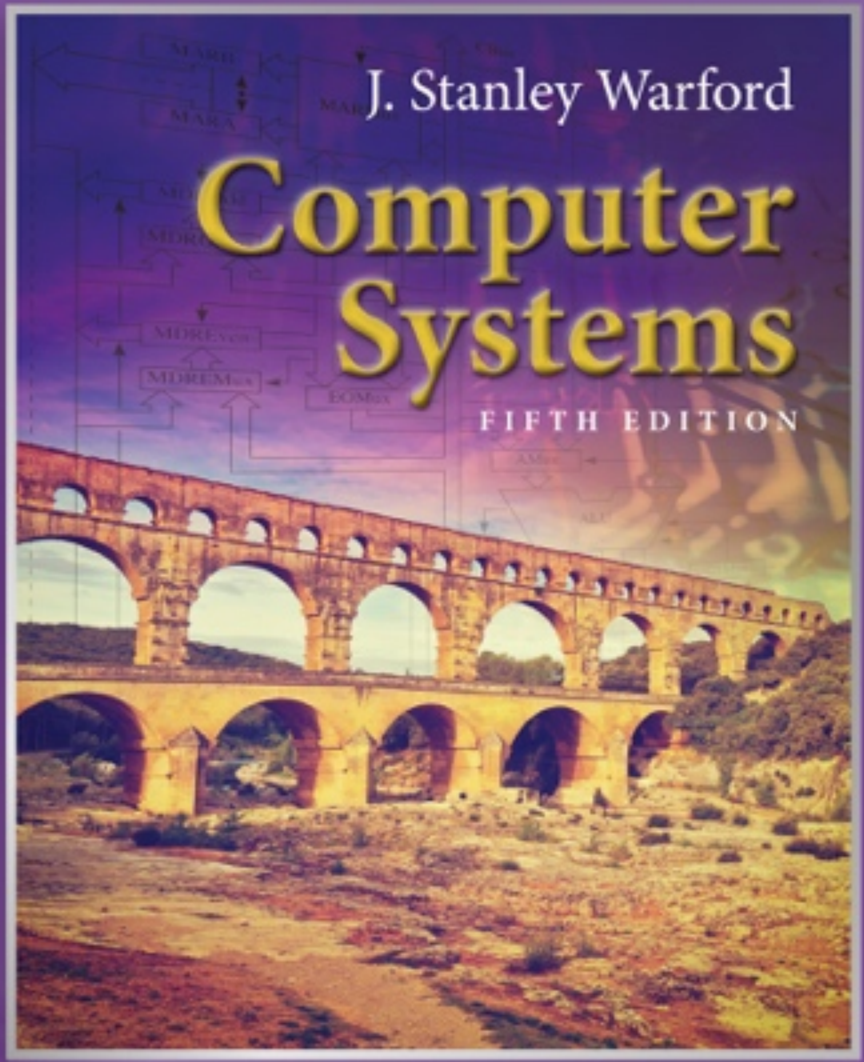
Chapter 9

Storage Management



Memory allocation techniques

- Uniprogramming
- Fixed-partition multiprogramming
- Variable-partition multiprogramming
- Paging
- Virtual memory

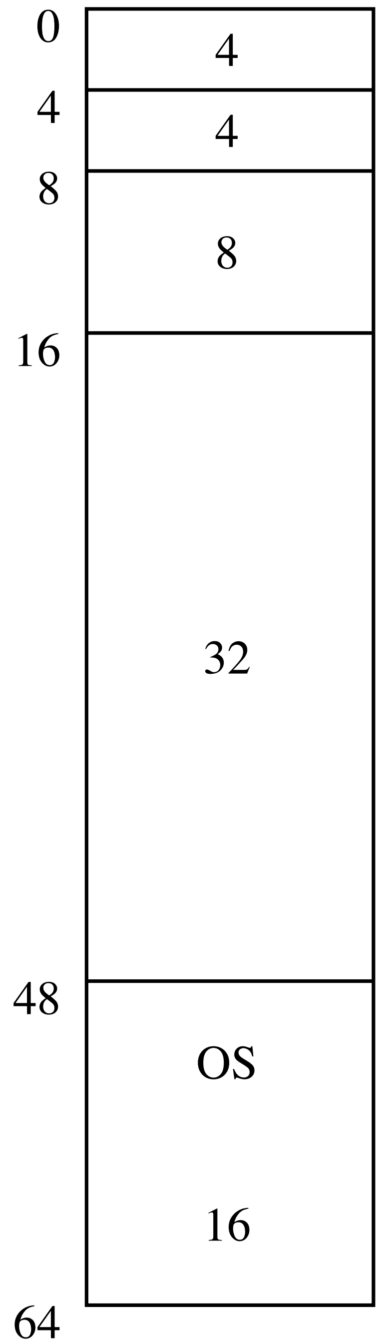
Uniprogramming

- Operating system resides at one end of memory
- Application at the other end
- System only executes one job at time
- Example: Pep/8 operating system
- Disadvantages: Inflexible, CPU time wasted waiting for I/O

Fixed-partition multiprogramming

- Operating system in one fixed reserved partition of memory
- Multiple processes in fixed partitions of memory
- Must solve the address problem

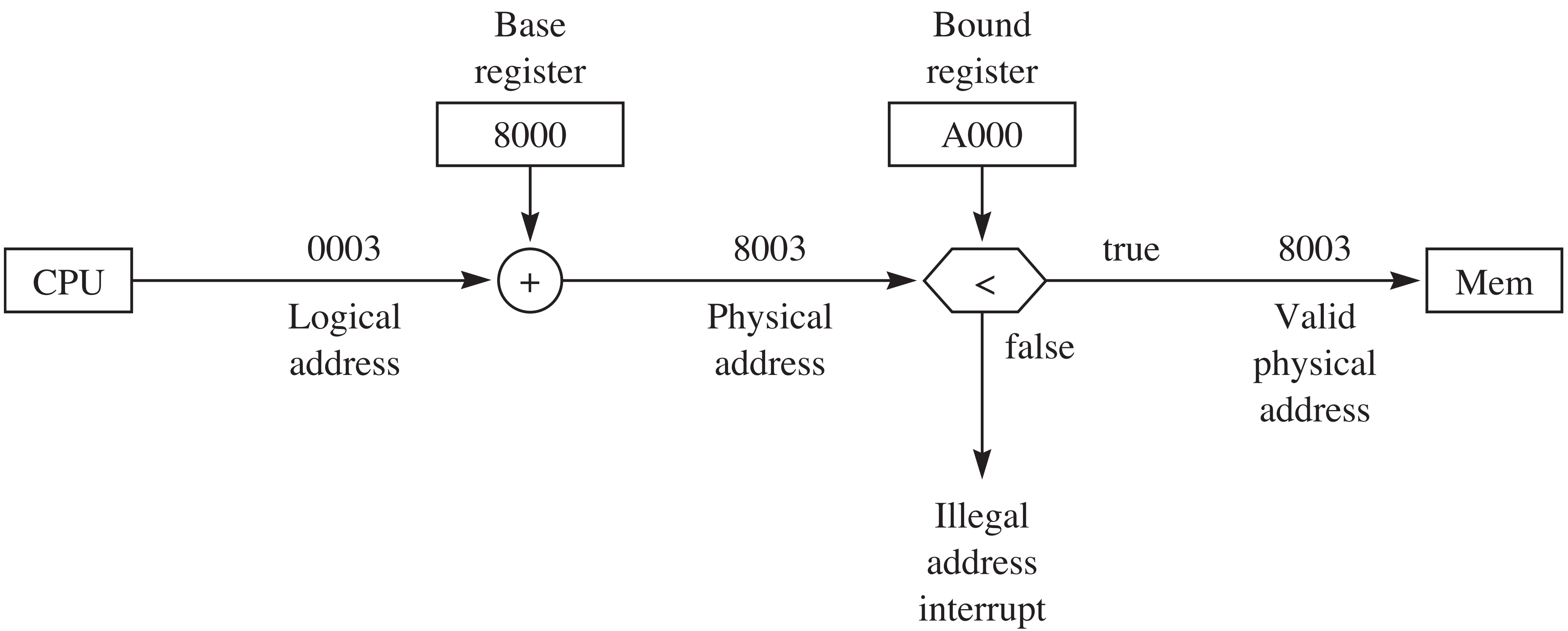
Figure 9.1



Logical addresses

- Logical address is the address generated by the assembler assuming the program begins at address 0
- Physical address =

  logical address + partition address
- Base register converts from logical to physical
- Bound register keeps program isolated

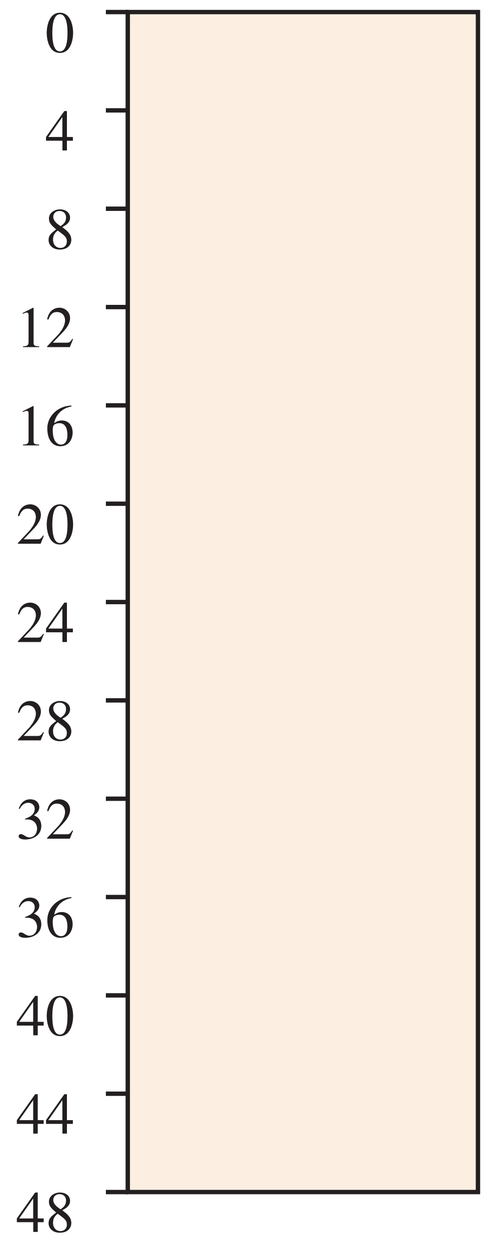

Base
register
8000
Bound
register
A000
CPU
0003
Logical
address
+
8003
Physical
address
<
true
8003
Valid
physical
address
Mem
false
Illegal
address
interrupt

Problems with fixed-partitions

- Scheduling a small job in a large partition because the small partitions are all used
- Determining the optimal partition in the first place

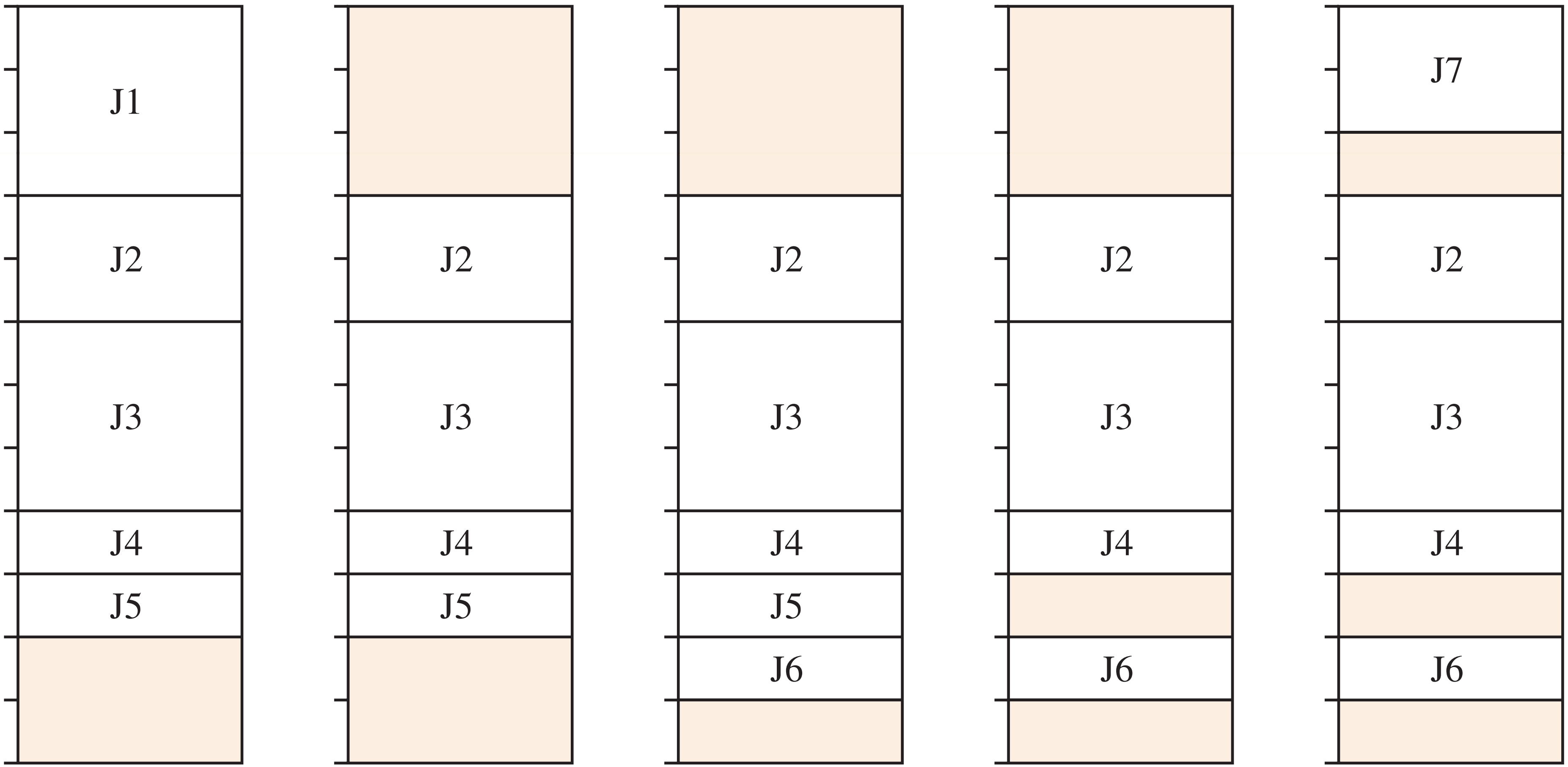
Variable-partition multiprogramming

- Establish a partition only when a job is loaded into memory
- The size of the partition can match the size of the job
- A region available for use by an incoming job is a *hole*


0
4
8
12
16
20
24
28
32
36
40
44
48

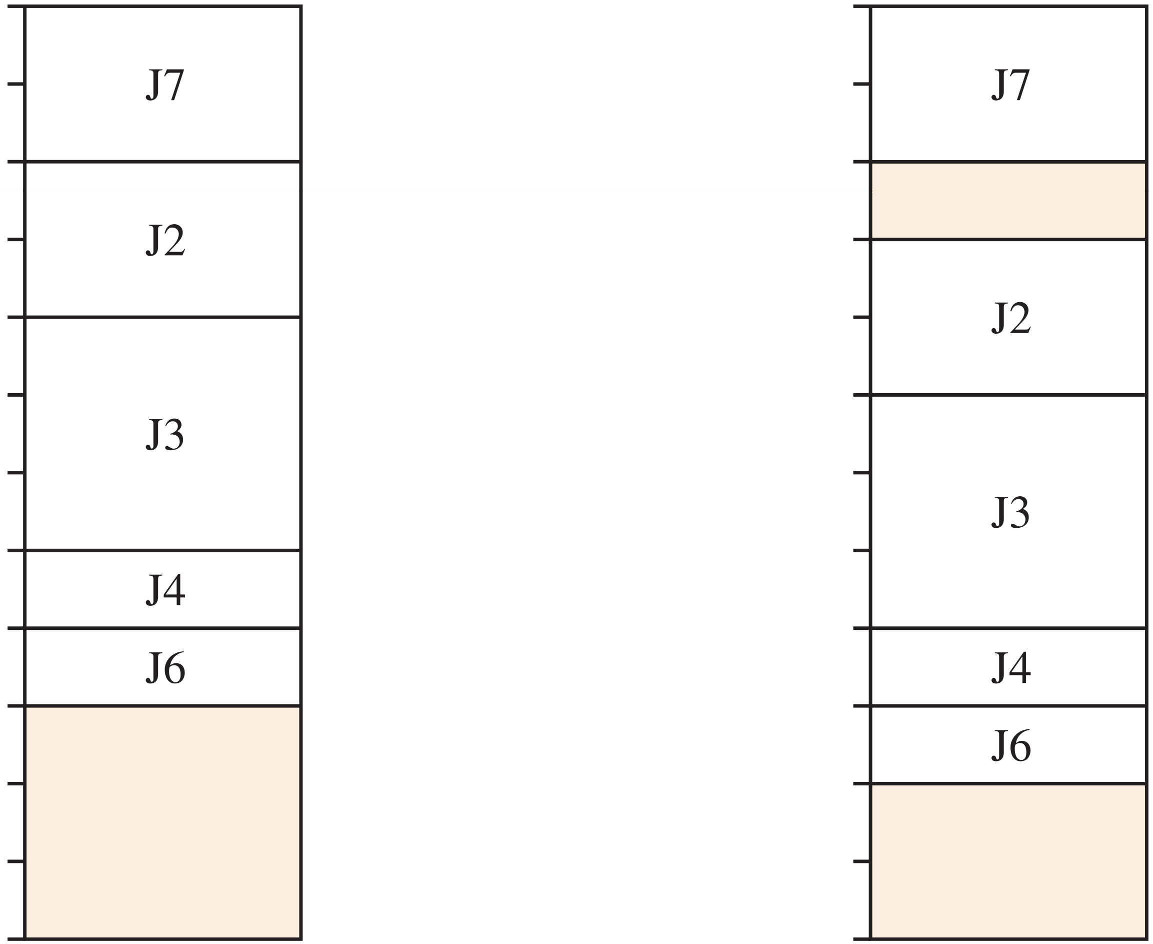
Job	Size	Action
J1	12	Start
J2	8	Start
J3	12	Start
J4	4	Start
J5	4	Start
J1	12	Stop
J6	4	Start
J5	4	Stop
J7	8	Start
J8	8	Start

Best-fit algorithm



(a) J1 to J5 starts. **(b)** J1 stops. **(c)** J6 starts. **(d)** J5 stops. **(e)** J7 starts.

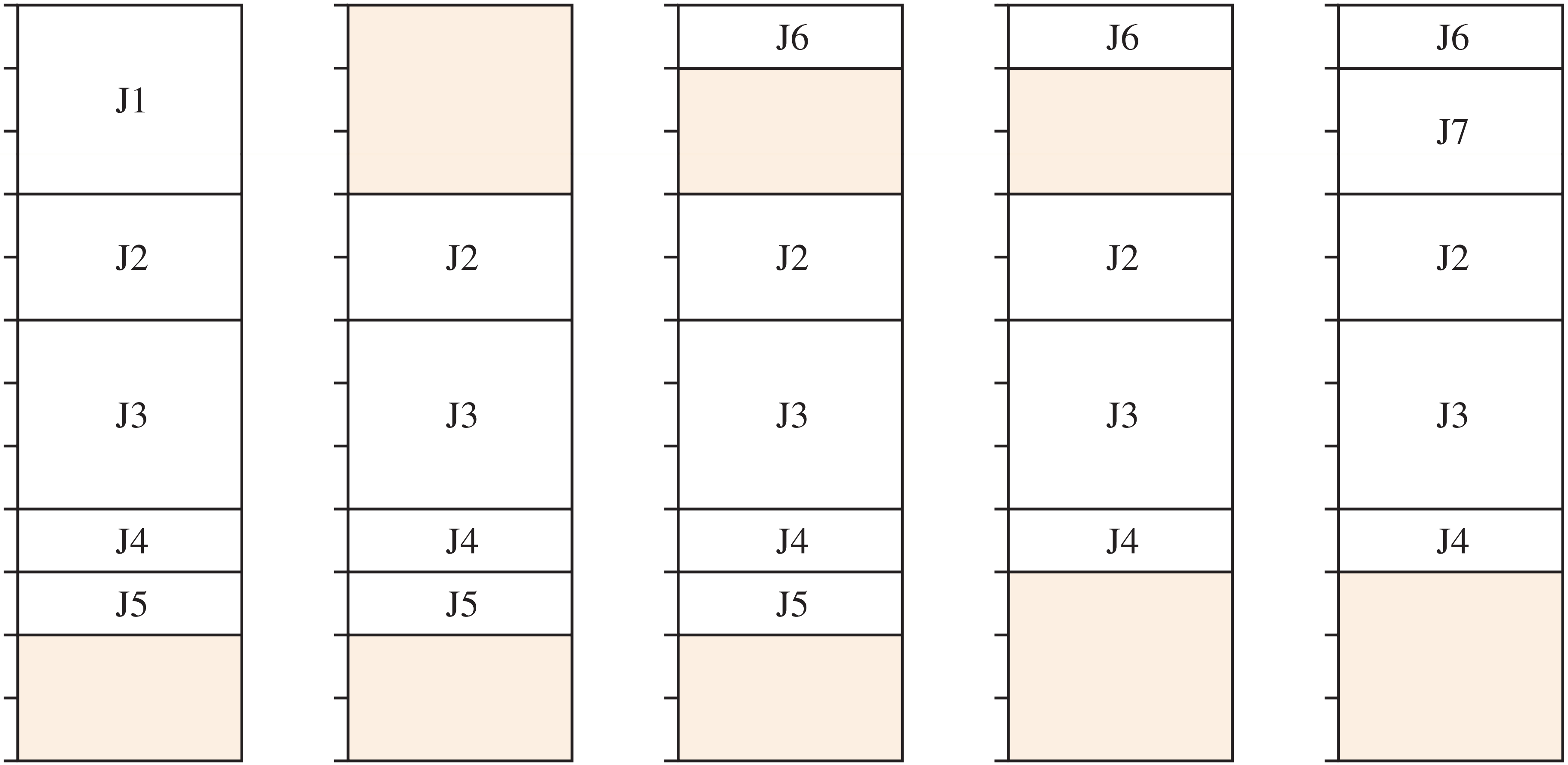
Compacting main memory



(a) Shifting all jobs to the top.

(b) Shifting only J6.

First-fit algorithm



(a) J1 to J5 starts.

(b) J1 stops.

(c) J6 starts.

(d) J5 stops.

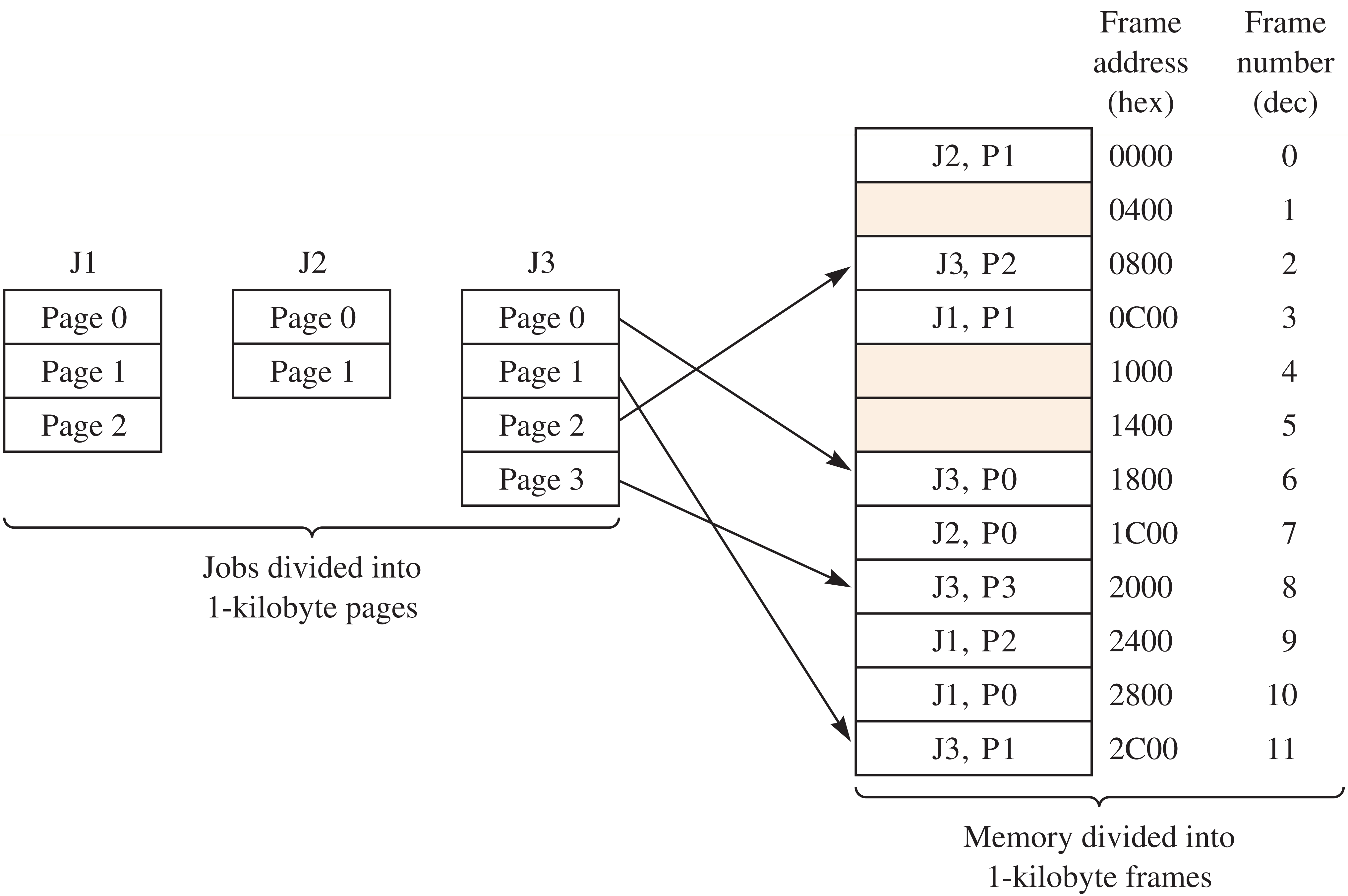
(e) J7 starts.

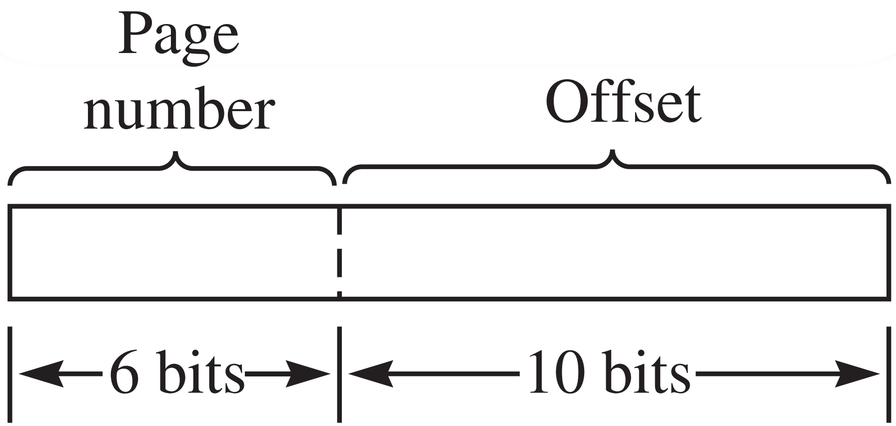
Problems with variable partitions

- Fragmentation
- Consolidating holes is time-consuming

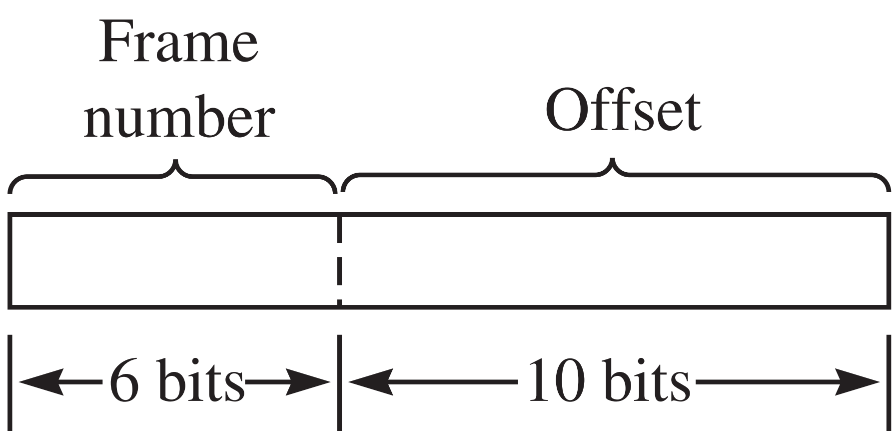
Paging

- Rather than coalesce several small holes to fit the program, fragment the program to fit the holes
- A job is divided into pages
- Main memory is divided into frames, each one the same size as a page
- No coalescing of holes is ever required


Frame address (hex)
Frame number (dec)
J1
J2
J3
Page 0
Page 1
Page 2
Page 0
Page 1
Page 0
Page 1
Page 2
Page 3
J2, P1
0000
0
0400
1
J3, P2
0800
2
J1, P1
0C00
3
1000
4
1400
5
J3, P0
1800
6
J2, P0
1C00
7
J3, P3
2000
8
J1, P2
2400
9
J1, P0
2800
10
J3, P1
2C00
11
Jobs divided into 1-kilobyte pages
Memory divided into 1-kilobyte frames

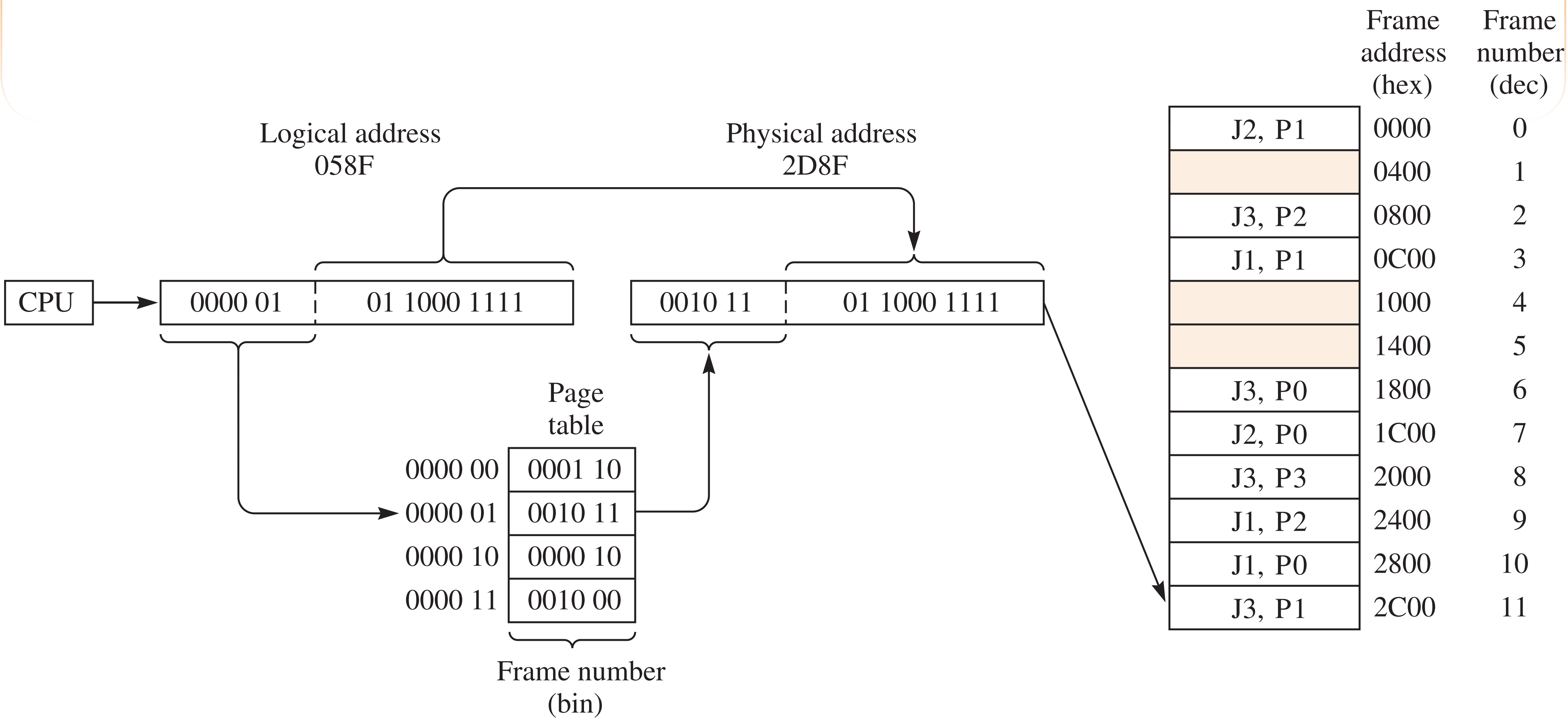


(a) Logical address.

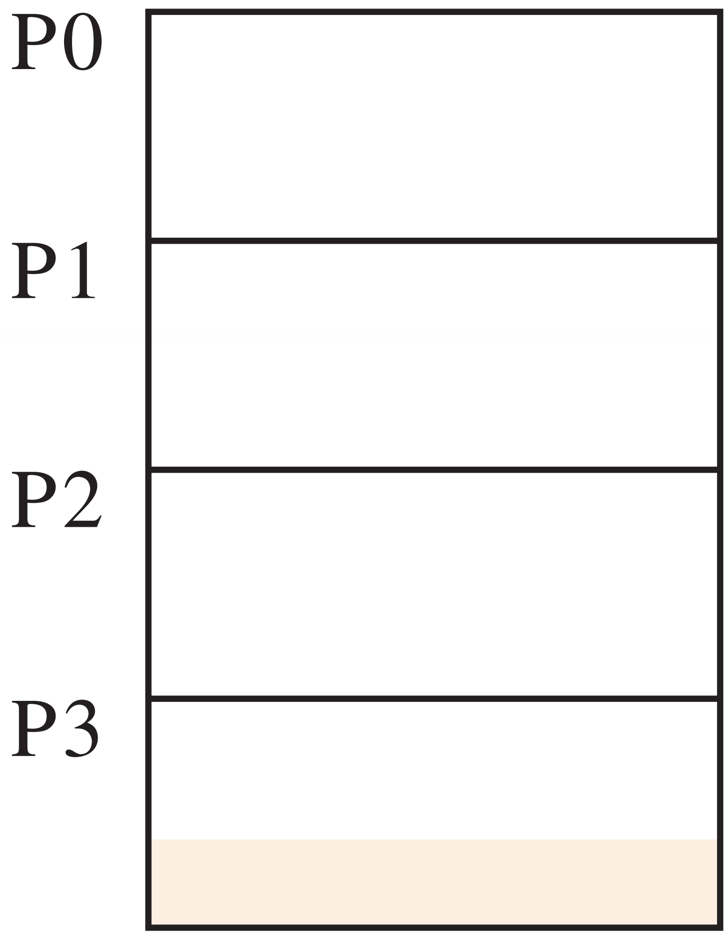


(b) Physical address.

Figure 9.10


Logical address
058F
Physical address
2D8F
CPU
0000 01
01 1000 1111
0010 11
01 1000 1111
Page
table
0000 00
0001 10
0000 01
0010 11
0000 10
0000 10
0000 11
0010 00
Frame number
(bin)
Frame address (hex)
Frame number (dec)
J2, P1
0000
0
0400
1
J3, P2
0800
2
J1, P1
0C00
3
1000
4
1400
5
J3, P0
1800
6
J2, P0
1C00
7
J3, P3
2000
8
J1, P2
2400
9
J1, P0
2800
10
J3, P1
2C00
11

Internal fragmentation



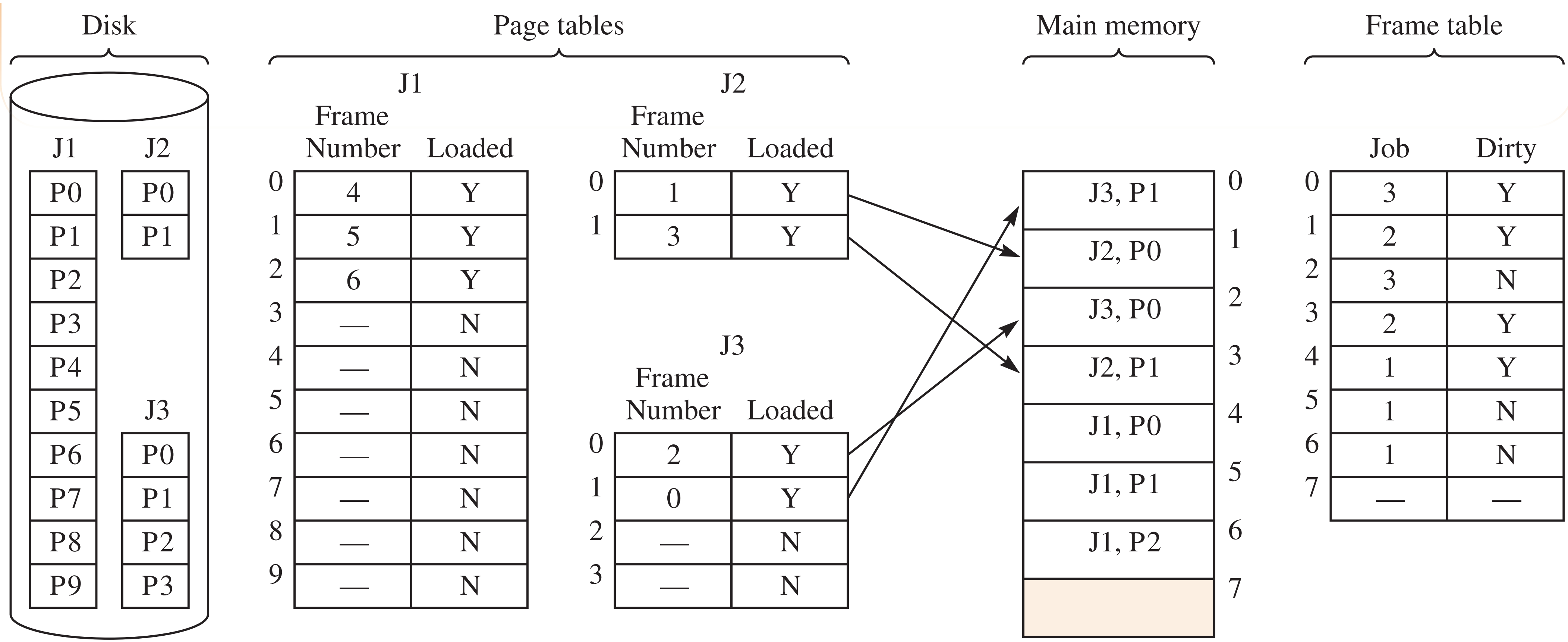
Problem with paging

- To execute a program, the entire program must be loaded into memory
- Most large programs have many sections of code that never execute
- Memory is used inefficiently with parts of the program taking up main memory unnecessarily

Virtual memory

- Cycle pages of the program from disk into memory only when they need to be executed
- The page that is executing in memory together with the pages in memory that have recently been executed is the program's *working set*
- As the program progresses, pages enter and leave the working set

Figure 9.12


Disk
J1
P0
P1
P2
P3
P4
P5
P6
P7
P8
P9
J2
P0
P1
J3
P0
P1
P2
P3
Page tables
J1
Frame Number
Loaded
0 4 Y
1 5 Y
2 6 Y
3 — N
4 — N
5 — N
6 — N
7 — N
8 — N
9 — N
J2
Frame Number
Loaded
0 1 Y
1 3 Y
J3
Frame Number
Loaded
0 2 Y
1 0 Y
2 — N
3 — N
Main memory
J3, P1 0
J2, P0 1
J3, P0 2
J2, P1 3
J1, P0 4
J1, P1 5
J1, P2 6
7
Frame table
Job
Dirty
0 3 Y
1 2 Y
2 3 N
3 2 Y
4 1 Y
5 1 N
6 1 N
7 — —

Page tables

- One page table for each program
- Converts logical address to physical address as in paging
- Loaded bit is 1 if the page is in memory
- A *page fault* occurs if the program needs to read or write a page that is not in memory
  - Page is loaded into an empty frame
  - If no empty frames, then a page is replaced

Frame tables

- One frame table with an entry for each frame
- Dirty bit initialized to 0 when page is first loaded into memory
- Set to 1 on a `STWr` to the frame, not on a `LDWr` from the frame
- When a page is replaced it is written back to disk only if the dirty bit is set to 1

Frame allocation

- Before a job starts executing, how many frames should be allocated for that job?
- System can allocate frames proportional to the physical size of the code

Page replacement

- First in, first out (FIFO)
  - On a page fault, select the page to be replaced as the one that first entered the set of loaded pages
- Least recently used (LRU)
  - On a page fault, select the page to be replaced as the one that was least recently read from or written to

First In, First Out (FIFO)

Page references		6	8	3	8	6	0	3	6	3	5	3	6
Loaded pages	—	6	8	3	3	3	0	0	6	6	5	3	3
	—	—	6	8	8	8	3	3	0	0	6	5	5
	—	—	—	6	6	6	8	8	3	3	0	6	6
Page fault		F	F	F			F		F		F	F	

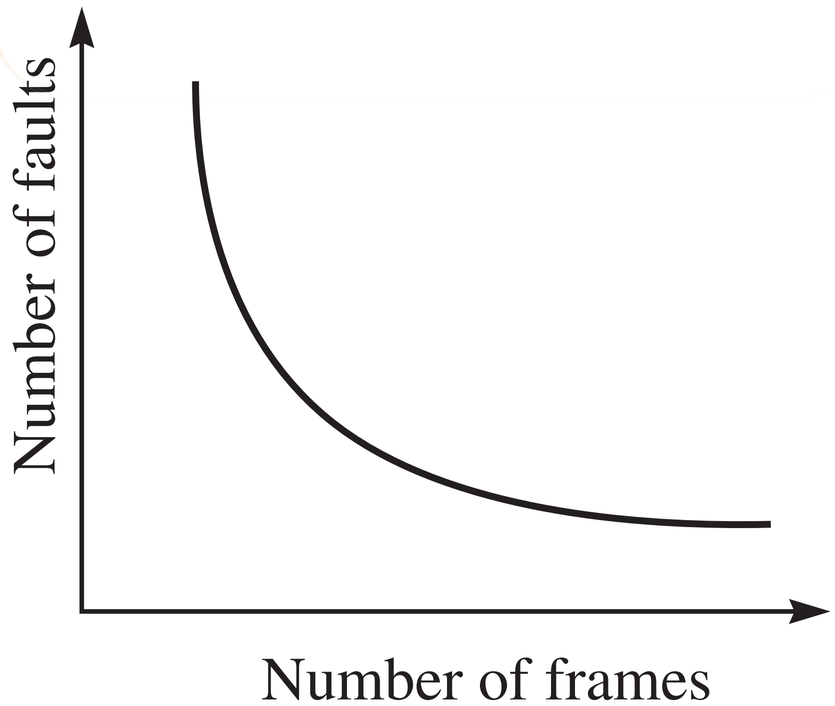
First In, First Out (FIFO)

Page references		6	8	3	8	6	0	3	6	3	5	3	6
Loaded pages	—	6	8	3	3	3	0	0	0	0	5	5	6
	—	—	6	8	8	8	3	3	3	3	0	0	5
	—	—	—	6	6	6	8	8	8	8	3	3	0
	—	—	—	—	—	—	6	6	6	6	8	8	3
Page fault		F	F	F			F				F		F

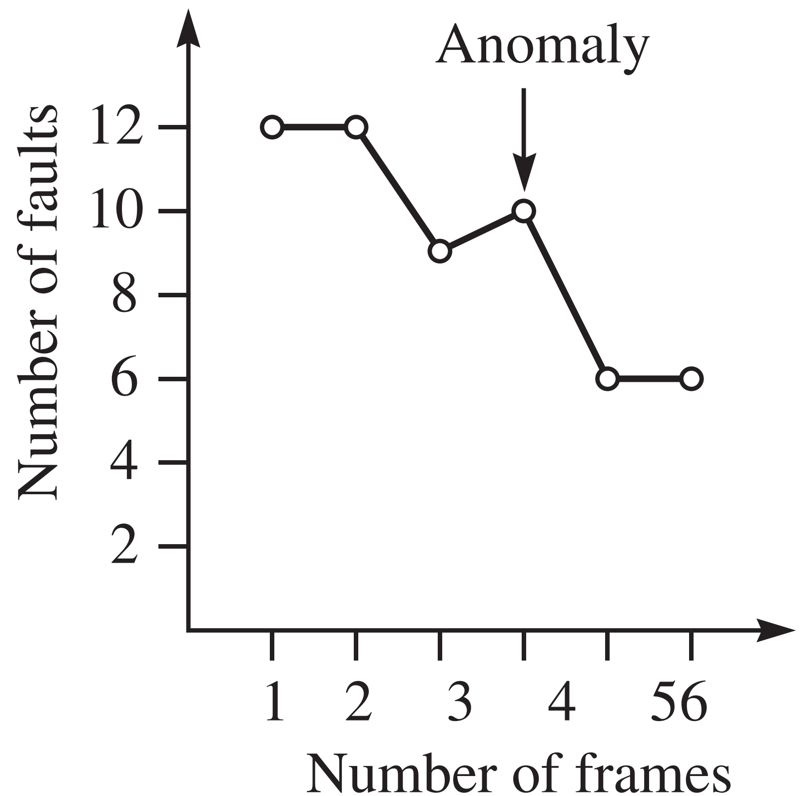
Bélády's anomoly

- In general, the greater the number of frames allocated to a program, the fewer the number of page faults
- In a few cases with FIFO, an increase in the number of frames increases the number of page faults
- Example page reference sequence

0, 1, 2, 3, 0, 1, 4, 0, 1, 2, 3, 4



(a) Expected effect of more frames on the number of page faults.



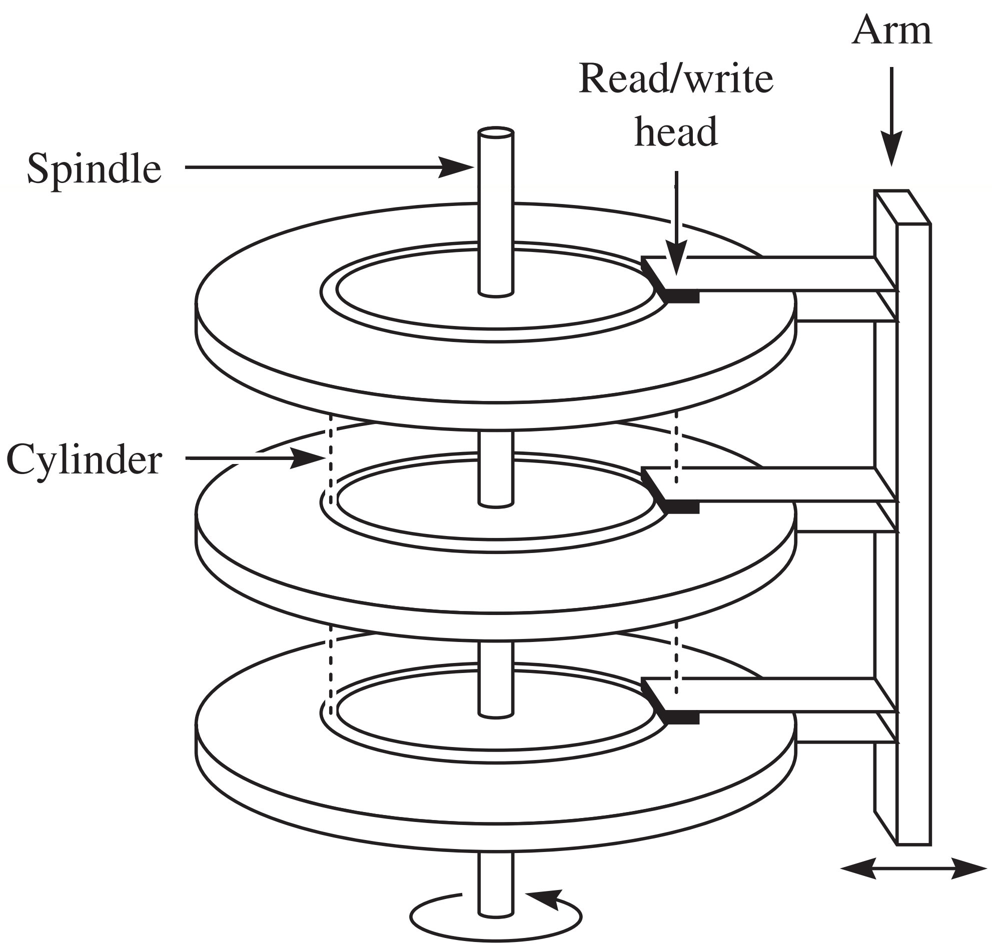
(b) Bélády's anomaly with the FIFO replacement algorithm.

Least Recently Used (LRU)

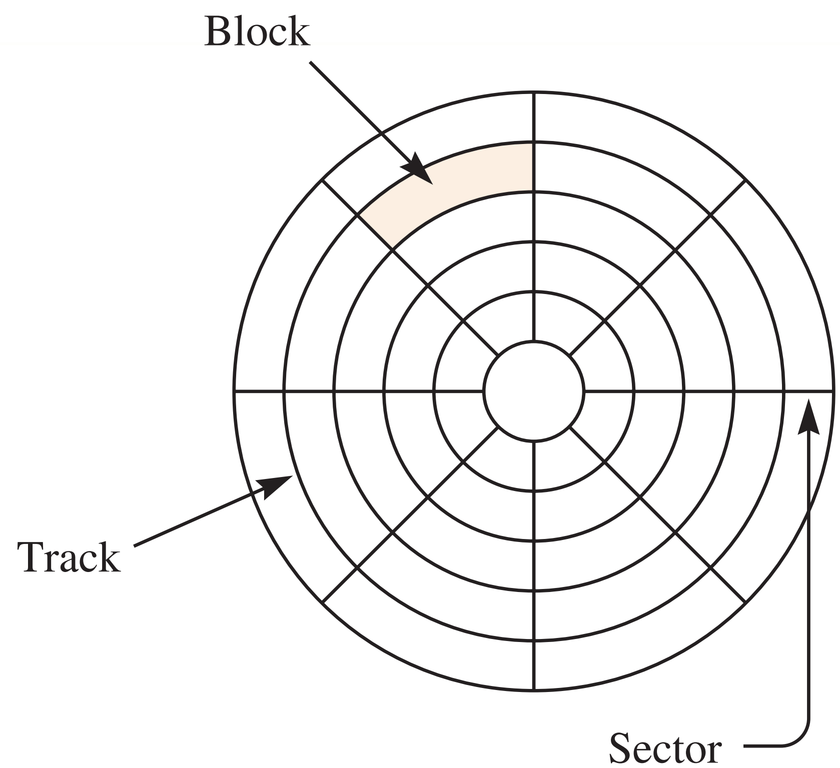
	6	8	3	8	6	0	3	6	3	5	3	6
—	6	8	3	8	6	0	3	6	3	5	3	6
—	—	6	8	3	8	6	0	3	6	3	5	3
—	—	—	6	6	3	8	6	0	0	6	6	5
	F	F	F			F	F			F		

File management

- Create a new file
- Delete a file
- Rename a file
- Open a file for editing
- Read the next data item from the file



(a) A hard disk drive.



(b) A single disk.

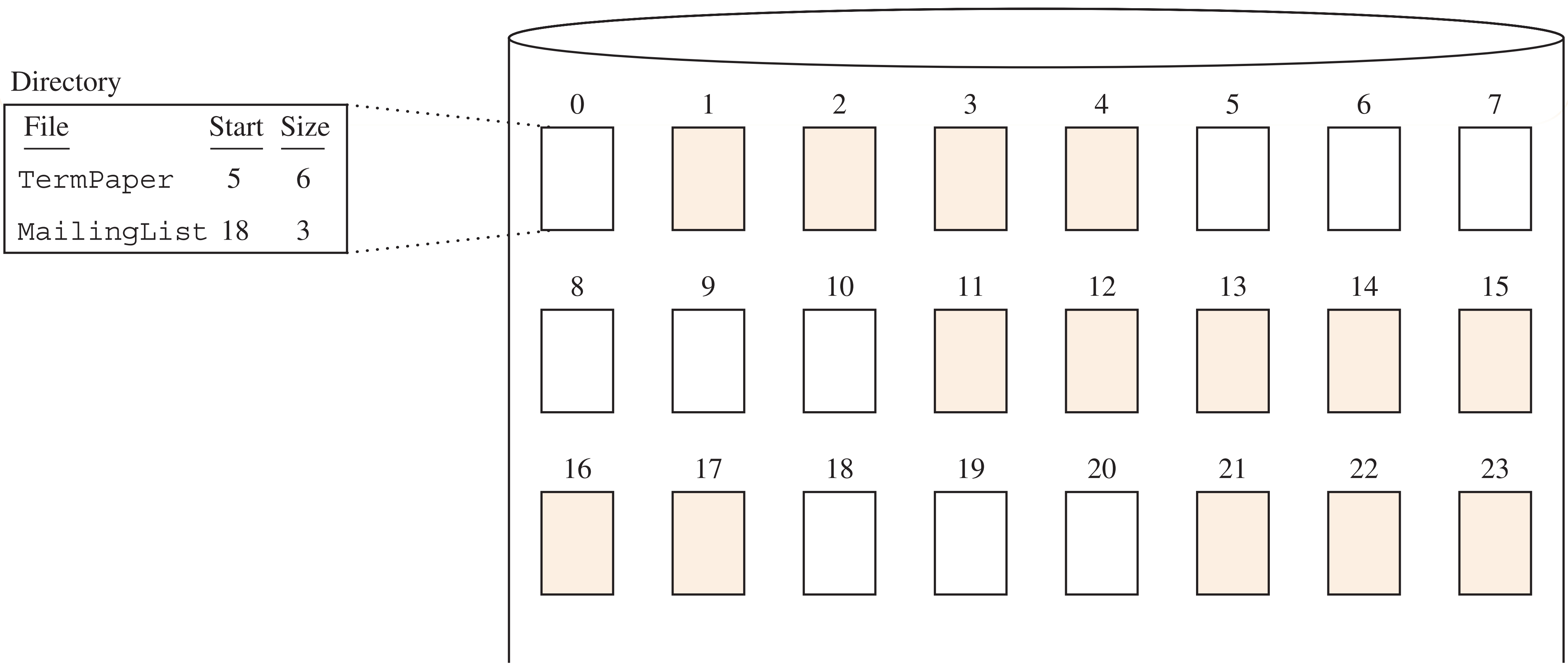
Contributions to the disk access time

- Seek time
  - Time for head to reach cylinder
- Latency
  - Time for start of sector to rotate to head
- Transmission time
  - Time for sector to pass under head

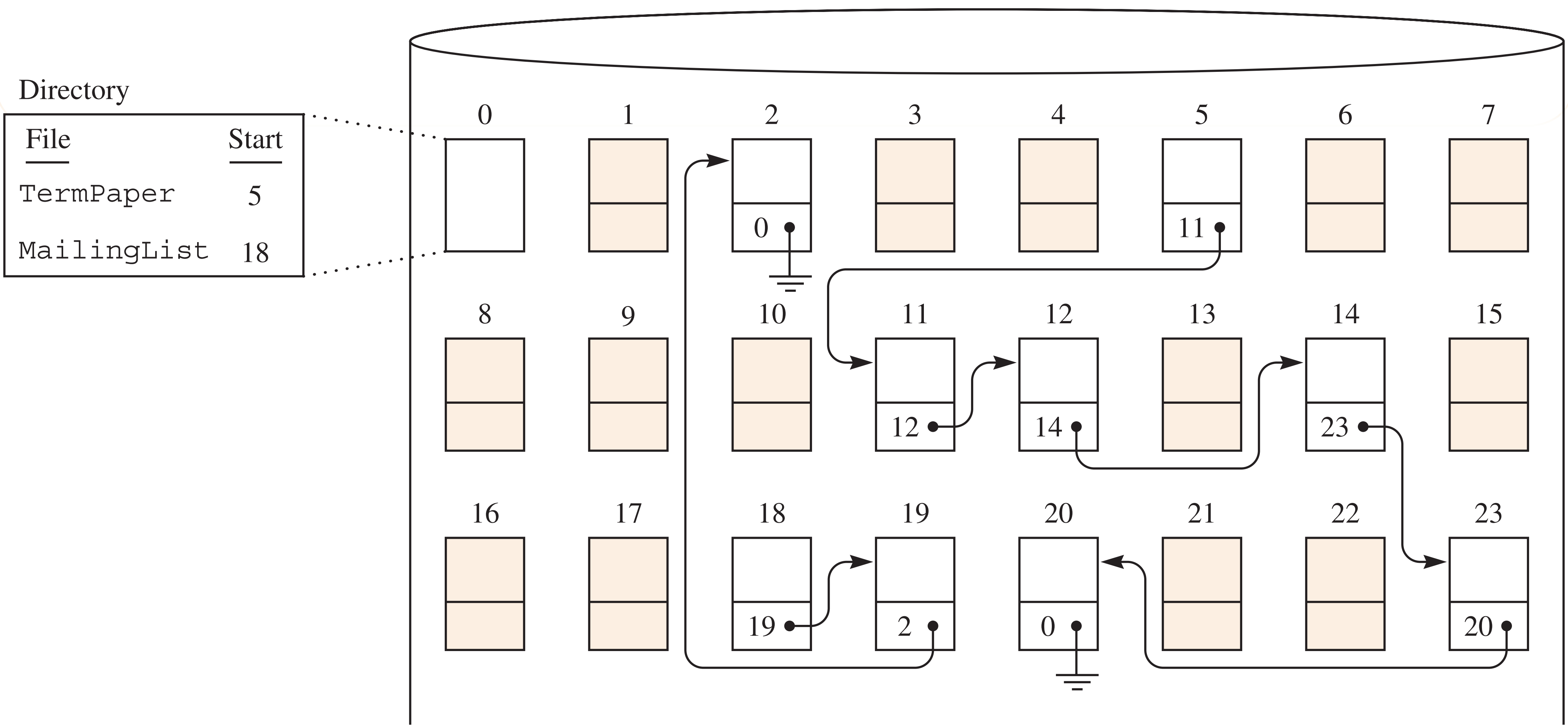
File allocation techniques

- Contiguous
- Linked
- Indexed

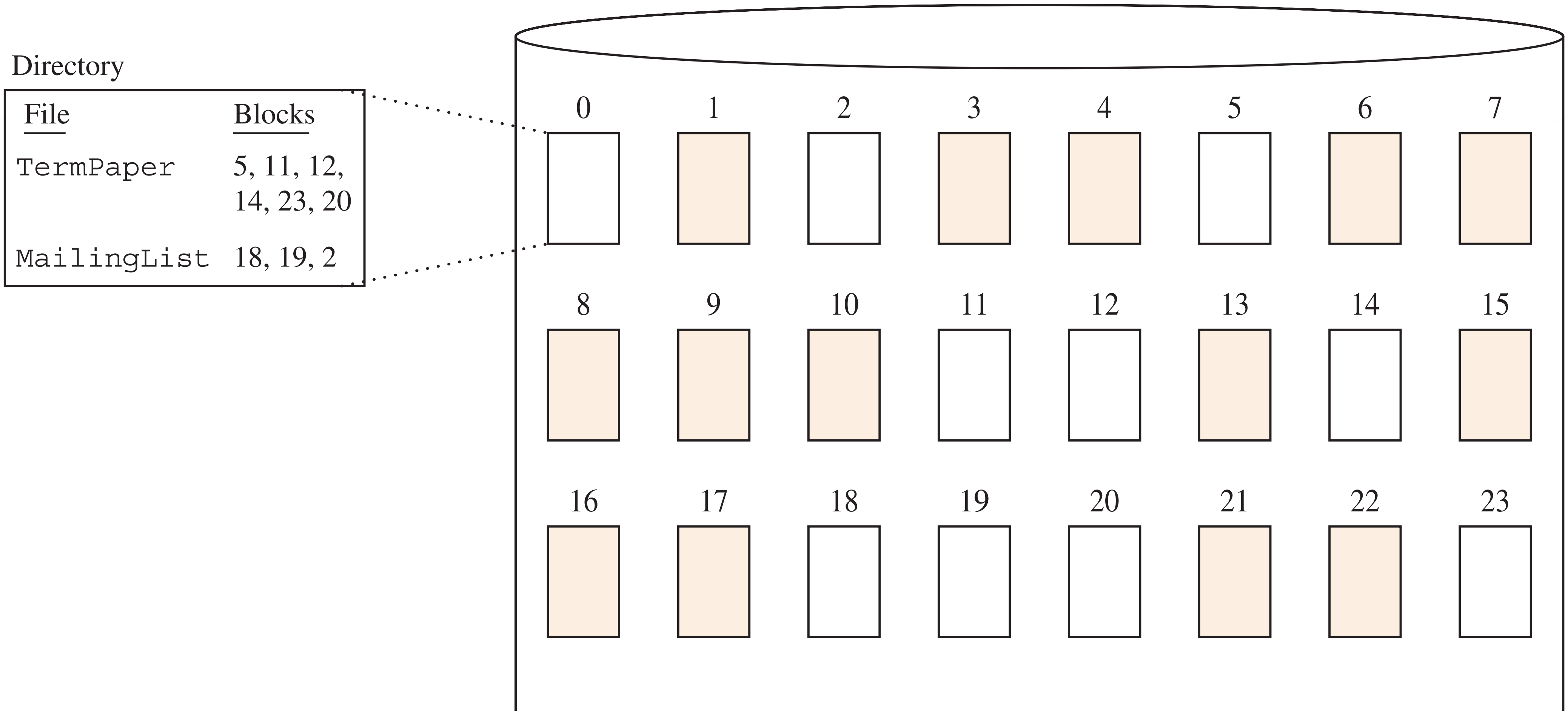
Contiguous allocation



Linked allocation



Indexed allocation



Physical errors

- Parity bits are redundant bits transmitted in addition to the data bits
- Two approaches to the error problem
  - Detect the error and retransmit or discard the received message
  - Correct the error

Codes

- Code
  - The set of data plus parity bit patterns that are sent
- Code word
  - An individual pattern from the code

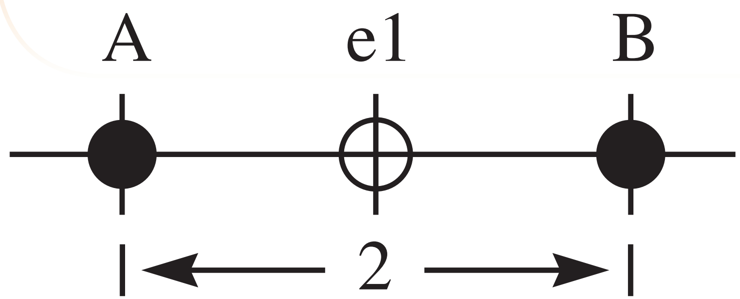
Code requirements

- The *Hamming distance* between two code words of the same length is the number of positions in which the bits differ
- The *code distance* is the minimum of the Hamming distance between all possible pairs of code words in the code

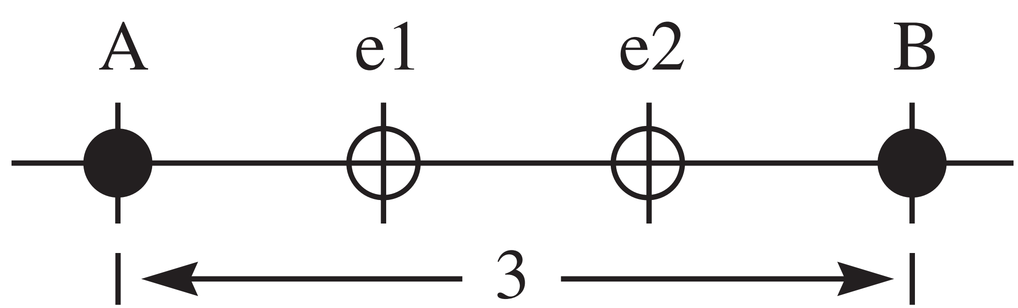
Error detection

- To detect d errors the code distance must satisfy the equation

$$\text{code distance} = d + 1$$



(a) Single-error detecting.

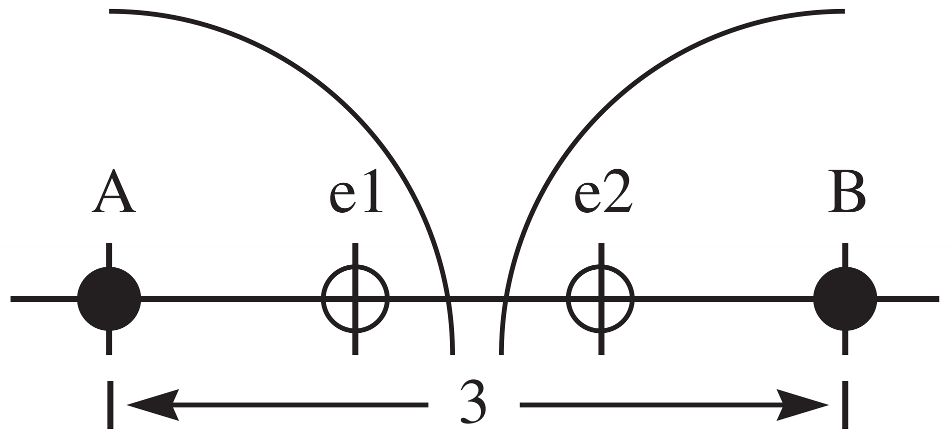


(b) Double-error detecting.

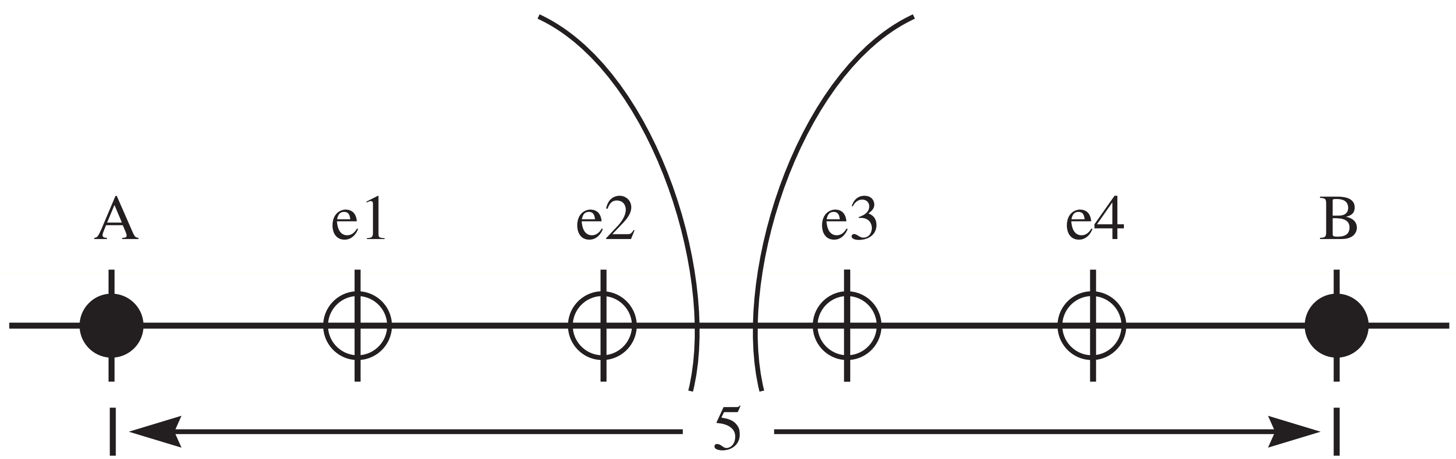
Error correction

- To correct d errors the code distance must satisfy the equation

$$\text{code distance} = 2\,d + 1$$



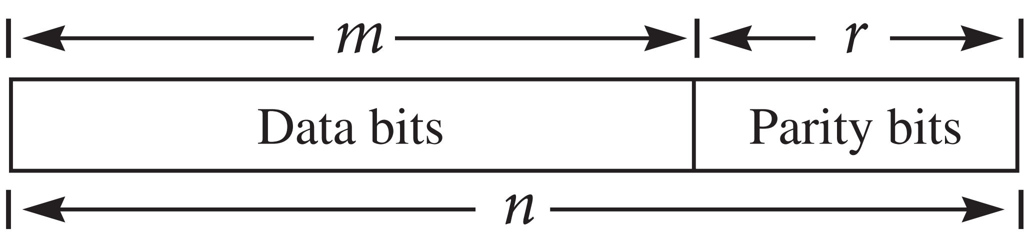
(a) Single-error correcting.



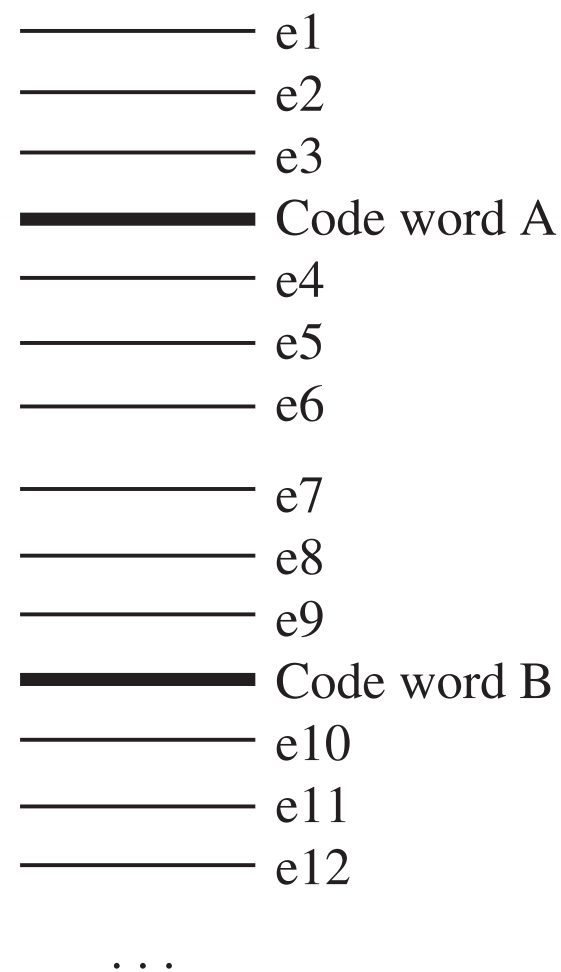
(b) Double-error correcting.

Single-error-correcting codes

- m data bits
- r parity bits
- $n = m + r$ total number of bits



(**a**) Code word structure.



(**b**) Grouping of received words with zero or one error.

Code requirement

- $m + r + 1 \leq 2^r$
- A *perfect code* is a code for which

  $m + r + 1 = 2^r$

Data Bits *m*	Parity Bits *r*	Percent Overhead
4	3	75
8	4	50
16	5	31
32	6	19
64	7	11
128	8	6

How to determine the parity bits

- Do not append the parity bits at the end of the code word
- Instead, distribute them throughout the code word
- Advantage: A single error can be corrected without having to compare the received word with every code word

Position of parity bits

1	2	3	4	5	6	7	8	9	10	11	12
		1		0	0	1		1	1	0	0

1 = 1	5 = 1 + 4	9 = 1 + 8
2 = 2	6 = 2 + 4	10 = 2 + 8
3 = 1 + 2	7 = 1 + 2 + 4	11 = 1 + 2 + 8
4 = 4	8 = 8	11 = 4 + 8

Parity bit 1 checks 1, 3, 5, 7, 9, 11

Parity bit 2 checks 2, 3, 6, 7, 10, 11

Parity bit 4 checks 4, 5, 6, 7, 12

Parity bit 8 checks 8, 9, 10, 11, 12

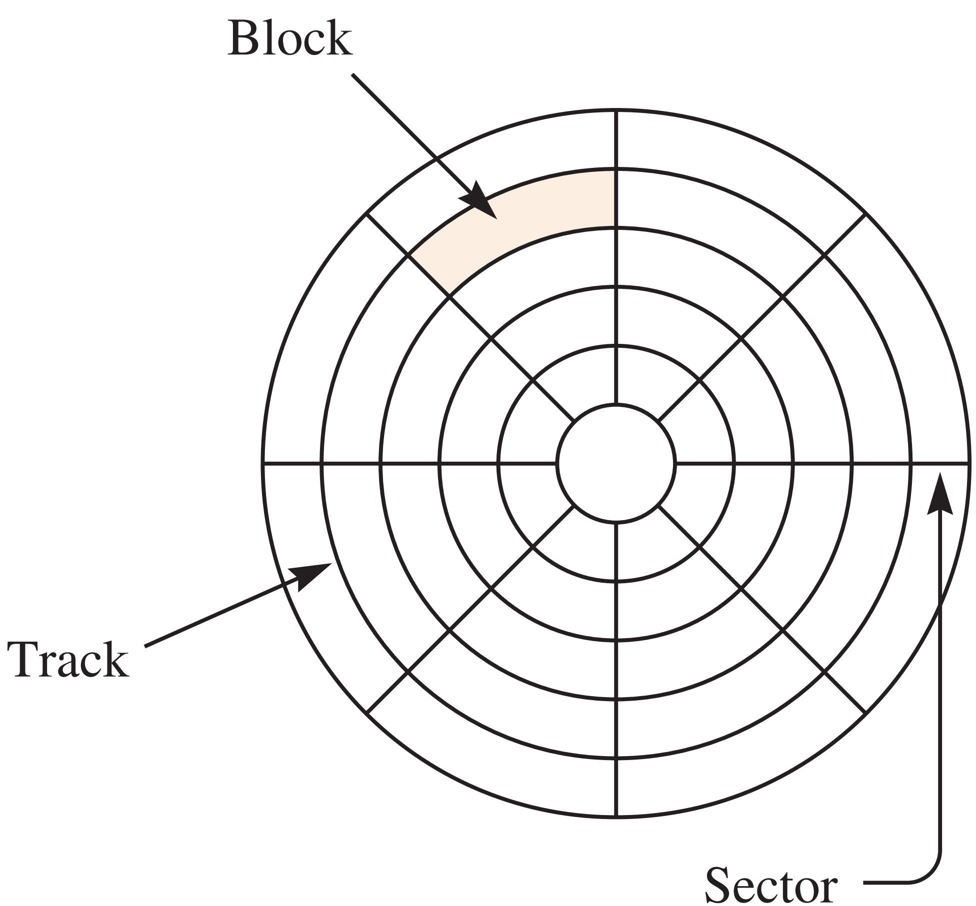
RAID

- Redundant Array of Inexpensive Disks
- RAID Levels

  0: Nonredundant striped
  1: Mirrored
  01, 10: Striped and mirrored
  2: Memory style ECC
  3: Bit-interleaved parity
  4: Block-interleaved parity
  5: Block-interleaved distributed parity

Stripes

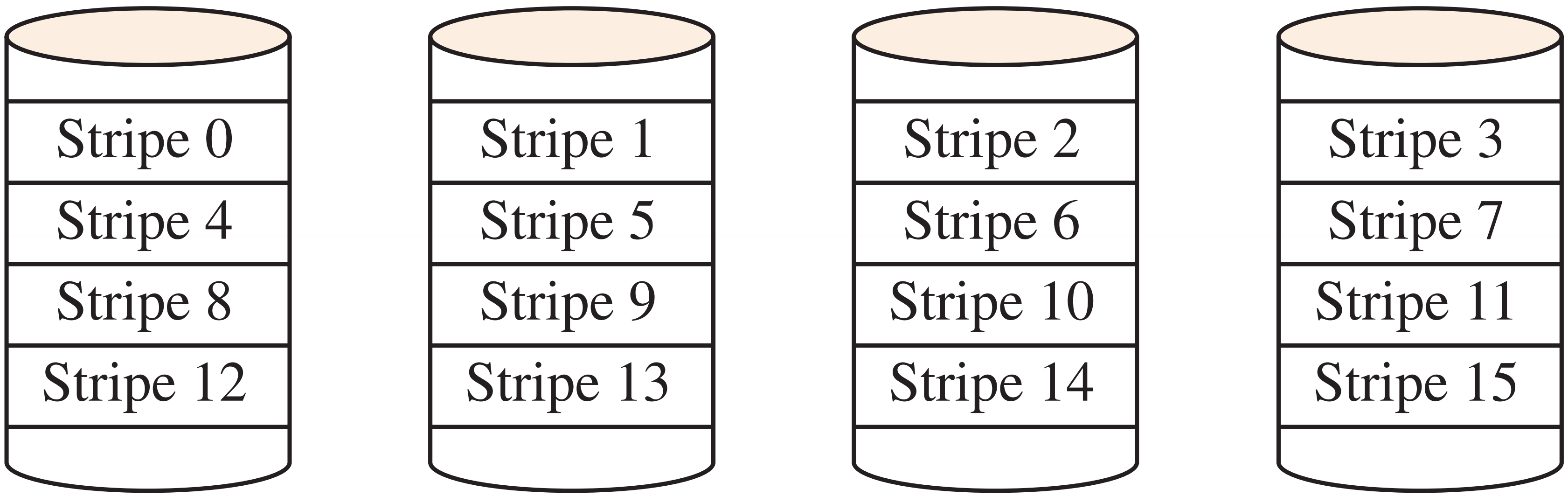
- A stripe consists of several blocks



RAID Level 0

- Nonredundant striped
- Not true RAID because there is no redundancy, therefore no enhanced reliability
- Advantage: Increased performance

RAID Level 0


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

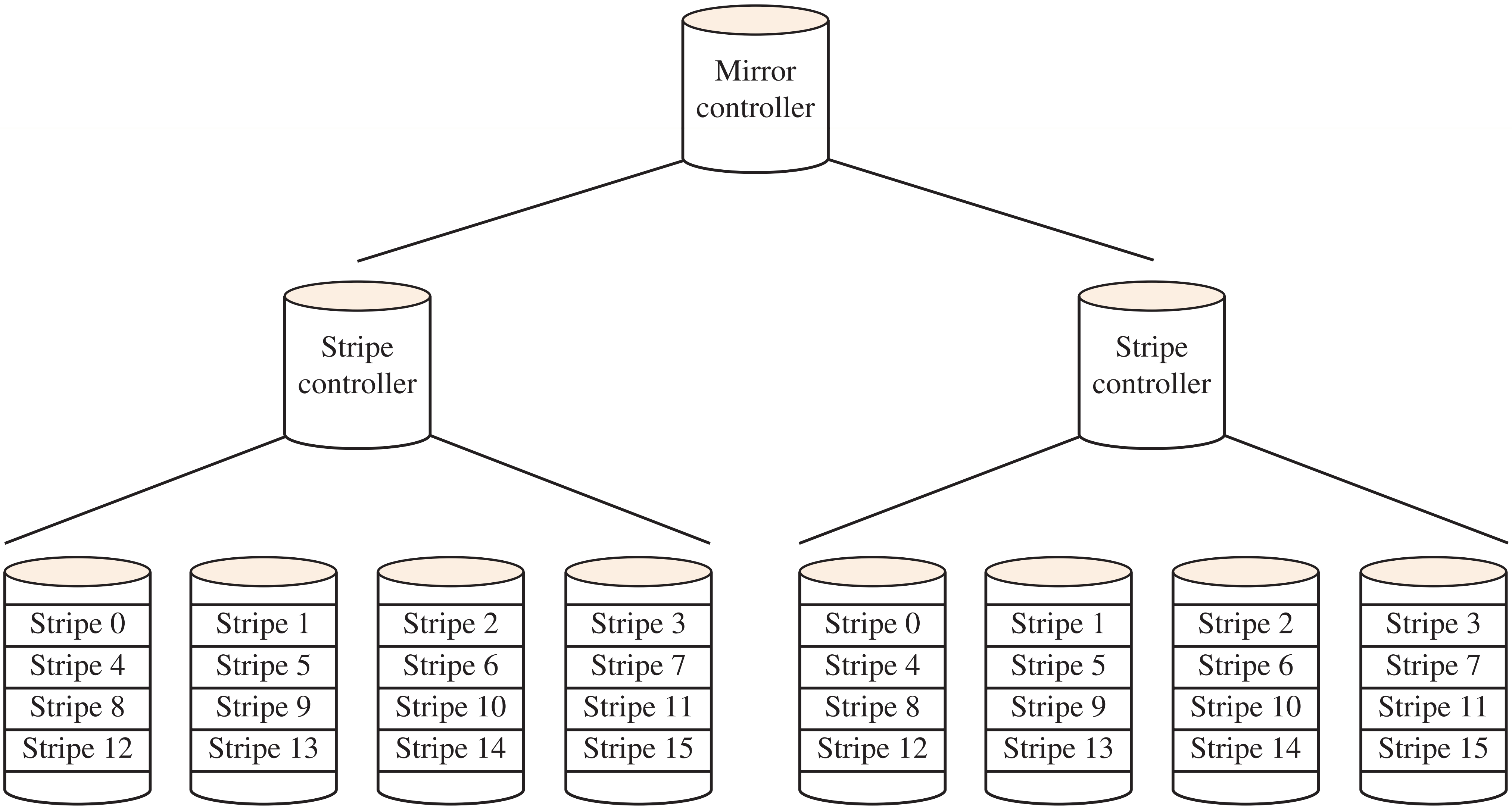
RAID Level 1

- Level 1: Mirrored
- An exact mirror image of each disk on a separate drive
- Disk writes are done in parallel to each drive
- Advantage: Reliability
- No performance hit, but no increase in perfomance either

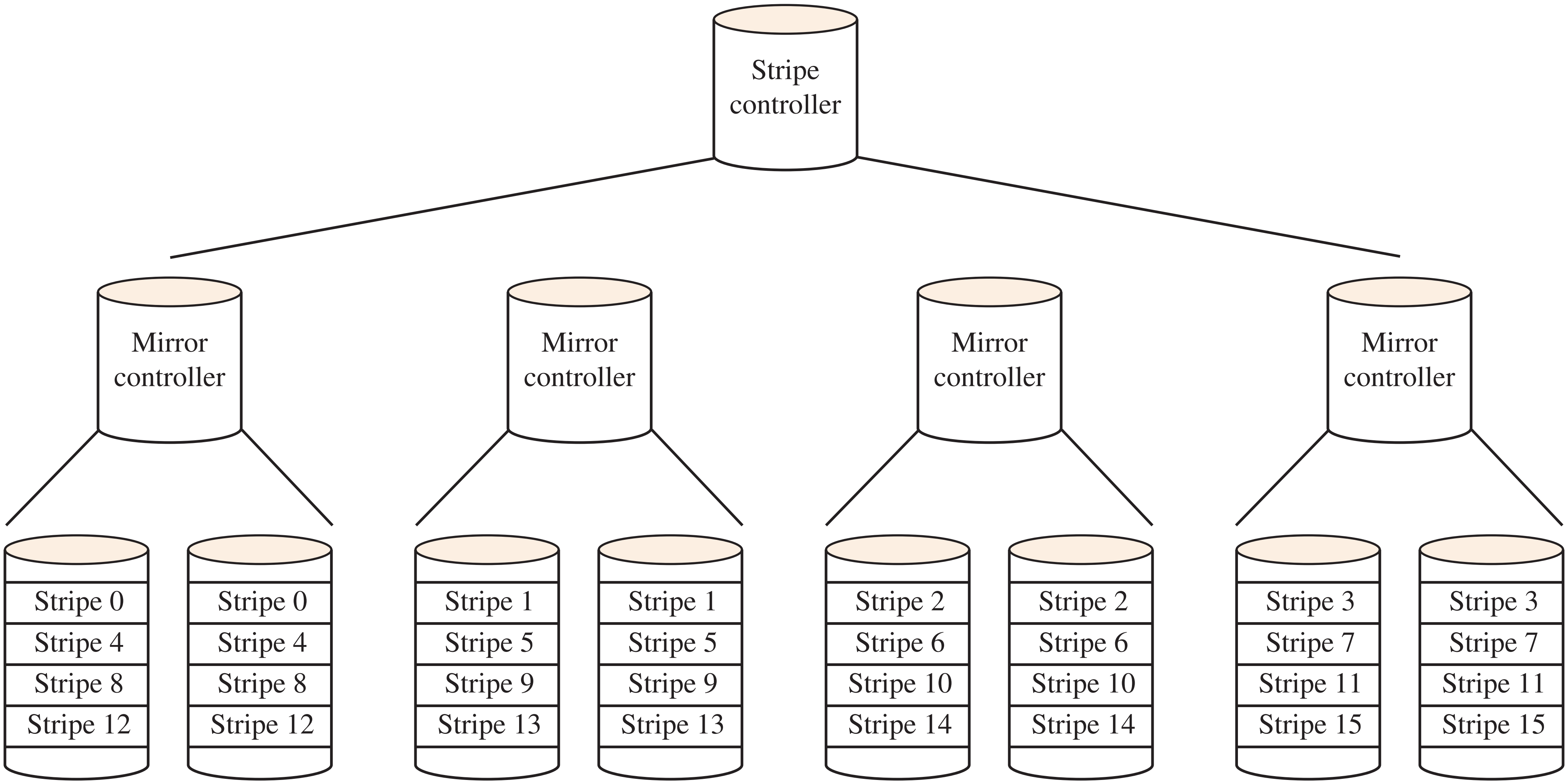
RAID Level 1

RAID Levels 01, 10

- Level 01: Mirrored stripes
- Level 10: Striped mirrors
- Both increased reliability and increased performance
- Advantage: Easy to hot-swap an entire drive
- Disadvantage: Huge storage overhead. Not as efficient as using parity to correct errors



(a) RAID level 01: Mirrored stripes.

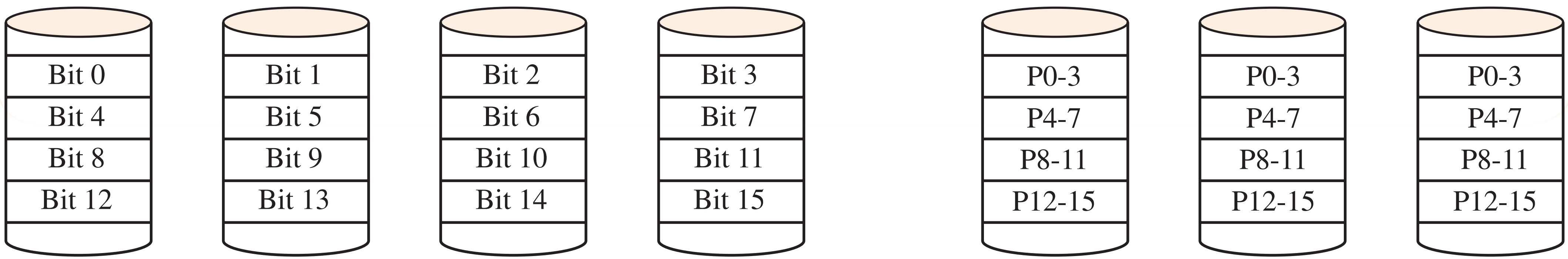


(b) RAID level 10: Striped mirrors.

RAID Level 2

- Memory-style ECC
- Error correction at the bit level
- Disks must be synchronized
- Used on older supercomputers with 32 data bits and 6 parity bits
- Modern disks have internal bit-level ECC, so Level 2 is no longer used commercially

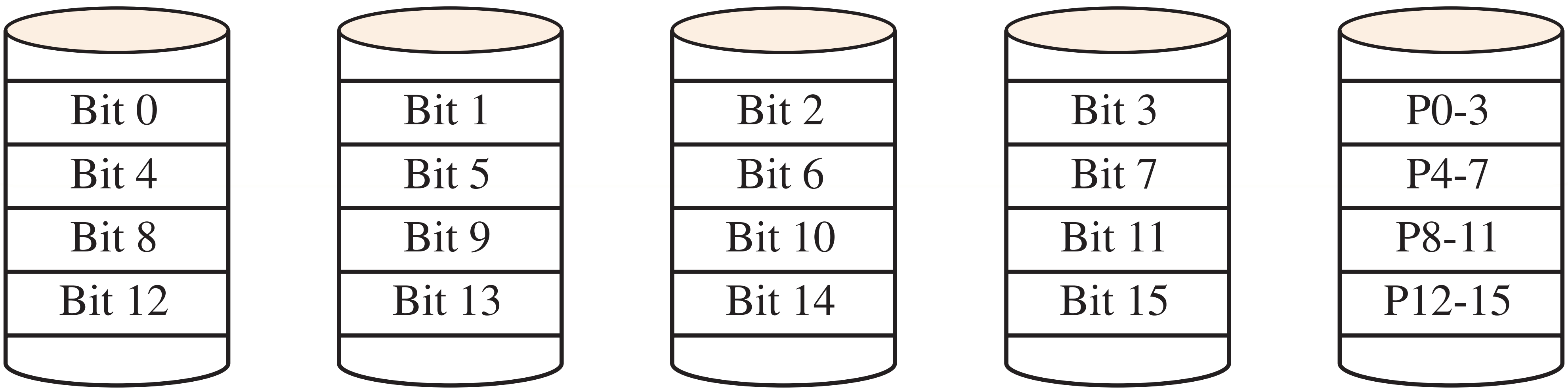
RAID Level 2


Bit 0
Bit 4
Bit 8
Bit 12
Bit 1
Bit 5
Bit 9
Bit 13
Bit 2
Bit 6
Bit 10
Bit 14
Bit 3
Bit 7
Bit 11
Bit 15
P0-3
P4-7
P8-11
P12-15
P0-3
P4-7
P8-11
P12-15
P0-3
P4-7
P8-11
P12-15

RAID Level 3

- Bit-interleaved parity
- When a drive fails, you know which disk drive it is
- Knowing which bit fails, it takes only *one* parity bit to *correct* the error
- More efficient use of disk space

RAID Level 3


Bit 0
Bit 4
Bit 8
Bit 12
Bit 1
Bit 5
Bit 9
Bit 13
Bit 2
Bit 6
Bit 10
Bit 14
Bit 3
Bit 7
Bit 11
Bit 15
P0-3
P4-7
P8-11
P12-15

RAID Level 3 disadvantages

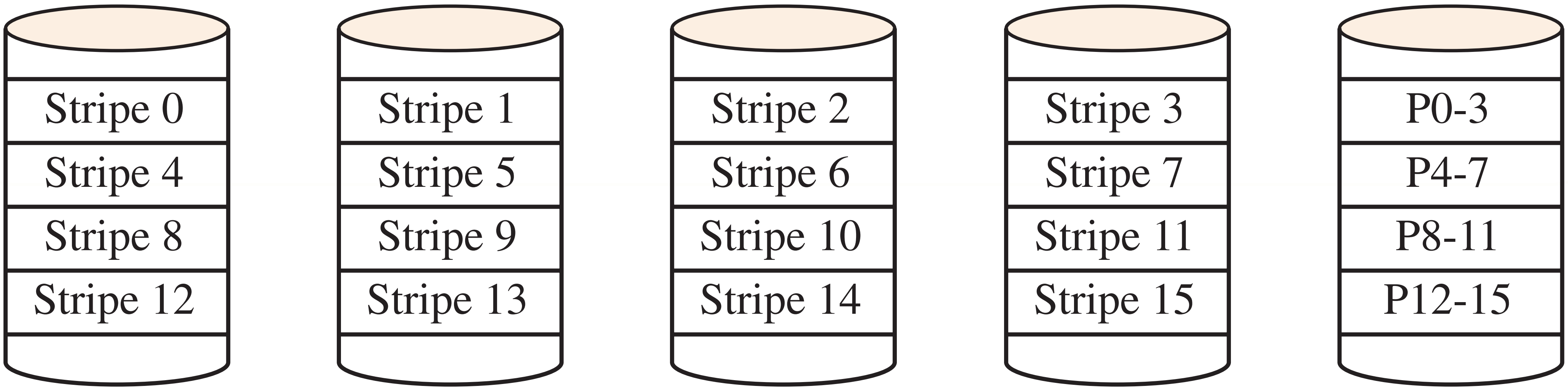
- To replace a failed drive, you must read *all* the drives to correct the error and compute the data in the substitute drive
- Every read/write request requires you to access every drive
- No concurrency, so no performance benefit

RAID Level 4

- Block-interleaved parity
- Example
  Stripe 0: Bits 0 through 1023
  Stripe 1: Bits 1024 through 2047
  Stripe 2: Bits 2048 through 3071
  Stripe 3: Bits 3072 through 4095

  Bit 1 of P0-3 checks 0, 1024, 2048, 3072
  Bit 2 of P0-3 checks 1, 1025, 2049, 3073
  ...

RAID Level 4


Stripe 0
Stripe 4
Stripe 8
Stripe 12
Stripe 1
Stripe 5
Stripe 9
Stripe 13
Stripe 2
Stripe 6
Stripe 10
Stripe 14
Stripe 3
Stripe 7
Stripe 11
Stripe 15
P0-3
P4-7
P8-11
P12-15

Level 4 advantages

- Striping is not at the bit level, so rotational synchronization is not required
- Better performance than Level 3 for small random read requests
  - With Level 4, many small files on different disks allow for concurrency
  - With Level 3, many small files must be read sequentially

Level 4 disk write without shortcut

- To write to stripe 0:
  - Read stripes 1, 2, 3
  - Compute parity with stripe 0
  - Write to stripe 0 and P0-3
- Result is *three* reads and *two* writes

Level 4 disk write with shortcut

- To write to stripe 0:
  - Read old stripe 0 and old stripe P0-3
  - If the new stripe 0 bit differs from the old stripe 0 bit, flip the corresponding bit in P0-3
  - Write to stripe 0 and P0-3
- Result is *two* reads and *two* writes

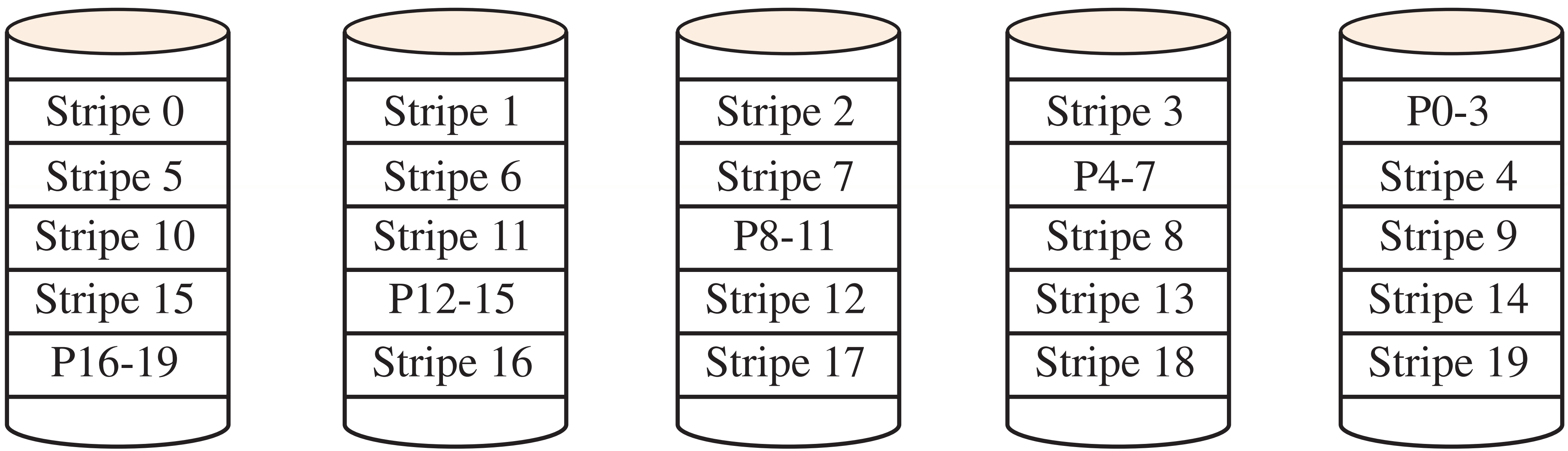
Level 4 disadvantage

- With or without the shortcut, every write request must write to the parity disk
- The parity disk becomes the performance bottleneck

RAID Level 5

- Block-interleaved distributed parity
- Rather than store all the parity on one disk, the parity information is scattered among all the disks
- No one disk has the responsibility for the parity information of the whole array
- Better performance with reliability maintained

RAID Level 5


Stripe 0
Stripe 5
Stripe 10
Stripe 15
P16-19
Stripe 1
Stripe 6
Stripe 11
P12-15
Stripe 16
Stripe 2
Stripe 7
P8-11
Stripe 12
Stripe 17
Stripe 3
P4-7
Stripe 8
Stripe 13
Stripe 18
P0-3
Stripe 4
Stripe 9
Stripe 14
Stripe 19