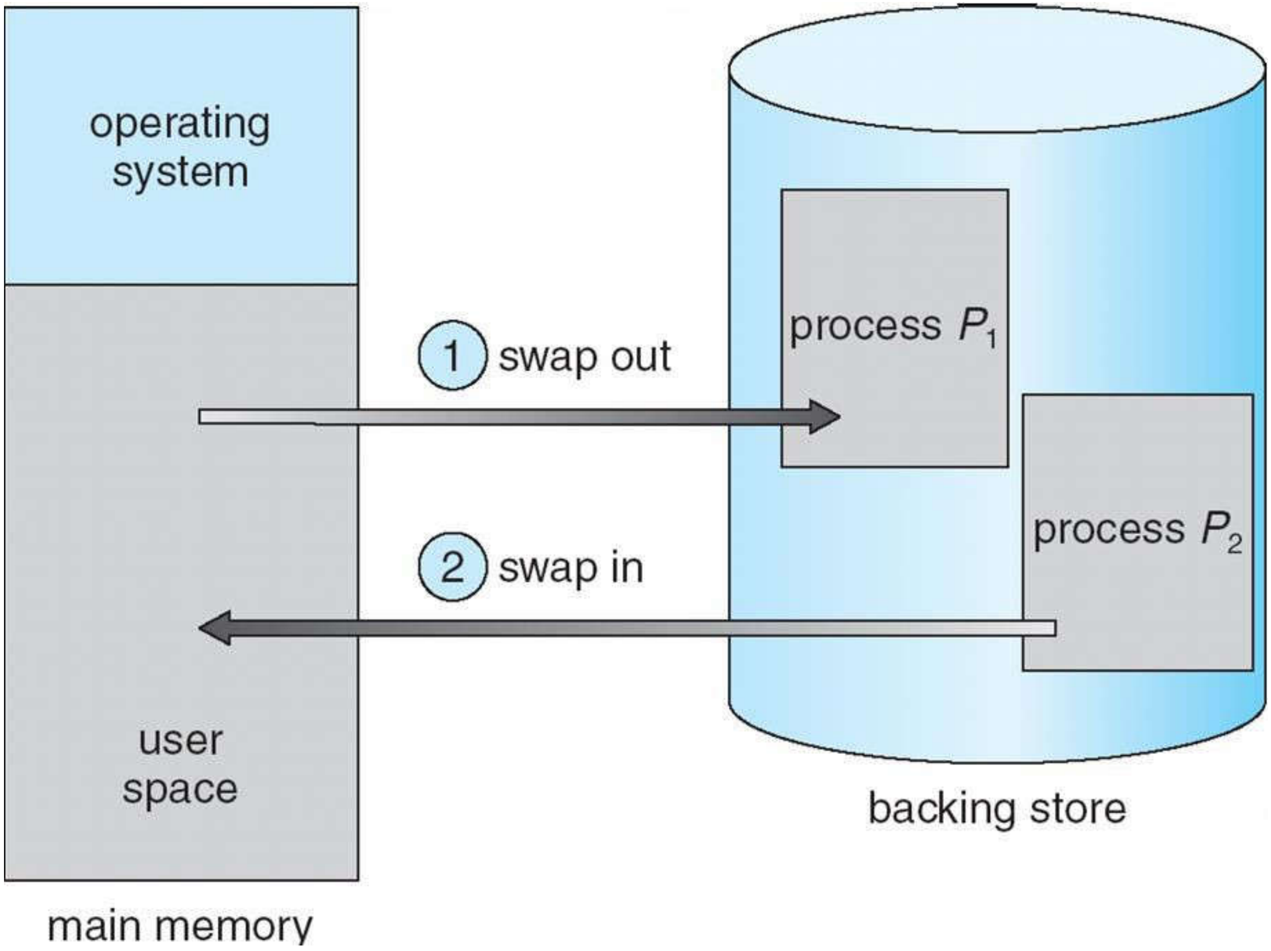
- **Backing store** – fast disk large enough to accommodate copies of all memory images for all users; must provide direct access to these memory images

- **Roll out, roll in** – swapping variant used for priority-based scheduling algorithms; lower-priority process is swapped out so higher-priority process can be loaded and executed

- Major part of swap time is transfer time; total transfer time is directly proportional to the amount of memory swapped

- Modified versions of swapping are found on many systems (i.e., UNIX, Linux, and Windows)
- System maintains a **ready queue** of ready-to-run processes which have memory images on disk

Schematic View of Swapping



Swapping

- Pending I/O →swapping → problem

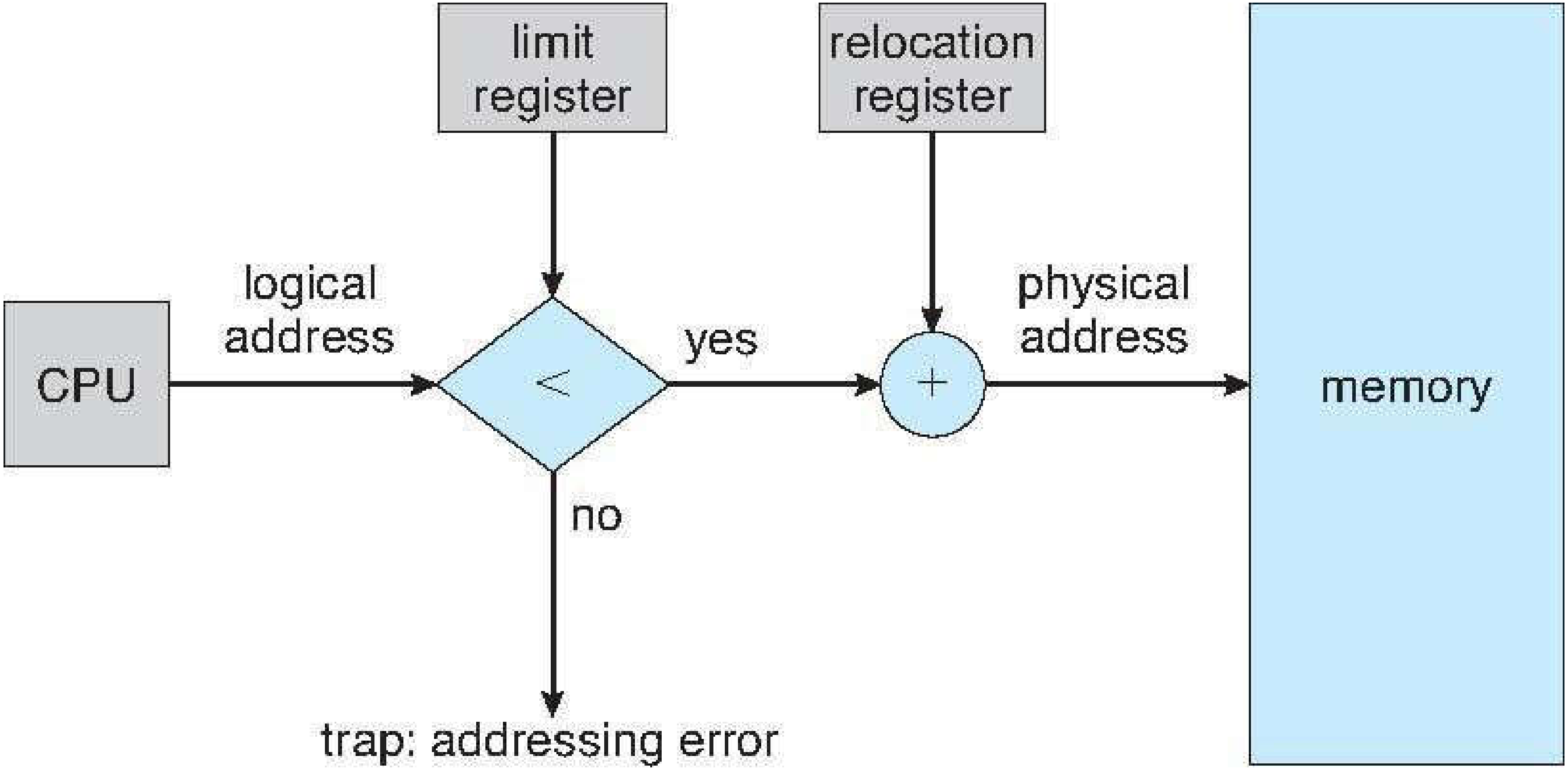
1. Never swap a process with pending I/O
2. Execute I/O operations only into operating system buffers

Contiguous Allocation

- Main memory - partitions:
  - **Resident operating system**, usually held in low memory with interrupt vector
  - **User processes** then held in high memory

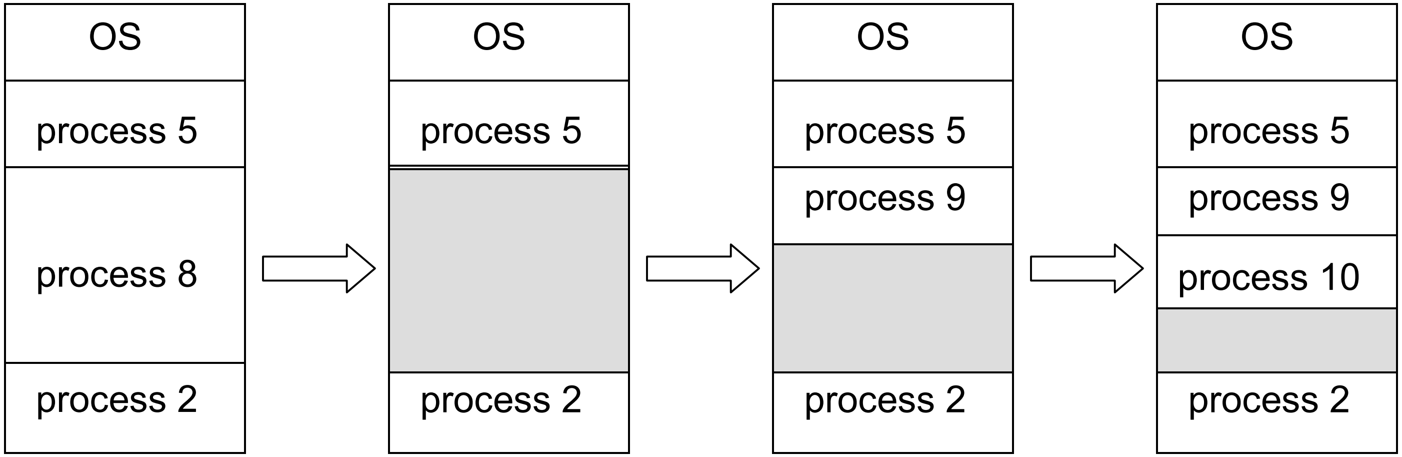
- Relocation registers used to protect user processes from each other, and from changing operating-system code and data
  - Base register contains value of smallest physical address
  - Limit register contains range of logical addresses – each logical address must be less than the limit register
  - MMU maps logical address *dynamically*

Hardware Support for Relocation and Limit Registers



Contiguous Allocation (Cont)

- Multiple-partition allocation
  - Hole – block of available memory; holes of various size are scattered throughout memory
  - When a process arrives, it is allocated memory from a hole large enough to accommodate it
  - Operating system maintains information about:
    a) allocated partitions   b) free partitions (hole)



Dynamic Storage-Allocation Problem

How to satisfy a request of size *n* from a list of free holes?

- **First-fit**: Allocate the *first* hole that is big enough
- **Best-fit**: Allocate the *smallest* hole that is big enough; must search entire list, unless ordered by size
  - Produces the smallest leftover hole
- **Worst-fit**: Allocate the *largest* hole; must also search entire list
  - Produces the largest leftover hole

First-fit and best-fit better than worst-fit in terms of speed and storage utilization

Fragmentation

- **External Fragmentation** – total memory space exists to satisfy a request, but it is not contiguous
- 50-percent rule
- **Internal Fragmentation** – allocated memory may be slightly larger than requested memory
- Reduce external fragmentation by compaction
  - Shuffle memory contents to place all free memory together in one large block
  - Compaction is possible *only* if relocation is dynamic, and is done at execution time

Fragmentation

- Solutions to external fragmentation:
  1. Paging
  2. segmentation

Paging

- Logical address space of a process can be noncontiguous; process is allocated physical memory whenever the latter is available
- Physical memory → frames
- Logical memory → pages

Address Translation Scheme

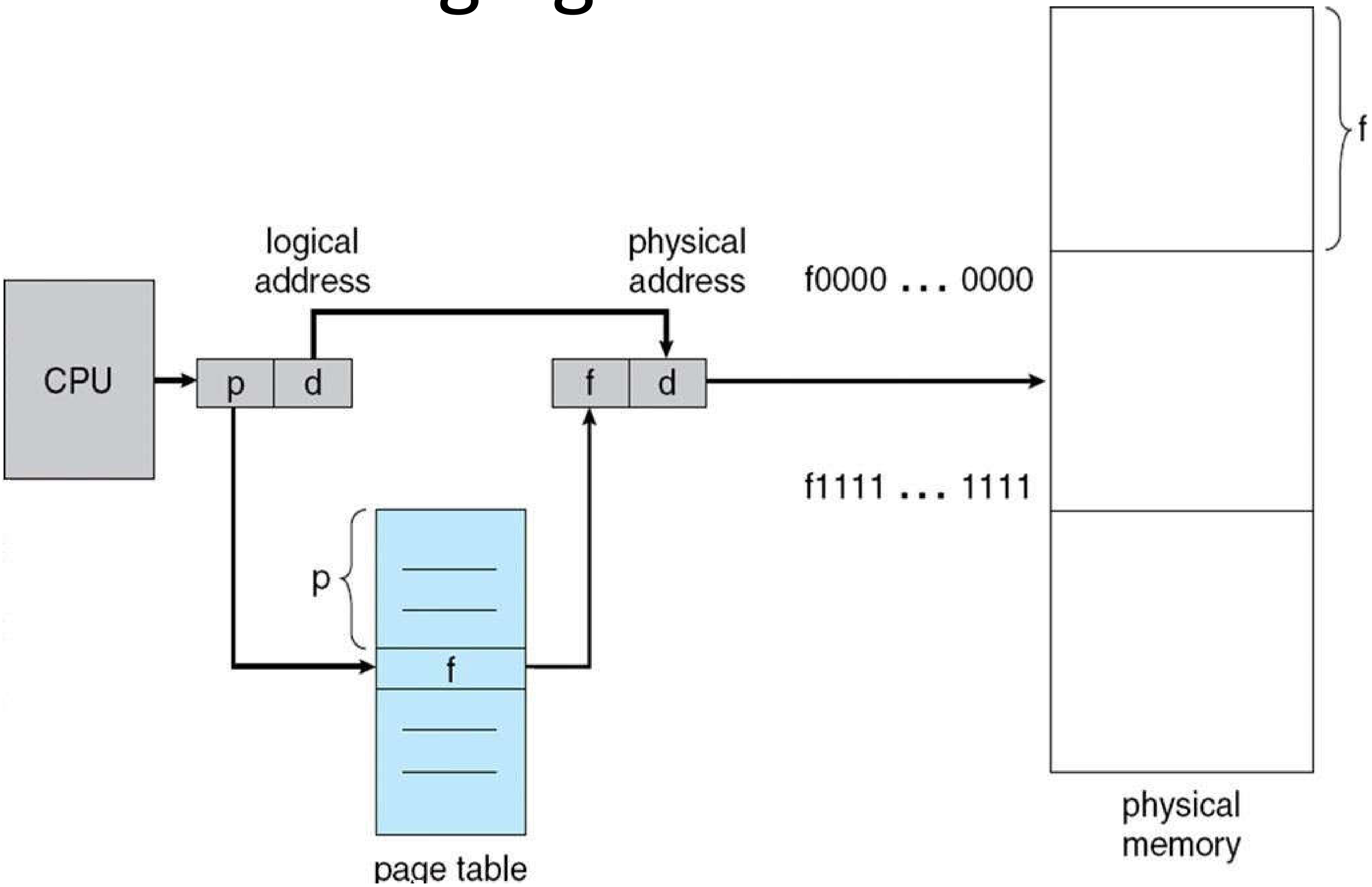
- Address generated by CPU is divided into:
  - **Page number (*p*)** – used as an index into a *page table*
  - **Page offset (d)** – combined with base address to define the physical memory address that is sent to the memory unit

Logical address:

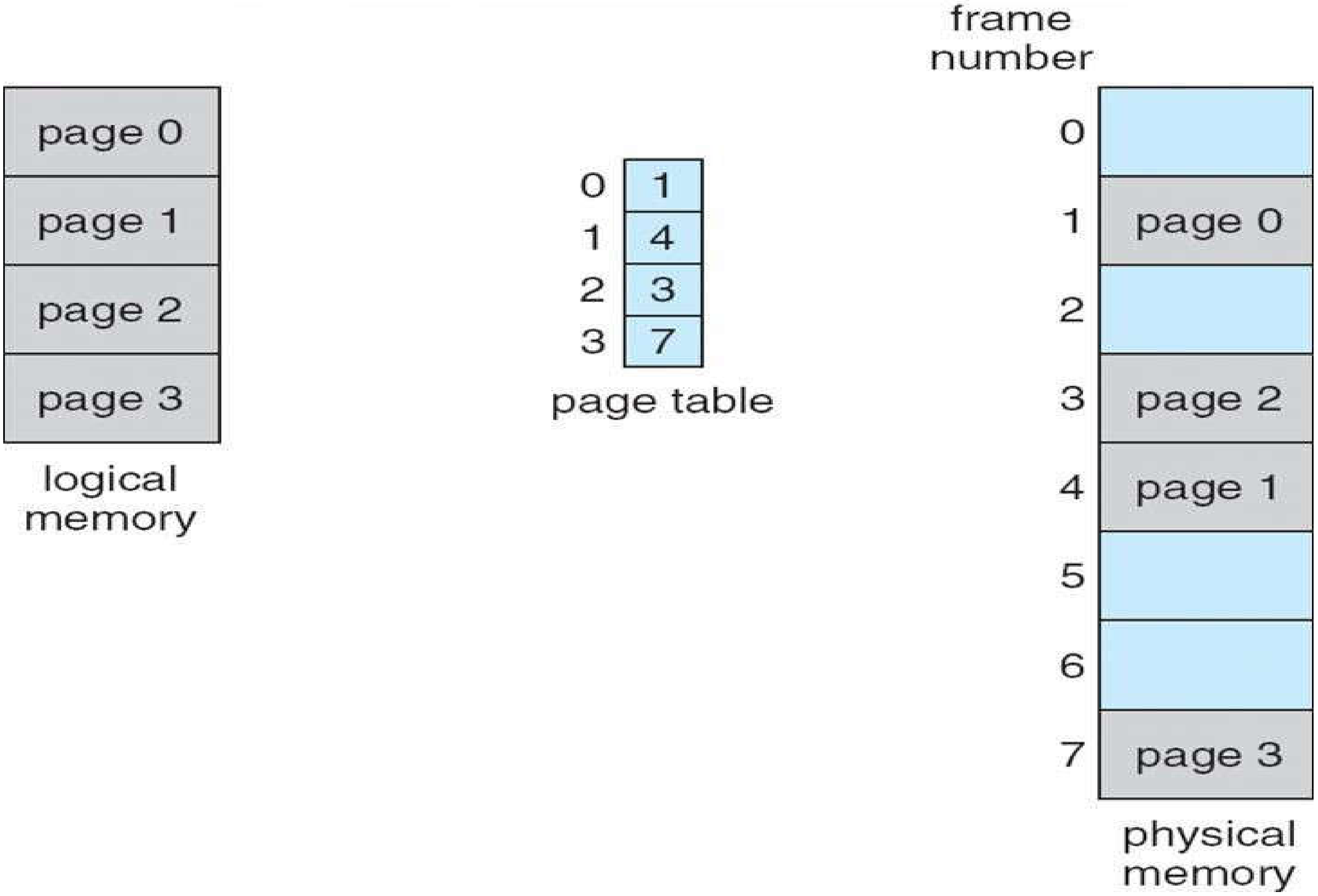
page number	page offset
p	**d**
m - n	***n***

- – logical address space → 2^m *and page size* → 2^n

Paging Hardware



Paging Model of Logical and Physical Memory



page 0

page 1

page 2

page 3

logical memory

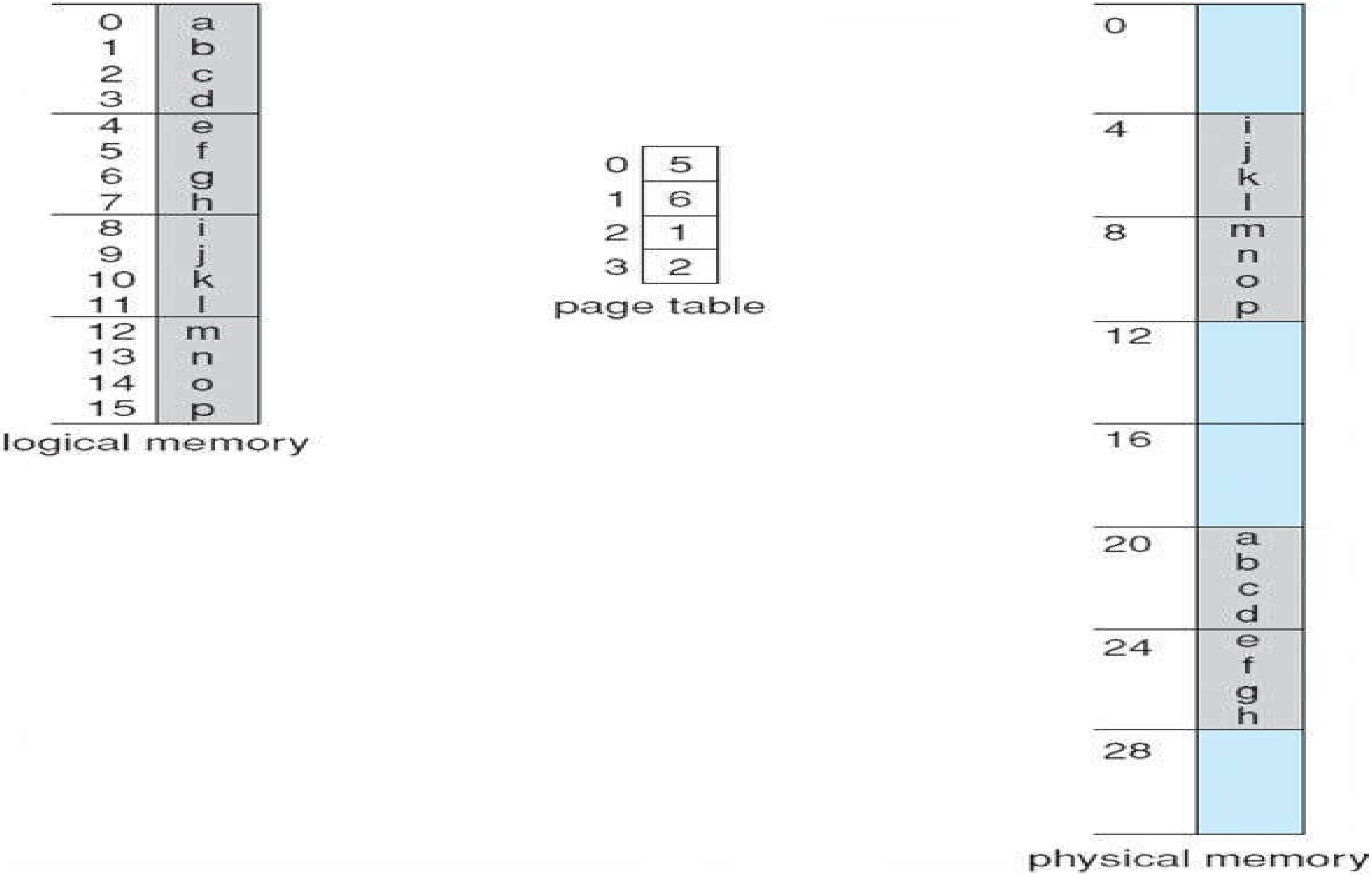
0	1
1	4
2	3
3	7

page table

frame number	
0	
1	page 0
2	
3	page 2
4	page 1
5	
6	
7	page 3

physical memory

Paging Example 32-byte memory and 4-byte pages



0	a
1	b
2	c
3	d
4	e
5	f
6	g
7	h
8	i
9	j
10	k
11	l
12	m
13	n
14	o
15	p

logical memory

0	5
1	6
2	1
3	2

page table

0	
4	i j k l
8	m n o p
12	
16	
20	a b c d
24	e f g h
28	

physical memory

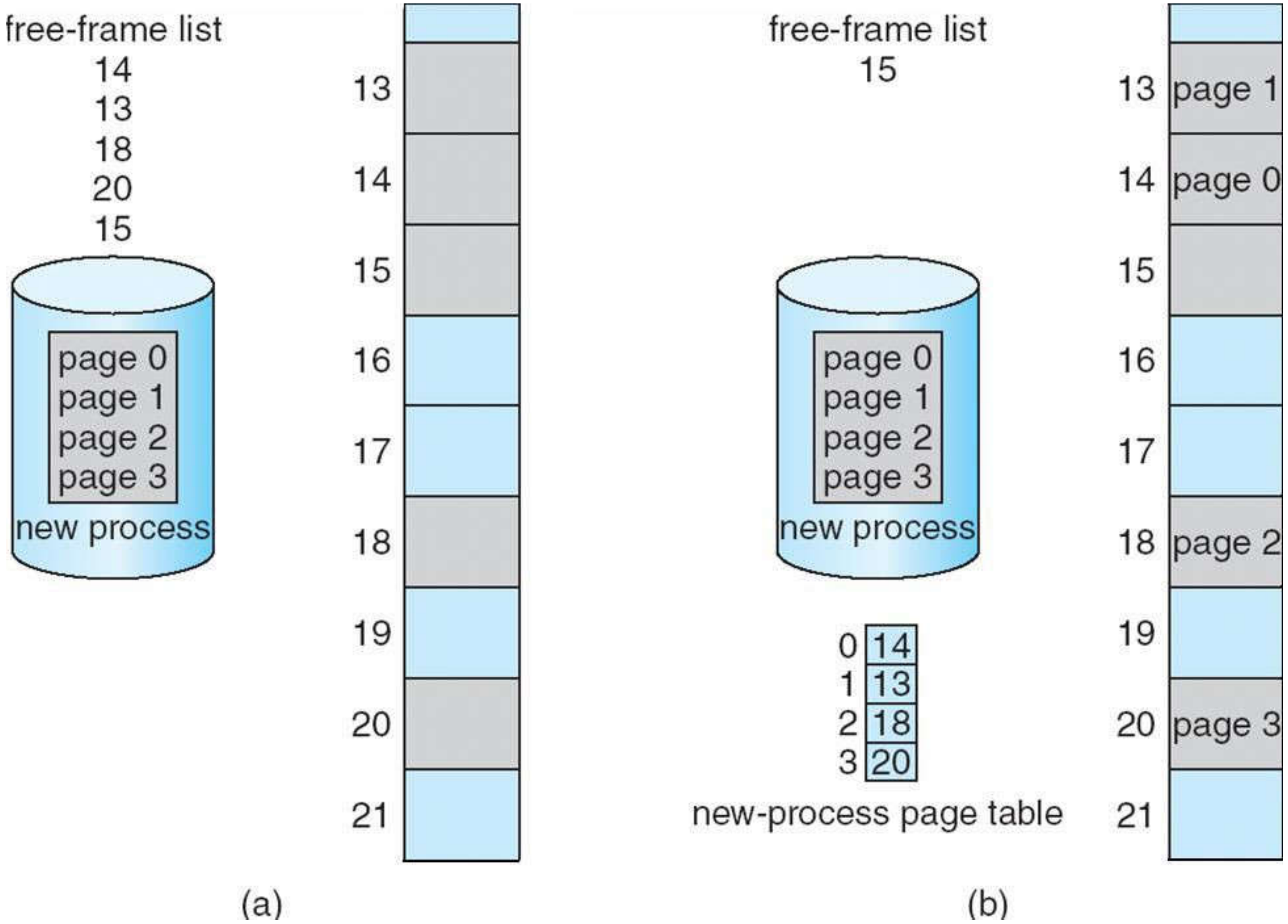
paging

1. No external fragmentation
2. May have internal fragmentation

Frame table: data structure with information on

- Which frames are allocated
- Which frames are available
- How many total frames are there etc.,

Free Frames



Before allocation

After allocation

Hardware support

- Page table is kept in main memory

1. Page-table base register (PTBR) points to the page table
   - In this scheme every data/instruction access requires two memory accesses. One for the page table and one for the data/instruction.

- Solution: special fast-lookup h/w cache called associative memory or translation look-aside buffers (TLBs)
- Some TLBs store address-space identifiers (ASIDs) in each TLB entry – uniquely identifies each process to provide address-space protection for that process.

Associative Memory

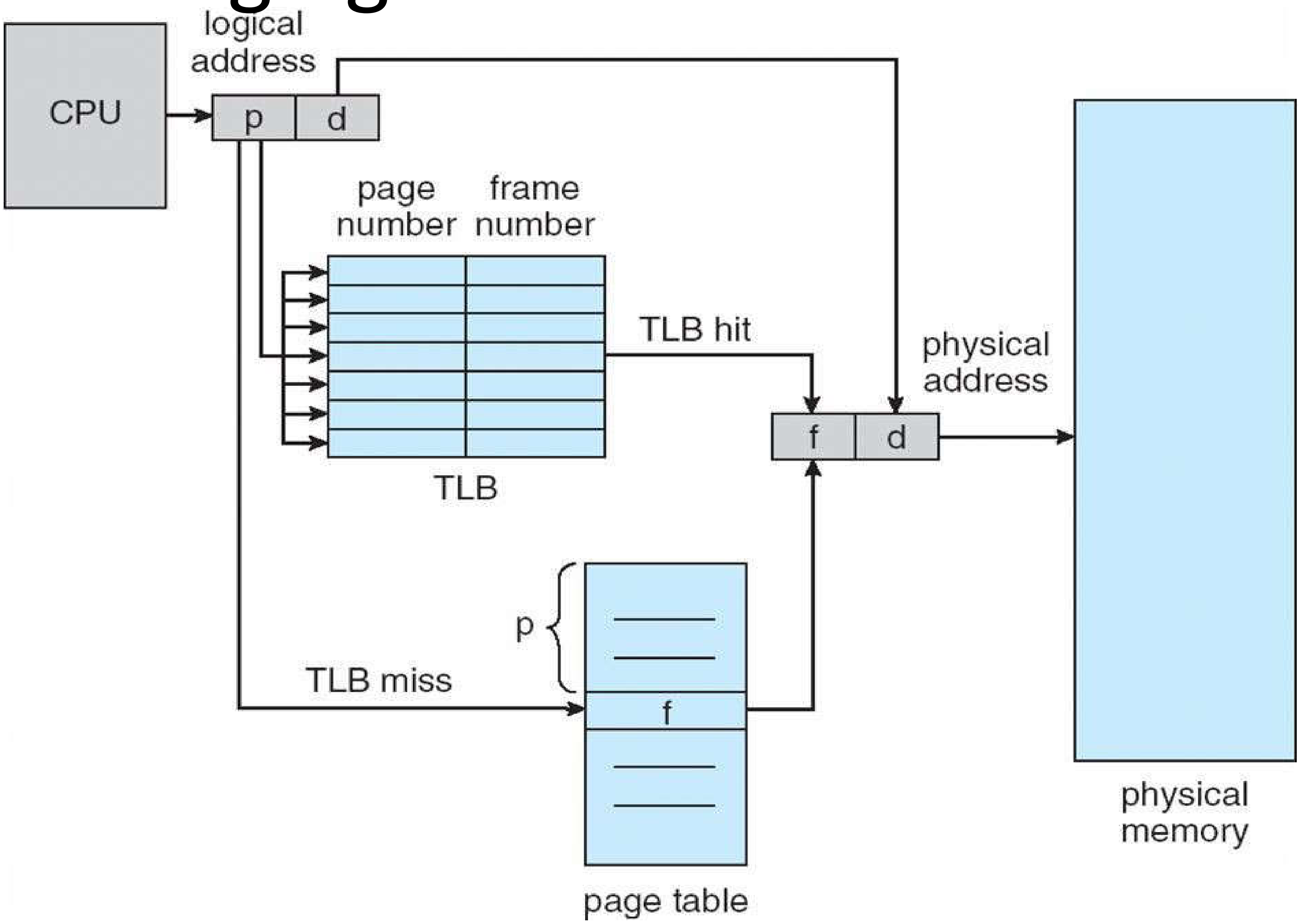
- Associative memory – parallel search

Page #	Frame #

Address translation (p, d)

- If p is in associative register, get frame # out
- Otherwise get frame # from page table in memory

Paging Hardware With TLB



Effective Access Time

- Associative Lookup = ε time unit
- Assume memory cycle time is 1 microsecond
- Hit ratio – percentage of times that a page number is found in the associative registers; ratio related to number of associative registers
- Hit ratio = α
- Effective Access Time (EAT)

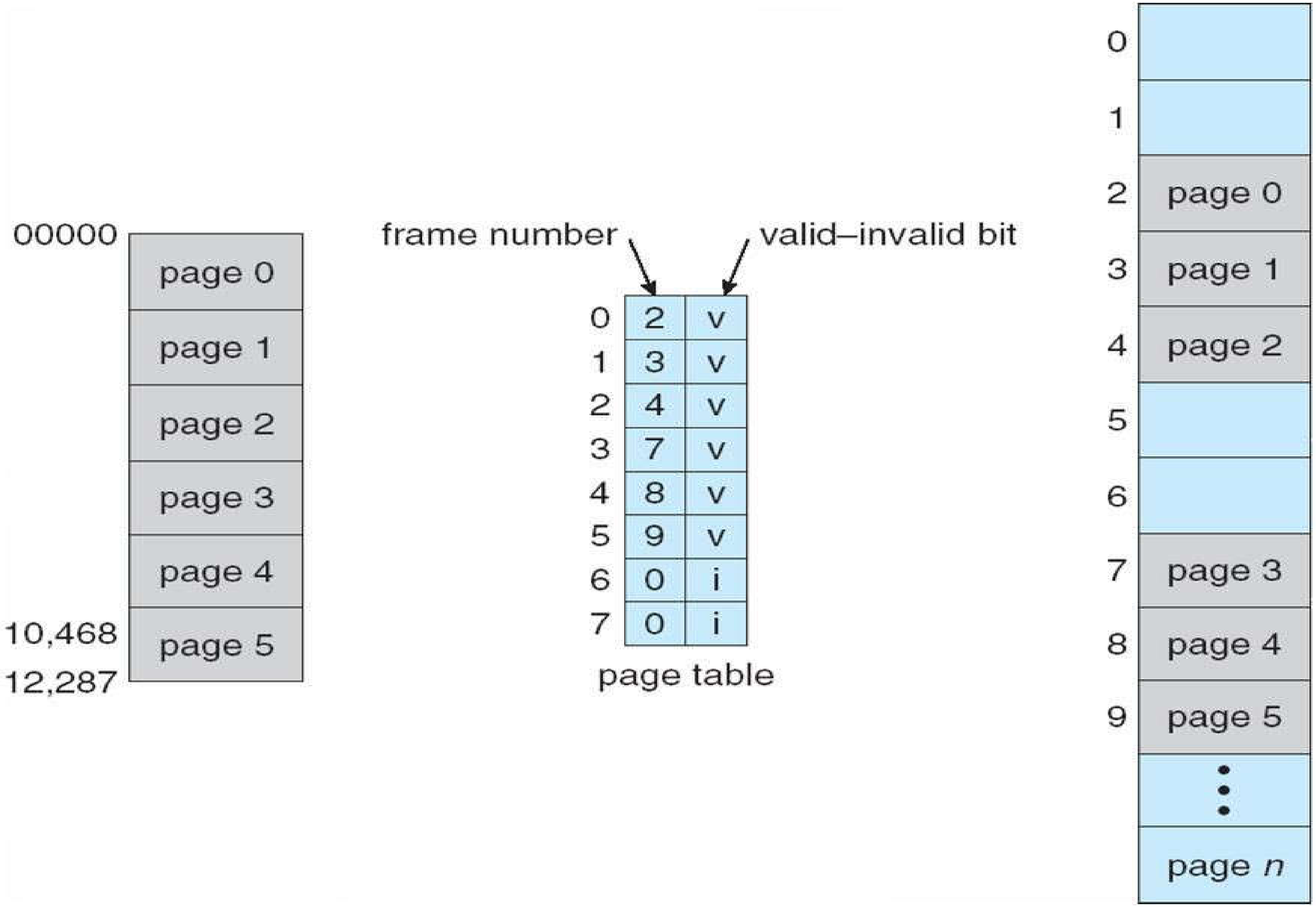
$$\text{EAT} = (1 + \varepsilon)\,\alpha + (2 + \varepsilon)(1 - \alpha)$$
$$= 2 + \varepsilon - \alpha$$

Protection

- Memory protection implemented by associating protection bit with each frame

- **Valid-invalid** bit attached to each entry in the page table:
  - "valid" indicates that the associated page is in the process' logical address space, and is thus a legal page
  - "invalid" indicates that the page is not in the process' logical address space

Valid (v) or Invalid (i) Bit In A Page Table



	frame number	valid–invalid bit
0	2	v
1	3	v
2	4	v
3	7	v
4	8	v
5	9	v
6	0	i
7	0	i

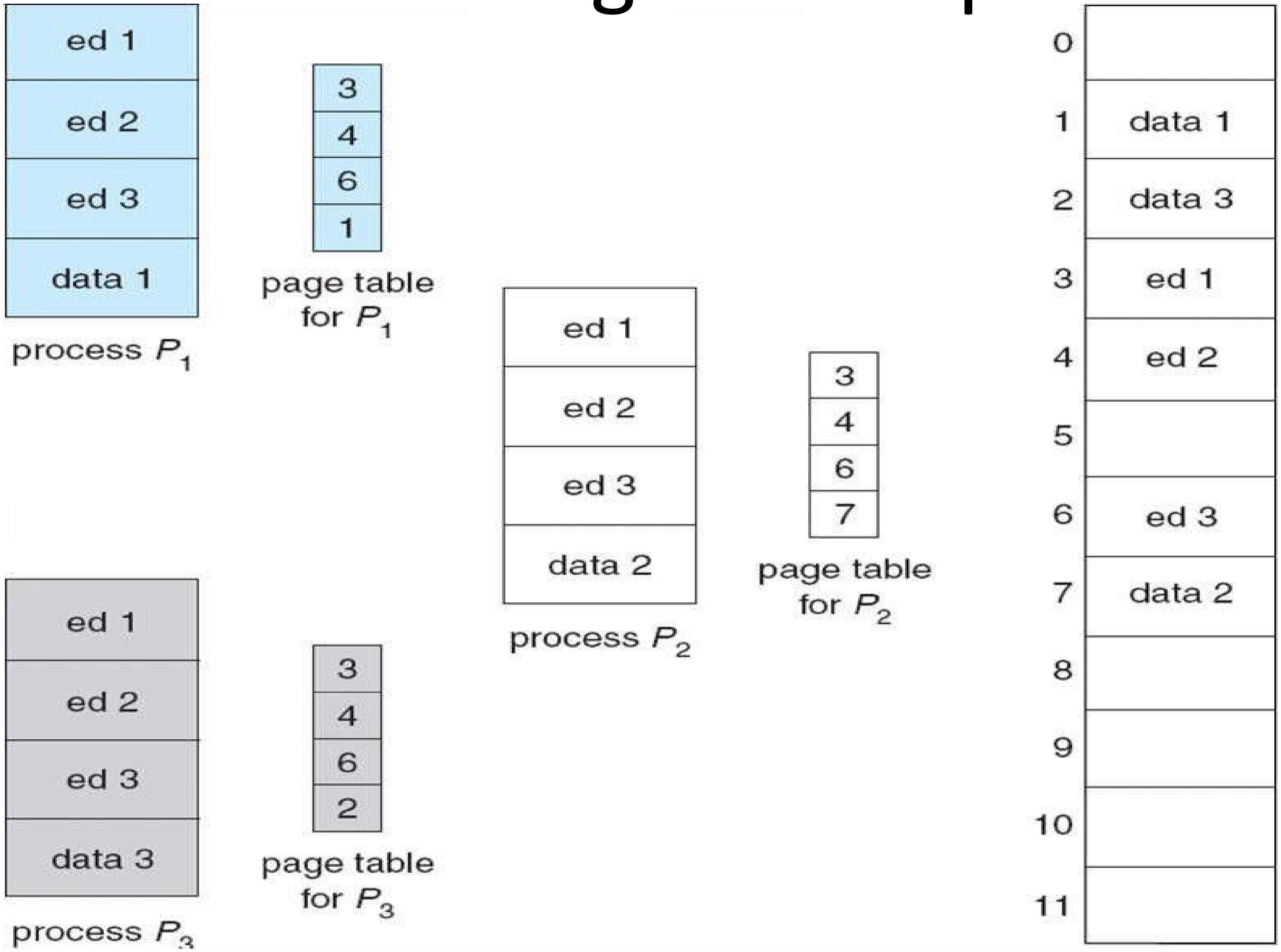
page table

Shared Pages

- **Shared code**
  - One copy of read-only (reentrant) code shared among processes (i.e., text editors, compilers, window systems).
  - Shared code must appear in same location in the logical address space of all processes

- **Private code and data**
  - Each process keeps a separate copy of the code and data
  - The pages for the private code and data can appear anywhere in the logical address space

Shared Pages Example



Structure of the Page Table

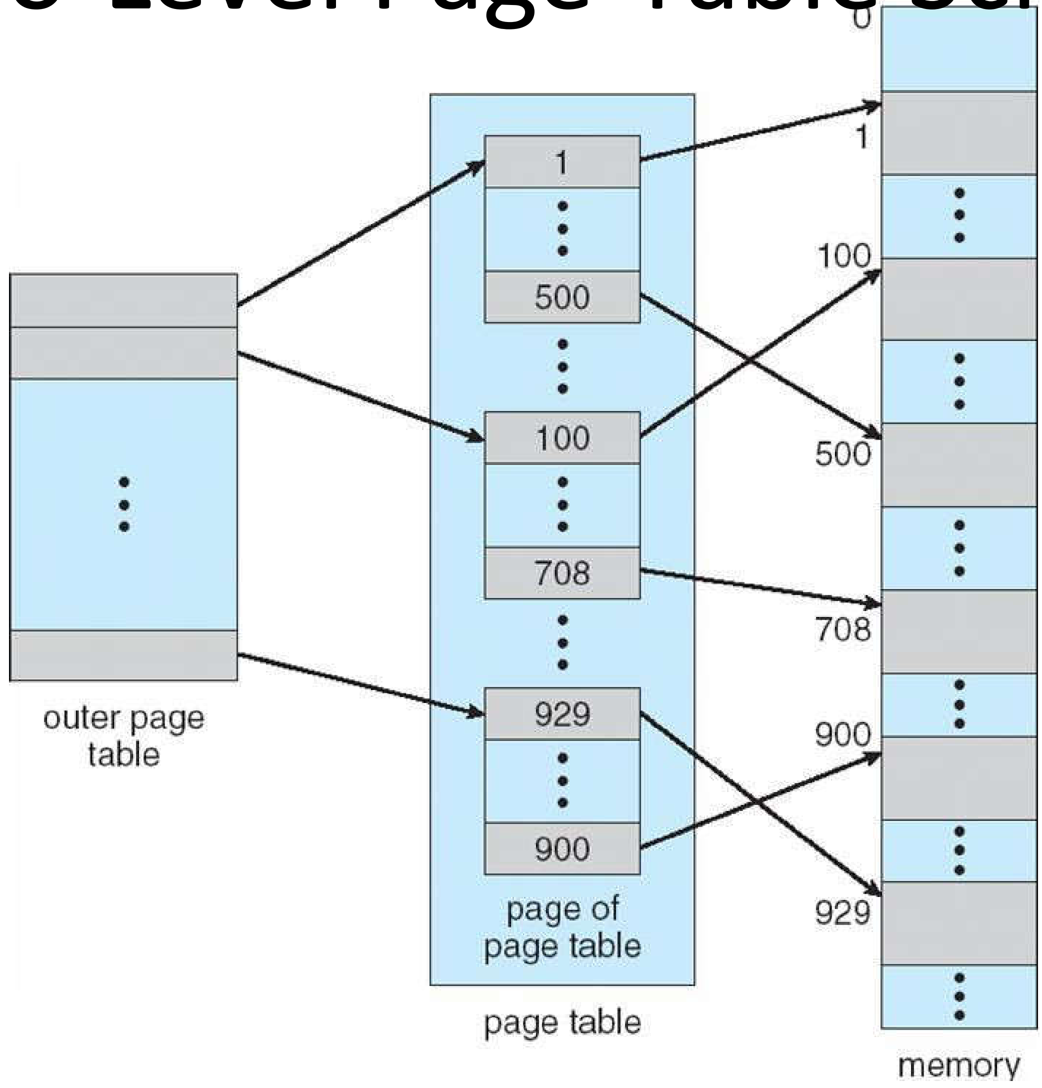
- Hierarchical Paging
- Hashed Page Tables
- Inverted Page Tables

Hierarchical Page Tables

- Break up the logical address space into multiple page tables

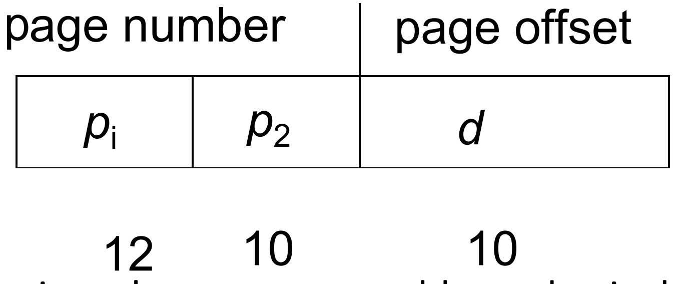
- A simple technique is a two-level page table

Two-Level Page-Table Scheme



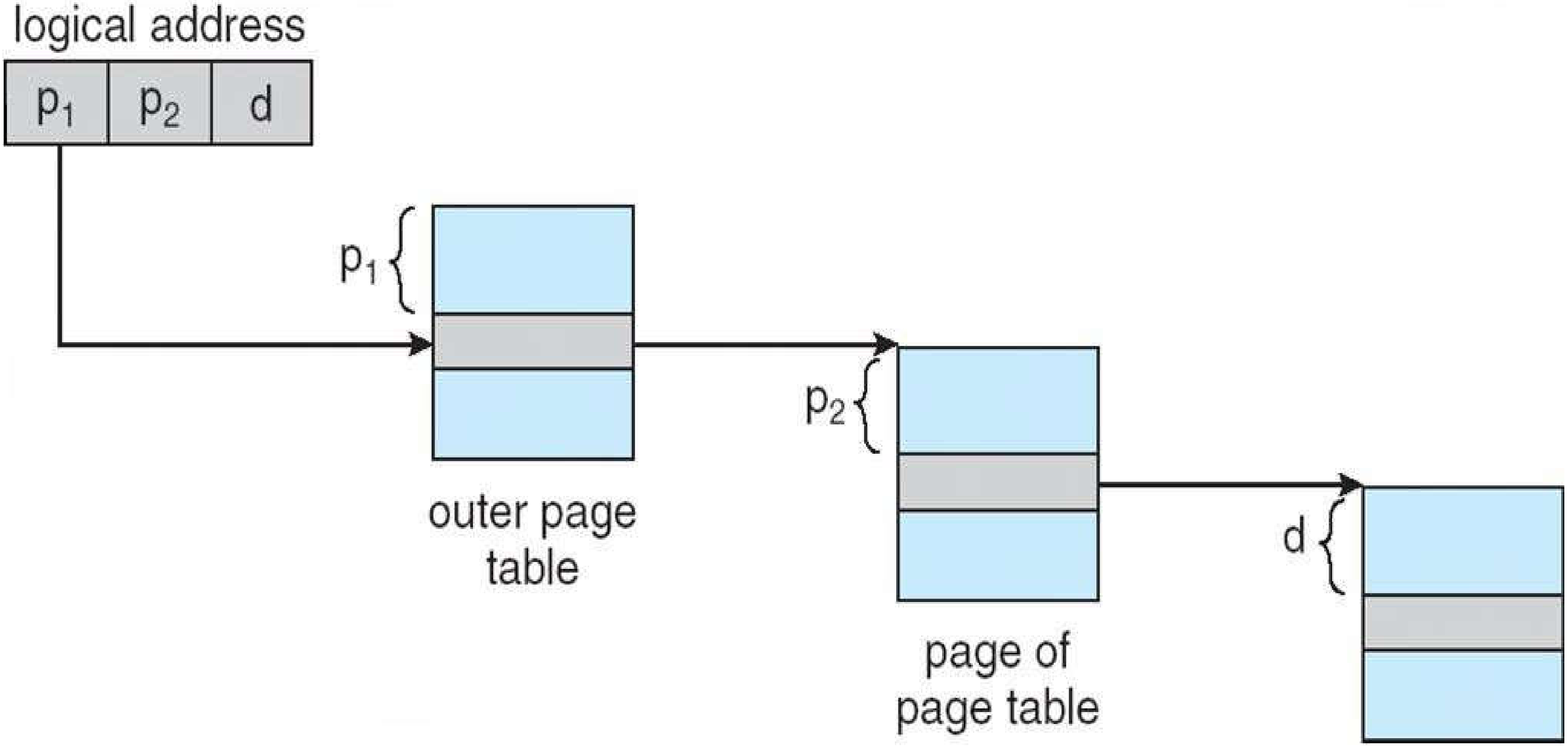
Two-Level Paging Example

- A logical address (on 32-bit machine with 1K page size) is divided into:
  - a page number consisting of 22 bits
  - a page offset consisting of 10 bits
- Since the page table is paged, the page number is further divided into:
  - a 12-bit page number
  - a 10-bit page offset
- Thus, a logical address is as follows:

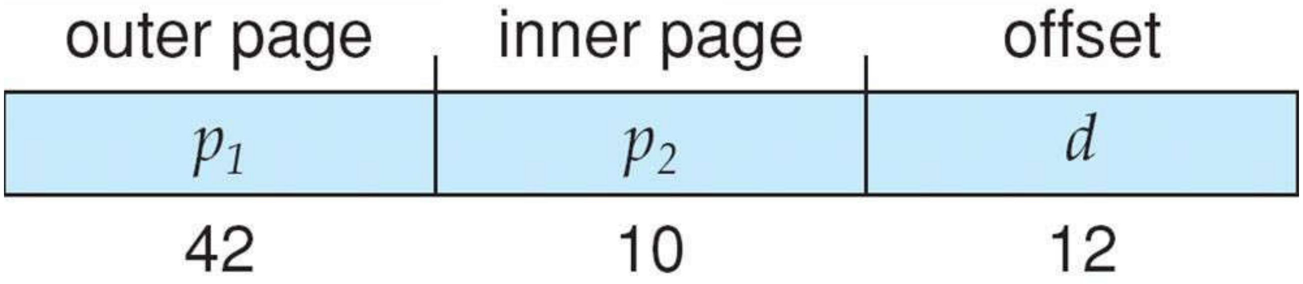


where p_i is an index into the outer page table, and p_2 is the displacement within the page of the outer page table

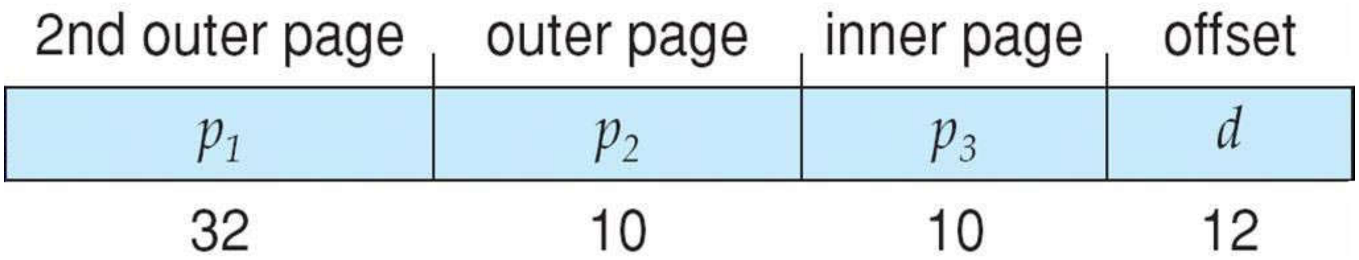
Address-Translation Scheme



Three-level Paging Scheme



outer page	inner page	offset
p_1	p_2	d
42	10	12



2nd outer page	outer page	inner page	offset
p_1	p_2	p_3	d
32	10	10	12

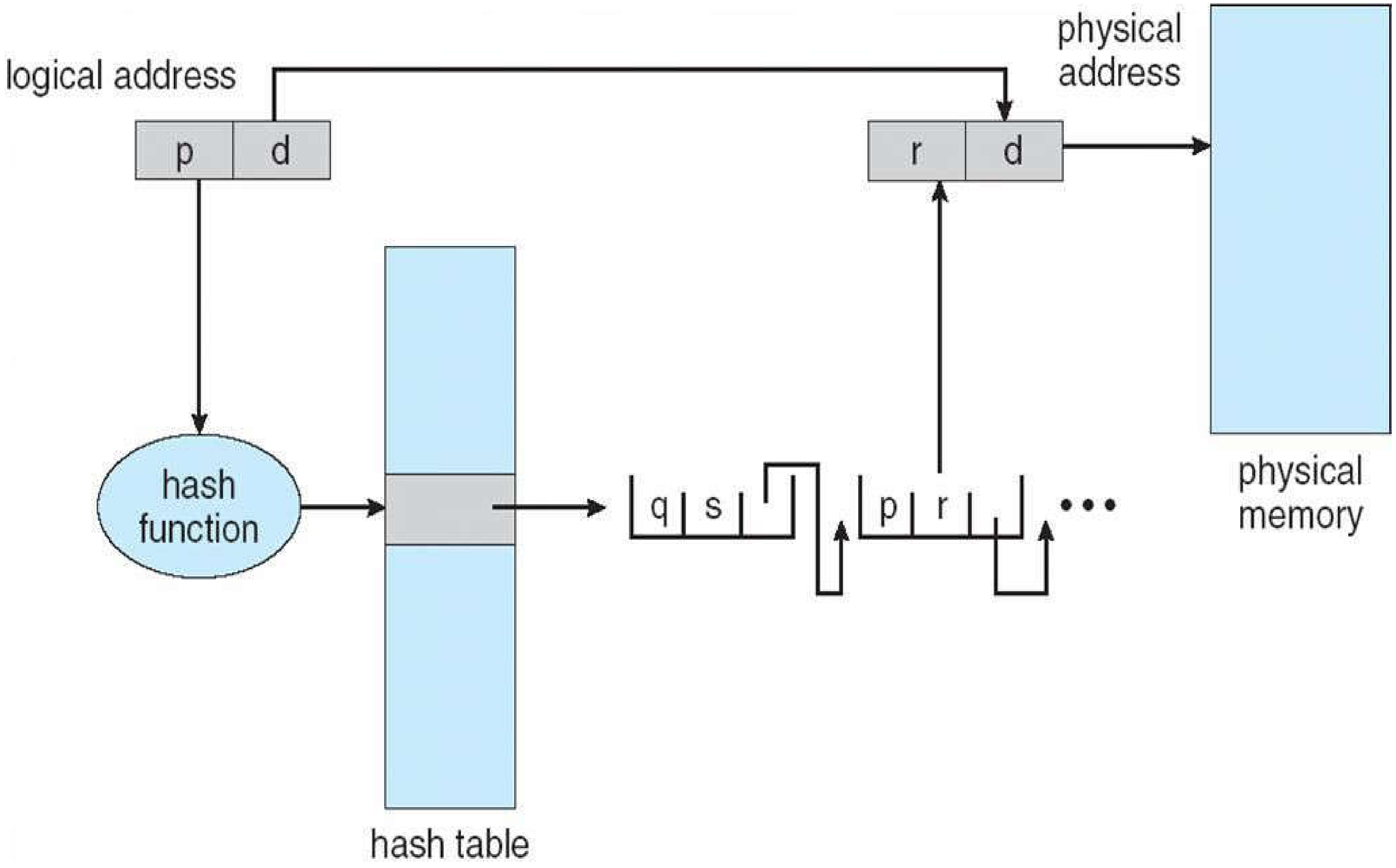
Hashed Page Tables

- Common in address spaces > 32 bits

- The virtual page number is hashed into a page table
  - This page table contains a chain of elements hashing to the same location

- Virtual page numbers are compared in this chain searching for a match
  - If a match is found, the corresponding physical frame is extracted

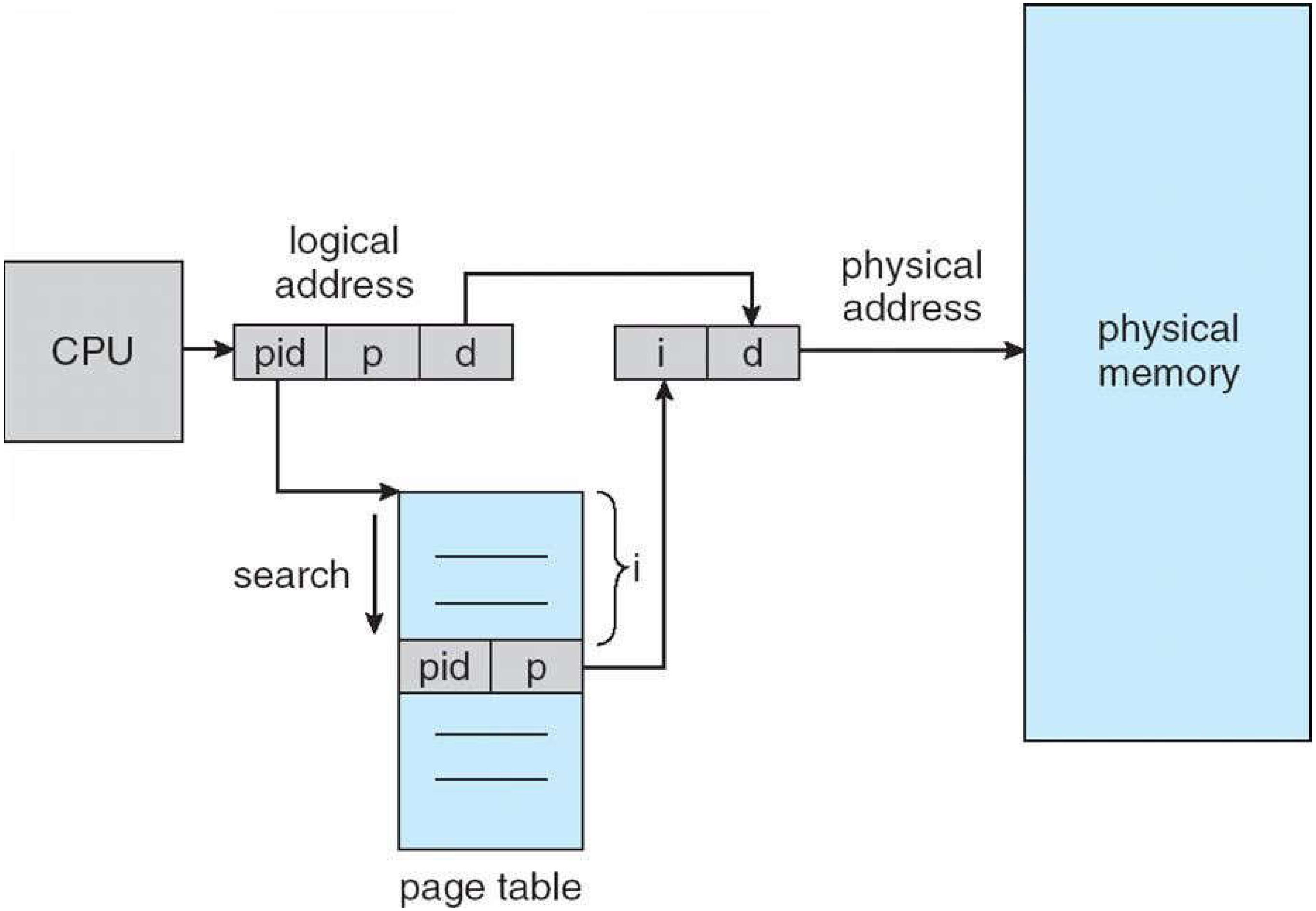
Hashed Page Table



Inverted Page Table

- One entry for each real page of memory
- Entry consists of the virtual address of the page stored in that real memory location, with information about the process that owns that page
- Decreases memory needed to store each page table, but increases time needed to search the table when a page reference occurs
- Use hash table to limit the search to one — or at most a few — page-table entries

Inverted Page Table Architecture

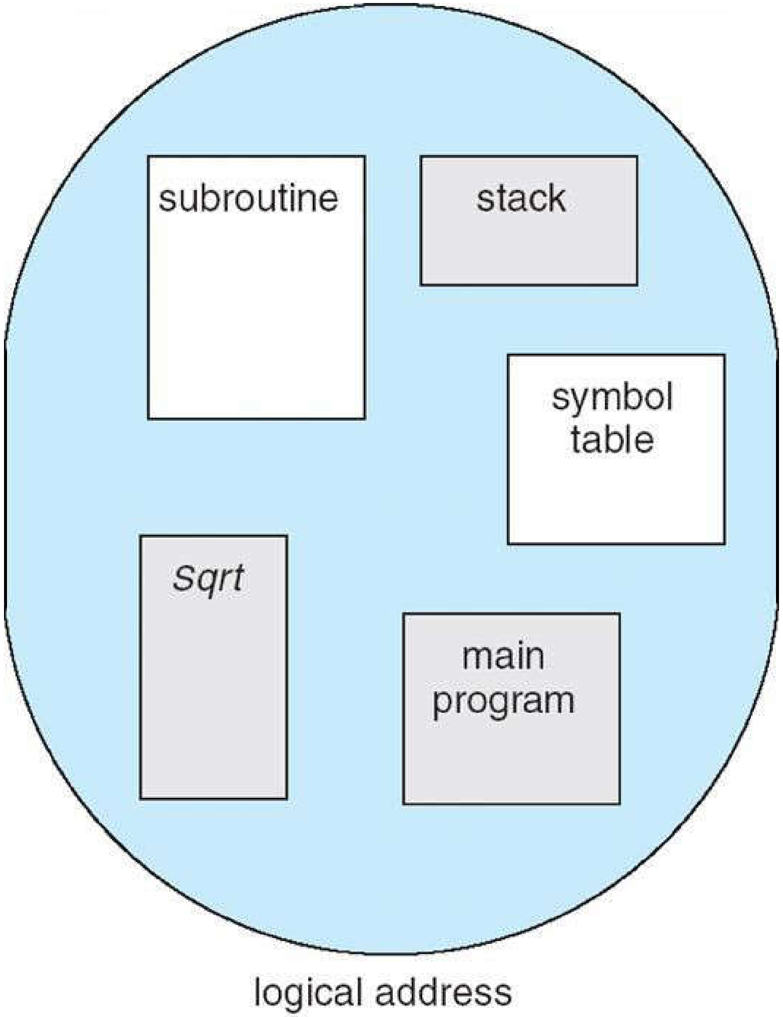


Segmentation

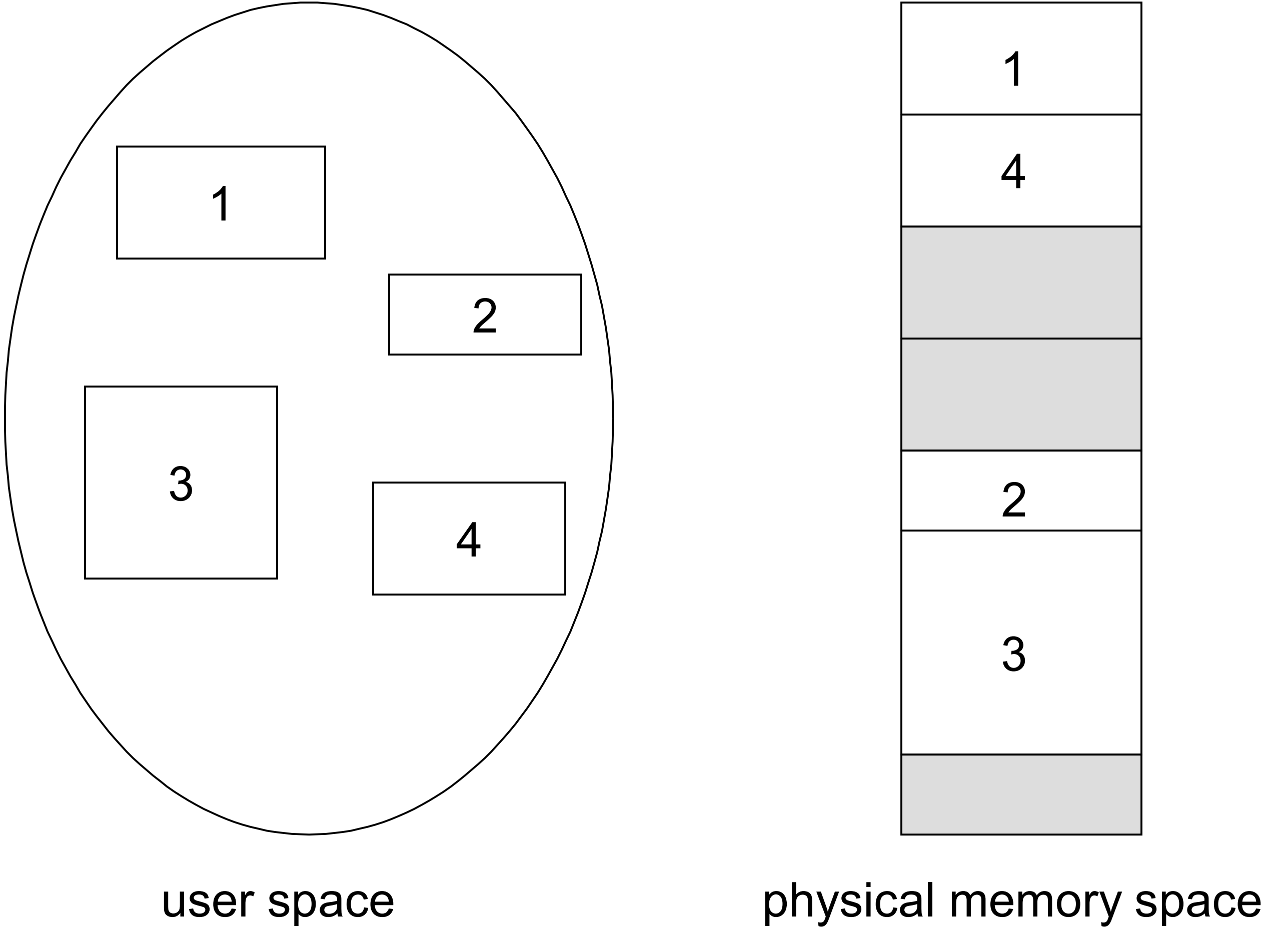
- Memory-management scheme that supports user view of memory
- A program is a collection of segments
  - A segment is a logical unit such as:

    main program
    procedure
    function
    method
    object
    local variables, global variables
    common block
    stack
    symbol table
    arrays

User’s View of a Program



Logical View of Segmentation



Segmentation Architecture

- Logical address consists of a two tuple:

  <segment-number, offset>,

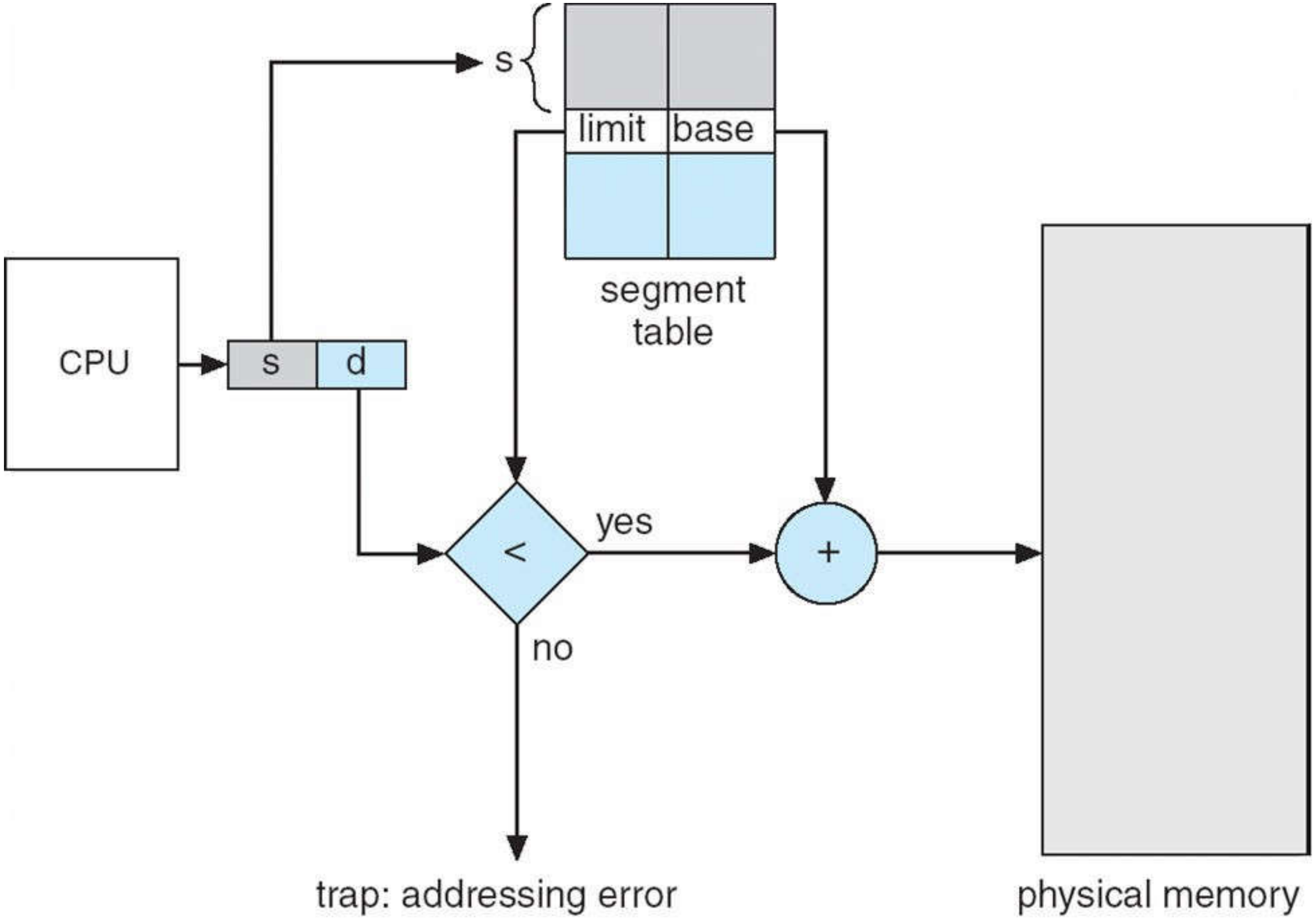
- **Segment table** – maps two-dimensional physical addresses; each table entry has:
  - **base** – contains the starting physical address where the segments reside in memory
  - **limit** – specifies the length of the segment
- **Segment-table base register (STBR)** points to the segment table's location in memory
- **Segment-table length register (STLR)** indicates number of segments used by a program;

  segment number ***s*** is legal if ***s*** < **STLR**

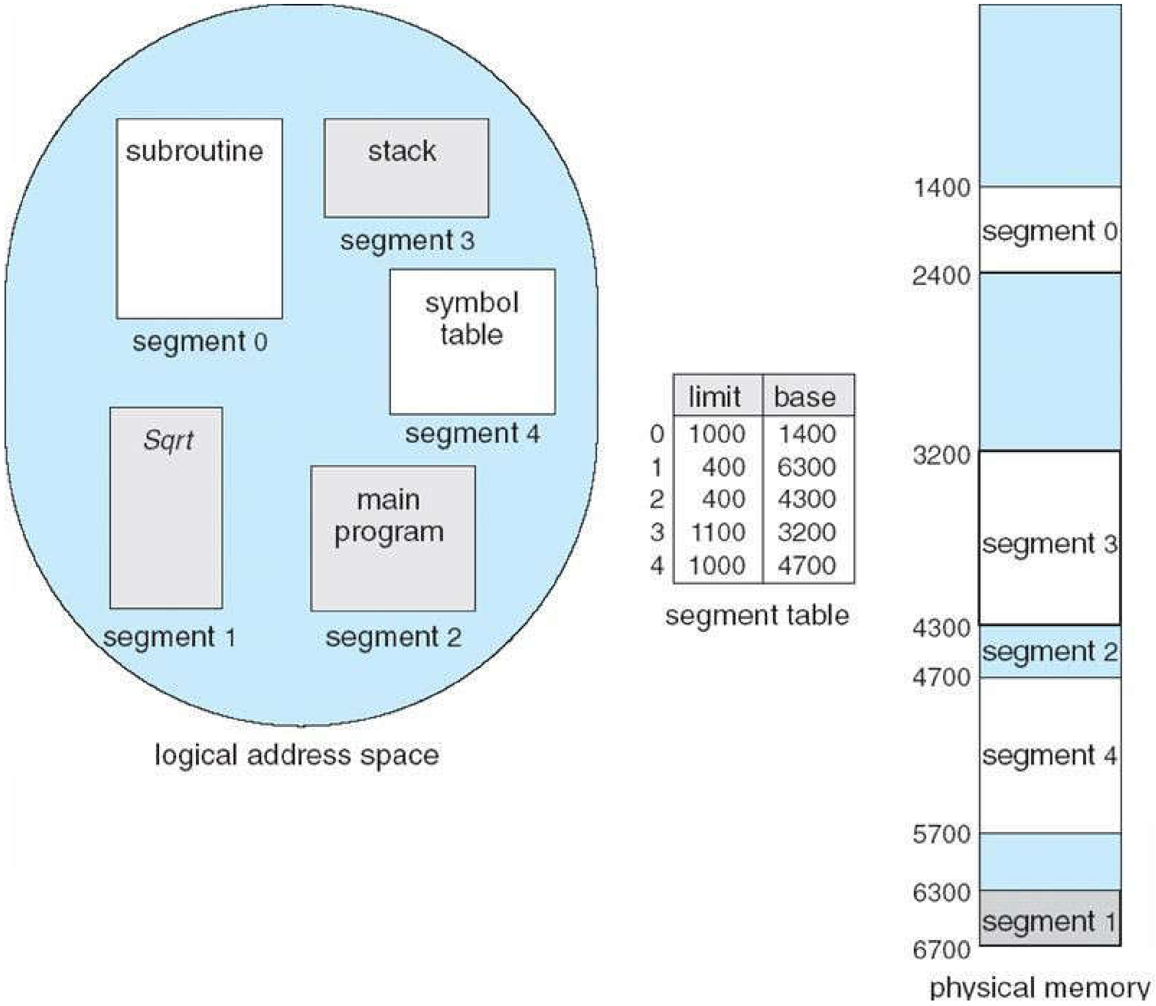
Segmentation Architecture (Cont.)

- Protection
  - With each entry in segment table associate:
    - validation bit = 0 ⇒ illegal segment
    - read/write/execute privileges
- Protection bits associated with segments; code sharing occurs at segment level
- Since segments vary in length, memory allocation is a dynamic storage-allocation problem
- A segmentation example is shown in the following diagram

Segmentation Hardware



Example of Segmentation



	limit	base
0	1000	1400
1	400	6300
2	400	4300
3	1100	3200
4	1000	4700

segment table

Virtual Memory

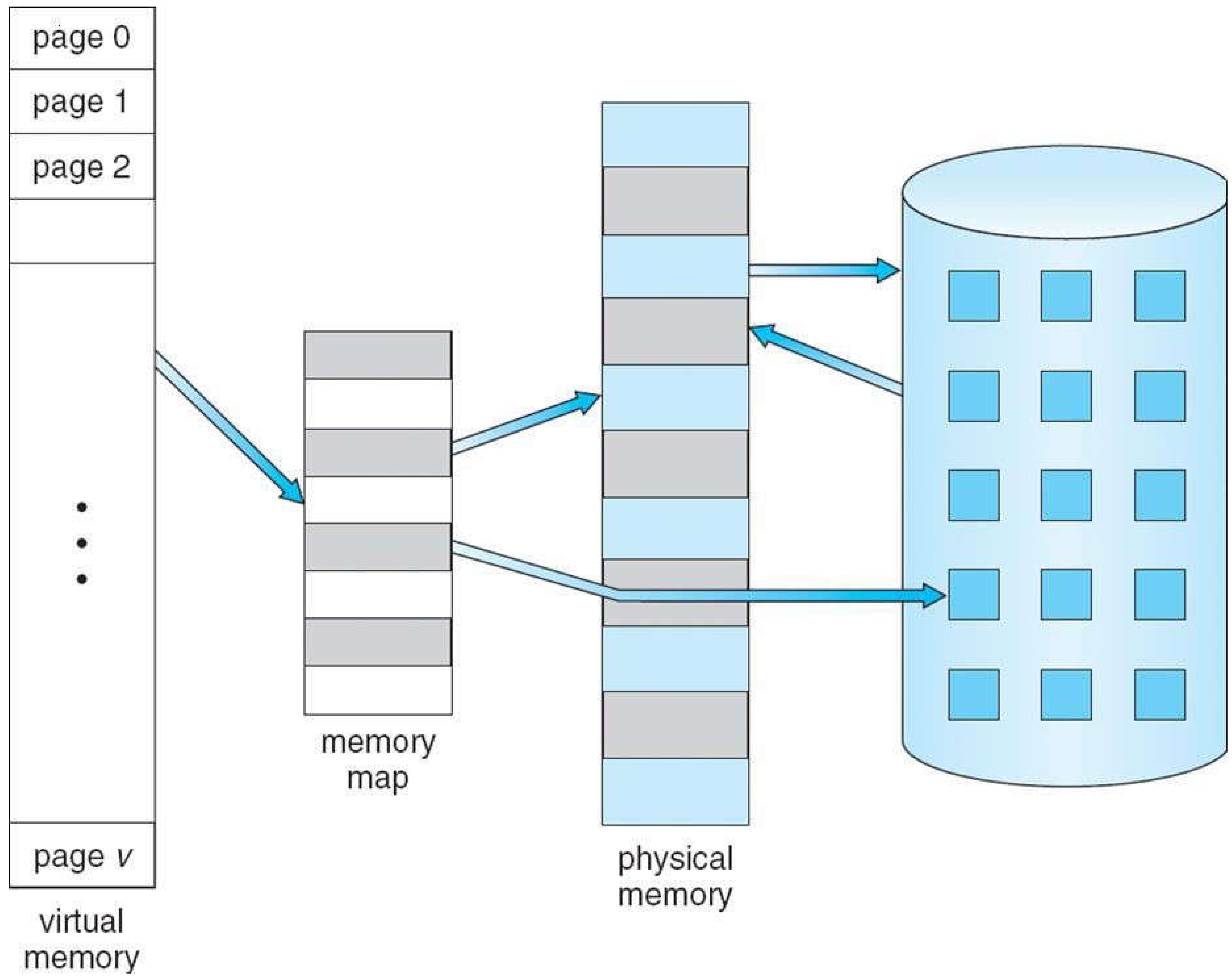
- Background
- Demand Paging
- Page Replacement
- Allocation of Frames
- Thrashing

Background

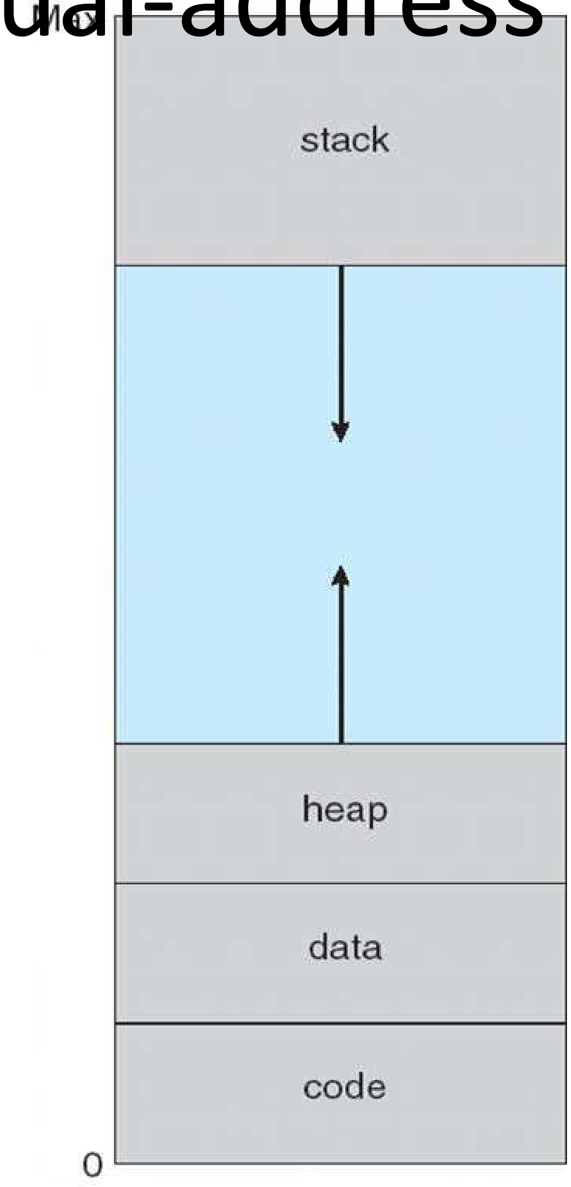
- Virtual memory – separation of user logical memory from physical memory.
  - Only part of the program needs to be in memory for execution
  - Logical address space can therefore be much larger than physical address space
  - Allows address spaces to be shared by several processes
  - Allows for more efficient process creation

- Virtual memory can be implemented via:
  - Demand paging
  - Demand segmentation

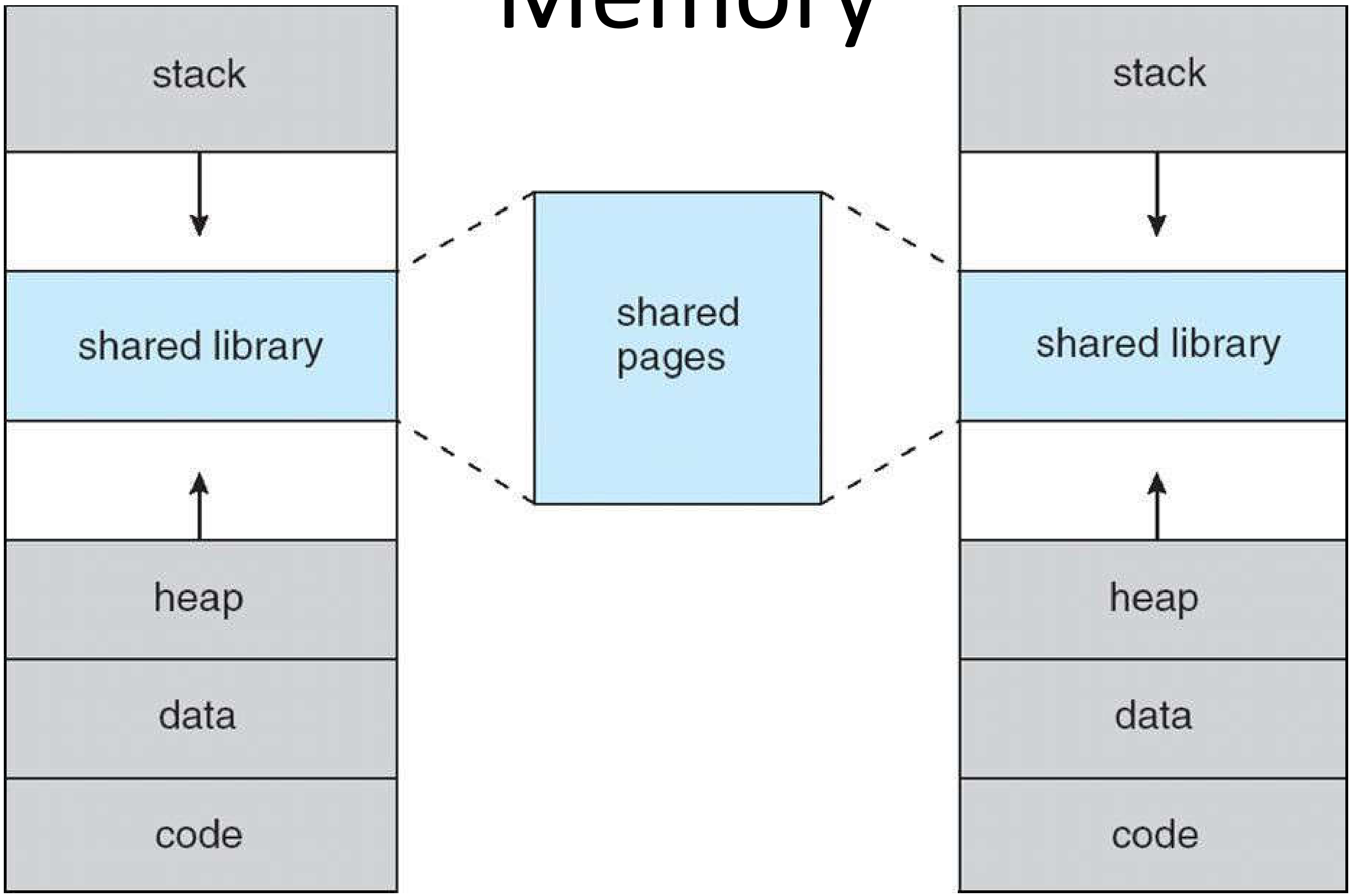
Virtual Memory That is Larger Than Physical Memory



Virtual-address Space



Shared Library Using Virtual Memory

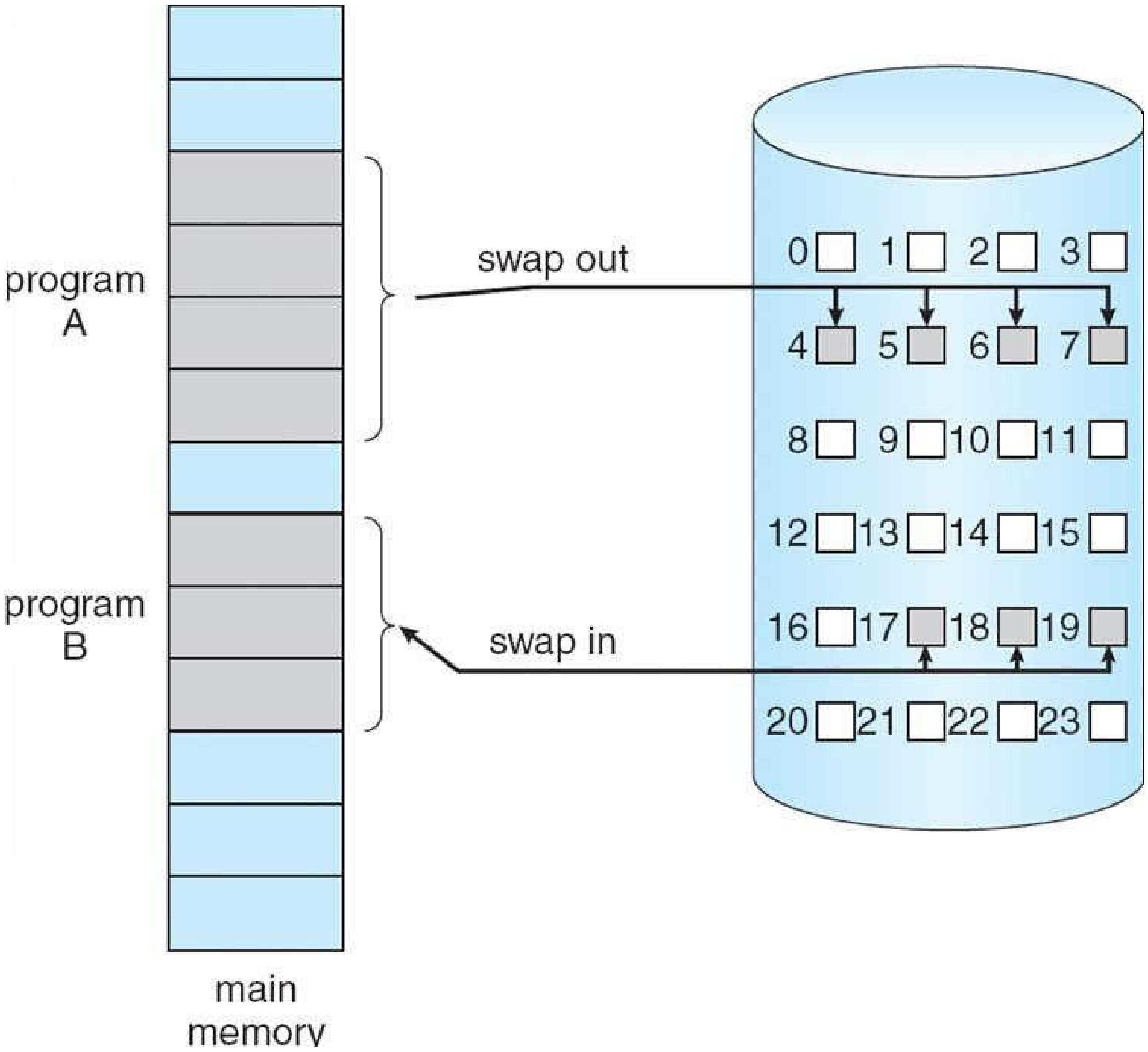


Demand Paging

- Bring a page into memory only when it is needed
  - Less I/O needed
  - Less memory needed
  - Faster response
  - More users

- Page is needed $\Rightarrow$ reference to it
  - invalid reference $\Rightarrow$ abort
  - not-in-memory $\Rightarrow$ bring to memory
- Lazy swapper – never swaps a page into memory unless page will be needed
  - Swapper that deals with pages is a **pager**

Transfer of a Paged Memory to Contiguous Disk Space



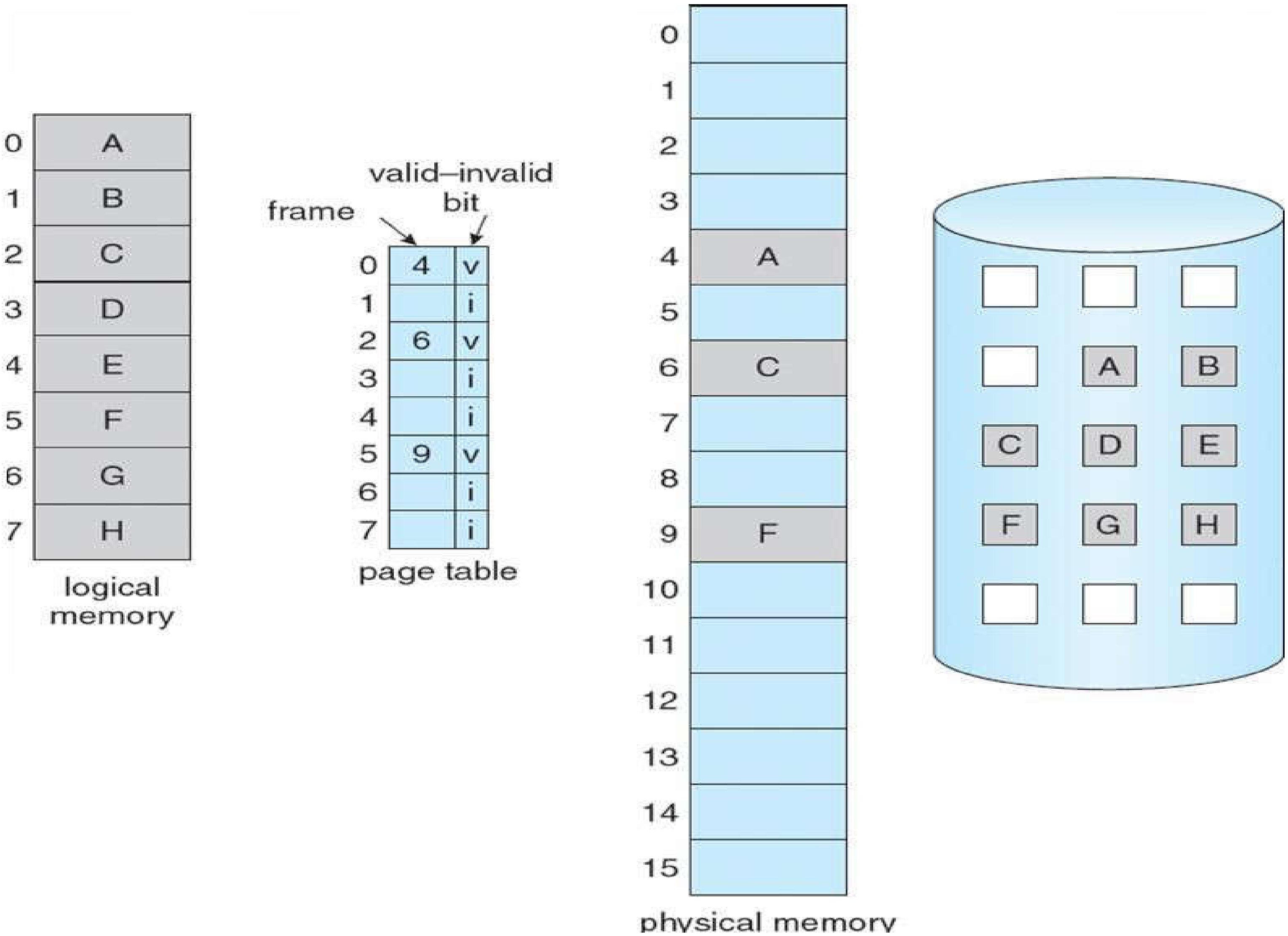
Valid-Invalid Bit

- **V/I bit** is a hardware support
  - associated with each page table entry (PTE)

    1: page is legal & in-memory

    0: page is invalid OR valid but on the disk → page fault

Page Table When Some Pages Are Not in Main Memory



	frame	valid–invalid bit
0	4	v
1		i
2	6	v
3		i
4		i
5	9	v
6		i
7		i

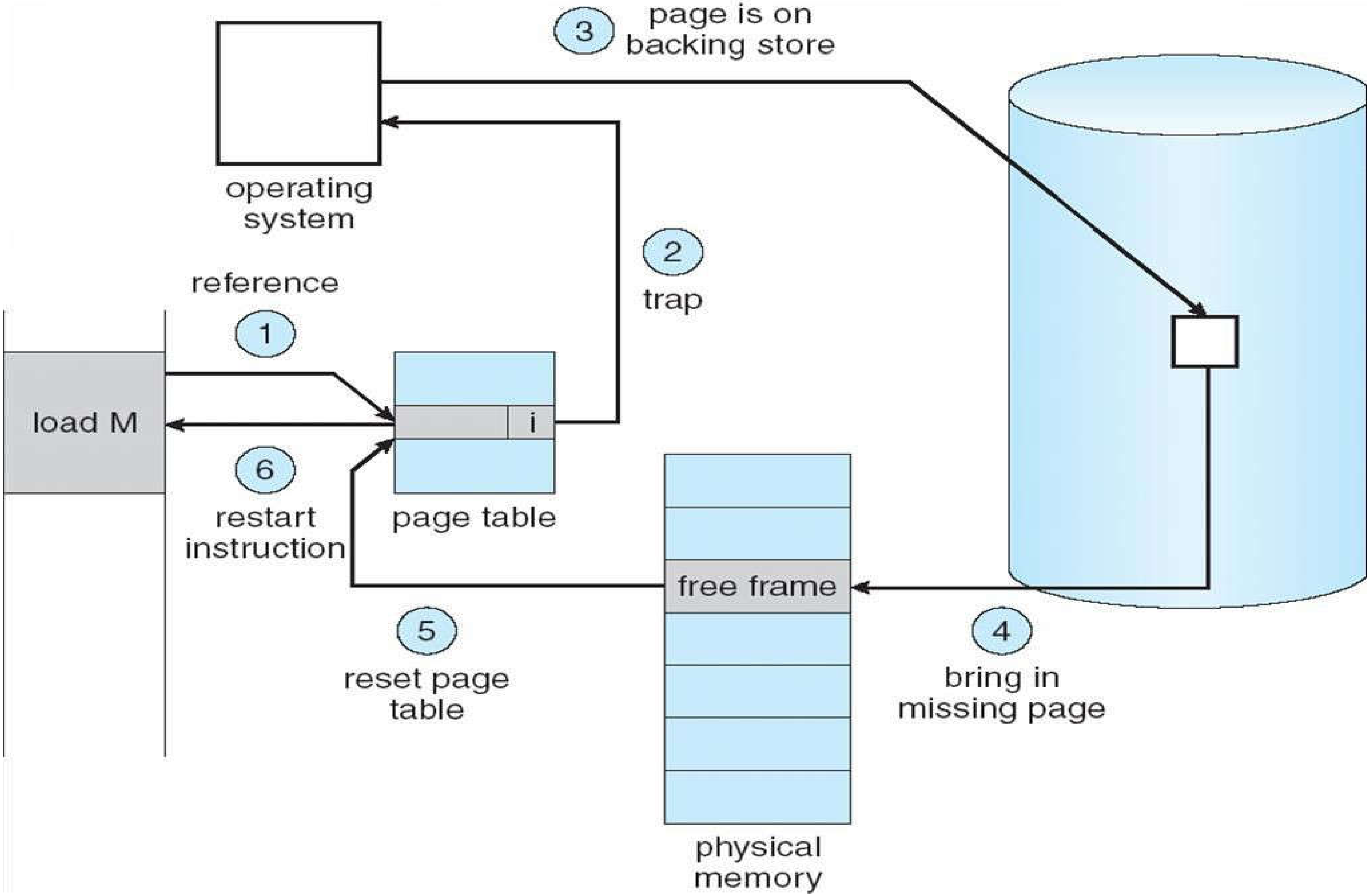
Page Fault

- If there is a reference to a page, first reference to that page will trap to operating system:

  page fault

1. Operating system looks at another table to decide:
   - Invalid reference ⇒ abort
   - Just not in memory
2. Get free frame
3. Swap page into frame
4. Reset tables
5. Set validation bit = **v**
6. Restart the instruction that caused the page fault

Steps in Handling a Page Fault



Page Fault (Cont..)

- **Pure demand paging** : Never bring a page into memory until it is required
- H/w for demand paging: (same as for paging and swapping)
  - Page-table
  - Secondary memory
- Restart instruction – save state
  - Computes and attempts to access both ends of both blocks
  - Temporary registers to hold values of overwritten locations

Performance of Demand Paging

- Page Fault Rate $0 \leq p \leq 1.0$
  - if $p = 0$ no page faults
  - if $p = 1$, every reference is a fault

- Effective Access Time = $(1 – p)$ x ma + p x page fault time
- page fault overhead:

  swap page out

  swap page in

  restart overhead

Demand Paging Example

- Memory access time = 200 nanoseconds
- Average page-fault service time = 8 milliseconds

- EAT = (1 – p) x 200 + p (8 milliseconds)

  = (1 – p x 200 + p x 8,000,000

  = 200 + p x 7,999,800

- If one access out of 1,000 causes a page fault, then

  EAT = 8.2 microseconds.

  This is a slowdown by a factor of 40!!

Process Creation

- Virtual memory allows other benefits during process creation:

  - Copy-on-Write

  - Memory-Mapped Files (later)

Copy-on-Write

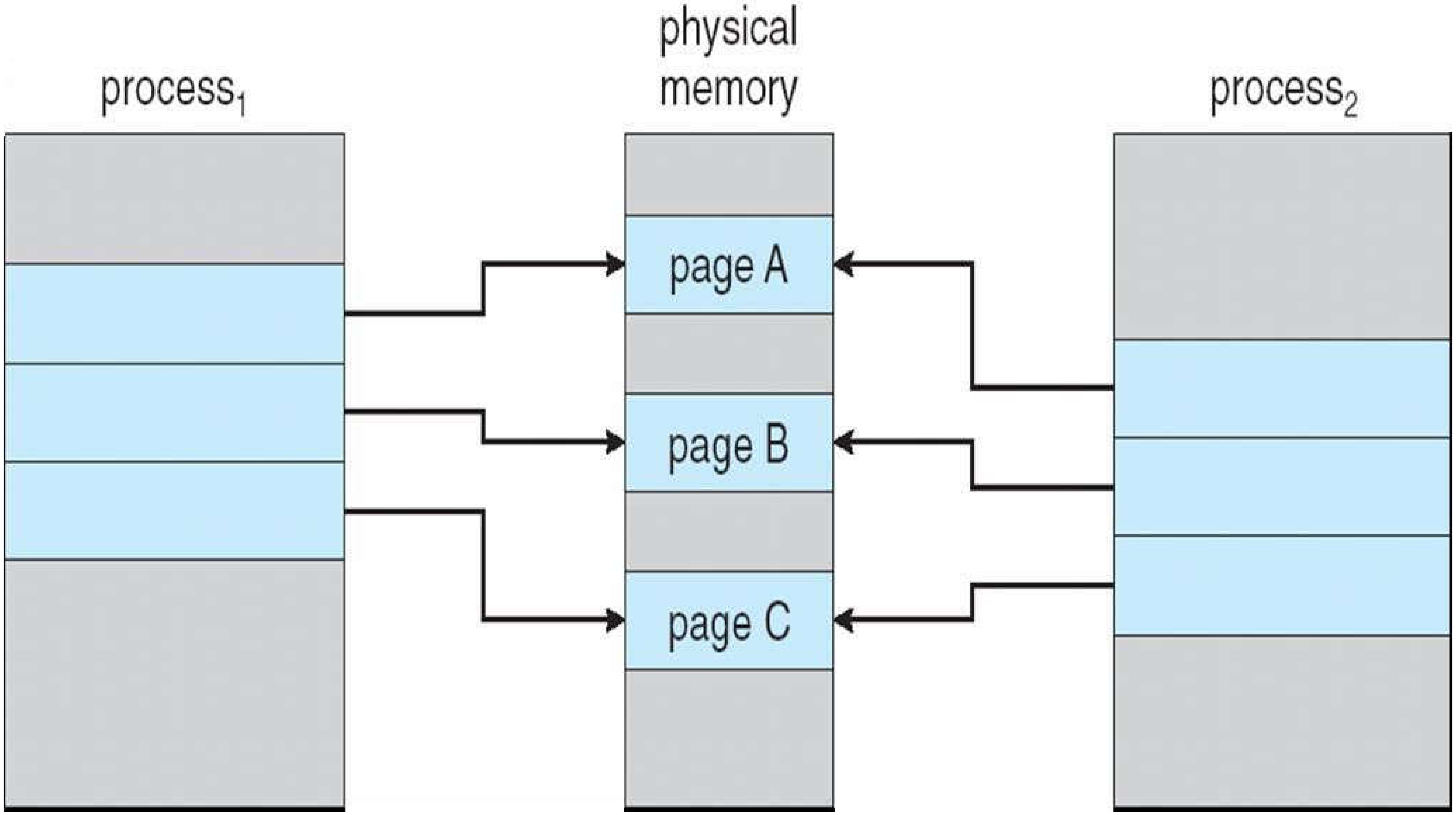
- Copy-on-Write (COW) allows both parent and child processes to initially *share* the same pages in memory

  If either process modifies a shared page, only then is the page copied

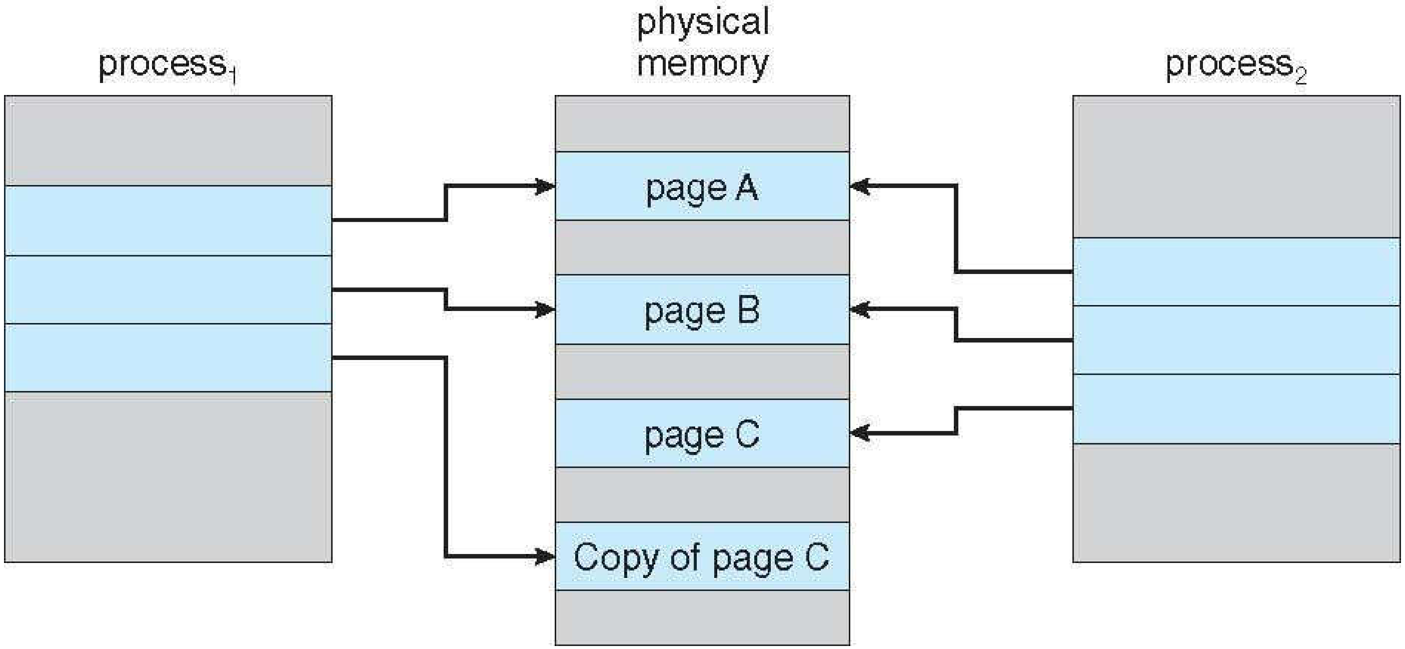
- COW allows more efficient process creation as only modified pages are copied

- Free pages are allocated from a **pool** of zeroed-out pages

Before Process 1 Modifies Page C



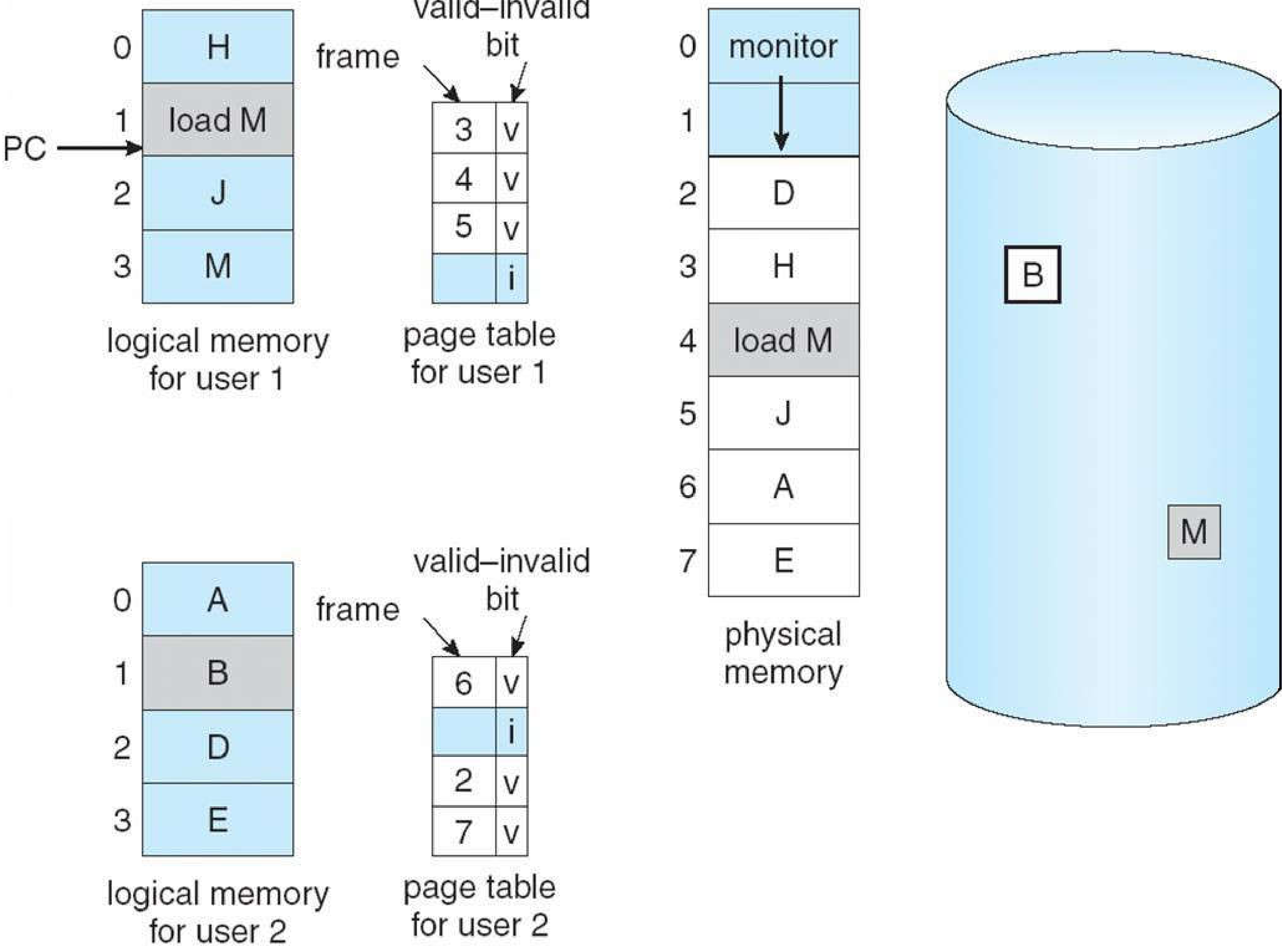
After Process 1 Modifies Page C



What happens if there is no free frame?

- Page replacement – find some page in memory, but not really in use, swap it out
  - algorithm
  - performance – want an algorithm which will result in minimum number of page faults
- Same page may be brought into memory several times

Need For Page Replacement



Basic Page Replacement

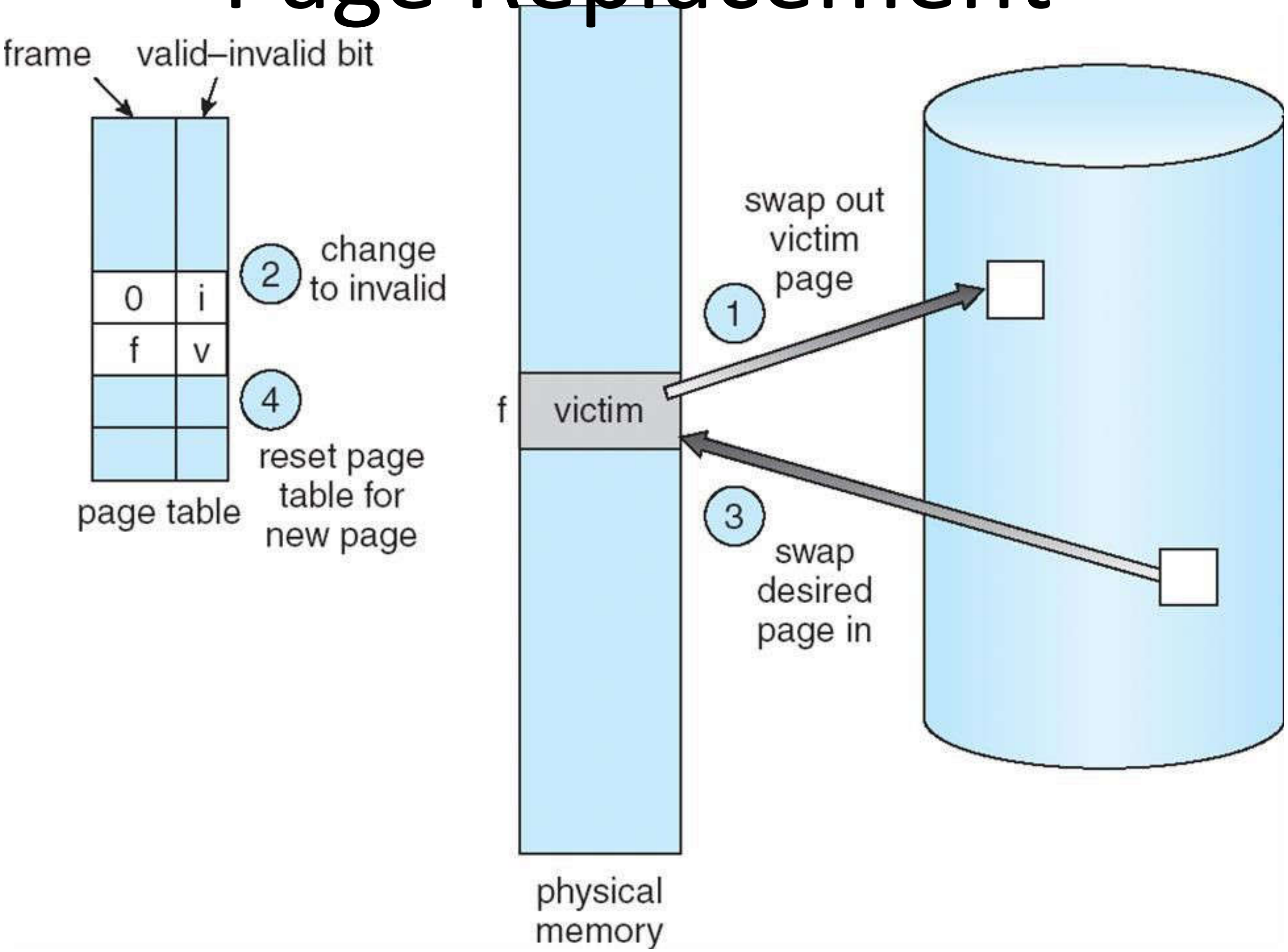
1. Find the location of the desired page on disk

2. Find a free frame:
   - If there is a free frame, use it
   - If there is no free frame, use a page replacement algorithm to select a **victim** frame

3. Bring the desired page into the (newly) free frame; update the page and frame tables

4. Restart the process

Page Replacement



Page Replacement

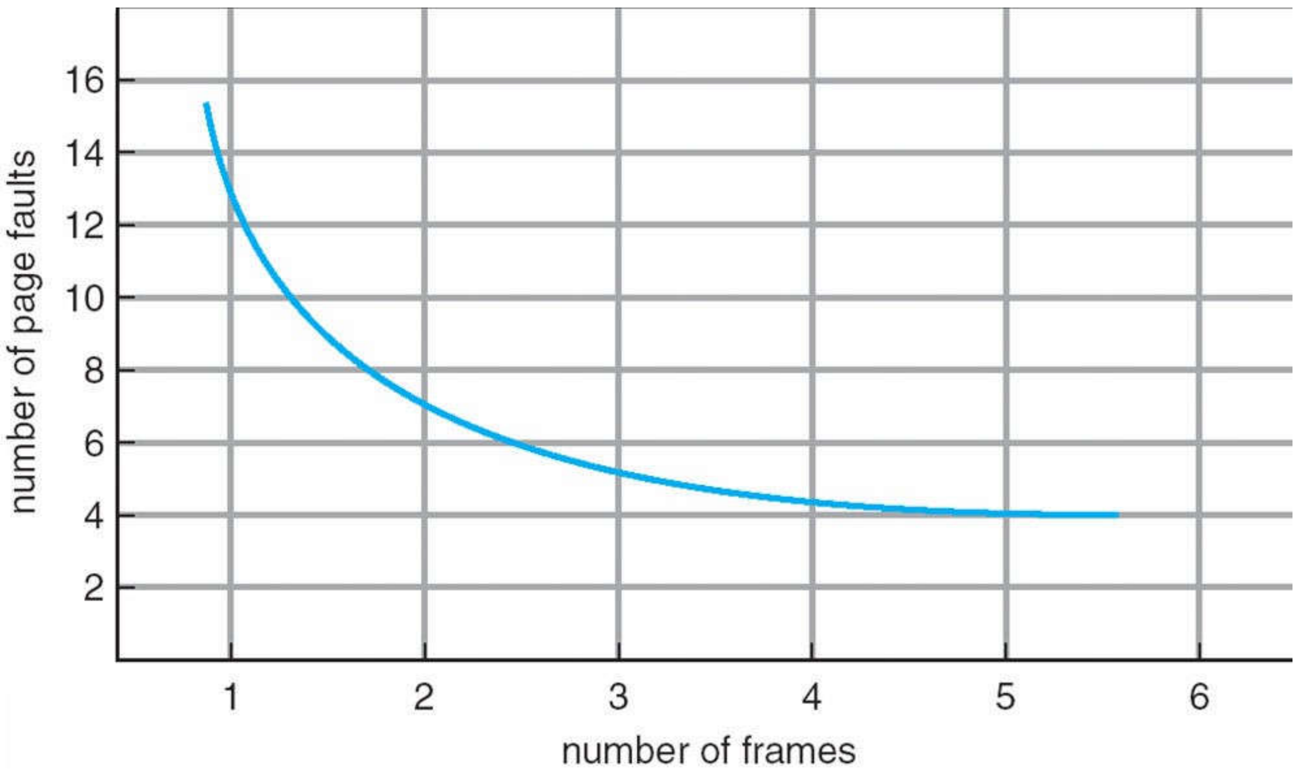
- No free frames ➔ two page transfers (one out & one in)

Sol: **modify (dirty) bit --** only modified pages are written to disk

- Frame-allocation algorithm
- Page-replacement algorithm → lowest page fault rate
- Evaluate algorithm by running it on a particular string of memory references (reference string) and computing the number of page faults on that string
- In all our examples, the reference string is

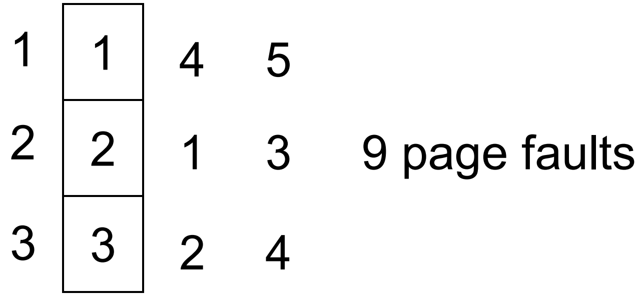
1, 4, 1, 6, 1, 6, 1, 6, 1, 6, 1

Graph of Page Faults Versus The Number of Frames



First-In-First-Out (FIFO) Algorithm

- Reference string: 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5
- 3 frames (3 pages can be in memory at a time per process)

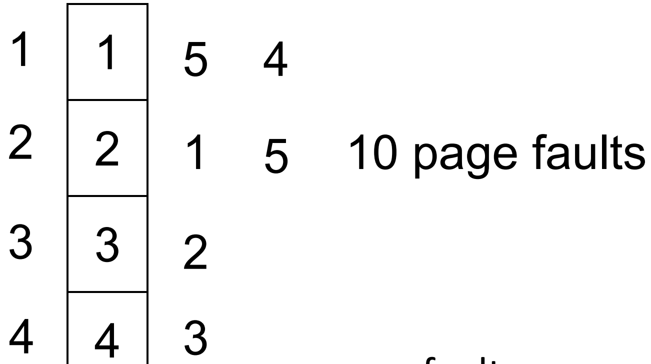


1 1 4 5

2 2 1 3 9 page faults

3 3 2 4

- 4 frames



1 1 5 4

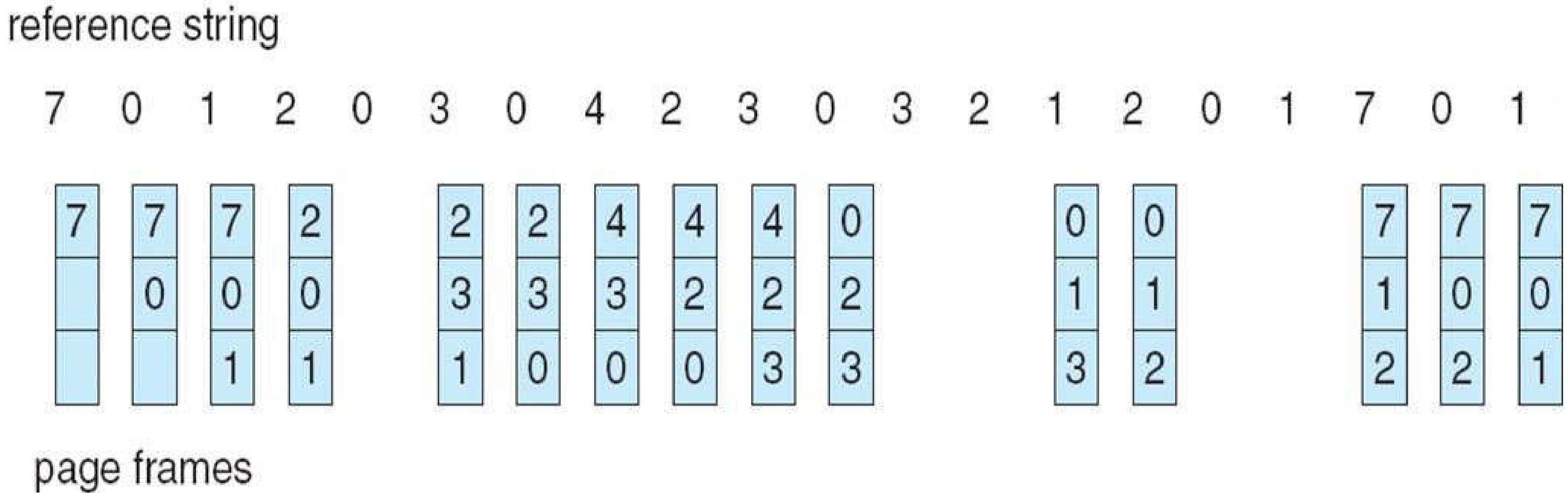
2 2 1 5 10 page faults

3 3 2

4 4 3

- Belady’s Anomaly: more frames ⇒ more page faults

FIFO Page Replacement

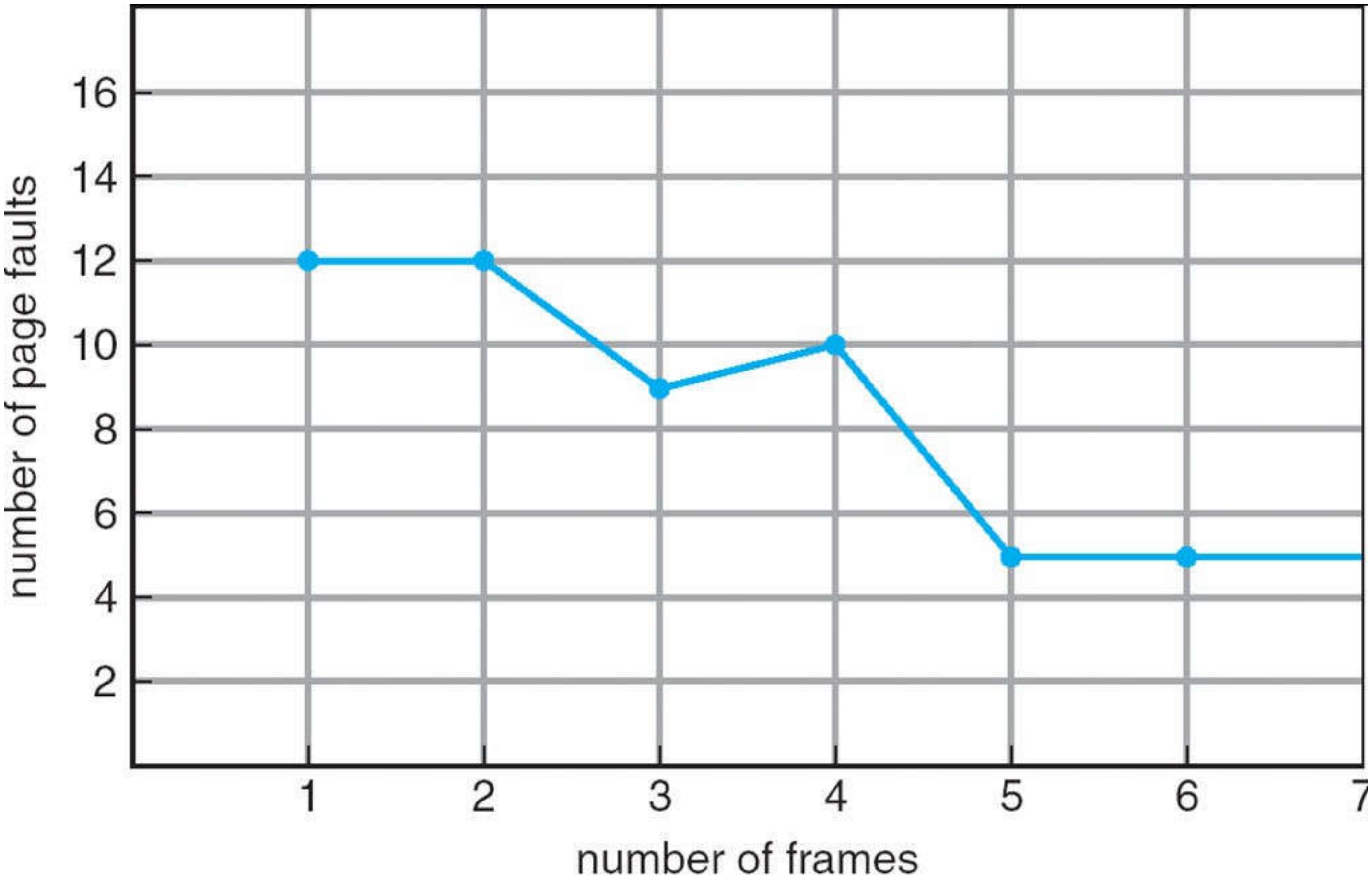


reference string

7	0	1	2	0	3	0	4	2	3	0	3	2	1	2	0	1	7	0	1
7	7	7	2		2	2	4	4	4	0			0	0			7	7	7
	0	0	0		3	3	3	2	2	2			1	1			1	0	0
		1	1		1	0	0	0	3	3			3	2			2	2	1

page frames

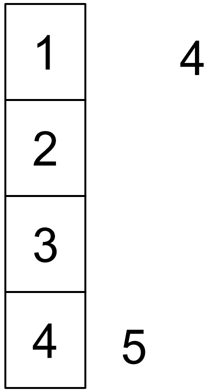
FIFO Illustrating Belady’s Anomaly



Optimal Algorithm

- Replace page that will not be used for longest period of time
- 4 frames example

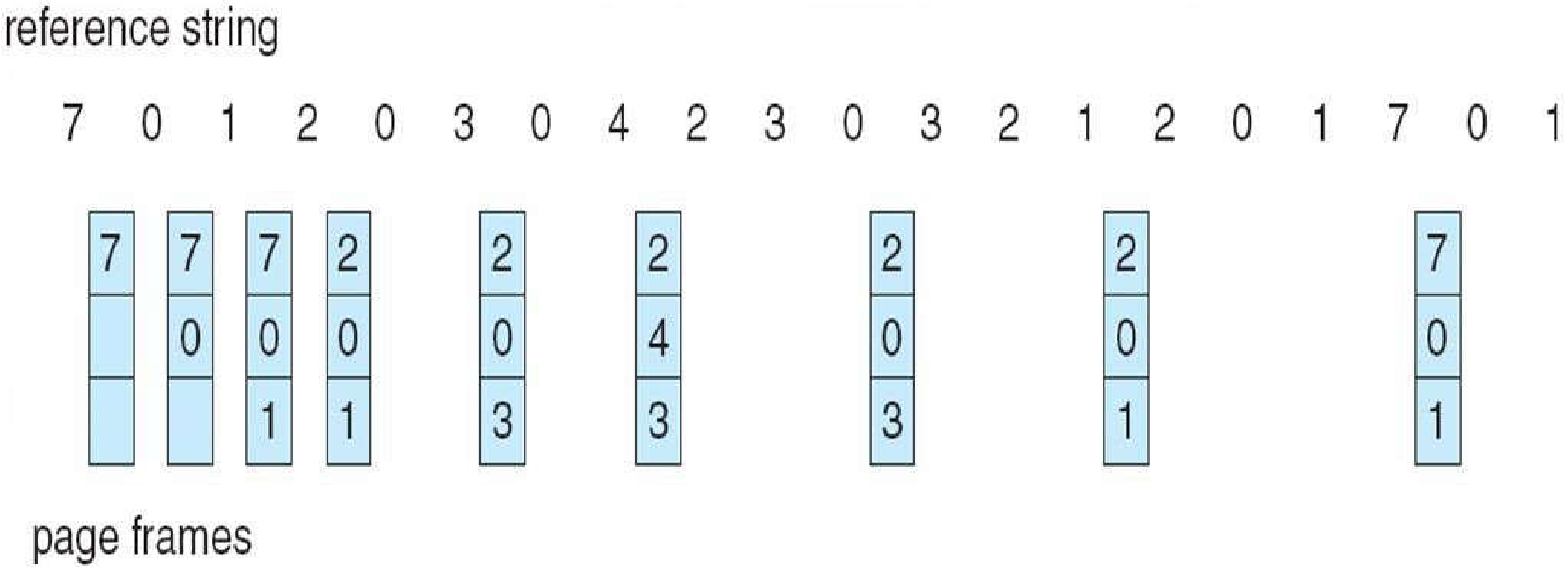
1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5



6 page faults

- How do you know this?
- Used for measuring how well your algorithm performs

Optimal Page Replacement



reference string

7 0 1 2 0 3 0 4 2 3 0 3 2 1 2 0 1 7 0 1

page frames

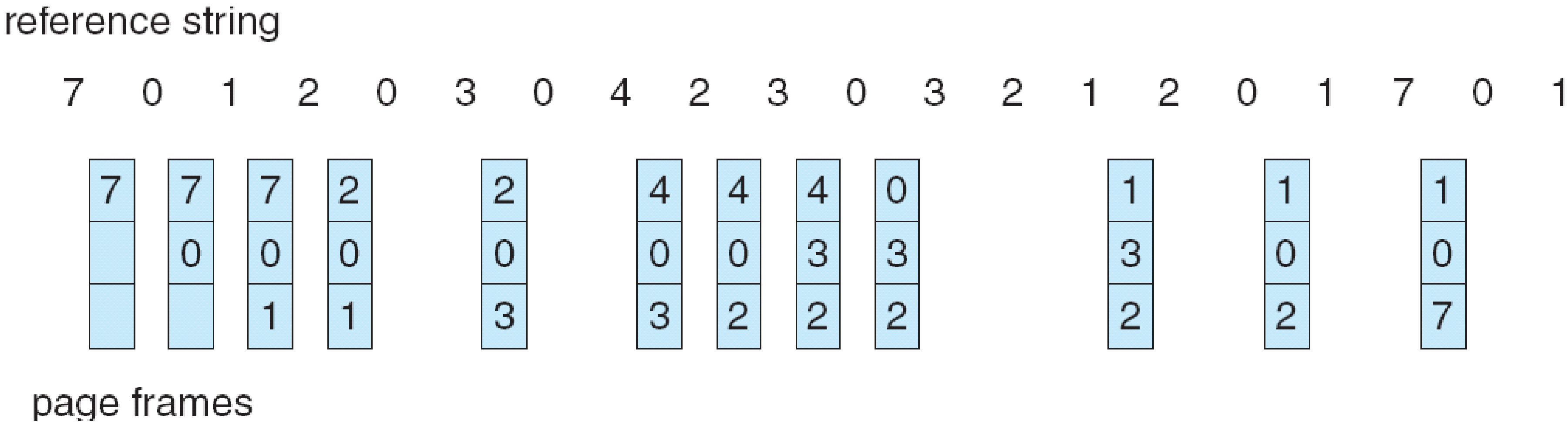
Least Recently Used (LRU) Algorithm

- Reference string: 1, 2, 3, 4, 1, 2, **5**, 1, 2, **3**, **4**, **5**

1	1	1	1	**5**
2	2	2	2	2
3	**5**	5	**4**	4
4	4	**3**	3	3

- Counter implementation
  - Every page entry has a counter; every time page is referenced through this entry, copy the clock into the counter
  - When a page needs to be changed, look at the counters to determine which are to change

LRU Page Replacement



reference string

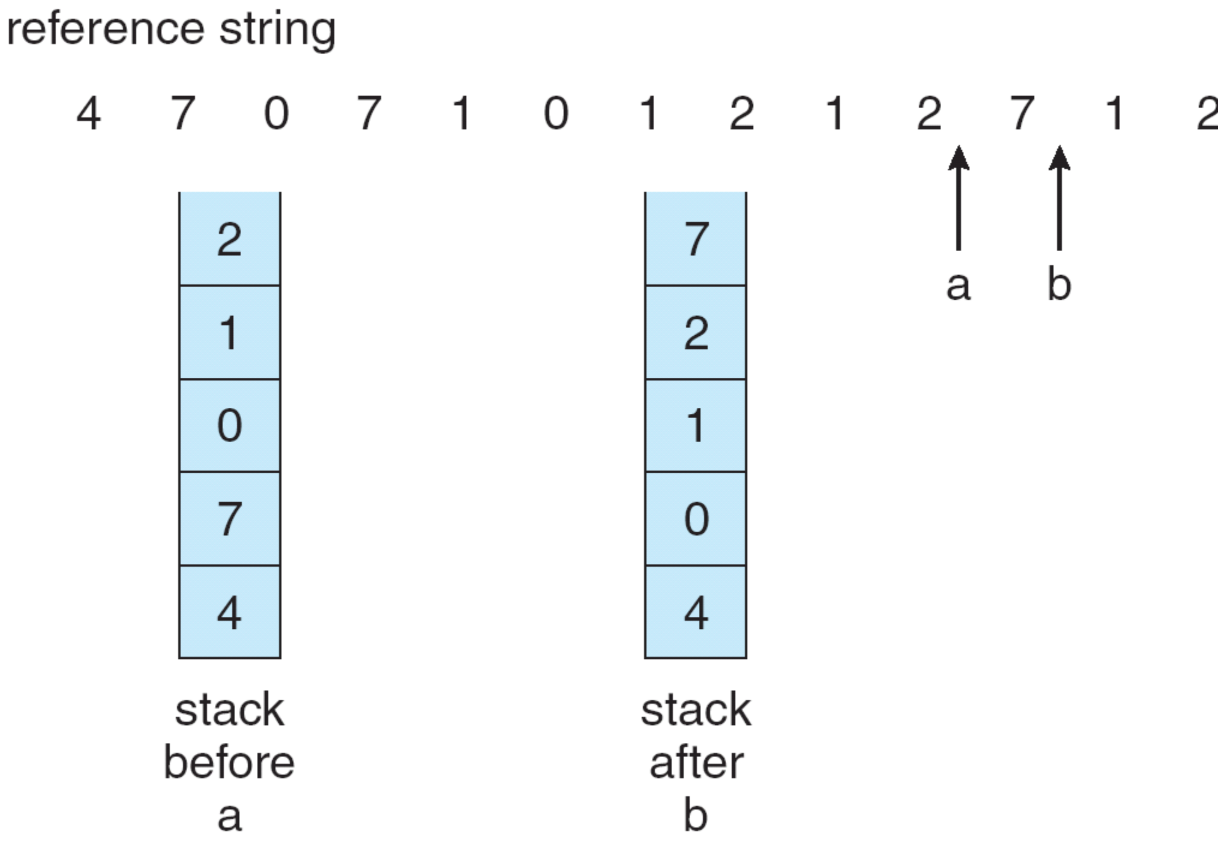
7	0	1	2	0	3	0	4	2	3	0	3	2	1	2	0	1	7	0	1
7	7	7	2		2		4	4	4	0			1		1		1		
	0	0	0		0		0	0	3	3			3		0		0		
		1	1		3		3	2	2	2			2		2		7		

page frames

LRU Algorithm (Cont.)

- Stack implementation – keep a stack of page numbers in a double link form:
  - Page referenced:
    - move it to the top
    - requires 6 pointers to be changed
  - No search for replacement

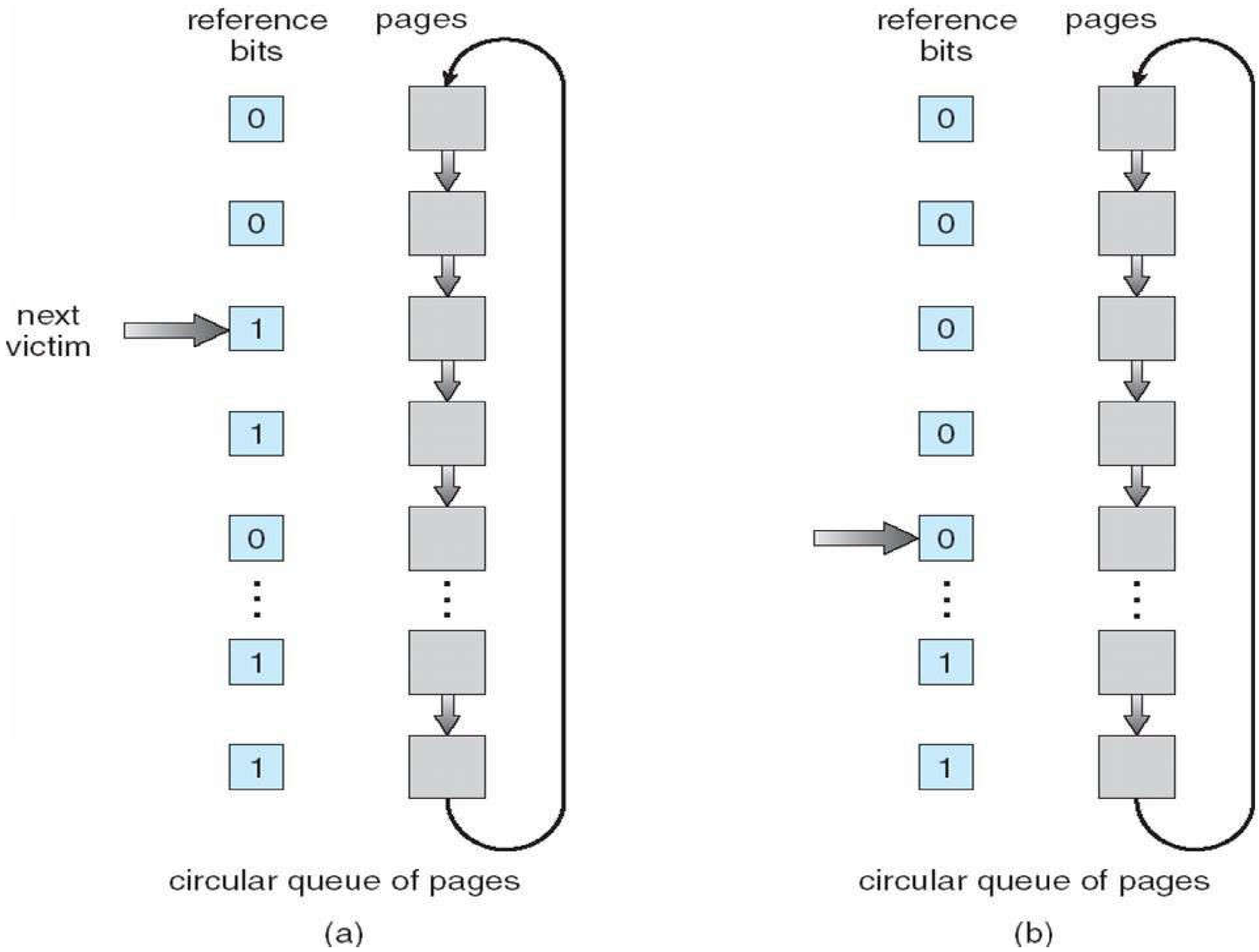
Use Of A Stack to Record The Most Recent Page References



LRU Approximation Algorithms

- Reference bit
  - With each page associate a bit, initially = 0
  - When page is referenced bit set to 1
  - Replace the one which is 0 (if one exists)
    - We do not know the order, however
- Second chance
  - Need reference bit
  - Clock replacement
  - If page to be replaced (in clock order) has reference bit = 1 then:
    - set reference bit 0
    - leave page in memory
    - replace next page (in clock order), subject to same rules

Second-Chance (clock) Page-Replacement Algorithm



Counting Algorithms

- Keep a counter of the number of references that have been made to each page

- **LFU Algorithm**: replaces page with smallest count

- **MFU Algorithm**: based on the argument that the page with the smallest count was probably just brought in and has yet to be used

Allocation of Frames

- Each process needs *minimum* number of pages
- Example: IBM 370 – 6 pages to handle SS MOVE instruction:
  - instruction is 6 bytes, might span 2 pages
  - 2 pages to handle *from*
  - 2 pages to handle *to*
- Two major allocation schemes
  - fixed allocation
  - priority allocation

Fixed Allocation

- Equal allocation – For example, if there are 100 frames and 5 processes, give each process 20 frames.
- Proportional allocation – Allocate according to the size of process
  - $s_i =$ size of process p_i
  - $S = \sum s_i$
  - $m =$ total number of frames
  - $a_i =$ allocation for $p_i = \frac{s_i}{S} \times m$

$$m = 64$$

$$s_i = 10$$

$$s_2 = 127$$

$$a_1 = \frac{10}{137} \times 64 \approx 5$$

$$a_2 = \frac{127}{137} \times 64 \approx 59$$

Priority Allocation

- Use a proportional allocation scheme using priorities rather than size

- If process P_i generates a page fault,
  - select for replacement one of its frames
  - select for replacement a frame from a process with lower priority number

Global vs. Local Allocation

- **Global replacement** – process selects a replacement frame from the set of all frames; one process can take a frame from another

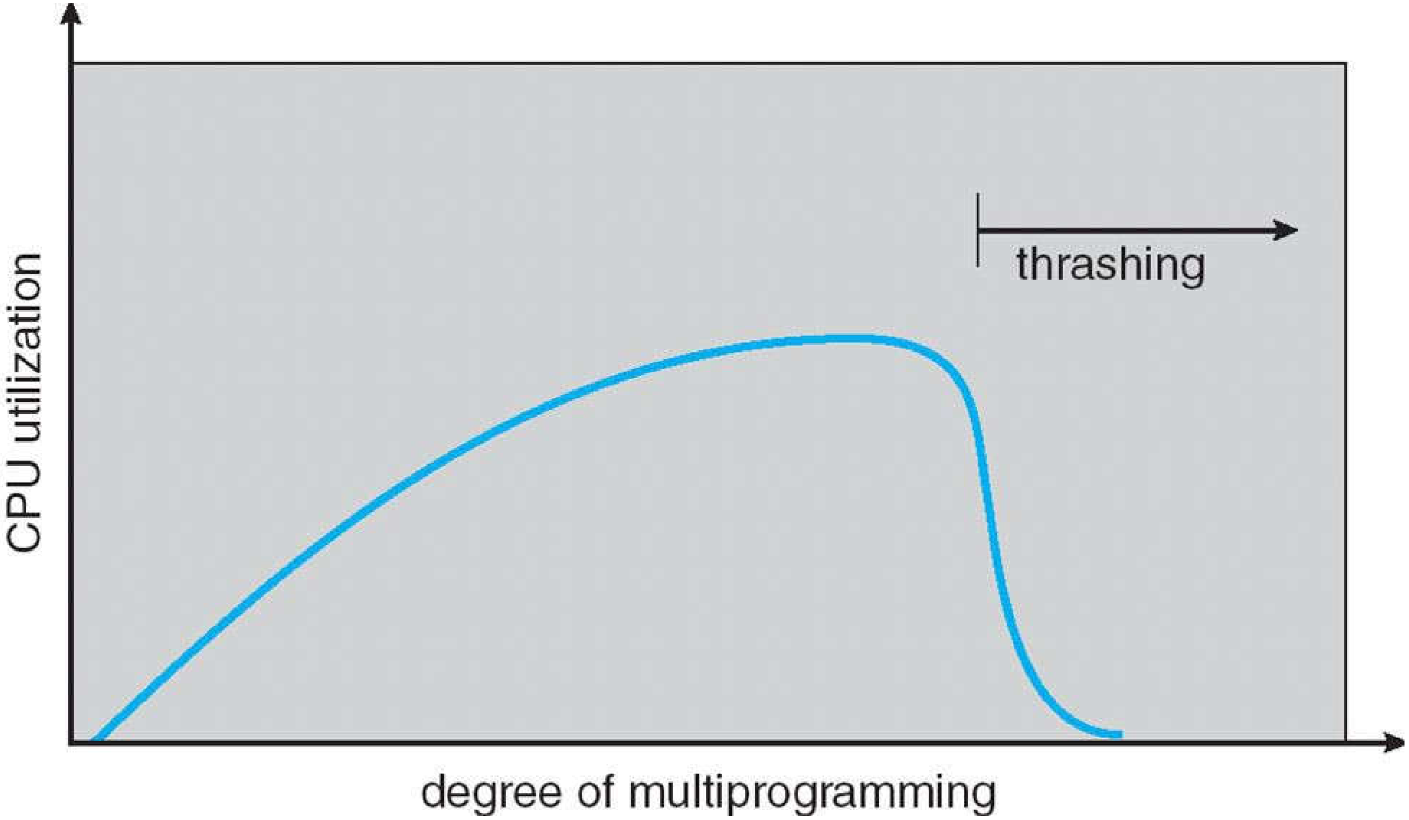
- **Local replacement** – each process selects from only its own set of allocated frames

Thrashing

- If a process does not have “enough” pages, the page-fault rate is very high. This leads to:
  - low CPU utilization
  - operating system thinks that it needs to increase the degree of multiprogramming
  - another process added to the system

- **Thrashing** ≡ a process is busy swapping pages in and out

Thrashing (Cont.)

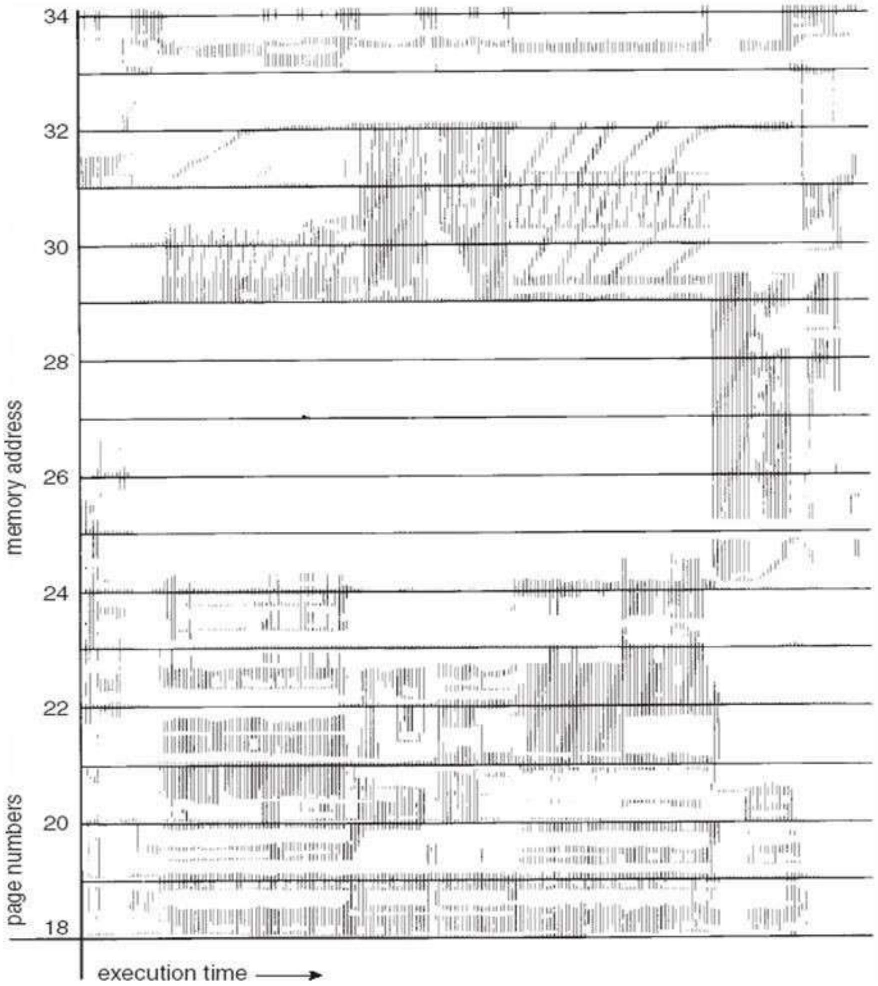


Demand Paging and Thrashing

- Why does demand paging work?
  Locality model
  - Process migrates from one locality to another
  - Localities may overlap

- Why does thrashing occur?
  Σ size of locality > total memory size

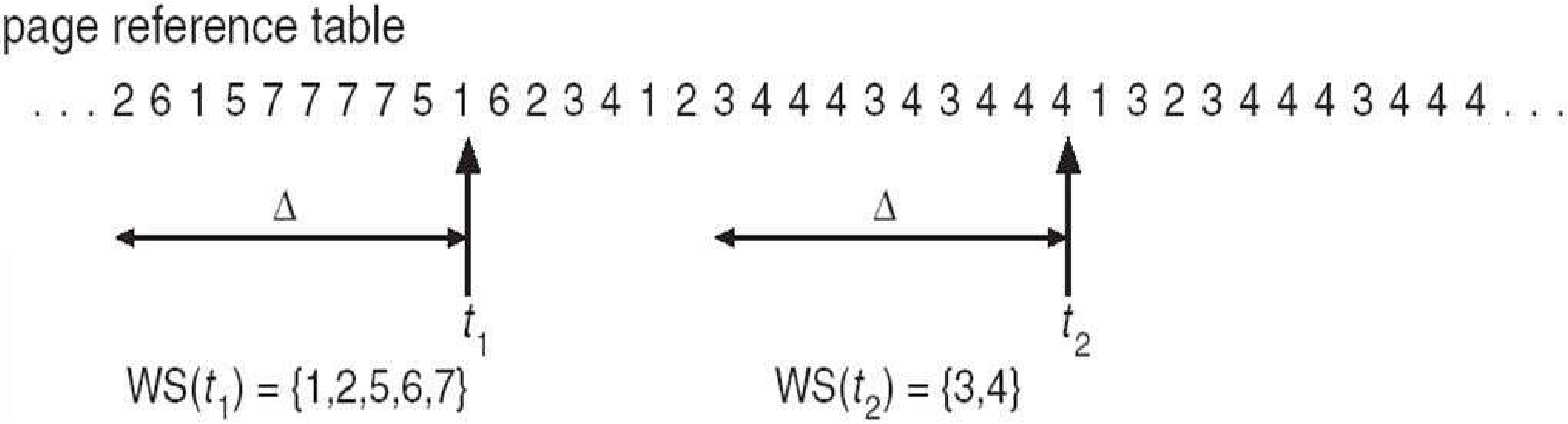
Locality In A Memory-Reference Pattern



Working-Set Model

- $\Delta \equiv$ working-set window $\equiv$ a fixed number of page references
  Example: 10,000 instruction
- WSS_i (working set of Process P_i) =
  total number of pages referenced in the most recent Δ (varies in time)
  - if Δ too small will not encompass entire locality
  - if Δ too large will encompass several localities
  - if $\Delta = \infty \Rightarrow$ will encompass entire program
- $D = \Sigma\, WSS_i \equiv$ total demand frames
- if $D > m \Rightarrow$ Thrashing
- Policy if $D >$ m, then suspend one of the processes

Working-set model

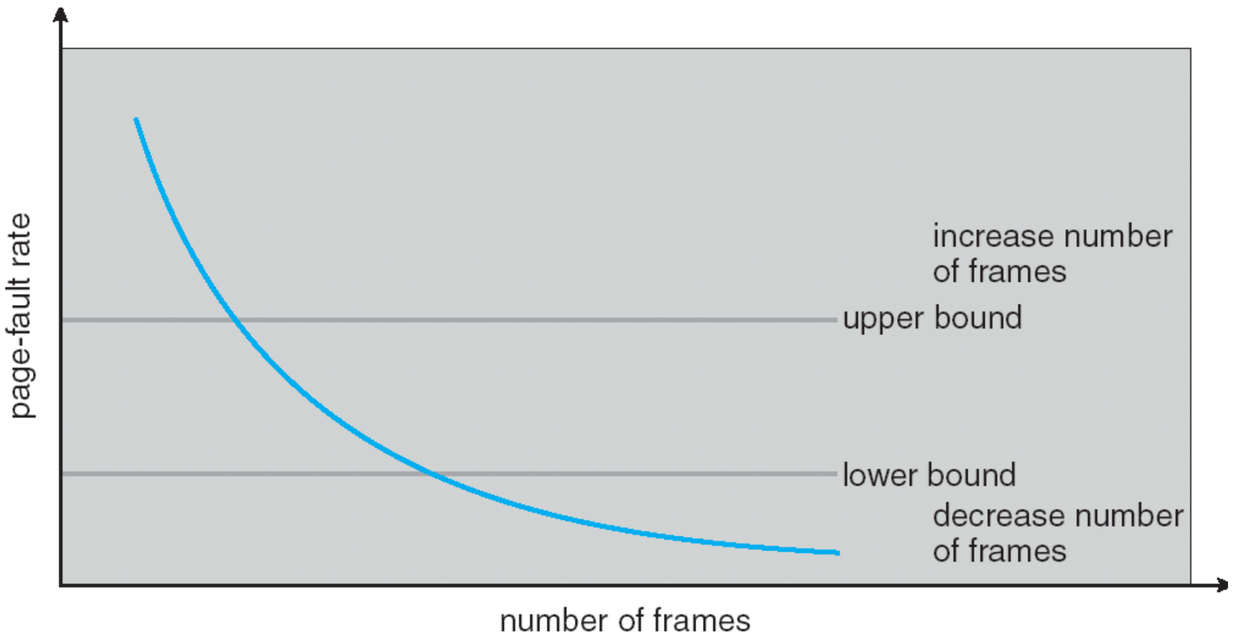


Keeping Track of the Working Set

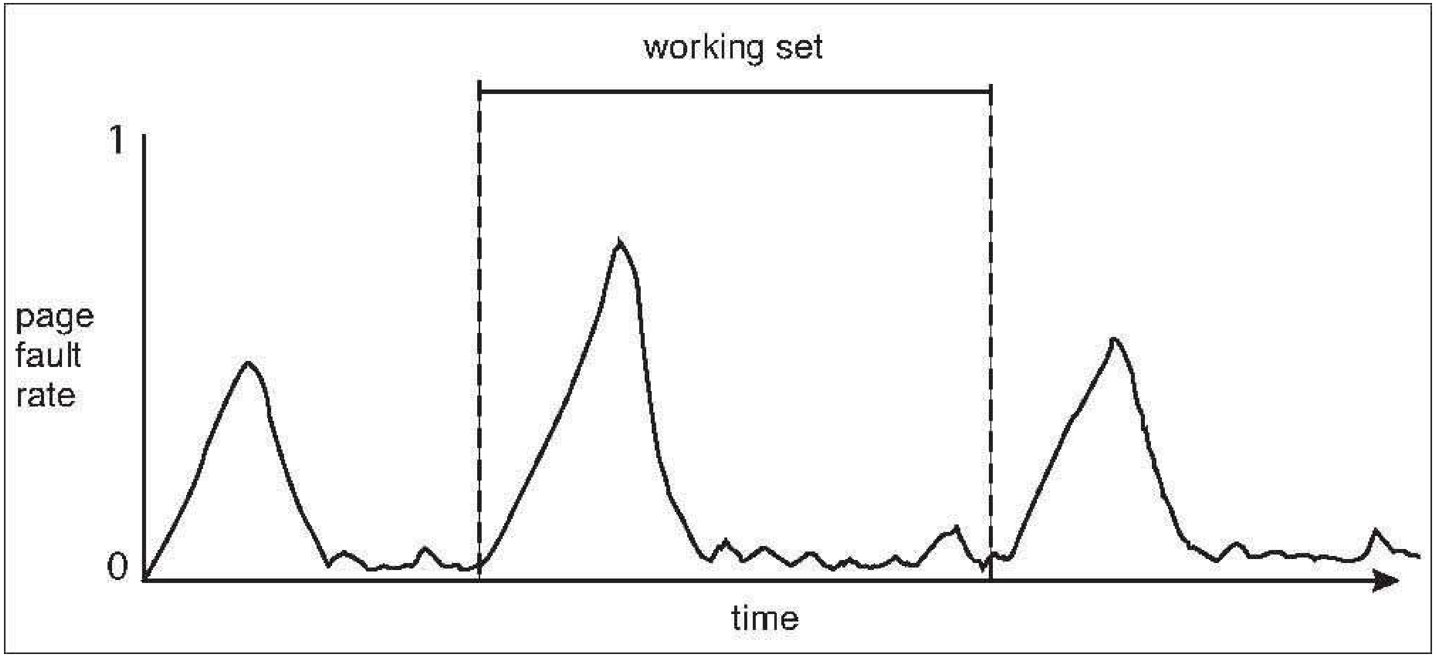
- Approximate with interval timer + a reference bit
- Example: $\Delta = 10{,}000$
  - Timer interrupts after every 5000 time units
  - Keep in memory 2 bits for each page
  - Whenever a timer interrupts copy and sets the values of all reference bits to 0
  - If one of the bits in memory = 1 $\Rightarrow$ page in working set
- Why is this not completely accurate?
- Improvement = 10 bits and interrupt every 1000 time units

Page-Fault Frequency Scheme

- Establish "acceptable" page-fault rate
  - If actual rate too low, process loses frame
  - If actual rate too high, process gains frame



Working Sets and Page Fault Rates



Unit - 5
Deadlocks

Deadlocks

- The Deadlock Problem
- System Model
- Deadlock Characterization
- Methods for Handling Deadlocks
- Deadlock Prevention
- Deadlock Avoidance
- Deadlock Detection
- Recovery from Deadlock

The Deadlock Problem

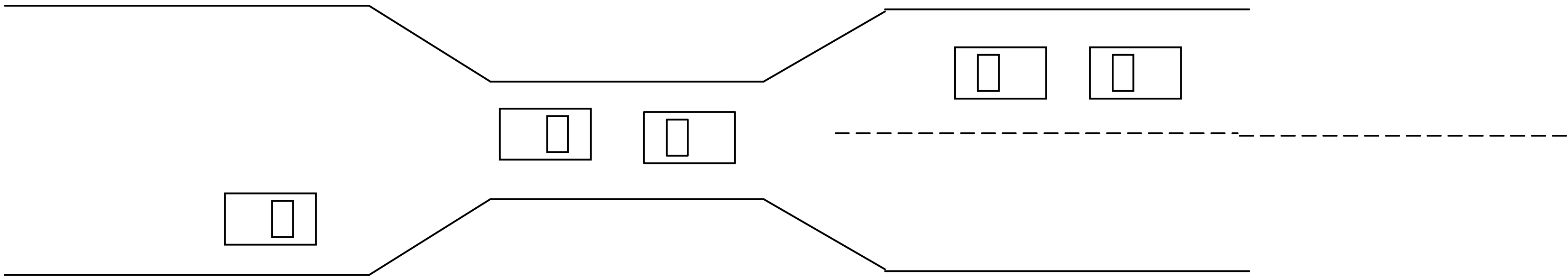
- A set of blocked processes each holding a resource and waiting to acquire a resource held by another process in the set

- Example
  - System has 2 disk drives
  - P_1 and P_2 each hold one disk drive and each needs another one

- Example
  - semaphores *A* and *B*, initialized to 1

P_0	P_1
wait (A);	**wait(B)**
wait (B);	**wait(A)**

Bridge Crossing Example

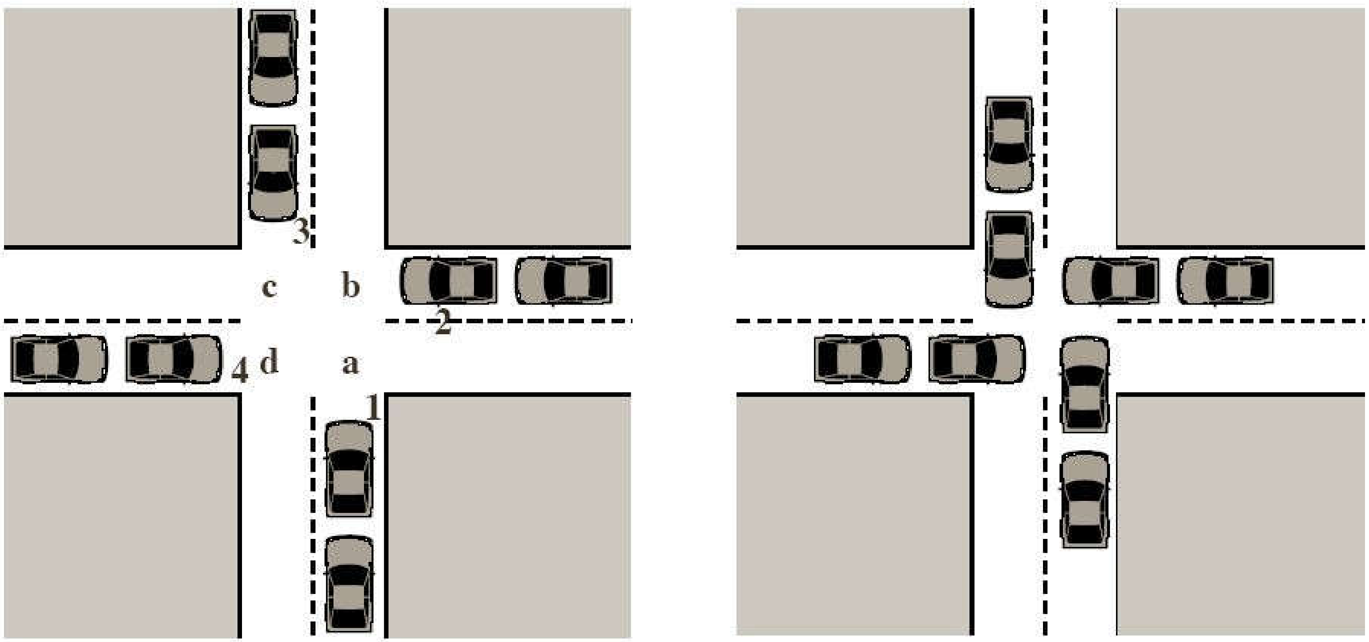


- Traffic only in one direction
- Each section of a bridge can be viewed as a resource
- If a deadlock occurs, it can be resolved if one car backs up
  - preempt resources and rollback
- Several cars may have to be backed up if a deadlock occurs
- Starvation is possible
- **Note :** Most OSes do not prevent or deal with deadlocks

Deadlock Principles

□ A deadlock is a permanent blocking of a set of threads

✓ a deadlock can happen while threads/processes are competing for system resources or communicating with each other



(a) Deadlock possible

(b) Deadlock

Illustration of a deadlock

Tanenbaum, A. S. (2001)
Modern Operating Systems (2nd Edition).

System Model

- Resource types $R_1, R_2, \ldots, R_m$

  CPU cycles, memory space, I/O devices

- Each resource type R_i has W_i instances.

- Each process utilizes a resource as follows:
  - **request**
  - **use**
  - **release**

Deadlock Characterization

Deadlock can arise if four conditions hold simultaneously.

- **Mutual exclusion:** only one process at a time can use a resource
- **Hold and wait:** a process holding at least one resource is waiting to acquire additional resources held by other processes
- **No preemption:** a resource can be released only voluntarily by the process holding it, after that process has completed its task
- **Circular wait:** there exists a set $\{P_0, P_1, \ldots, P_0\}$ of waiting processes such that P_0 is waiting for a resource that is held by P_1, P_1 is waiting for a resource that is held by

  P_2, …, P_{n-1} is waiting for a resource that is held by P_n, and P_0 is waiting for a resource that is held by P_0.

NECESSARY CONDITIONS

ALL of these four **must** happen simultaneously for a deadlock to occur:

Mutual exclusion

One or more than one resource must be held by a process in a non-sharable (exclusive) mode.

Hold and Wait

A process holds a resource while waiting for another resource.

No Preemption

There is only voluntary release of a resource - nobody else can make a process give up a resource.

Circular Wait

Process A waits for Process B waits for Process C .... waits for Process A.

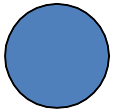
Resource-Allocation Graph

A set of vertices *V* and a set of edges *E*.

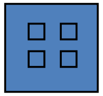
- V is partitioned into two types:
  - $P = \{P_1, P_2, \ldots, P_n\}$, the set consisting of all the processes in the system
  - $R = \{R_1, R_2, \ldots, R_m\}$, the set consisting of all resource types in the system
- request edge :directed edge $P_1 \rightarrow R_j$
- **assignment edge** : directed edge $R_j \rightarrow P_i$

Resource-Allocation Graph (Cont.)

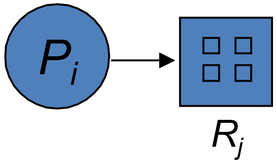
- **Process**



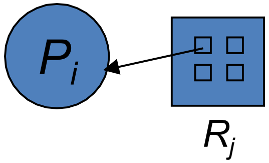
- **Resource Type with 4 instances**



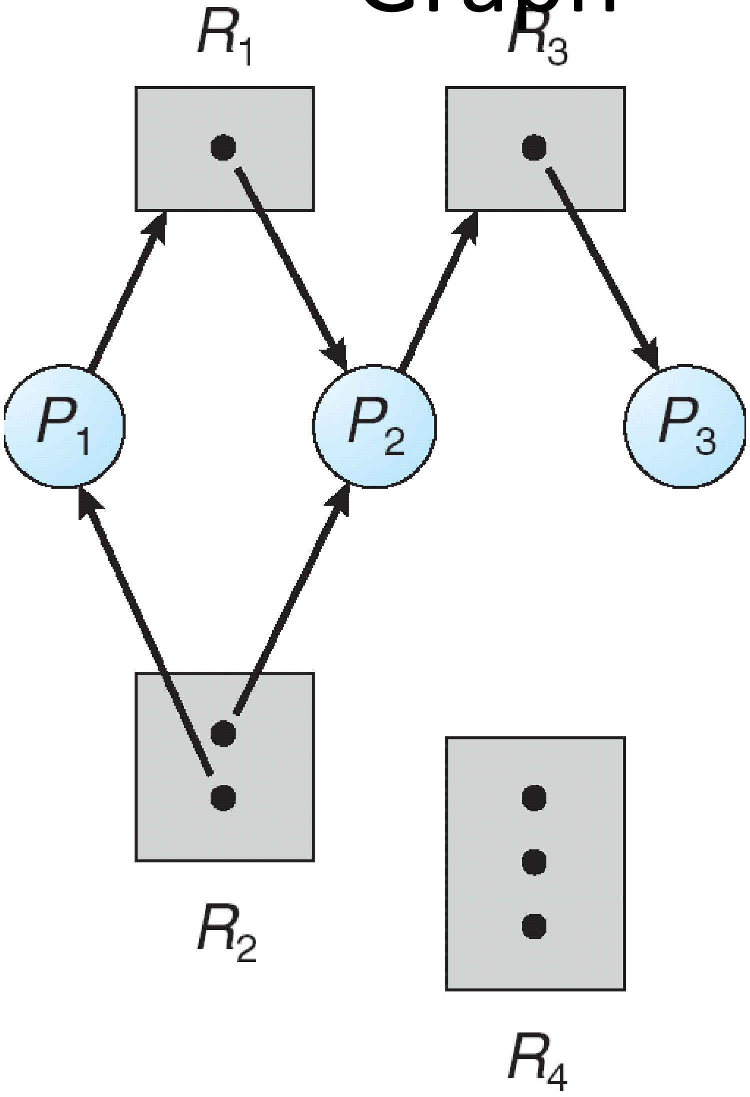
- **P_i requests instance of R_j**



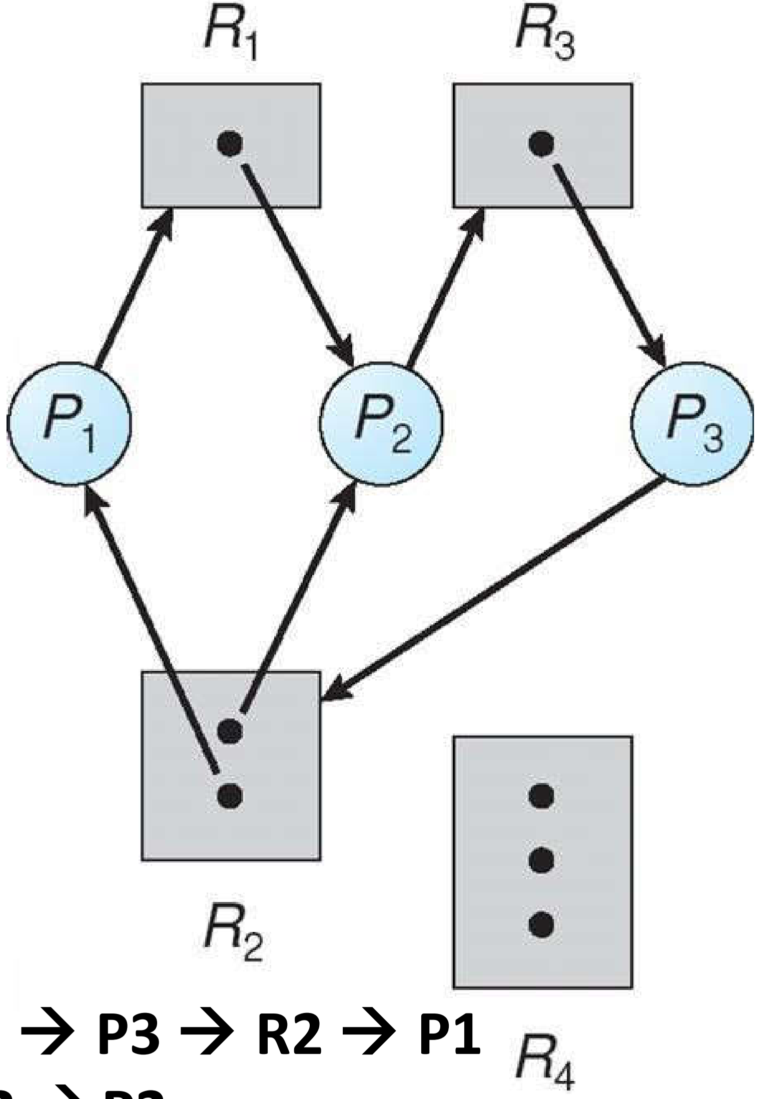
- **P_i is holding an instance of R_j**



Example of a Resource Allocation Graph

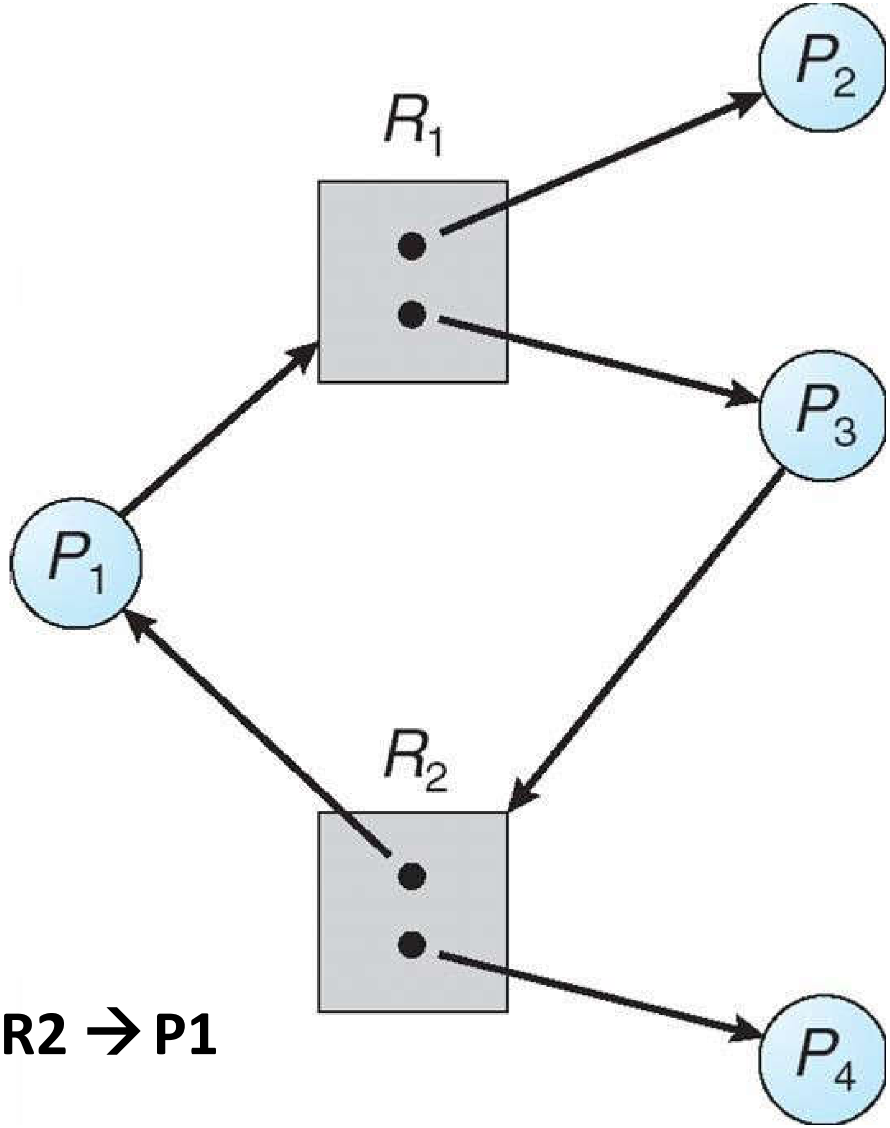


Resource Allocation Graph With A Deadlock



P1→ R1 → P2 → R3 → P3 → R2 → P1
P2 → R3 → P3 → R2 →P2

Graph With A Cycle But No Deadlock



P1→ R1 → P3 → R2 →P1

Basic Facts

- If graph contains no cycles $\Rightarrow$ no deadlock

- If graph contains a cycle $\Rightarrow$
  - if only one instance per resource type, then deadlock
  - if several instances per resource type, possibility of deadlock

Methods for handling deadlocks

- Ensure that the system will *never enter a deadlock state.*
  - **Prevention:** Prevent any one of the 4 conditions from happening.
  - **Avoidance** : Allow all deadlock conditions, but calculate cycles about to happen and stop dangerous operations.
- Allow the system to enter a deadlock state and then
  - detect and
  - recover
- Ignore the problem and pretend that deadlocks never occur in the system; used by most OS, including UNIX.

Deadlock prevention

1. **Mutual exclusion:**

- Hold for non-sharable resources ex: printer
- not required for sharable resources
- Can’t deny mutual exclusion condition

2. Hold and wait:

a. Collect all resources before execution→ resource utilization is low

b. Allow a process to request resources only when it has none → starvation possible

Deadlock prevention

3. No preemption:

- If a process that is holding some resources requests another resource that cannot be immediately allocated to it, then all resources currently being held are released.
- Preempted resources are added to the list of resources for which the process is waiting.
- Process will be restarted only when it can regain its old resources, as well as the new ones that it is requesting.

4. Circular wait:

- impose a total ordering of all resource types,
- each process requests resources in an increasing order

Deadlock Avoidance

Requires that the system has some additional *a priori* information available.

- Simplest and most useful model requires that each process declare the *maximum number* of resources of each type that it may need.

- The deadlock-avoidance algorithm dynamically examines the resource-allocation state to ensure that there can never be a circular-wait condition.

- Resource-allocation *state* is defined by the number of available and allocated resources, and the maximum demands of the processes.

Safe State

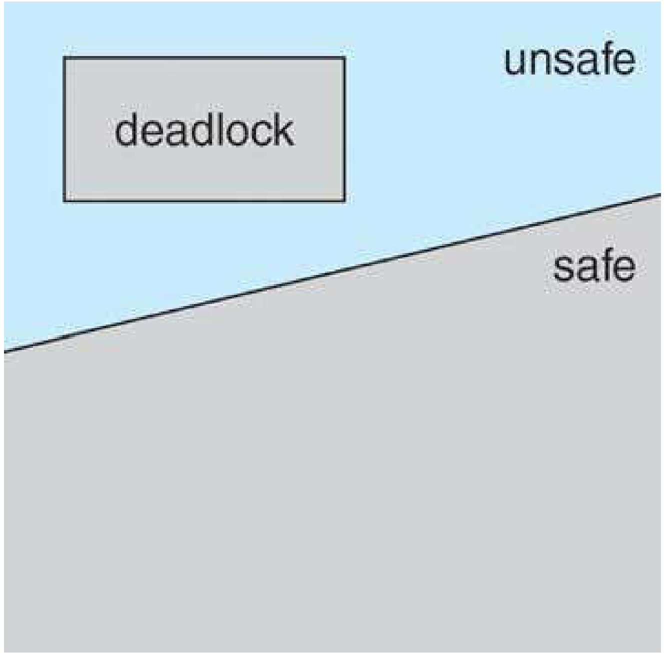
- When a process requests an available resource, system must decide if immediate allocation leaves the system in a safe state.

- System is in safe state if there exists a sequence $<P_1, P_2, \ldots, P_n>$ of ALL the processes is the systems such that for each P_i, the resources that P_i can still request can be satisfied by currently available resources + resources held by all the P_j, with $j < i$.

- That is:
  - If P_i resource needs are not immediately available, then P_i can wait until all P_j have finished.
  - When P_j is finished, P_i can obtain needed resources, execute, return allocated resources, and terminate.
  - When P_i terminates, P_{i+1} can obtain its needed resources, and so on.

Basic Facts

- If a system is in safe state ⇒ no deadlocks.
- If a system is in unsafe state ⇒ possibility of deadlock.
- Avoidance ⇒ ensure that a system will never enter an unsafe state.



Avoidance algorithms

- Single instance of a resource type → resource allocation graph algorithm.
- Multiple instances of a resource type → banker’s algorithm.

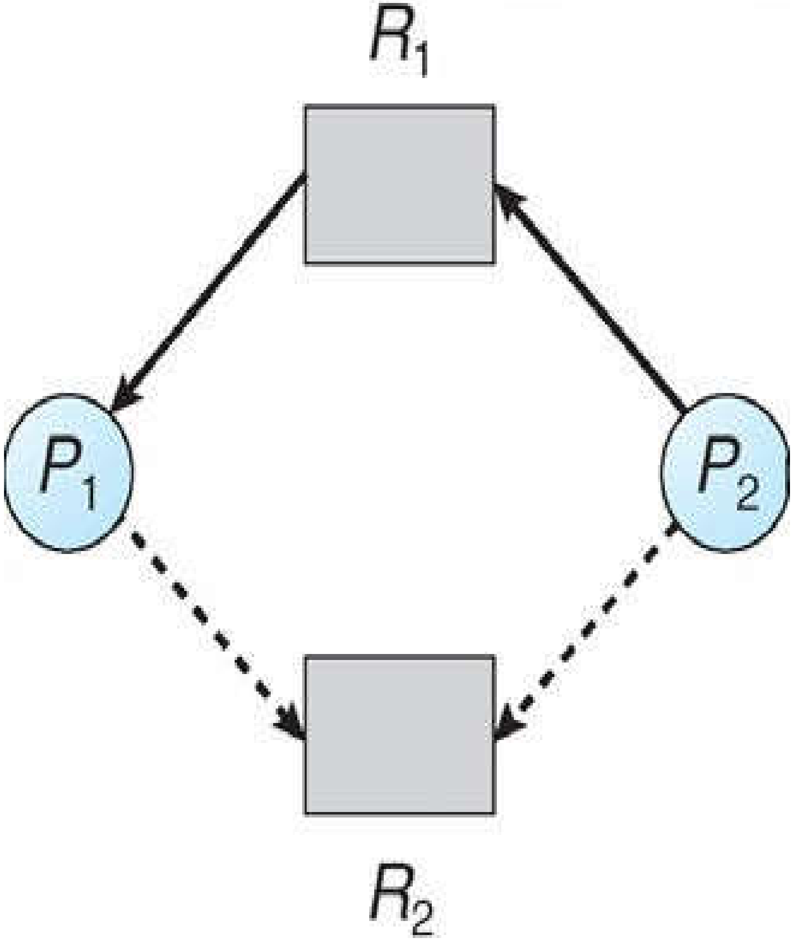
Resource-Allocation Graph Scheme

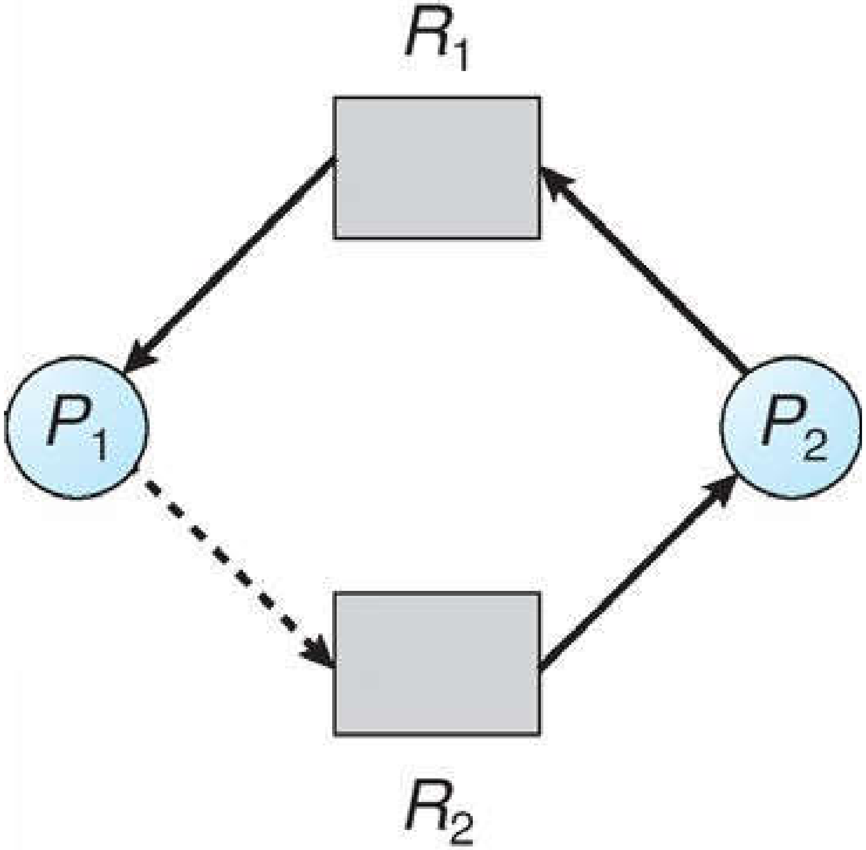
- *Claim edge Pi → Rj ==>process Pj may request* resource *Rj;(-->).*
- Claim edge →request edge when a process

requests a resource.

- Request edge →assignment edge when the resource is allocated to the process.
- When a resource is released by a process, assignment edge reconverts to a claim edge.
- Resources must be claimed *a priori in the system.*

Resource-Allocation Graph





Unsafe state in resource-allocation graph

Resource-Allocation Graph Algorithm

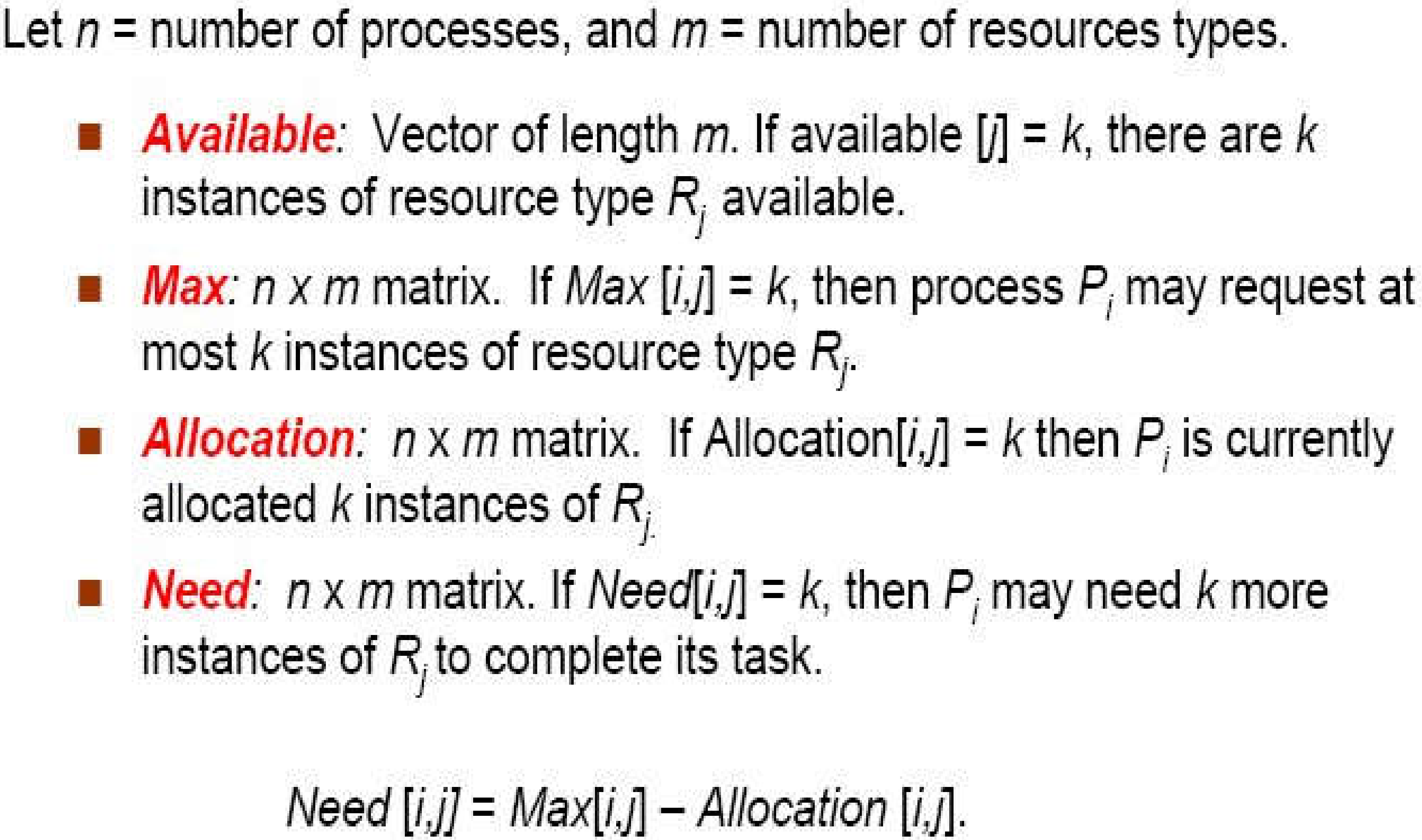
Pi → *Rj*

- The request can be granted only if converting the request edge to an assignment edge does not result in the **formation of a cycle** in the resource allocation graph

Banker’s Algorithm

- Multiple instances.
- Each process must a priori claim maximum use.
- When a process requests a resource it may have to wait.
- When a process gets all its resources it must return them in a finite amount of time

DATA STRUCTURES FOR THE BANKER'S ALGORITHM

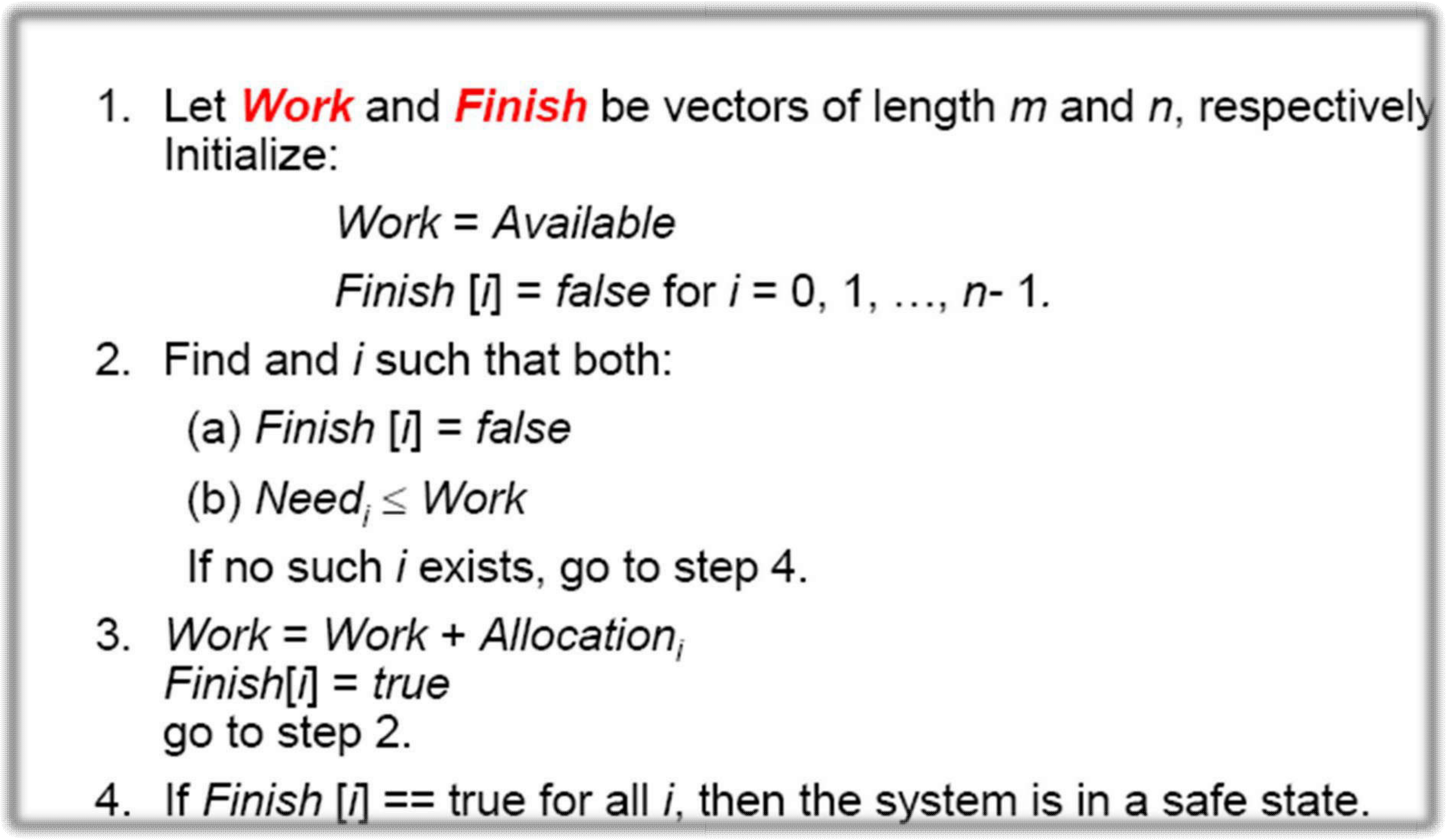


Let n = number of processes, and m = number of resources types.

- *Available*: Vector of length m. If available $[j] = k$, there are k instances of resource type R_j available.
- *Max*: $n \times m$ matrix. If *Max* $[i,j] = k$, then process P_i may request at most k instances of resource type R_j.
- *Allocation*: $n \times m$ matrix. If Allocation$[i,j] = k$ then P_i is currently allocated k instances of R_j.
- *Need*: $n \times m$ matrix. If $Need[i,j] = k$, then P_i may need k more instances of R_j to complete its task.

$$Need\,[i,j] = Max[i,j] - Allocation\,[i,j].$$

SAFETY ALGORITHM



1. Let ***Work*** and ***Finish*** be vectors of length *m* and *n*, respectively Initialize:

   Work = *Available*

   Finish [*i*] = *false* for *i* = 0, 1, …, *n*- 1.

2. Find and *i* such that both:

   (a) *Finish* [*i*] = *false*

   (b) $Need_i \leq Work$

   If no such *i* exists, go to step 4.

3. $Work = Work + Allocation_i$
   Finish[*i*] = *true*
   go to step 2.

4. If *Finish* [*i*] == true for all *i*, then the system is in a safe state.

Resource-Request Algorithm for Process *Pi*

- *If Request$_i$ [j] = k then* process *Pi wants k instances of resource type Rj.*

1. If *Request$_i$ ≤ Need$_i$ go to step 2. else, raise error* condition, since process has exceeded its maximum claim.

2. If *Request$_i$ ≤ Available, go to step 3. Otherwise Pi must* wait, since resources are not available.

3. Pretend to allocate requested resources to *Pi by modifying* the state as follows:

Resource-Request Algorithm for Process *Pi*

$$Available = Available – Request;$$

$$Allocation_i = Allocation_i + Request_i;$$

$$Need_i = Need_i – Request_i;$$

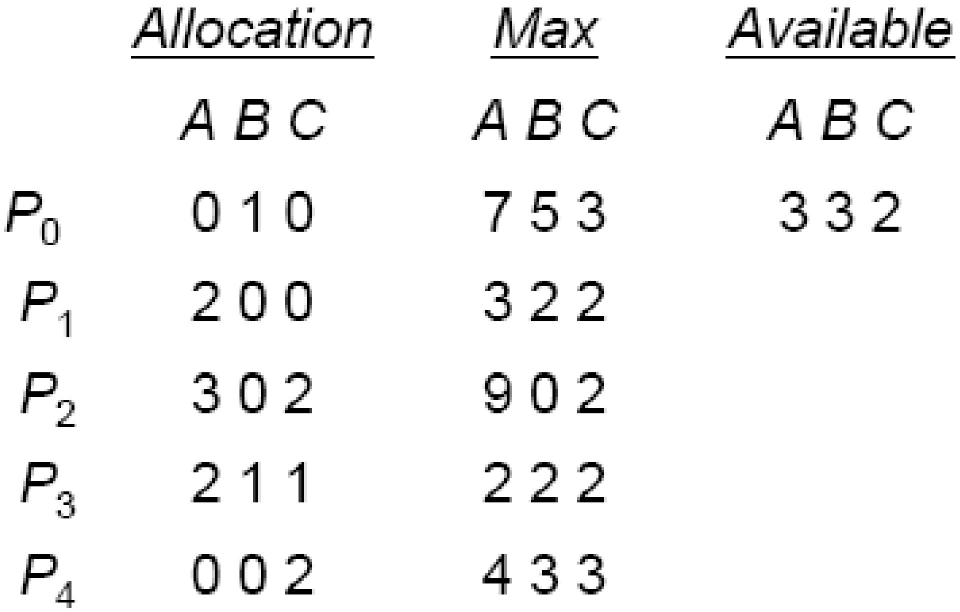
- *If safe ==> resources are allocated to Pi.*
- *If unsafe==> Pi must wait, and the old resource-allocation state is restored*

Example of Banker’s Algorithm

5 processes *P0 … P4;*

3 resource types: *A (10 instances), B (5), and C (7)*

Snapshot at time *T0:*



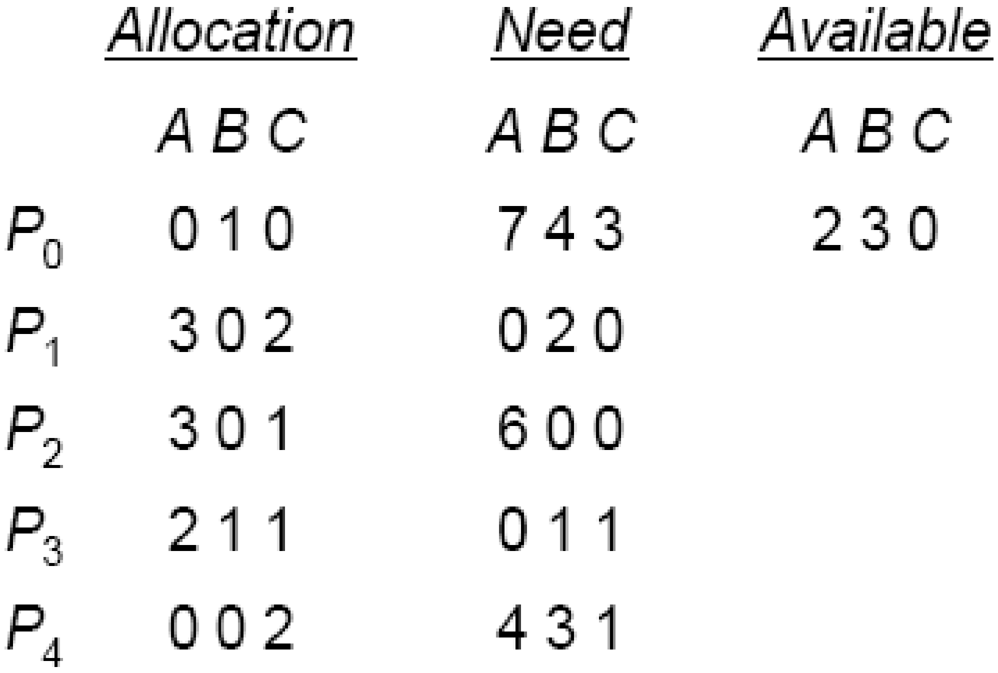
	Allocation	*Max*	*Available*
	A B C	*A B C*	*A B C*
P_0	0 1 0	7 5 3	3 3 2
P_1	2 0 0	3 2 2	
P_2	3 0 2	9 0 2	
P_3	2 1 1	2 2 2	
P_4	0 0 2	4 3 3	

System is in safe state -- sequence < *P1, P3, P4, P2, P0*>

	Need		
	A	*B*	*C*
P_0	7	4	3
P_1	1	2	2
P_2	6	0	0
P_3	0	1	1
P_4	4	3	1

Example: *P1 Request (1,0,2)*

- Check that Request ≤ Available (that is, (1,0,2) ≤ (3,3,2) ==> true.



	Allocation	Need	Available
	A B C	A B C	A B C
P_0	0 1 0	7 4 3	2 3 0
P_1	3 0 2	0 2 0	
P_2	3 0 1	6 0 0	
P_3	2 1 1	0 1 1	
P_4	0 0 2	4 3 1	

< *P1, P3, P4, P0, P2>* satisfies safety requirement.

Can request for (3,3,0) by *P4 be granted?*

Can request for (0,2,0) by *P0 be granted?*

Deadlock Detection

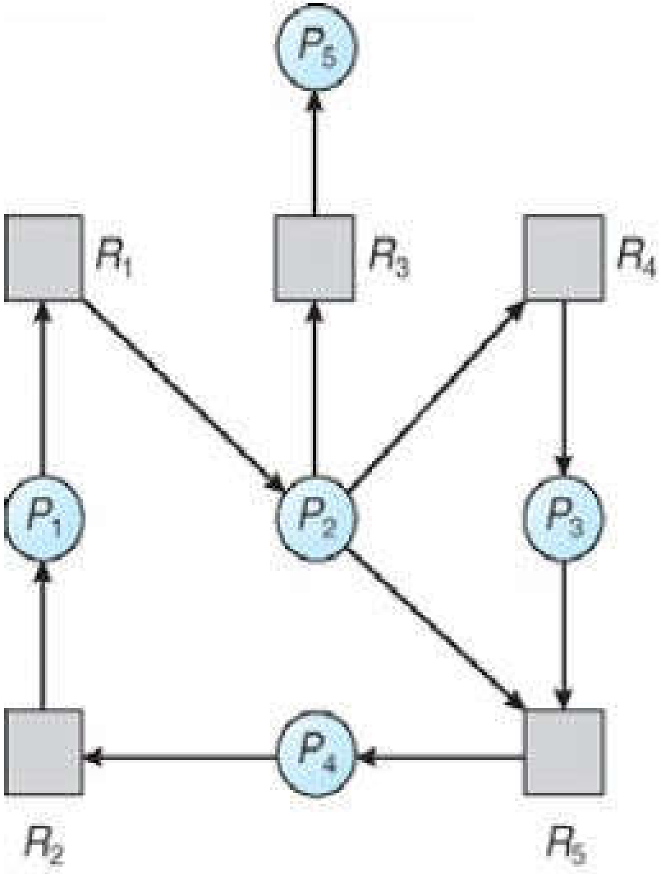
- Allow system to enter deadlock state
- Detection algorithm
- Recovery scheme

❑Single instance
❑Several instances

Single Instance of Each Resource Type

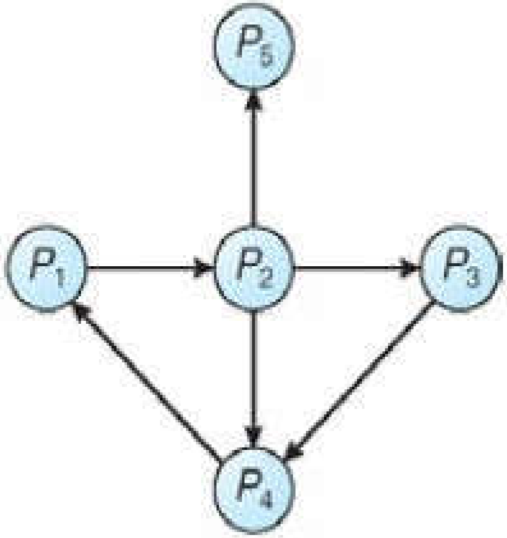
- Maintain *wait-for graph*
  - Nodes are processes.
  - *Pi → Pj if Pi is waiting for Pj.*
- Periodically invoke an algorithm that searches for a cycle in the graph. If there is a cycle, there exists a deadlock.

Resource-Allocation Graph and Wait-for Graph



(a)

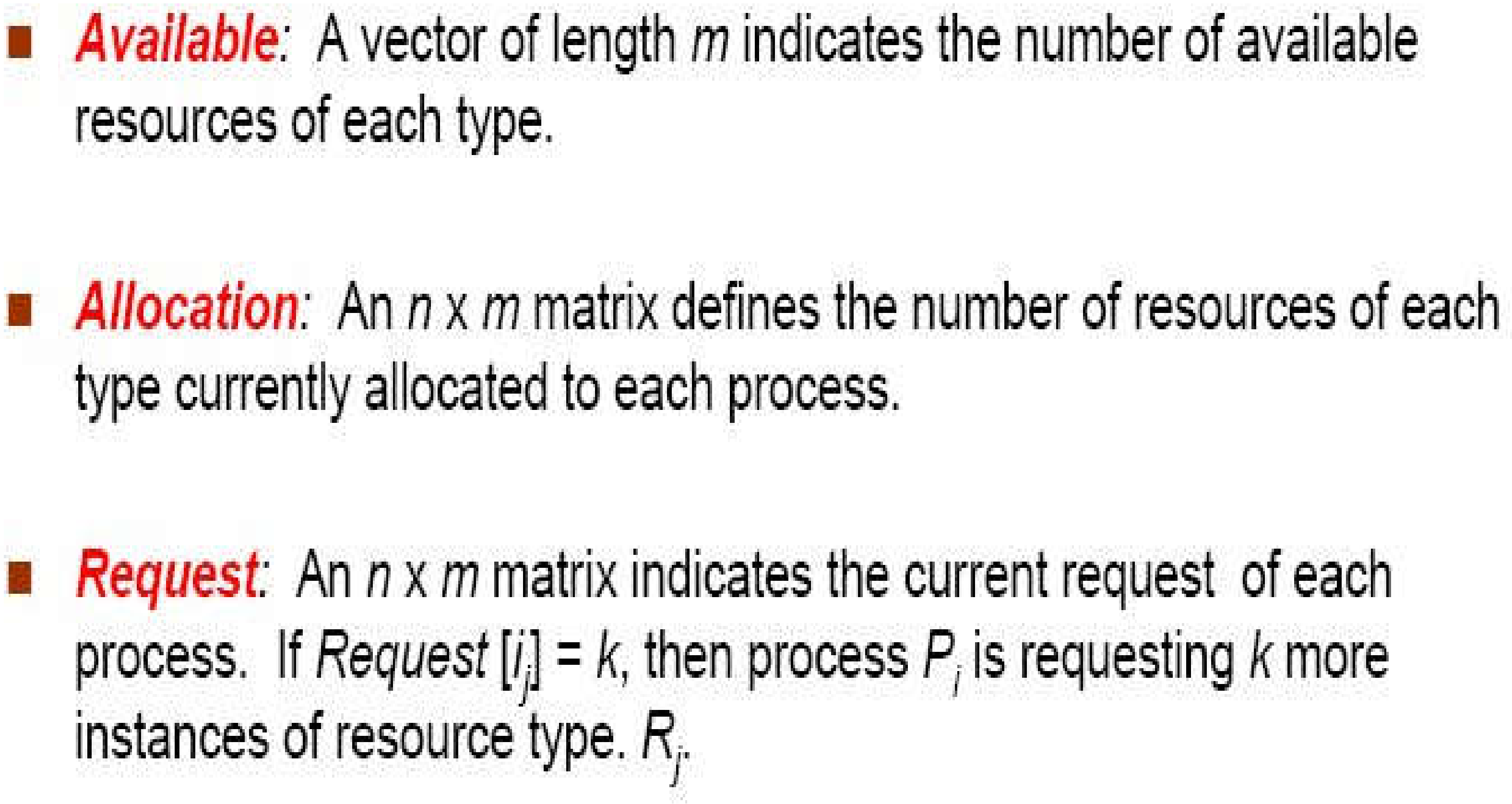
Resource-Allocation Graph



(b)

Corresponding wait-for graph

Several Instance of Resource Type



- *Available*: A vector of length *m* indicates the number of available resources of each type.

- *Allocation*: An *n* x *m* matrix defines the number of resources of each type currently allocated to each process.

- *Request*: An *n* x *m* matrix indicates the current request of each process. If *Request* $[i_j] = k$, then process P_i is requesting *k* more instances of resource type. R_j.

1. Let *Work* and *Finish* be vectors of length *m* and *n*, respectively Initialize

   (a) *Work* = *Available*

   (b) For $i = 1,2, \ldots, n$, if $Allocation_i \neq 0$, then *Finish*[i] = false;otherwise, *Finish*[i] = *true*.

2. Find an index *i* such that both:

   (a) *Finish*[*i*] == *false*

   (b) $Request_i \leq Work$

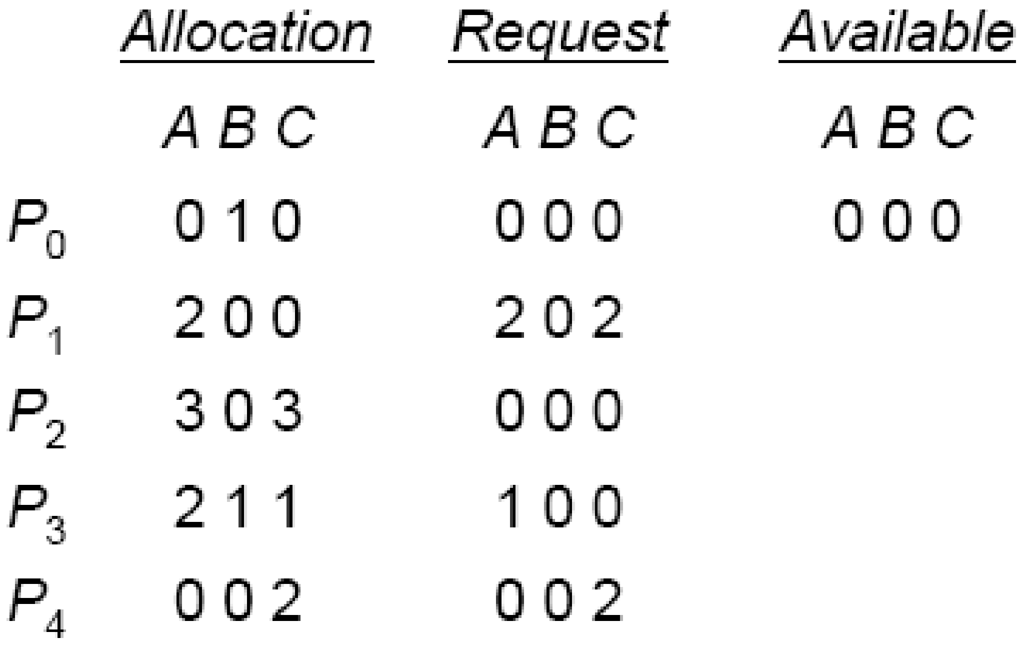
   If no such *i* exists, go to step 4.

3. $Work = Work + Allocation_i$
   Finish[*i*] = *true*
   go to step 2.

4. If *Finish*[*i*] == false, for some *i*, $1 \leq i \leq n$, then the system is in deadlock state. Moreover, if *Finish*[*i*] == *false*, then P_i is deadlocked.

Example of Detection Algorithm

- 5 processes *P0 … P4;*
- *3 resource types:* A (7 instances), *B (2), and C (6)*
- Snapshot at time *T0:*



	Allocation	Request	Available
	A B C	A B C	A B C
P_0	0 1 0	0 0 0	0 0 0
P_1	2 0 0	2 0 2	
P_2	3 0 3	0 0 0	
P_3	2 1 1	1 0 0	
P_4	0 0 2	0 0 2	

Sequence *<P0, P2, P3, P1, P4>*

Example (Cont.)

- P_2 requests an additional instance of type *C*.

	Request		
	A	*B*	*C*
P_0	0	0	0
P_1	2	0	1
P_2	0	0	1
P_3	1	0	0
P_4	0	0	2

- State of system?
  - Can reclaim resources held by process P_0, but insufficient resources to fulfill other processes; requests.
  - Deadlock exists, consisting of processes P_1, P_2, P_3, and P_4.

Detection-Algorithm Usage

- When, and how often, to invoke depends on:
  - How often a deadlock is likely to occur?
  - How many processes will need to be rolled back?
    - one for each disjoint cycle
- If detection algorithm is invoked arbitrarily, there may be many cycles in the resource graph and so we would not be able to tell which of the many deadlocked processes "caused" the deadlock.

Recovery from Deadlock

1. Process Termination
2. Resource Preemption

Process Termination

- **Abort all deadlocked processes.**
- **Abort one process at a time until the deadlock cycle is eliminated.**
- *In which order should we choose to abort?*
  - what the priority of the process is
  - How long process has computed, and how much longer to completion.
  - how many & what type of resources the process has used
  - How many more resources process needs to complete
  - How many processes will need to be terminated
  - Is process interactive or batch?

Resource Preemption

- Selecting a victim – minimize cost.
- Rollback – return to some safe state, restart process for that state.
- Starvation – same process may always be picked as victim, include number of rollback in cost factor.

Unit – 5

File-System Interface

File-System Interface

- File Concept
- Access Methods
- Directory Structure
- File-System Mounting
- File Sharing
- Protection

- **File System Implementation-**
- File system structure
- File system implementation
- Directory implementation
- Allocation methods
- Free-space management
- Efficiency and performance

File Concept

- File : named collection of related information that is recorded on secondary storage.
- Contiguous logical address space
- Types:
  - Data
    - numeric
    - character
    - binary
  - Program

File Attributes

- **Name** – only information kept in human-readable form
- **Identifier** – unique tag (number) identifies file within file system
- **Type** – needed for systems that support different types
- **Location** – pointer to file location on device
- **Size** – current file size
- **Protection** – controls who can do reading, writing, executing
- **Time, date, and user identification** – data for protection, security, and usage monitoring
- Information about files are kept in the directory structure, which is maintained on the disk

File Operations

- File is an **abstract data type**
  - **Create** → space in file system, directory entry
  - **Write** → system call (name, inf.), write pointer
  - **Read** → system call, read pointer
  - **Reposition within file (file seek)**
  - **Delete**
  - **Truncate**
- ***Open(F_i)*** – search the directory structure on disk for entry F_i, and move the content of entry to memory (open-file table)
- ***Close (F_i)*** – move the content of entry F_i in memory to directory structure on disk

Current file-position pointer

File Operations (Cont…)

- OS uses two levels of internal tables:

1. **Per-process table:**
   - Keeps track of all files that a process has open
   - Each entry in per-process table points to a system-wide table
2. **System-wide table:**
   - Contains process-independent information ex: file size, access dates

Open Files

- Several pieces of data are needed to manage open files:
  - **File pointer:** pointer to last read/write location, per process that has the file open
  - **File-open count:**
    - keeps track of number of times a file is open/close
    - to allow removal of data from open-file table when last processes closes it
  - **Disk location of the file:** cache of data access information
  - **Access rights:** per-process table stores this mode information (access)

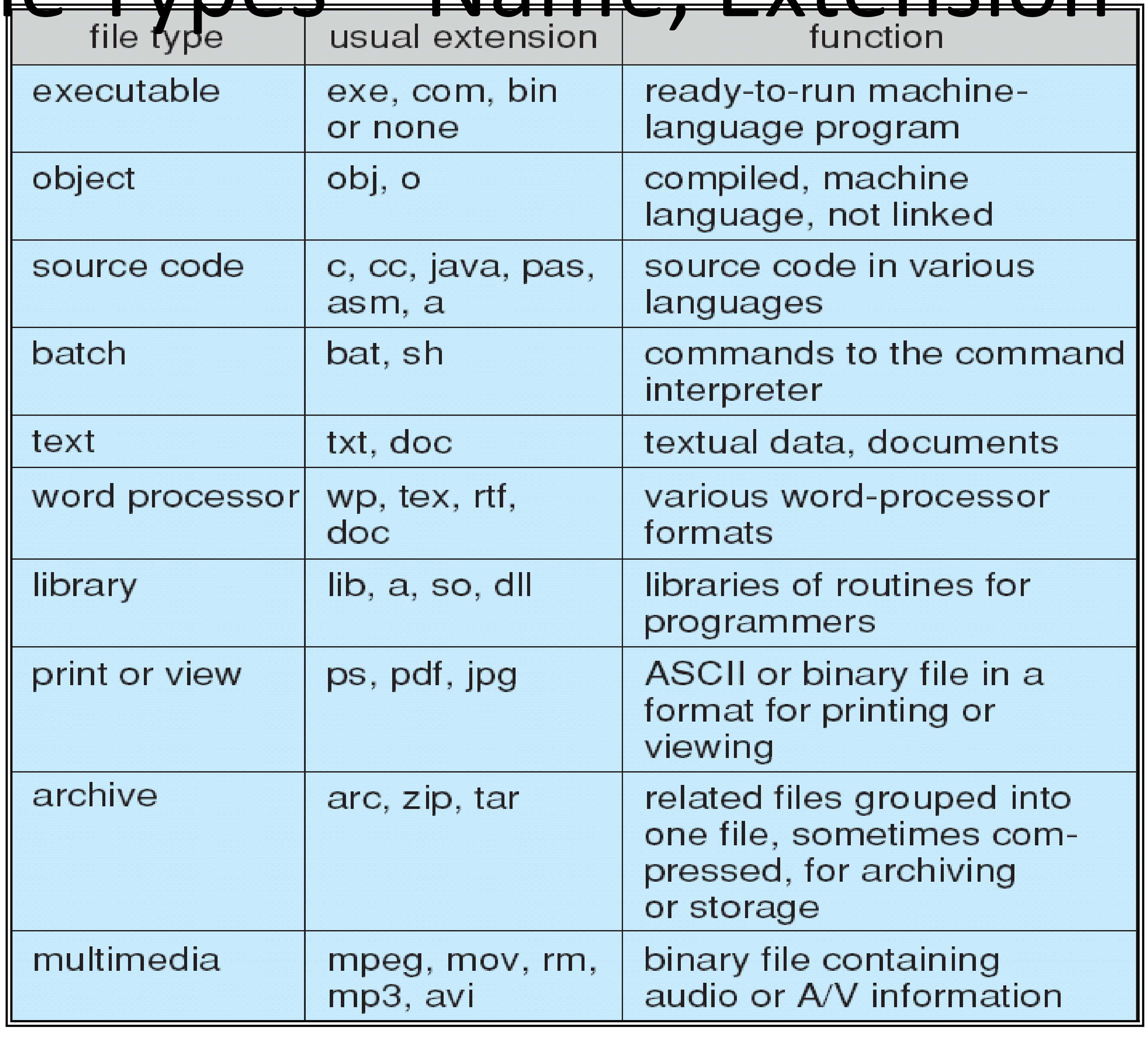
File Locking

- Provided by some operating systems and file systems
- File locks allow one process to lock a file and prevent other processes from gaining access to it

1. **Shared lock:** several processes can acquire lock concurrently (reader lock)
2. **Exclusive lock:** only one process can acquire such lock at a time (writer lock)

- **File locking mechanisms:**
  - **Mandatory** – access is denied depending on locks held and requested
  - **Advisory** – processes can find status of locks and decide what to do

File Types – Name, Extension



file type	usual extension	function
executable	exe, com, bin or none	ready-to-run machine-language program
object	obj, o	compiled, machine language, not linked
source code	c, cc, java, pas, asm, a	source code in various languages
batch	bat, sh	commands to the command interpreter
text	txt, doc	textual data, documents
word processor	wp, tex, rtf, doc	various word-processor formats
library	lib, a, so, dll	libraries of routines for programmers
print or view	ps, pdf, jpg	ASCII or binary file in a format for printing or viewing
archive	arc, zip, tar	related files grouped into one file, sometimes compressed, for archiving or storage
multimedia	mpeg, mov, rm, mp3, avi	binary file containing audio or A/V information

File Structure

- OS → multiple file structures
  - disadvantage→ code to support them
- Packing a no. of logical records into physical blocks.
  - User's application program
  - OS
- All files suffer from internal fragmentation
  - Larger the block size, greater the internal fragmentation

Access Methods

1. **Sequential Access**

   read next

   write next

   reset

   no read after last write

   (rewrite)

   Ex: editors, compilers

2. **Direct Access (**file operations → include block no. as parameter'n'**)**

   read *n*

   write *n*

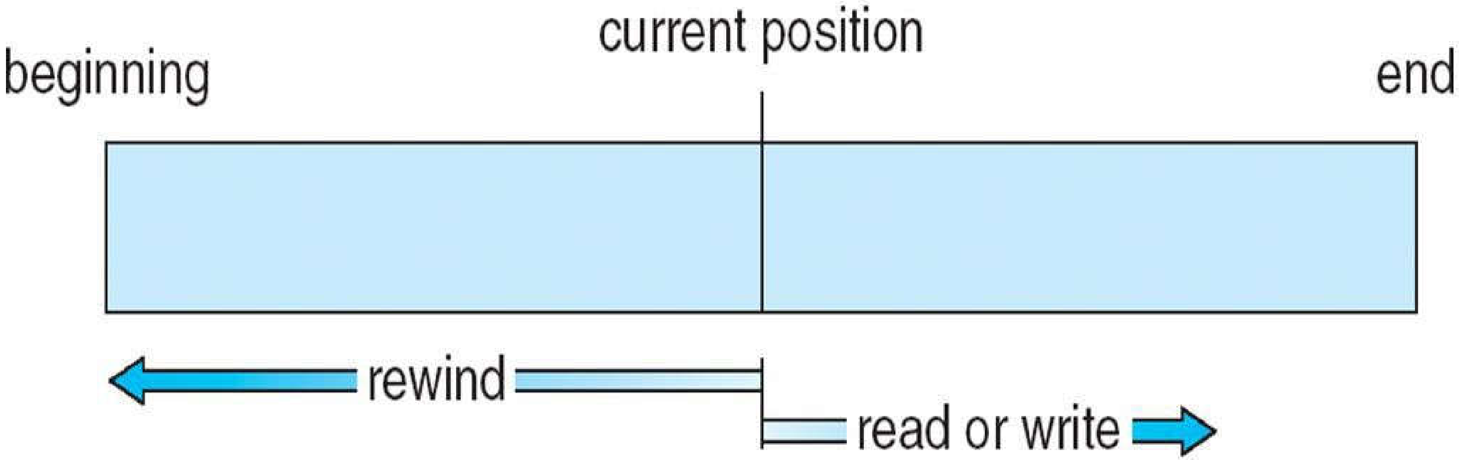
   position to *n*

   read next

   write next

   rewrite *n*

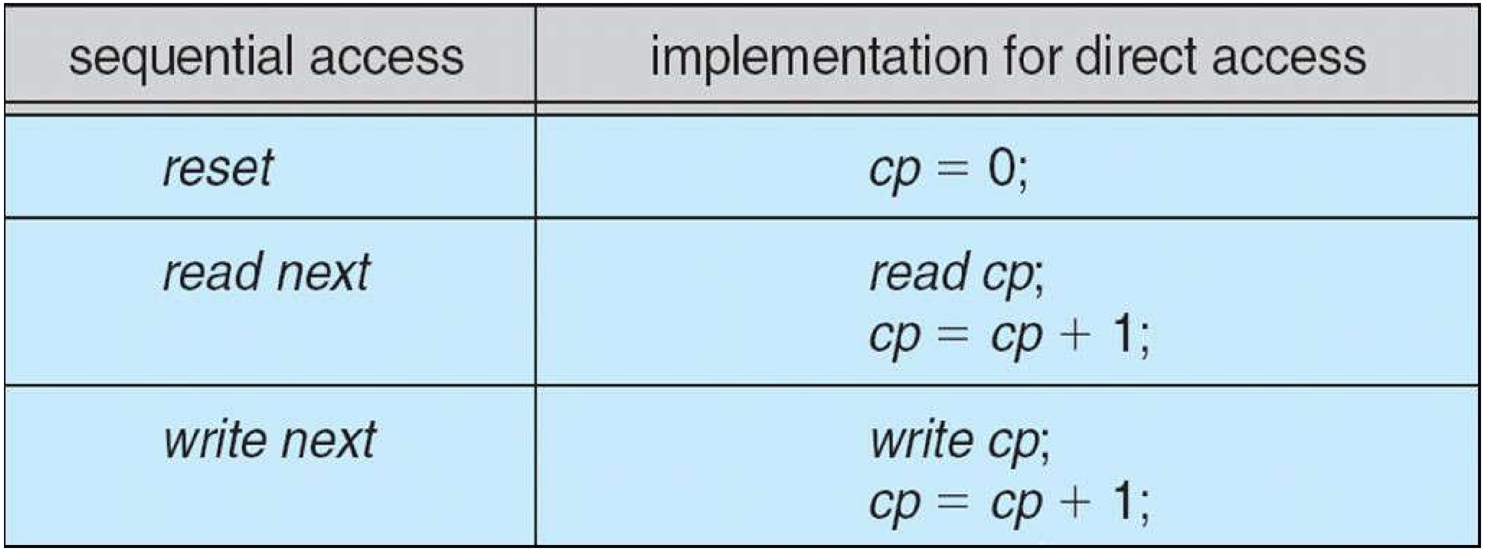
Sequential-access File



Direct access (or relative access)

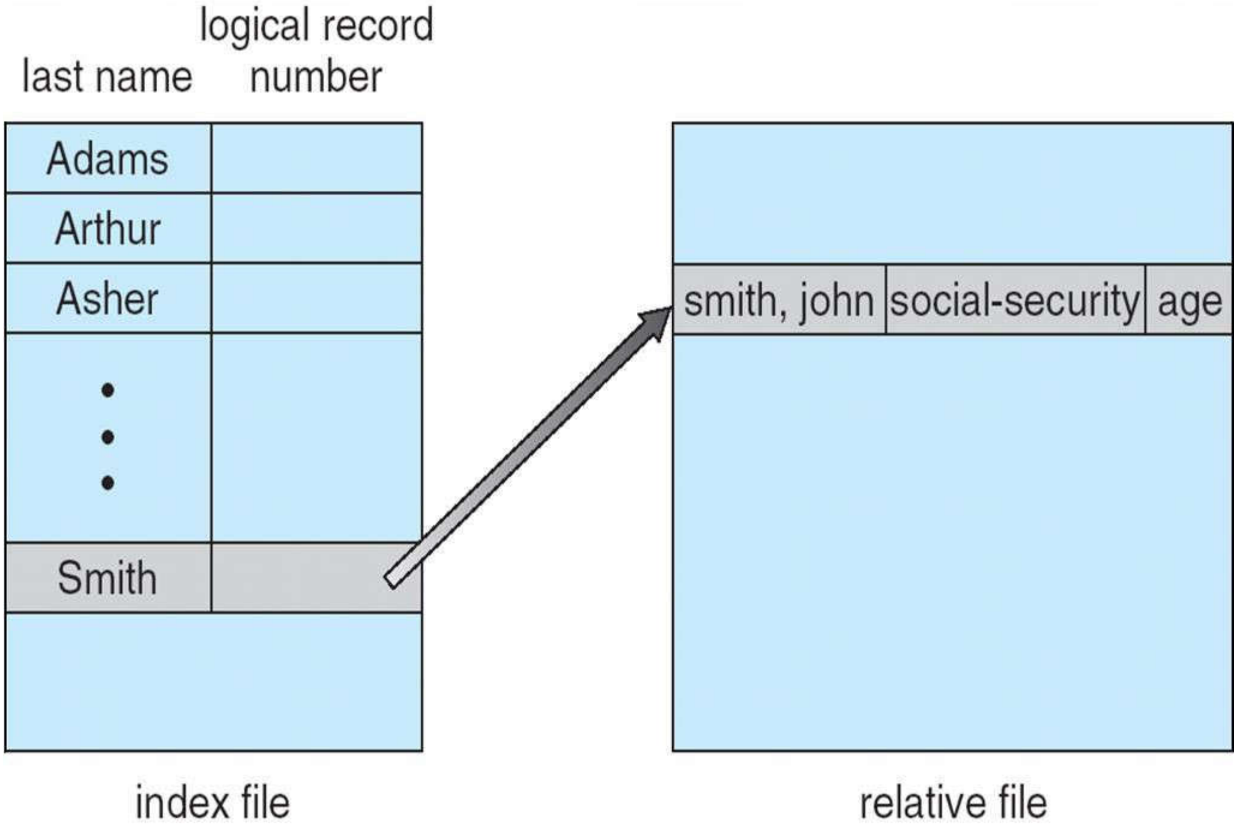
- File is viewed as a numbered sequence of blocks or records
  - Ex: read block 14 then read block 53 and then write block 7
- No restrictions on the ordering of reading or writing
- Great use for immediate access to large amounts of information

Simulation of Sequential Access on Direct-access File



sequential access	implementation for direct access
reset	*cp* = 0;
read next	*read cp*; *cp* = *cp* + 1;
write next	*write cp*; *cp* = *cp* + 1;

Example of Index and Relative Files



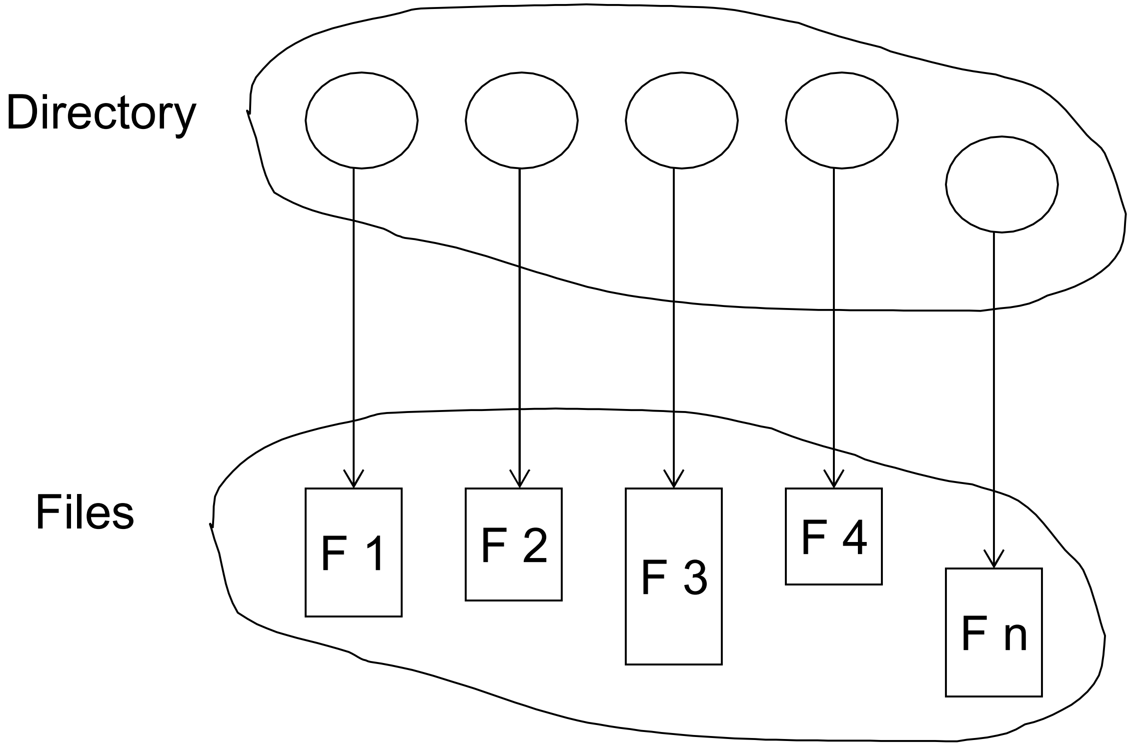
Index: contains pointers to the various blocks

Large files➔ large index

Sol: create an index for index file

Directory Structure

- A collection of nodes containing information about all files

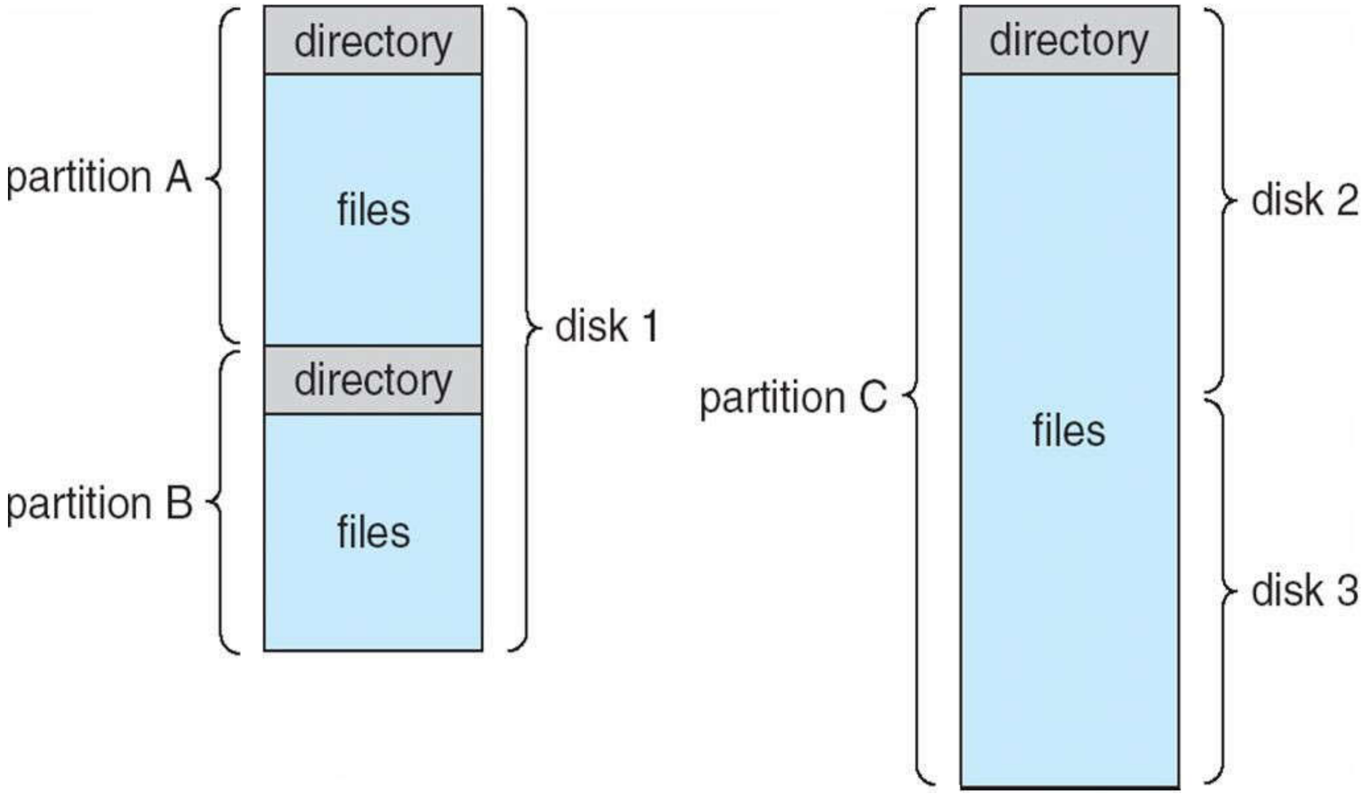


Both the directory structure and the files reside on disk
Backups of these two structures are kept on tapes

Disk Structure

- Disk can be subdivided into partitions
- Disks or partitions can be RAID protected against failure
- Disk or partition can be used raw – without a file system, or formatted with a file system
- Partitions also known as minidisks, slices
- Entity containing file system known as a volume
- Each volume containing file system also tracks that file system's info in device directory or volume table of contents
- As well as general-purpose file systems there are many special-purpose file systems, frequently all within the same operating system or computer

A Typical File-system Organization



Operations Performed on Directory

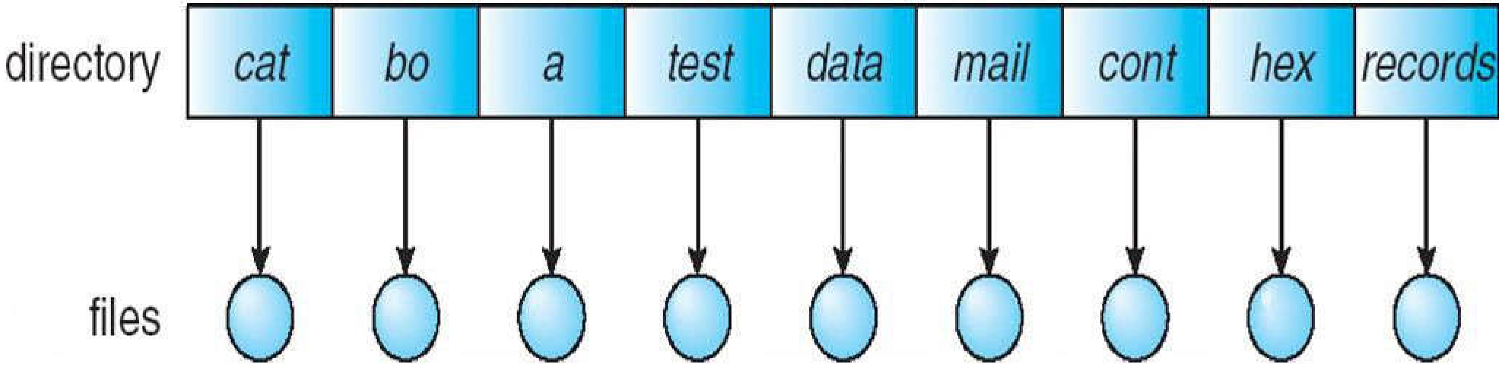
- Search for a file
- Create a file
- Delete a file
- List a directory
- Rename a file
- Traverse the file system

Organize the Directory (Logically) to Obtain

- Efficiency – locating a file quickly
- Naming – convenient to users
  - Two users can have same name for different files
  - The same file can have several different names
- Grouping – logical grouping of files by properties, (e.g., all Java programs, all games, …)

Single-Level Directory

- A single directory for all users

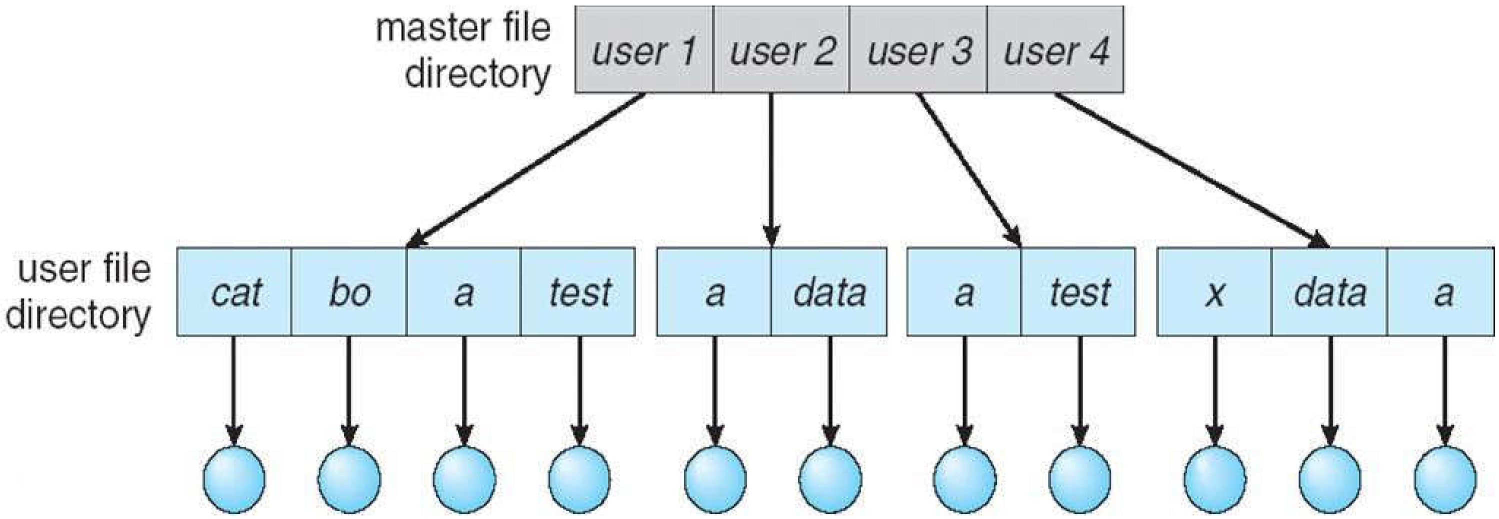


Naming problem

Grouping problem

Two-Level Directory

- Separate directory for each user



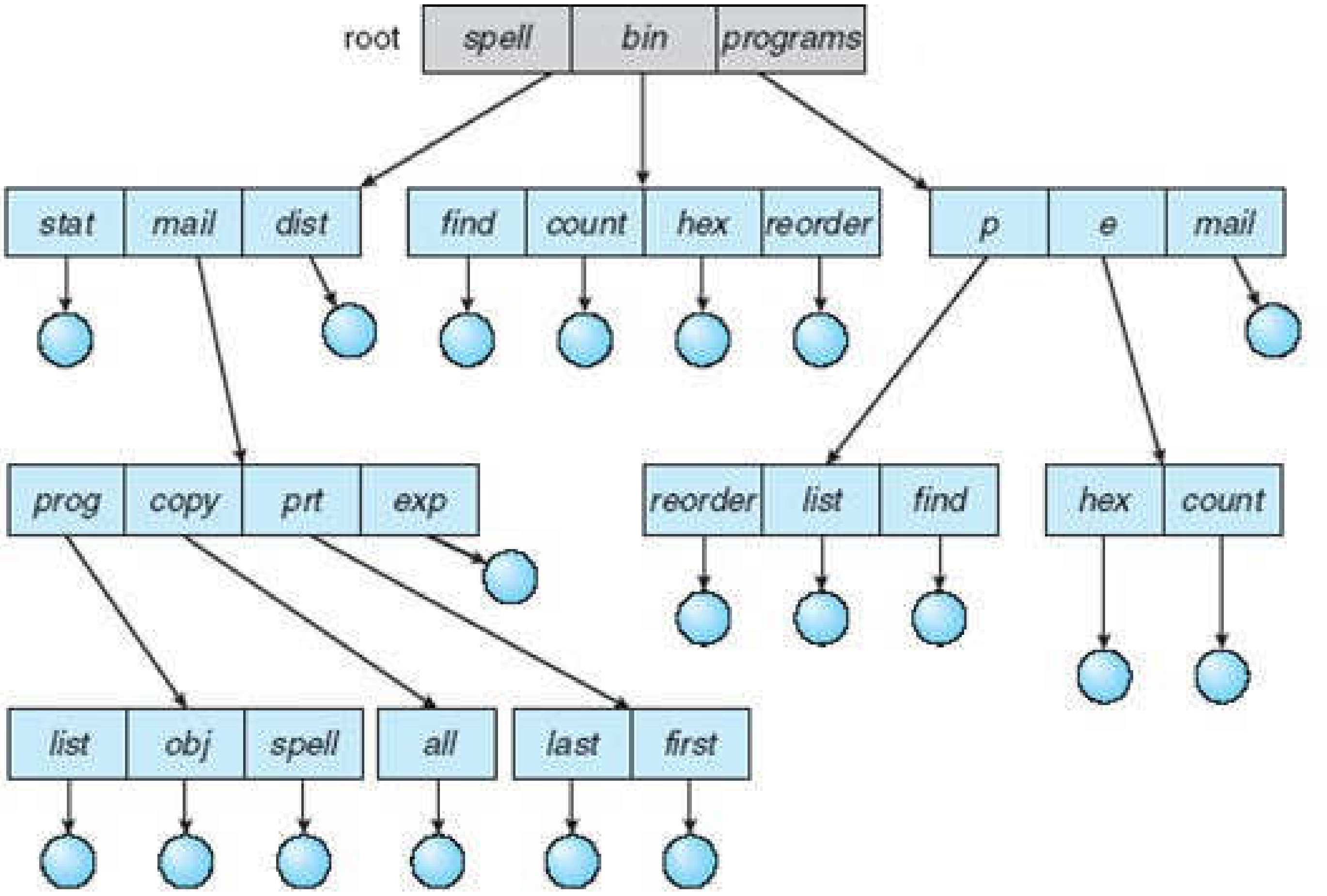
□ Path name

□ Can have the same file name for different user

□ Efficient searching

□ Isolation of one user from another

Tree-Structured Directories



Tree-Structured Directories (Cont)

- Directory entry: File – 0; subdirectory - 1
- Efficient searching
- Current directory (working directory)
  - cd /spell/mail/prog
  - type list
- **Absolute** or **relative** path name
- Absolute: begins at root & follows a path down to specified file
- Relative: defines a path from the current directory

Tree-Structured Directories (Cont)

- **Absolute** or **relative** path name
- Creating a new file is done in current directory
- Delete a file(directory empty?)

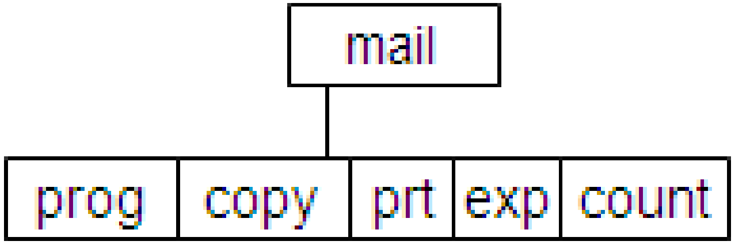
rm <file-name>

- Creating a new subdirectory is done in current directory

mkdir <dir-name>

Example: if in current directory /mail

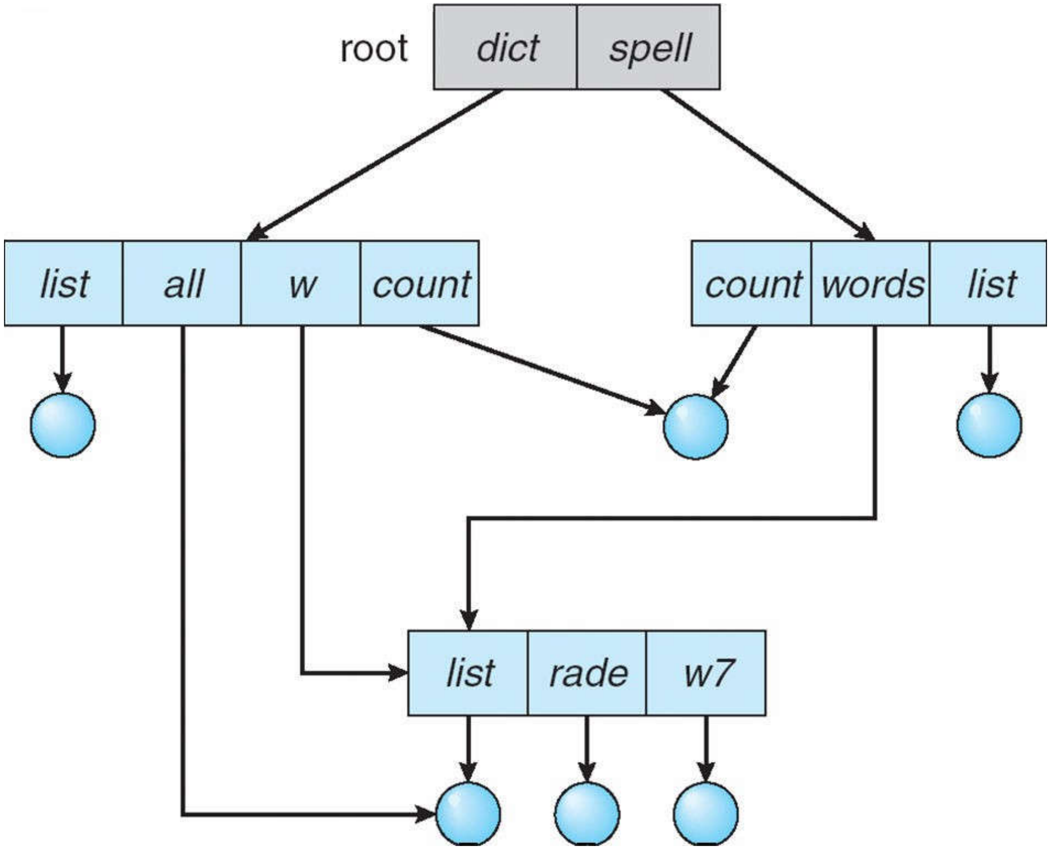
mkdir count



Deleting “mail” ⇒ deleting the entire subtree rooted by “mail”

Acyclic-Graph Directories

- Have shared subdirectories and files



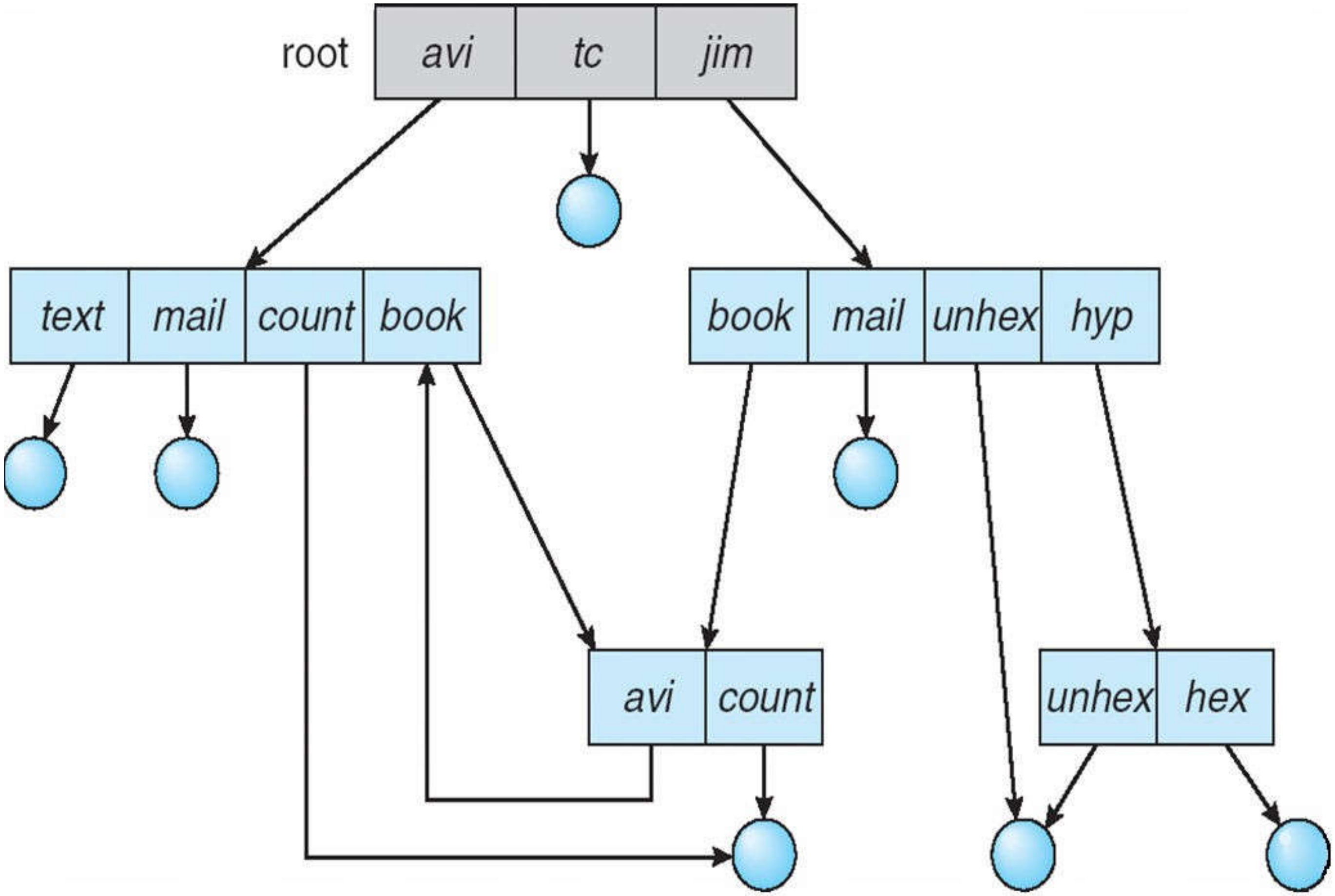
Acyclic-Graph Directories (Cont.)

- Two different names (aliasing)

- If *dict* deletes *list* ⇒ dangling pointer

  Solutions:
  - Backpointers, so we can delete all pointers
    Variable size records a problem
  - Backpointers using a daisy chain organization
  - Entry-hold-count solution
- New directory entry type
  - **Link** – another name (pointer) to an existing file
  - **Resolve the link** – follow pointer to locate the file

General Graph Directory



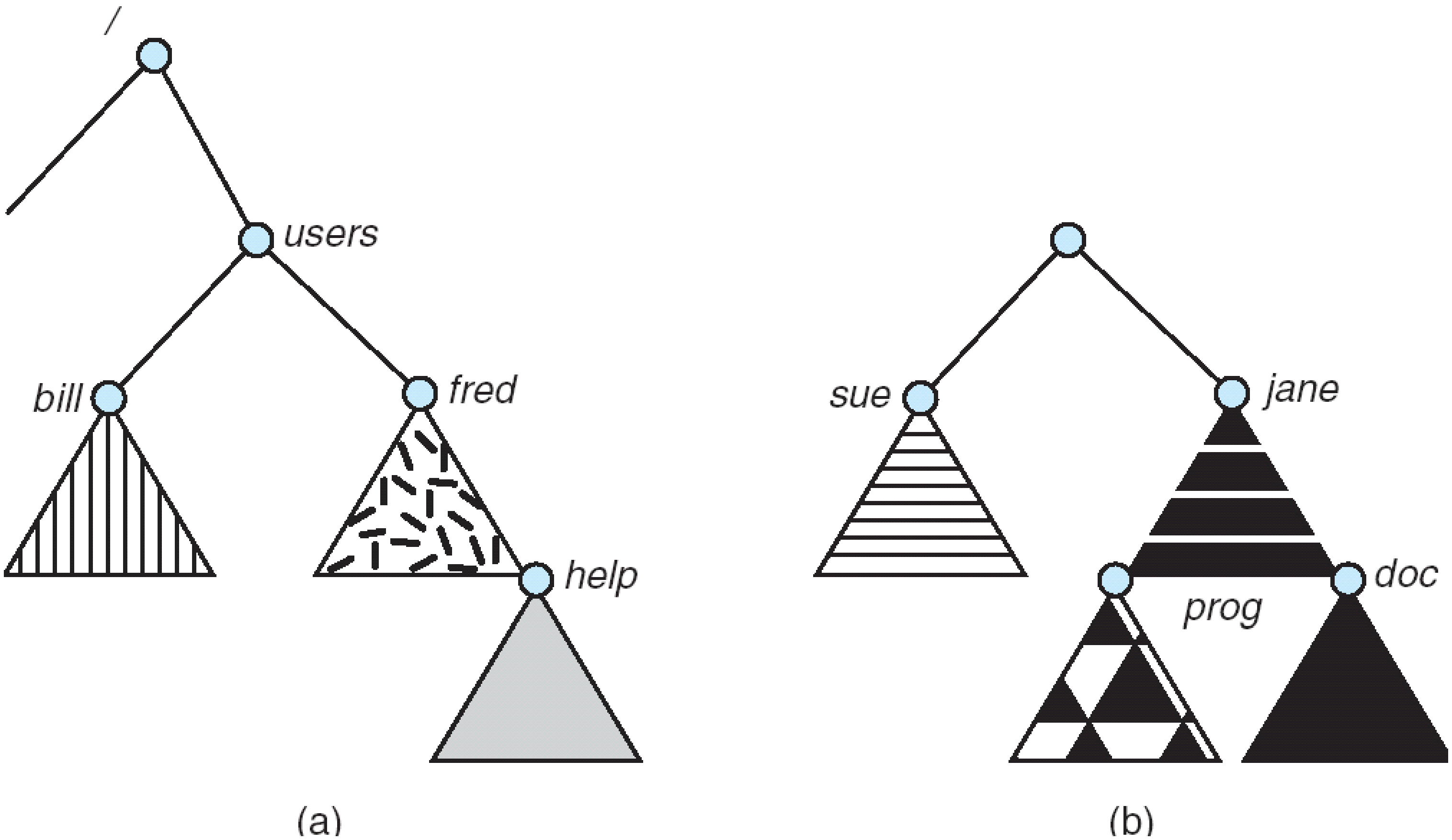
General Graph Directory (Cont.)

- How do we guarantee no cycles?
  - Allow only links to file not subdirectories
  - Garbage collection
  - Every time a new link is added use a cycle detection algorithm to determine whether it is OK

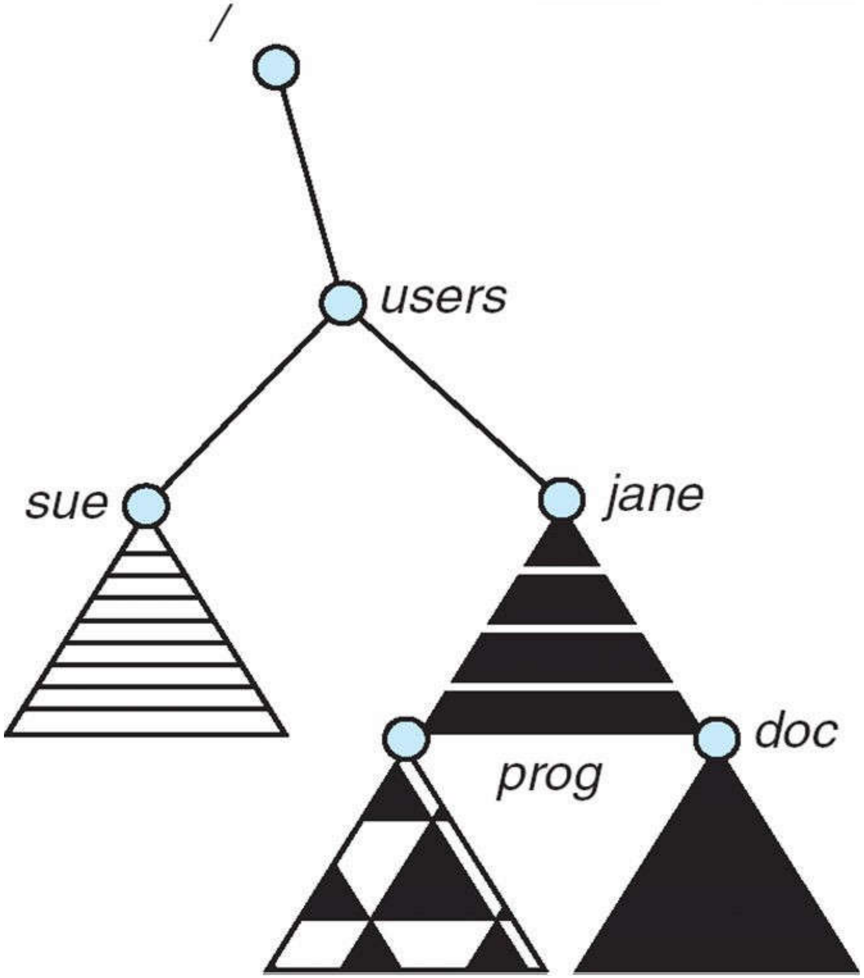
File System Mounting

- A file system must be **mounted** before it can be accessed
- A unmounted file system (i.e.(b)) is mounted at a **mount point**
- **Mount point:** location within file structure where the file system is to be attached.

(a) Existing. (b) Unmounted Partition



Mount Point



File Sharing

- Sharing of files on multi-user systems is desirable
- Sharing may be done through a **protection** scheme
- On distributed systems, files may be shared across a network
- Network File System (NFS) is a common distributed file-sharing method

File Sharing – Multiple Users

- **User IDs** identify users, allowing permissions and protections to be per-user

- **Group IDs** allow users to be in groups, permitting group access rights

File Sharing – Remote File Systems

- Uses networking to allow file system access between systems
  - Manually via programs like FTP
  - Automatically, seamlessly using **distributed file systems**
  - Semi automatically via the **world wide web**
- **Client-server** model allows clients to mount remote file systems from servers
  - Server can serve multiple clients
  - Client and user-on-client identification is insecure or complicated
  - **NFS** is standard UNIX client-server file sharing protocol
  - **CIFS** is standard Windows protocol
  - Standard operating system file calls are translated into remote calls
- **Distributed Information Systems (distributed naming services)** such as LDAP, DNS, NIS, Active Directory implement unified access to information needed for remote computing

File Sharing – Failure Modes

- Remote file systems add new failure modes, due to network failure, server failure
- Recovery from failure can involve state information about status of each remote request
- Stateless protocols such as NFS include all information in each request, allowing easy recovery but less security

File Sharing – Consistency Semantics

- **Consistency semantics** specify how multiple users are to access a shared file simultaneously
  - Similar to process synchronization algorithms
    - Tend to be less complex due to disk I/O and network latency (for remote file systems
  - Andrew File System (AFS) implemented complex remote file sharing semantics
  - Unix file system (UFS) implements:
    - Writes to an open file visible immediately to other users of the same open file
    - Sharing file pointer to allow multiple users to read and write concurrently
  - AFS has session semantics
    - Writes only visible to sessions starting after the file is closed

Protection

- File owner/creator should be able to control:
  - what can be done
  - by whom

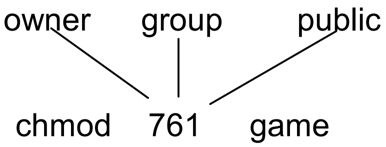
- Types of access
  - **Read**
  - **Write**
  - **Execute**
  - **Append**
  - **Delete**
  - **List**

Access Lists and Groups

- Mode of access: read, write, execute
- Three classes of users

			RWX
a) **owner access**	7	⇒	1 1 1
			RWX
b) **group access**	6	⇒	1 1 0
			RWX
c) **public access**	1	⇒	0 0 1

- Ask manager to create a group (unique name), say G, and add some users to the group.
- For a particular file (say *game*) or subdirectory, define an appropriate access.



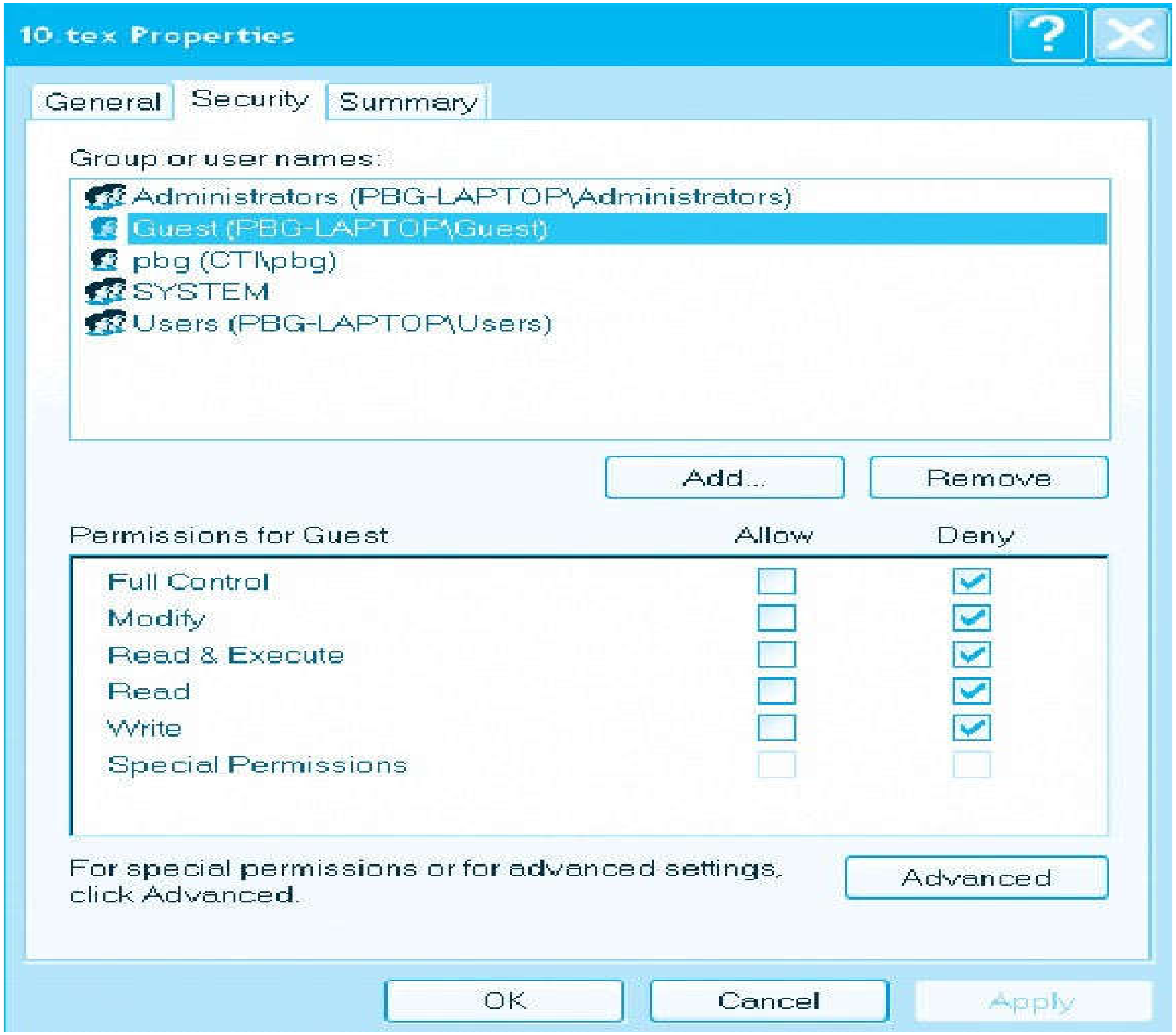
owner group public

chmod 761 game

Attach a group to a file

chgrp G game

Windows XP Access-control List Management



Group or user names:

- Administrators (PBG-LAPTOP\Administrators)
- Guest (PBG-LAPTOP\Guest)
- pbg (CTI\pbg)
- SYSTEM
- Users (PBG-LAPTOP\Users)

Permissions for Guest	Allow	Deny
Full Control	□	☑
Modify	□	☑
Read & Execute	□	☑
Read	□	☑
Write	□	☑
Special Permissions	□	□

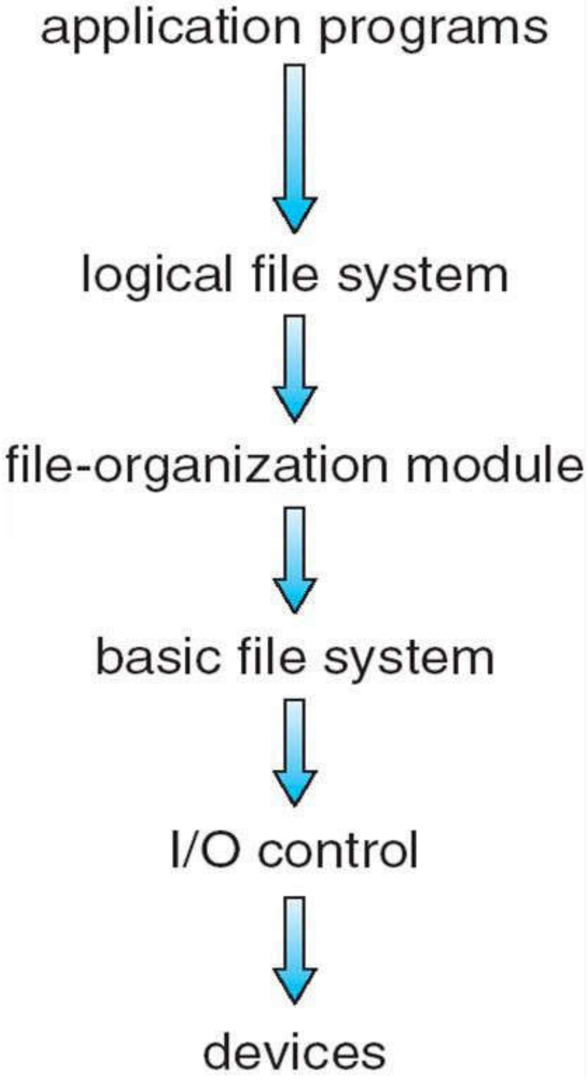
For special permissions or for advanced settings, click Advanced.

File System Implementation

File-System Structure

- File structure
  - Logical storage unit
  - Collection of related information
- File system resides on secondary storage (disks)
- File system organized into layers
- **File control block** – storage structure consisting of information about a file

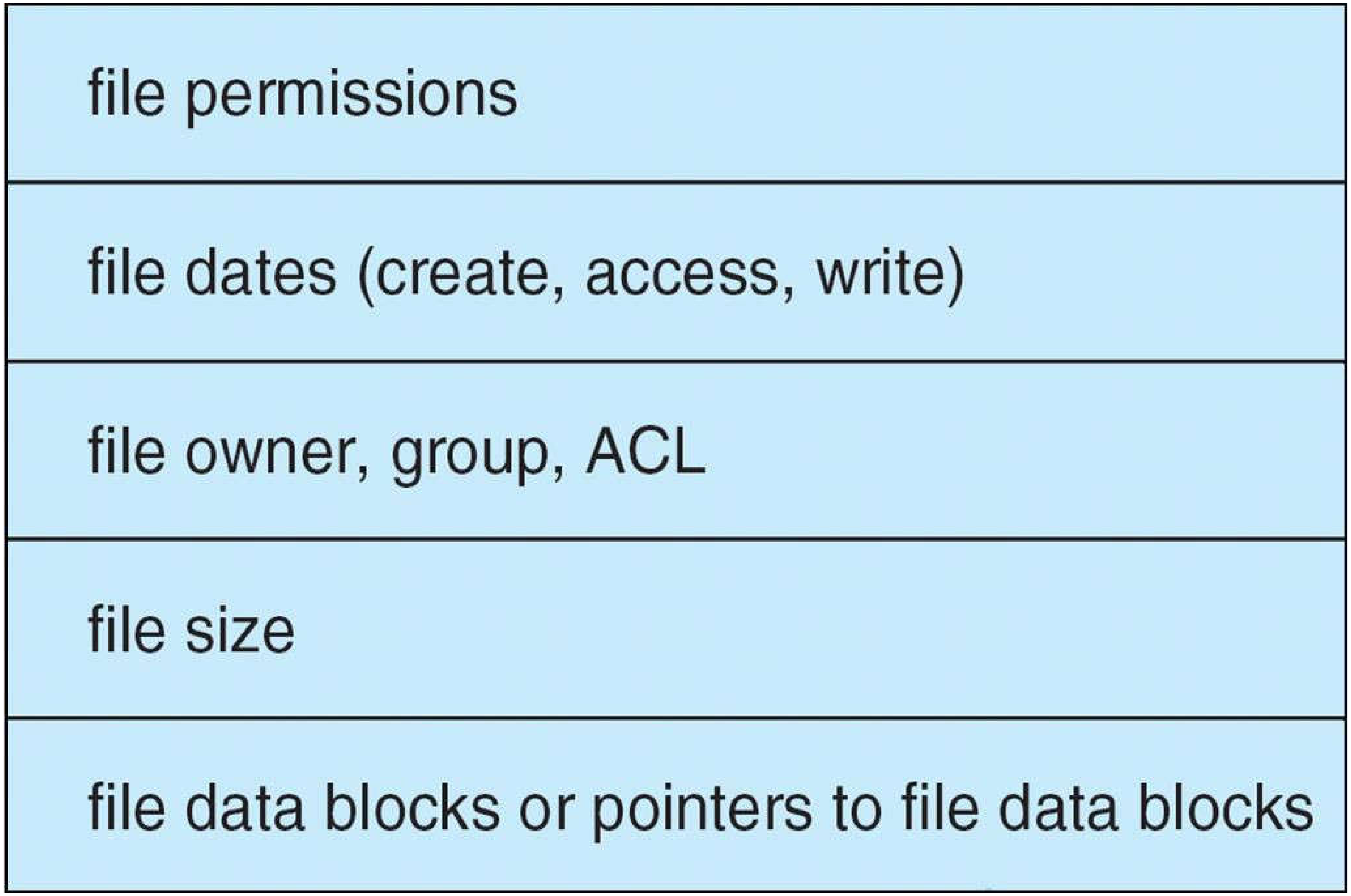
Layered File System



Layered File System

- **I/O control:** device drivers & interrupt handlers
- **Basic file system:** issues generic commands to appropriate device driver
- **File-organization module:** knows about files and their logical blocks & physical blocks. (includes free-space manager)
- **Logical file system**: manages metadata information (all of file system structure except the actual data) (FCB)
- **File control block:** information about file, including
  - ownership,
  - permissions, and
  - location of file contents.

A Typical File Control Block



file permissions

file dates (create, access, write)

file owner, group, ACL

file size

file data blocks or pointers to file data blocks

File system implementation

- **On disk:**

1. *Boot control block:* contains inf. Needed by the system to boot OS
    1. UFS: boot block; NTFS: partition boot sector

2. Volume control block: contains volume details (no. of blocks, size of blocks, free block count etc.)
    1. UFS: superblock; NTFS: master file table

3. Directory structure: to organize files

File system implementation

- **In-memory:**

1. *In-memory mount table:* information about each mounted volume
2. *In-memory directory structure cache:* information of recently accessed directories
3. *System-wide open-file table:* copy of FCB of each open file
4. *Per-process open-file table:* pointer to appropriate entry in system-wide open-file table

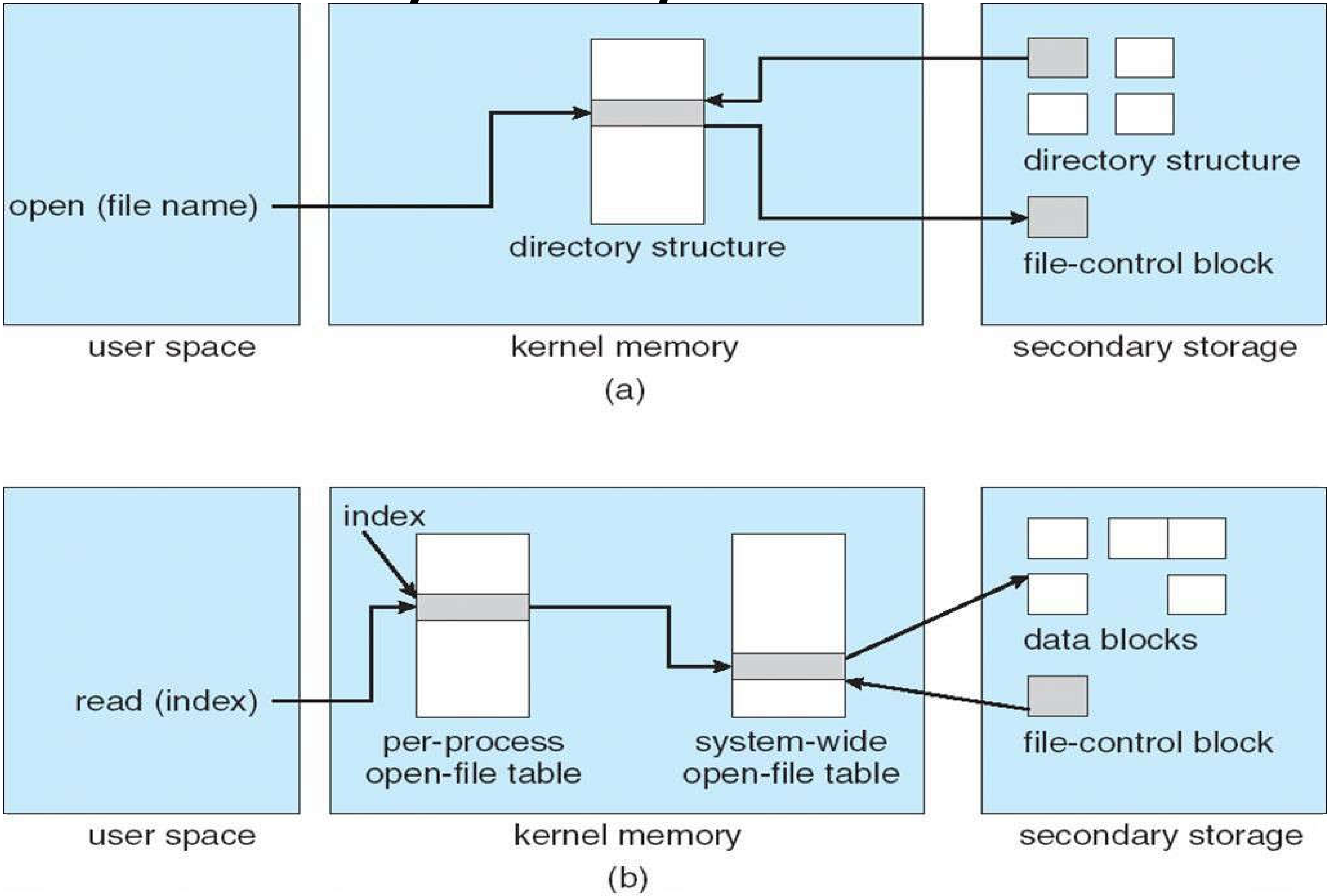
In-Memory File System Structures

- Fig. illustrates the necessary file system structures provided by the OS

- Figure (a) refers to opening a file.

- Figure (b) refers to reading a file.

In-Memory File System Structures



Partitions and mounting

- Disk– can be sliced into multiple partitions
- Raw disk – containing no file system
- Boot information: sequential series of blocks, loaded as an image into memory
- Systems can be dual-booted.
- Root partition: contains OS kernel & other system files is mounted at boot time

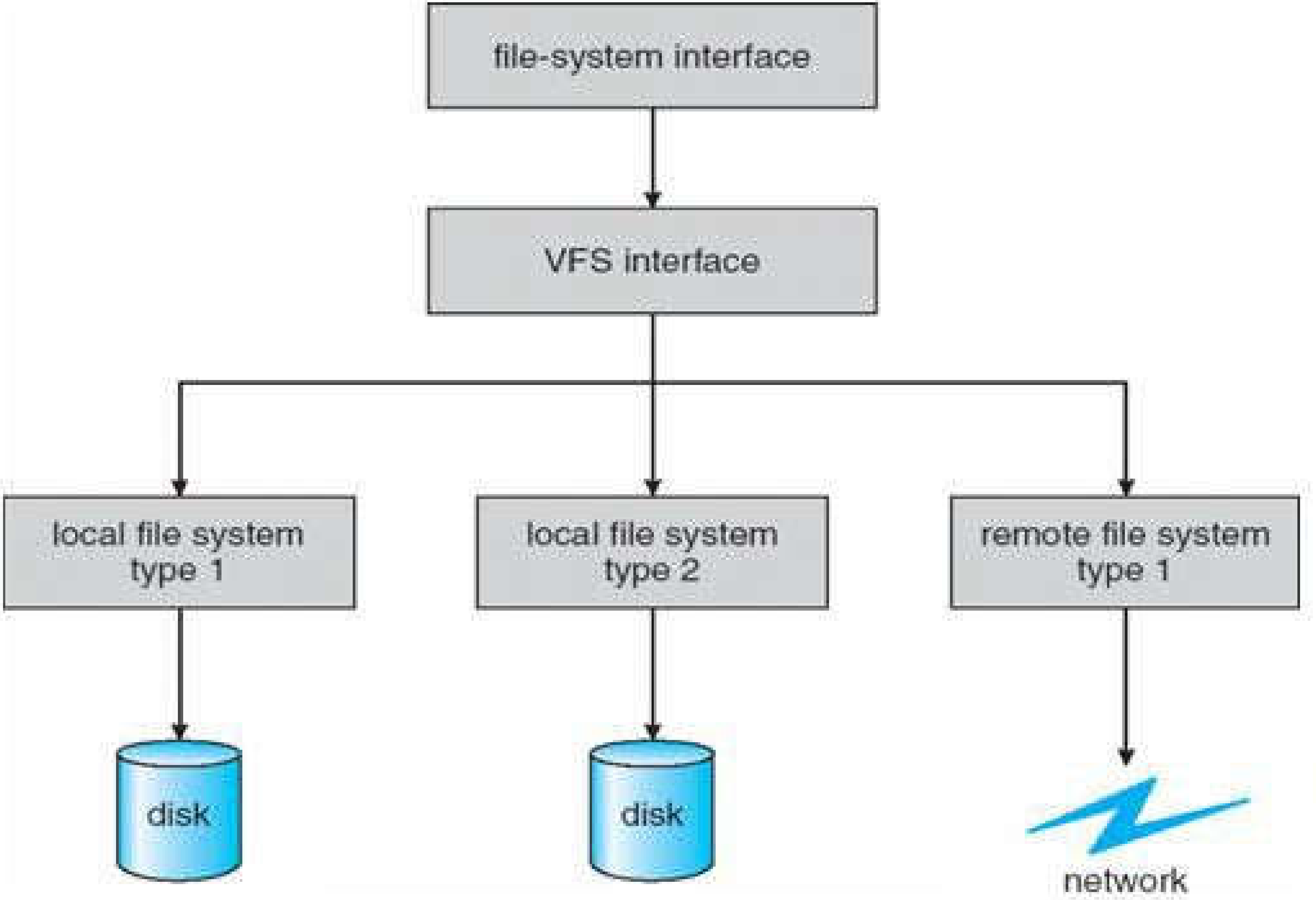
Virtual File Systems (VFS)

- VFS provide an object-oriented way of implementing file systems.

- VFS allows the same system call interface (the API) to be used for different types of file systems.

- The API is to the VFS interface, rather than any specific type of file system.

Schematic View of Virtual File System



VFS architecture in Linux

4 main object types defined by Linux VFS :

1. *inode object*: represents an individual file
2. *file object:* represents an open file
3. *superblock object:* represents an entire file system
4. *dentry object:* represents an individual directory entry

Directory Implementation

1. **Linear list** of file names with pointers to the data blocks.
   - simple to program
   - time-consuming to execute
   - finding a file requires linear search

2. **Hash Table** – linear list with hash data structure.
   - takes a value from file name & returns a pointer to the file name in the linear list
   - decreases directory search time
   - **collisions** – situations where two file names hash to the same location
   - fixed size

Allocation Methods

An allocation method refers to how disk blocks are allocated for files:

1. **Contiguous allocation**

2. **Linked allocation**

3. **Indexed allocation**

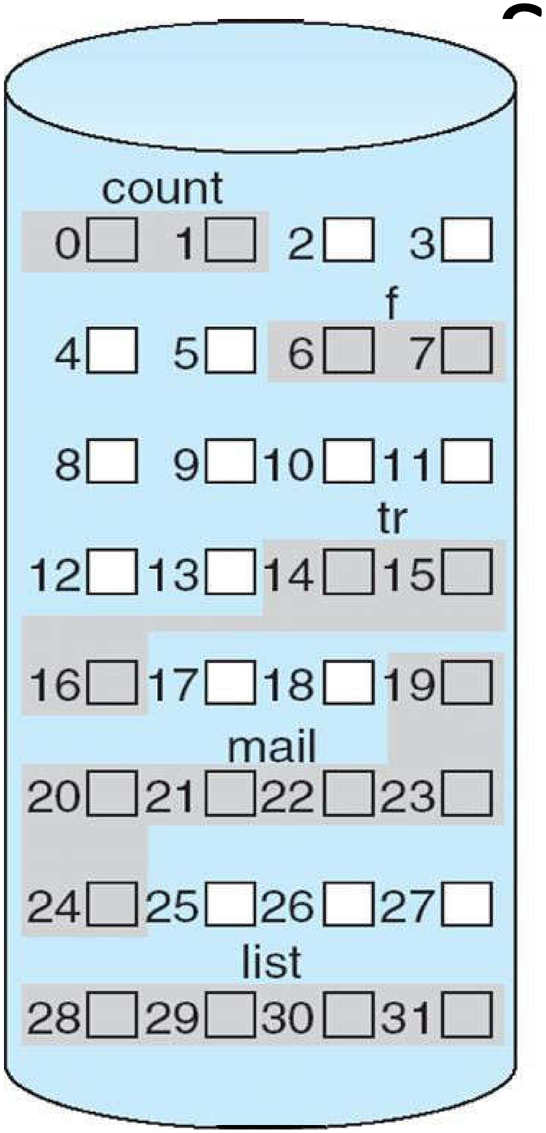
1. Contiguous Allocation

- Each file occupies a set of contiguous blocks on the disk
- Simple – disk address

  &

  length
- Random access
- Wasteful of space (dynamic storage-allocation problem)
- External fragmentation
- Files cannot grow

Contiguous Allocation of Disk S



directory

file	start	length
count	0	2
tr	14	3
mail	19	6
list	28	4
f	6	2

Extent-Based Systems

- Modified contiguous allocation scheme
- Extent-based file systems allocate disk blocks in **extents**
- Veritas file system uses extents
- An **extent** is a contiguous block of disks
  - Extents are allocated for file allocation
  - A file consists of one or more extents.

Linked Allocation

- Solves all problems of contiguous allocation
- Each file is a linked list of disk blocks,
- blocks may be scattered anywhere on the disk.

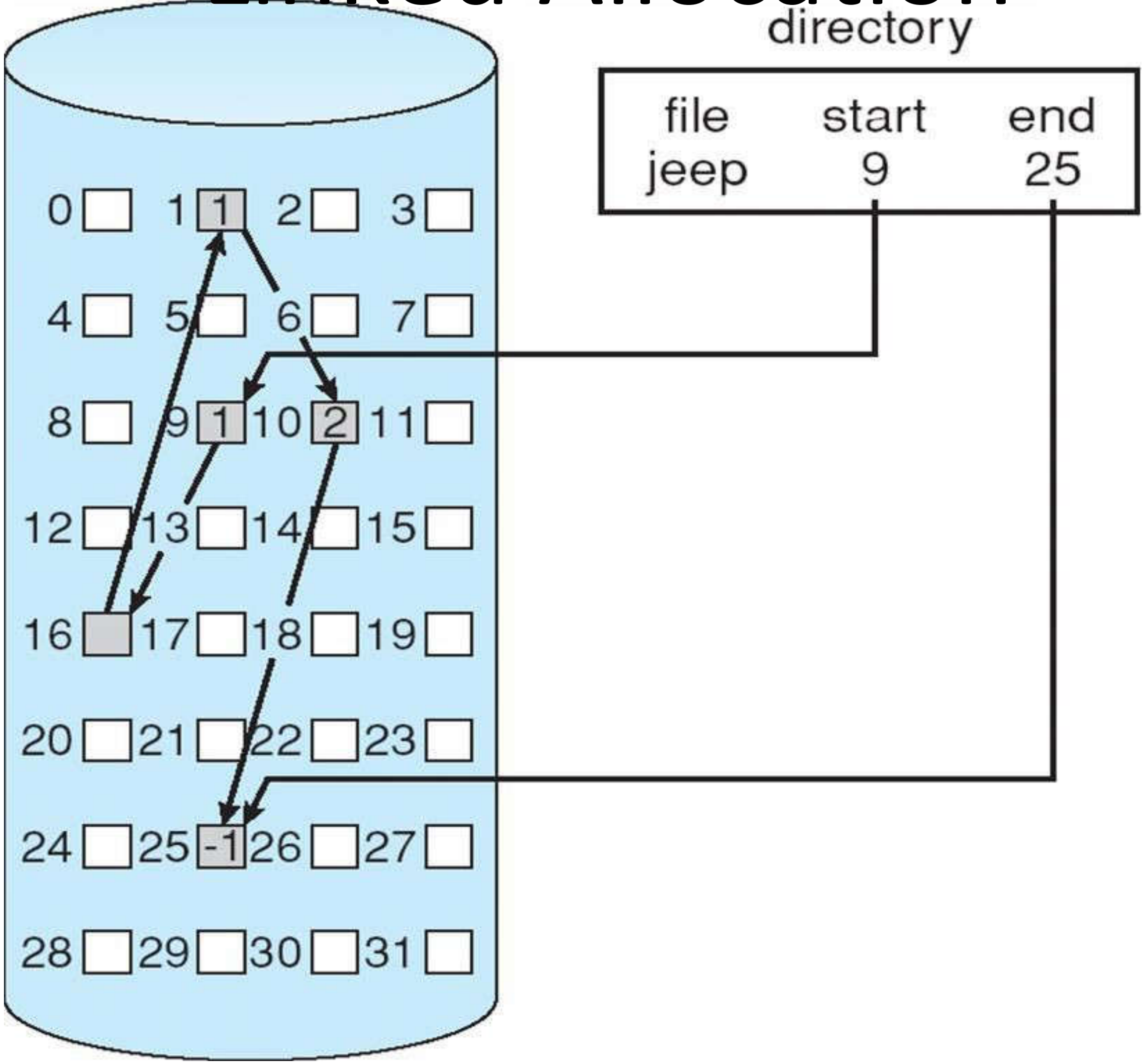
Linked Allocation (Cont.)

- Simple – need only starting address
- Free-space management system – no waste of space
- No random access
- Mapping

Disadvantages:

- Can be used only for sequential access files
- Space required for the pointers (sol: clusters – multiple blocks)
- Reliability (if a pointer were lost, sol: doubly linked list)

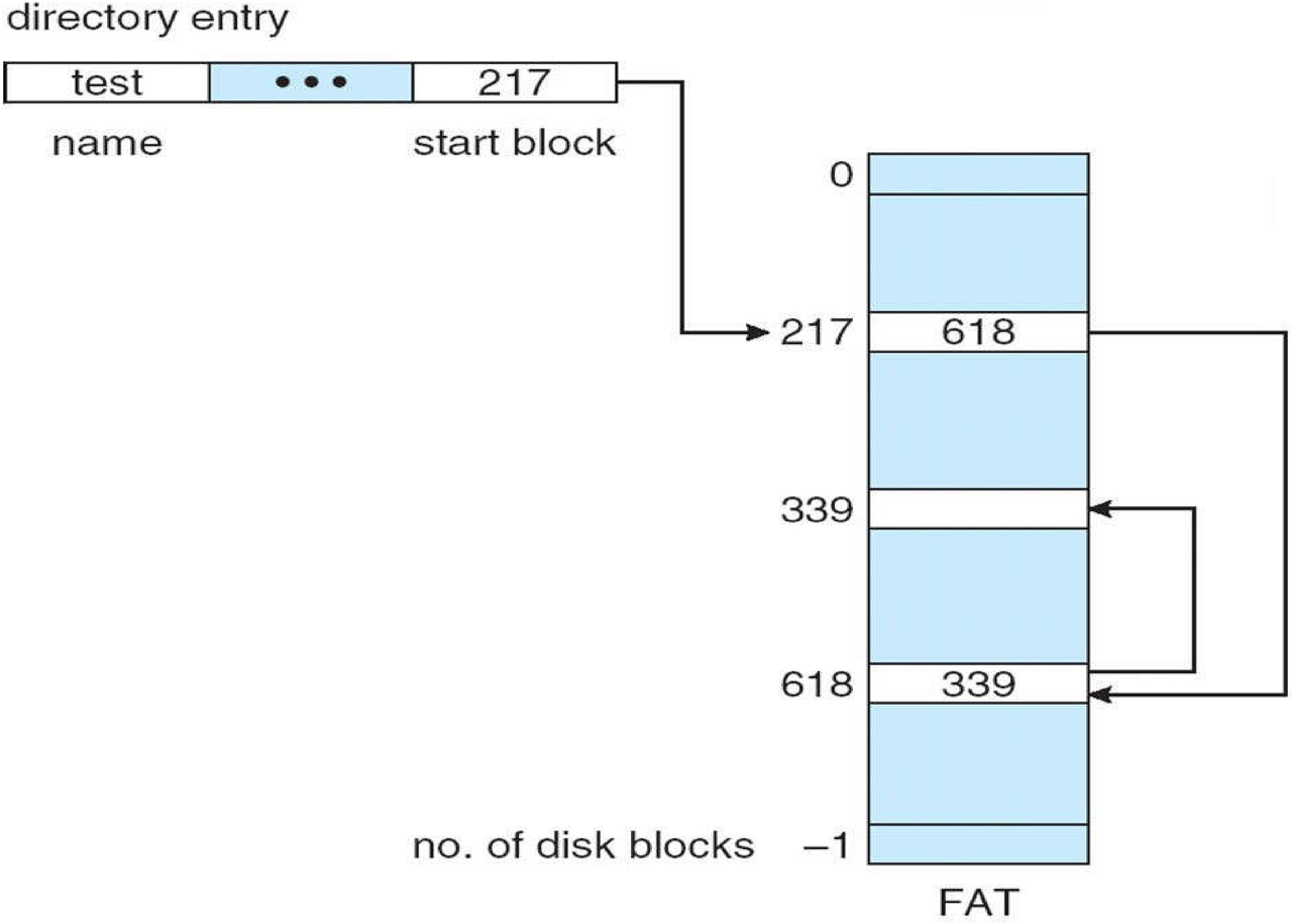
Linked Allocation



File allocation table

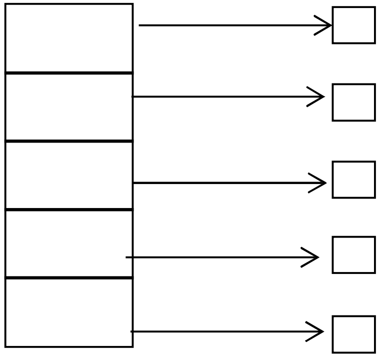
- Variation on linked allocation
  – Simple, but efficient
  – Section of disk beginning – set aside for table
  – One entry for each disk block

File-Allocation Table



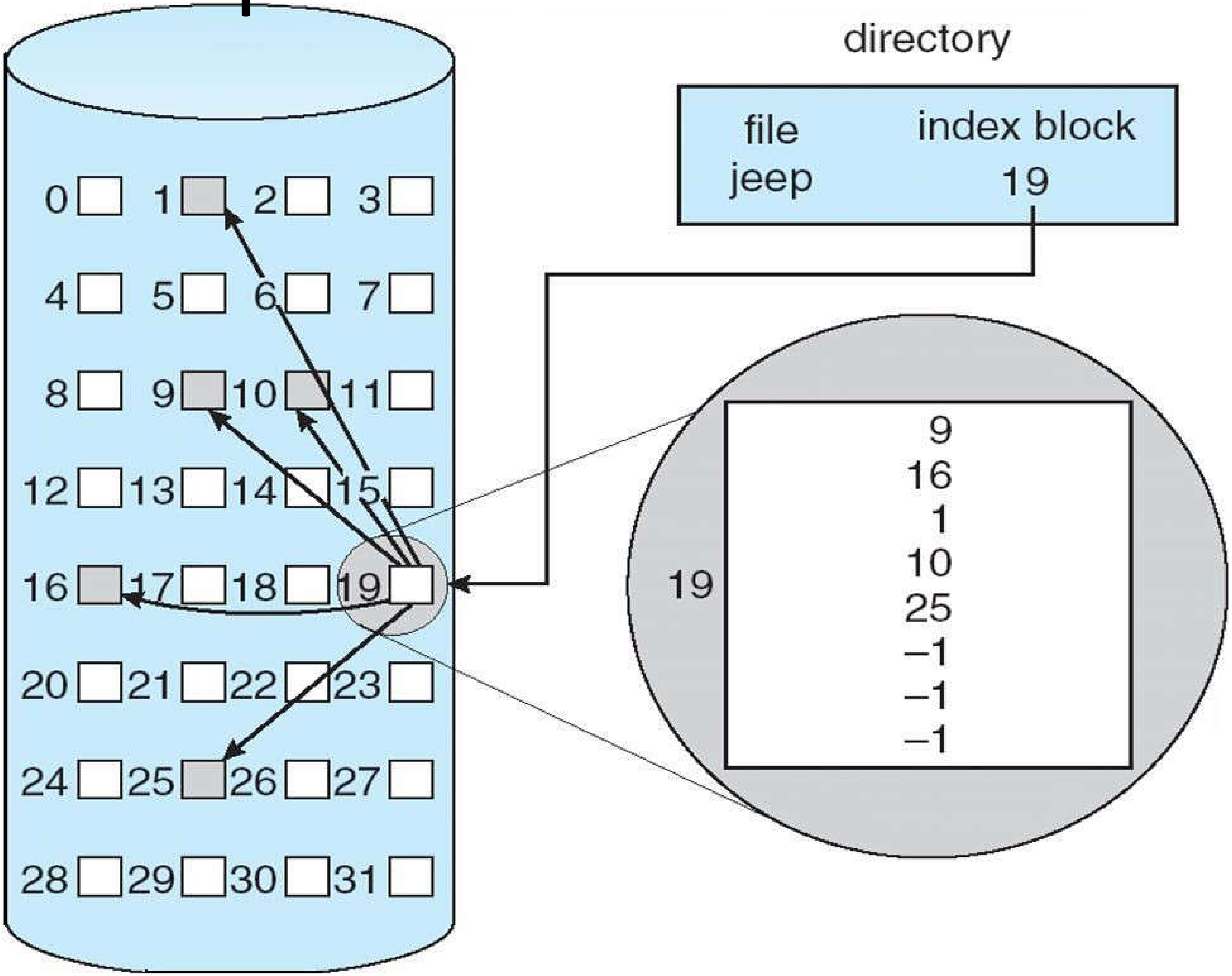
Indexed Allocation

- Brings all pointers together into the *index block.*
- Logical view.



index table

Example of Indexed Allocation



Indexed Allocation (Cont.)

- Need index table
- Random access
- Dynamic access without external fragmentation, but have overhead of index block.

Indexed Allocation (Cont.)

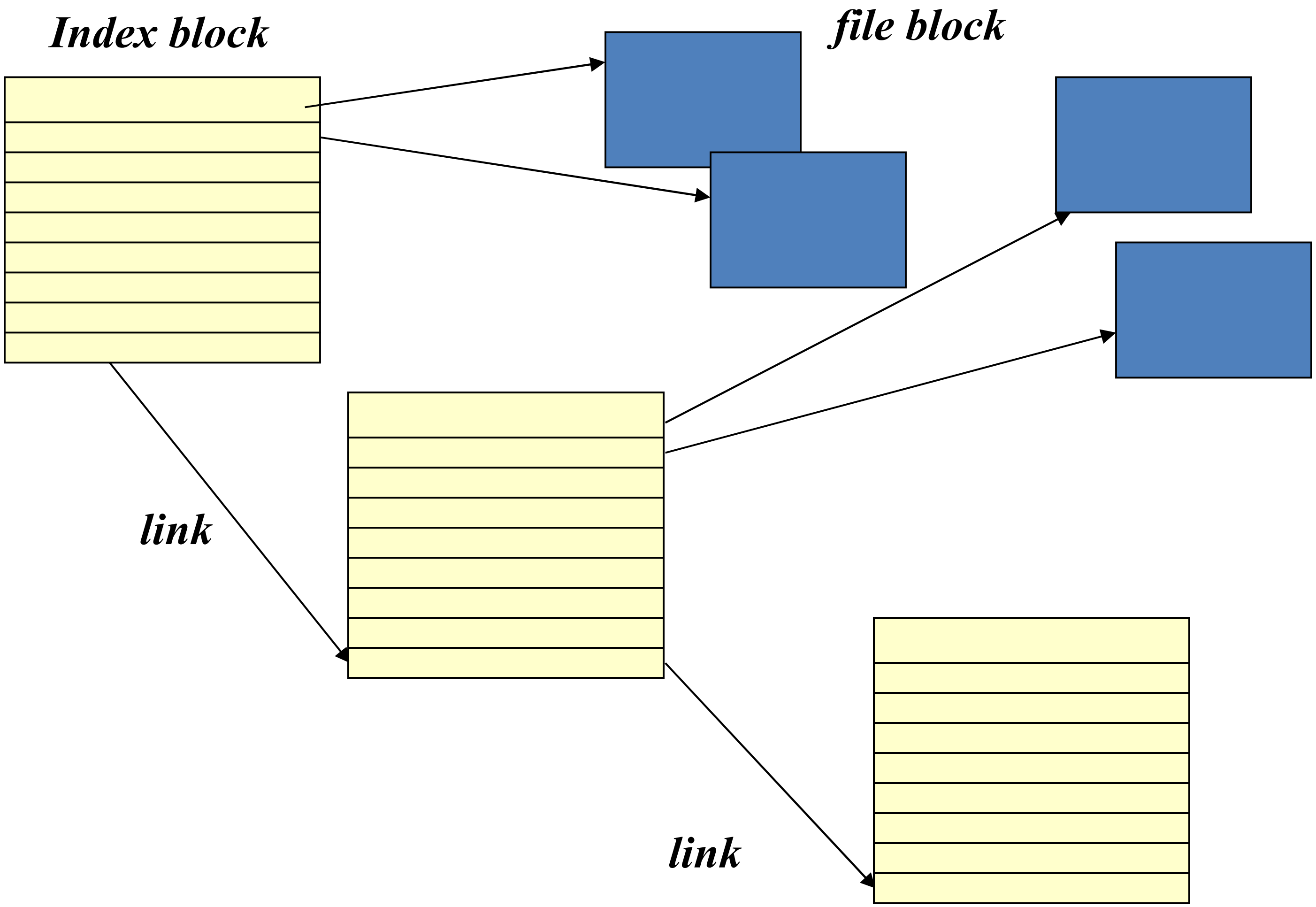
1. Linked scheme: normally one disk block

   Large files: link together several index blocks

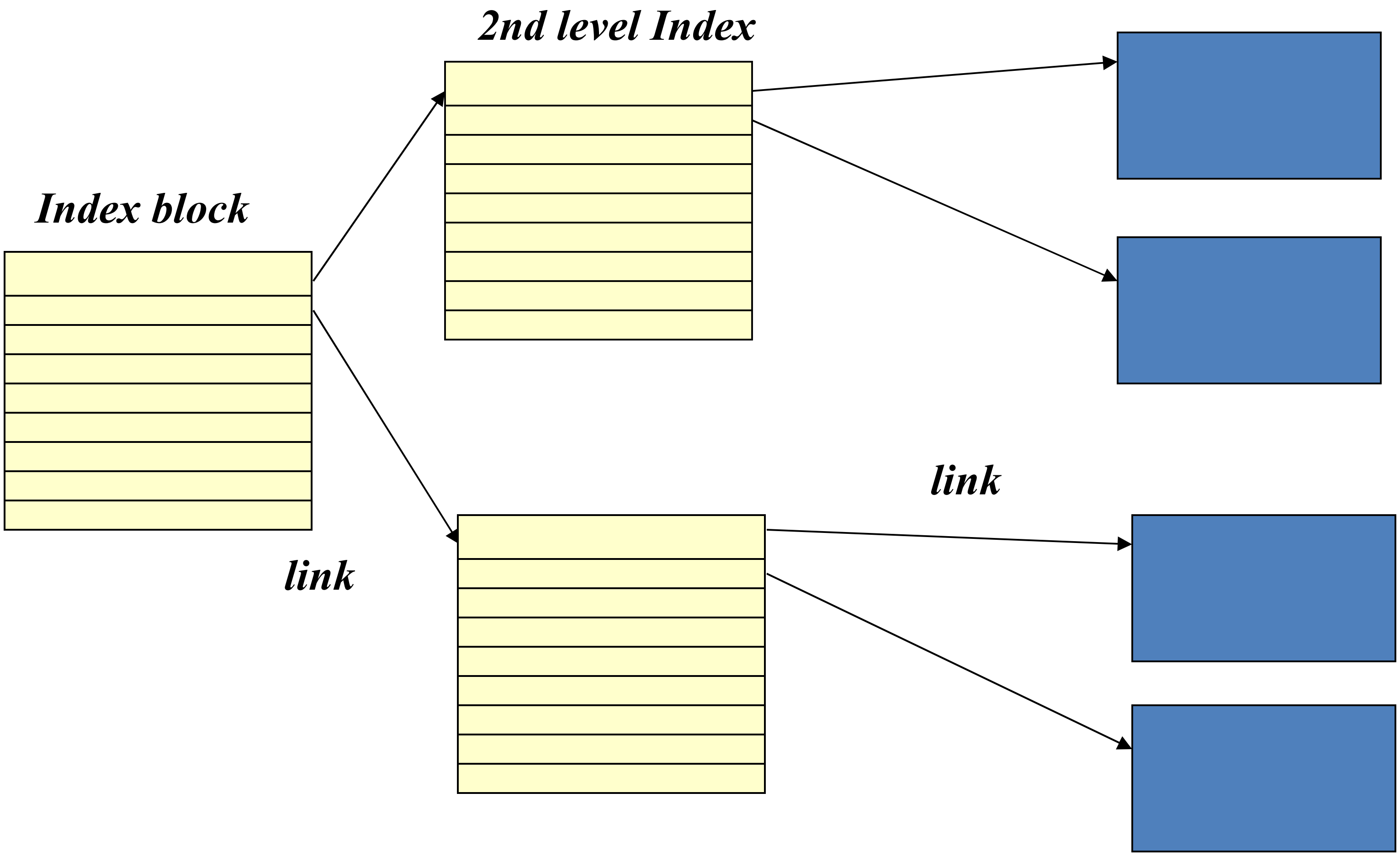
2. Multilevel index:

   - E.g. Two Level Index - first level index block points to a set of second level index blocks, which in turn point to file blocks.
   - Increase number of levels based on maximum file size desired.
   - Maximum size of file is bounded.

Indexed File - Linked Scheme



Indexed Allocation - Multilevel index



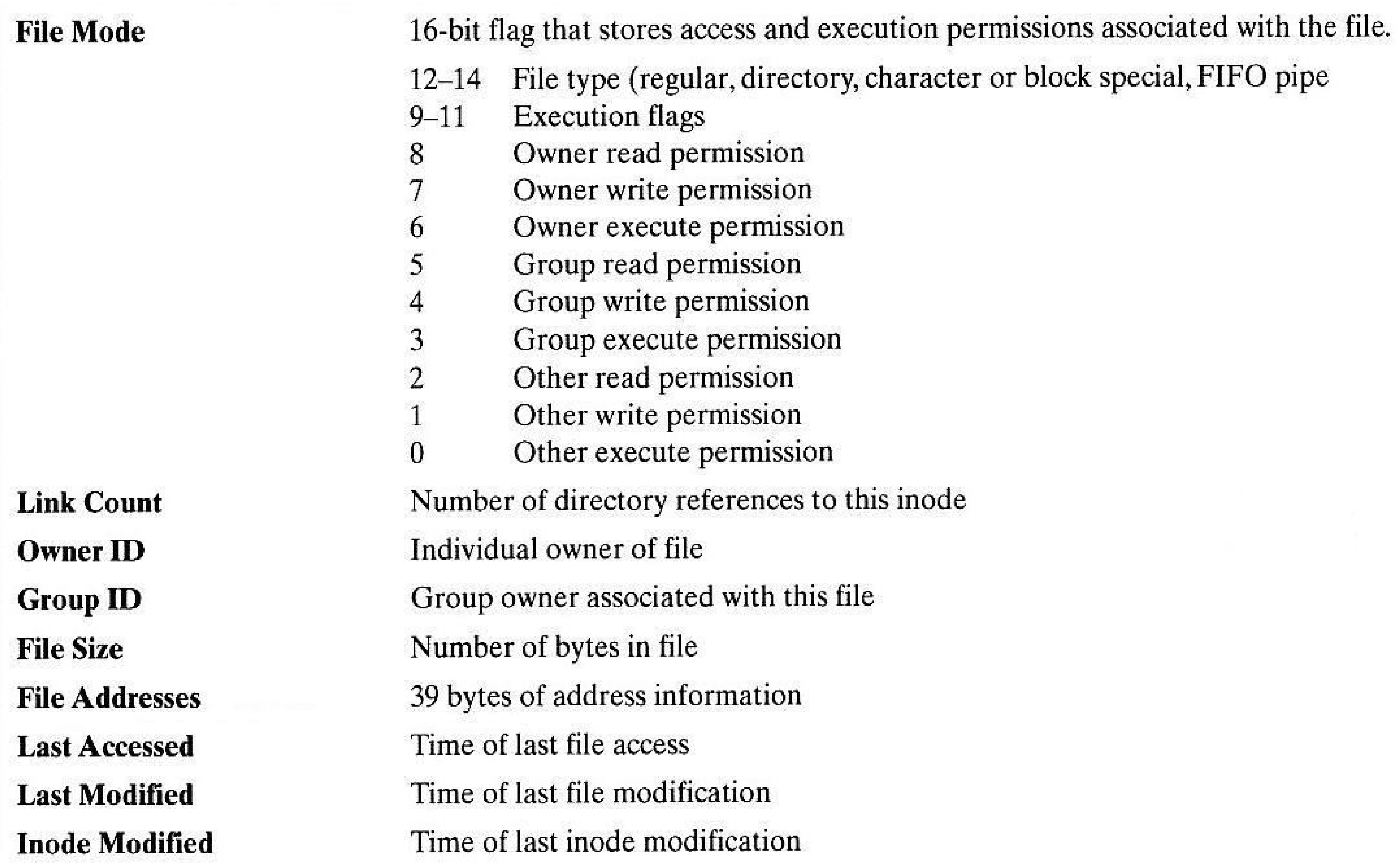
Indexed Allocation (Cont.)

3. Combined scheme:

- An inode (index node) is a control structure that contains key information needed by the OS to access a particular file. Several file names may be associated with a single inode, but each file is controlled by exactly ONE inode.

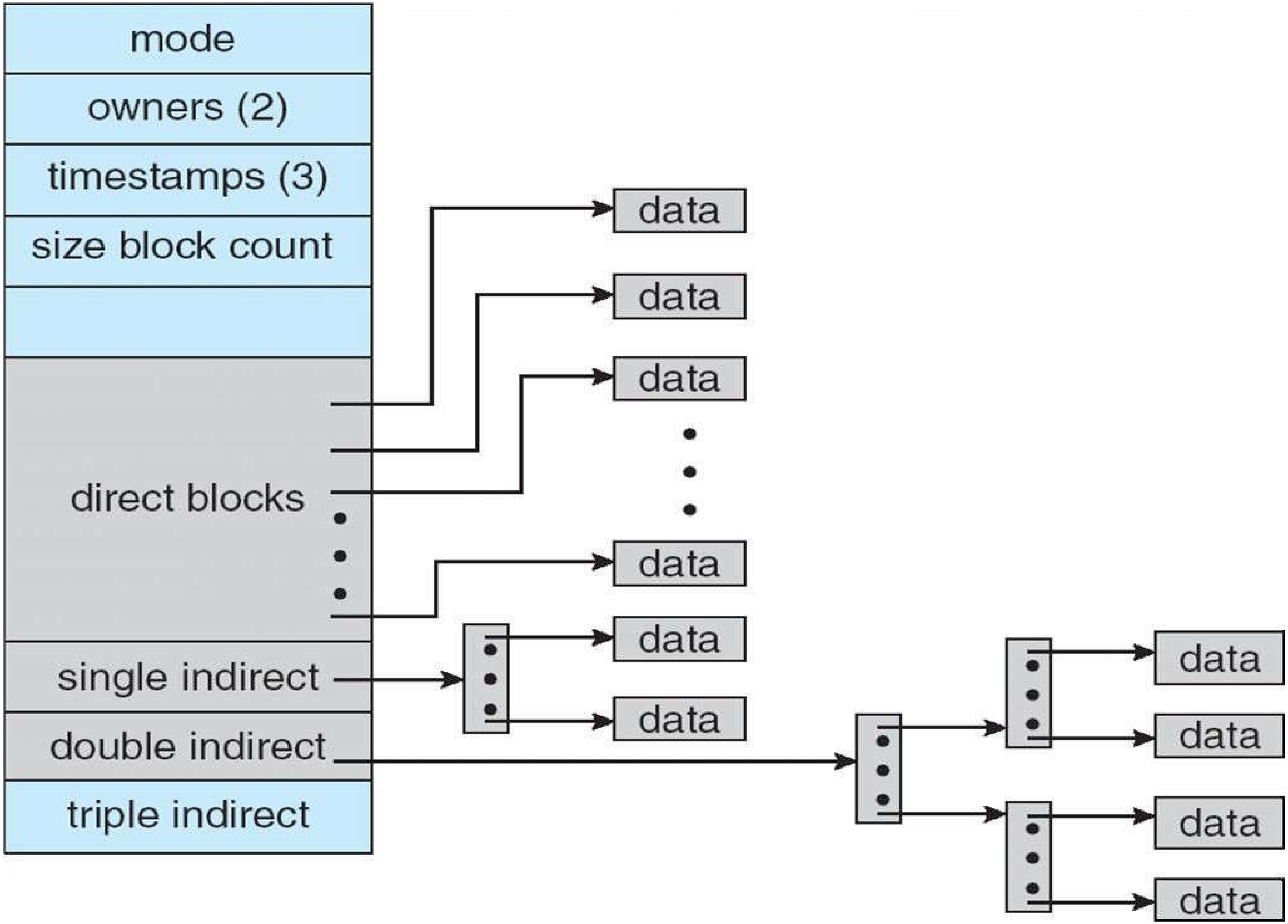
- On the disk, there is an inode table that contains the inodes of all the files in the filesystem. When a file is opened, its inode is brought into main memory and stored in a memory-resident inode table.

Information in the inode



File Mode	16-bit flag that stores access and execution permissions associated with the file.	
	12–14	File type (regular, directory, character or block special, FIFO pipe
	9–11	Execution flags
	8	Owner read permission
	7	Owner write permission
	6	Owner execute permission
	5	Group read permission
	4	Group write permission
	3	Group execute permission
	2	Other read permission
	1	Other write permission
	0	Other execute permission
Link Count	Number of directory references to this inode	
Owner ID	Individual owner of file	
Group ID	Group owner associated with this file	
File Size	Number of bytes in file	
File Addresses	39 bytes of address information	
Last Accessed	Time of last file access	
Last Modified	Time of last file modification	
Inode Modified	Time of last inode modification	

Combined Scheme: UNIX (4K bytes per block)



Allocation methods - Performance

Contiguous allocation	Linked allocation	Indexed allocation
direct access files	sequential access files	complex
Supports sequential access files also	Cannot be used for direct access files	Performance depends upon index structure, size of the file, position of the block desired
Efficient for small files	-----	For large files

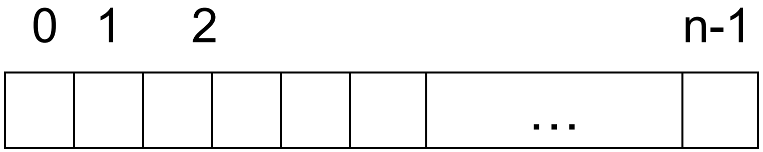
Free-Space Management

Free-space list: keeps track of free disk space

1. Bit vector
2. Linked list
3. Grouping
4. Counting

Free-Space Management

1. Bit vector or bit vector (*n* blocks)



$$\text{bit}[i] = \begin{cases} 1 \Rightarrow \text{block}[i] \text{ free} \\ 0 \Rightarrow \text{block}[i] \text{ occupied} \end{cases}$$

Block number calculation:

(number of bits per word) *
(number of 0-value words) +
offset of first 1 bit

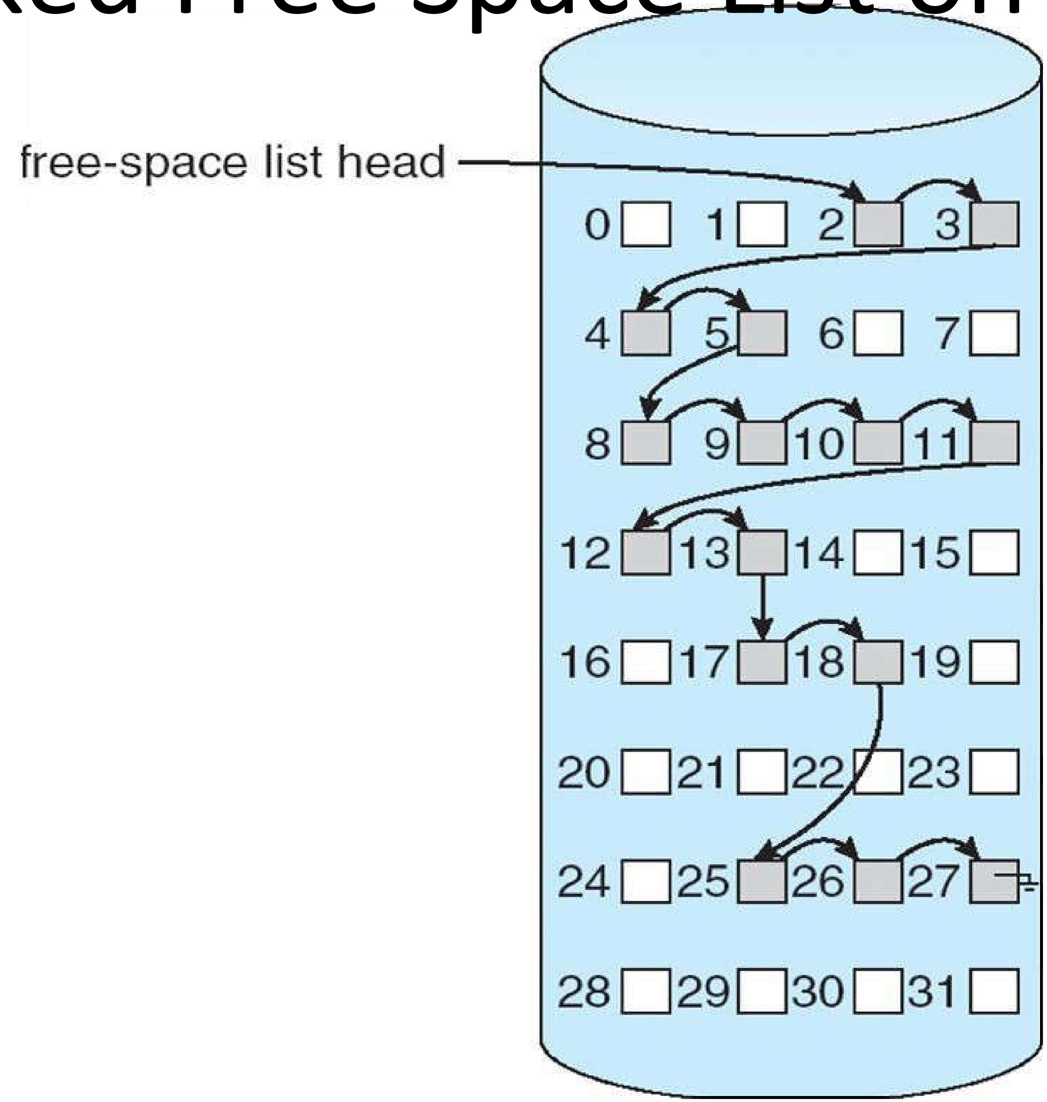
Bit map requires extra space

Easy to get contiguous files

Free-Space Management (Cont.)

2. Linked list (free list)
   - Link all free disk blocks – keep pointer to first free block & cache it in memory
   - Cannot get contiguous space easily
   - No waste of space

Linked Free Space List on Disk



Free-Space Management (Cont.)

3. Grouping:

- Stores addresses of n free blocks in first free block
- Last block - addresses of another n free blocks

4. Counting:

- Keeps address of first free block & number n of free contiguous blocks that follow first block
- Each entry – disk address & count

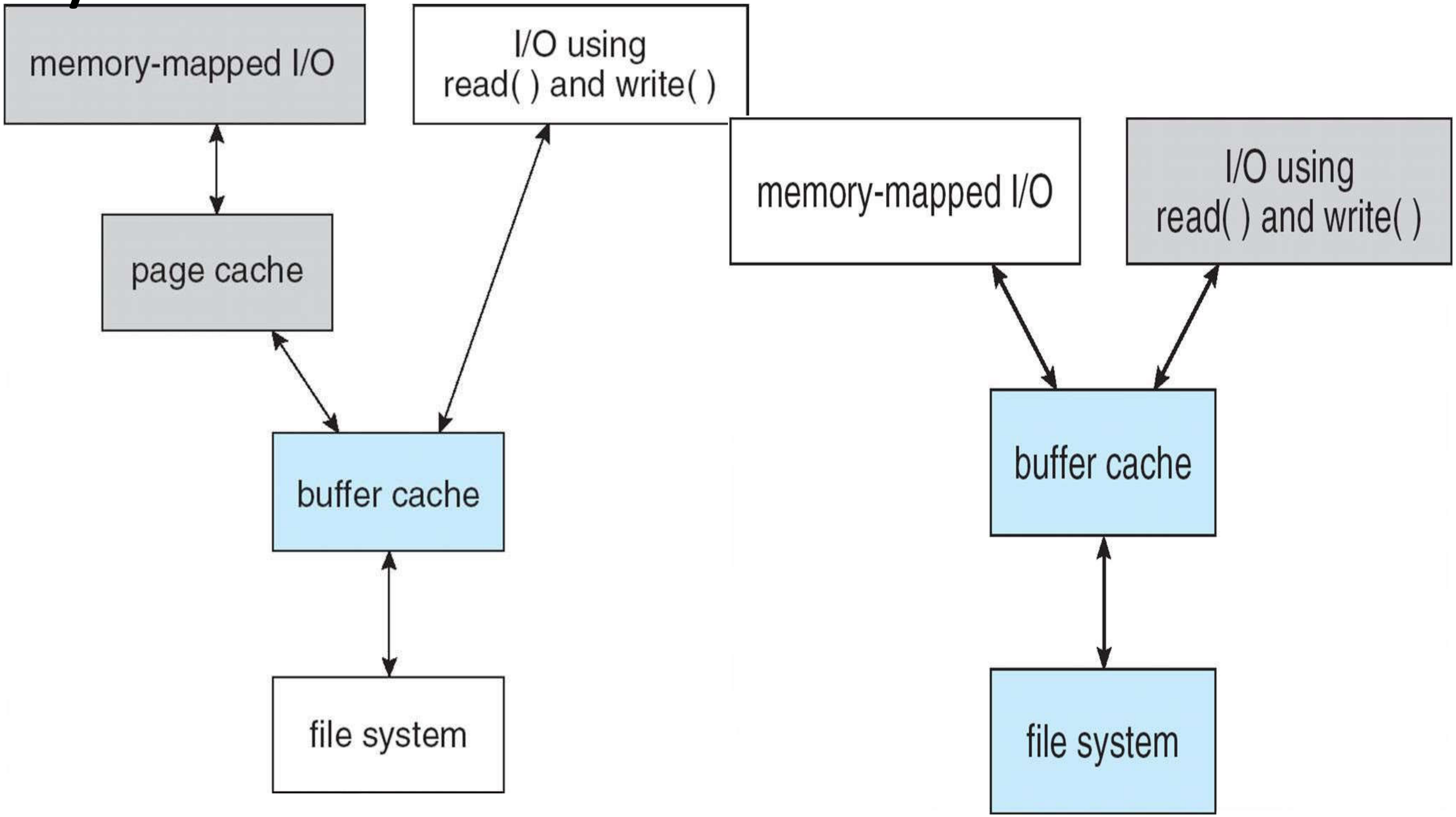
Efficiency and Performance

- **Efficiency** depends on:
  - disk allocation and directory algorithms
  - types of data kept in file’s directory entry
    - last write date or last access date

Efficiency and Performance

- **Performance:**
  - **disk cache** – separate section of main memory for frequently used blocks
  - **Buffer cache** – separate section of main memory for blocks that will be used again shortly
  - **Page cache** – caches file data as pages
  - **Unified virtual memory** – caches both pages & file data
  - **Unified buffer cache** – uses the same page cache for both memory-mapped pages and files

I/O Without a Unified Buffer Cache



memory-mapped I/O

I/O using read( ) and write( )

page cache

buffer cache

file system

memory-mapped I/O

I/O using read( ) and write( )

buffer cache

file system

Efficiency and Performance

- Block replacement mechanisms:
  - LRU
  - *Free-behind* - removes block from buffer as soon as next block is requested.
  - *Read-ahead* - request block and several subsequent blocks are read and cached.

End of Unit - 6