

INF239 Sistemas Operativos

MODERN OPERATING SYSTEMS

Fourth Edition

by Andrew S. TANENBAUM, Herbert BOS

3 MEMORY MANAGEMENT

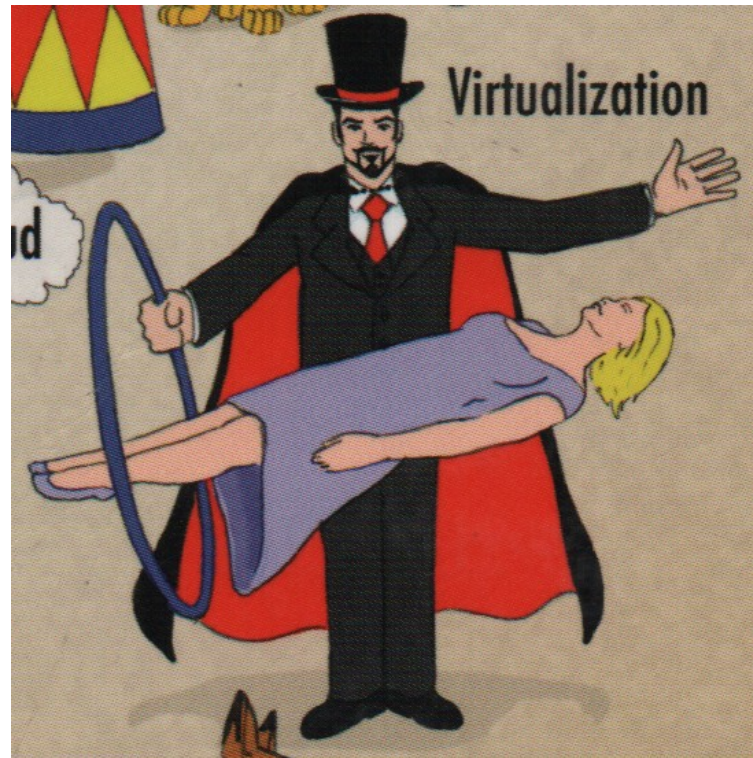
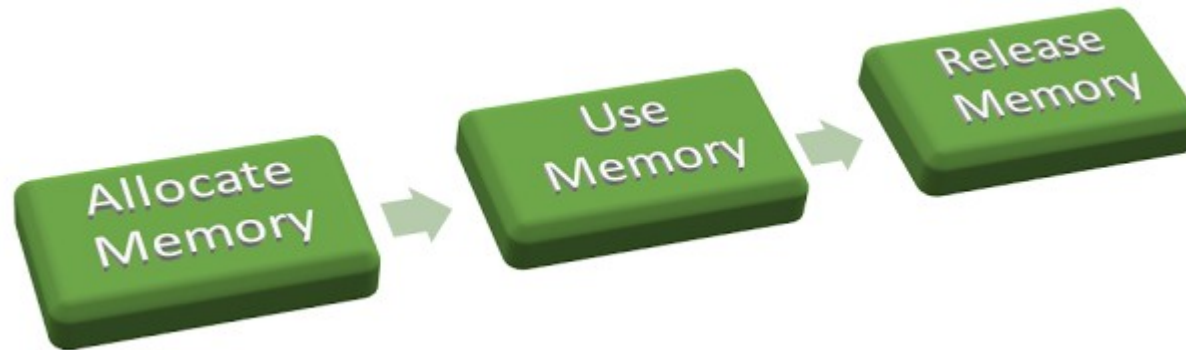
Clase 9

Viktor Khlebnikov

Ingeniería Informática
Departamento de Ingeniería
Pontificia Universidad Católica del Perú

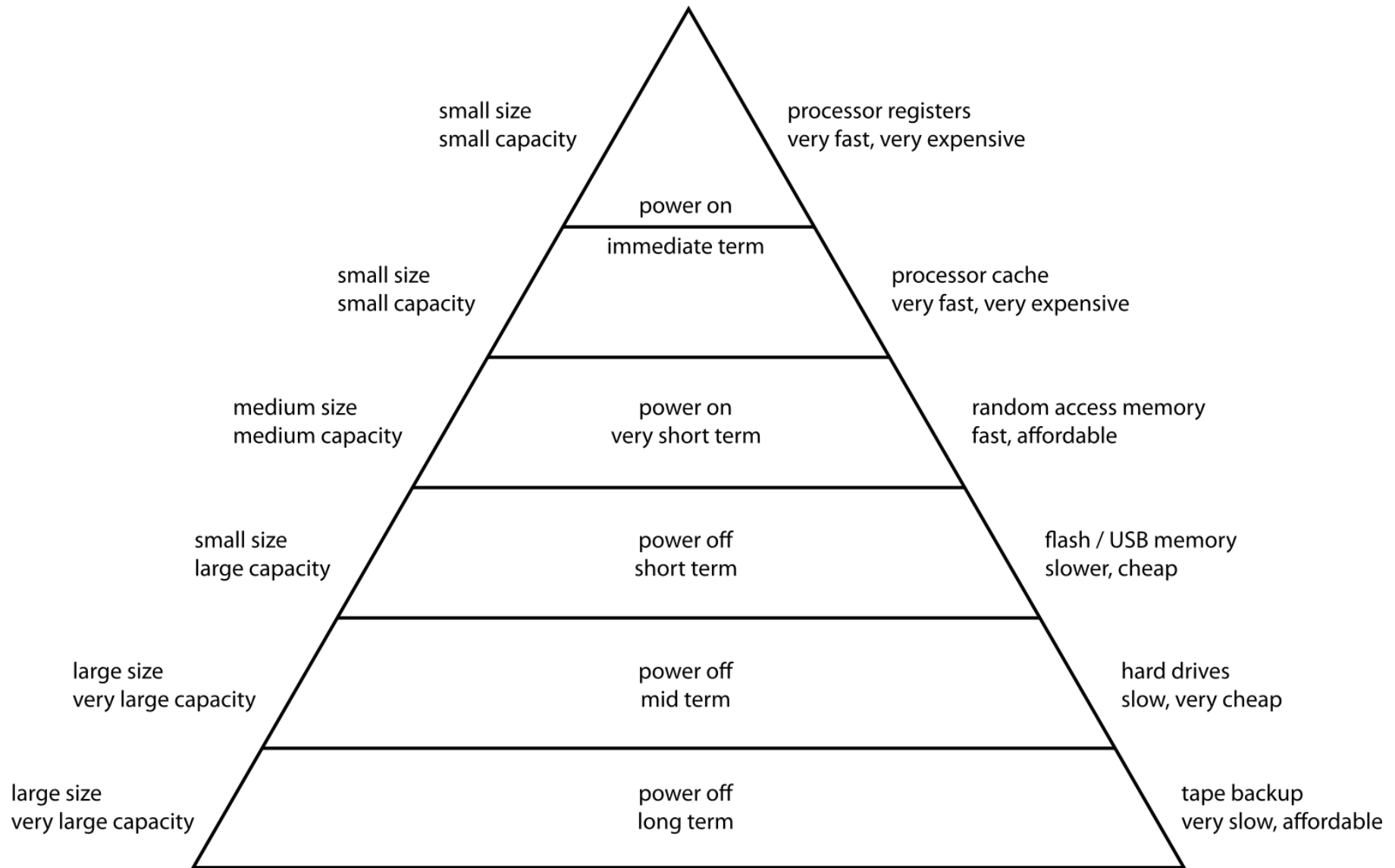
2020-2

Memory Management. Part 1



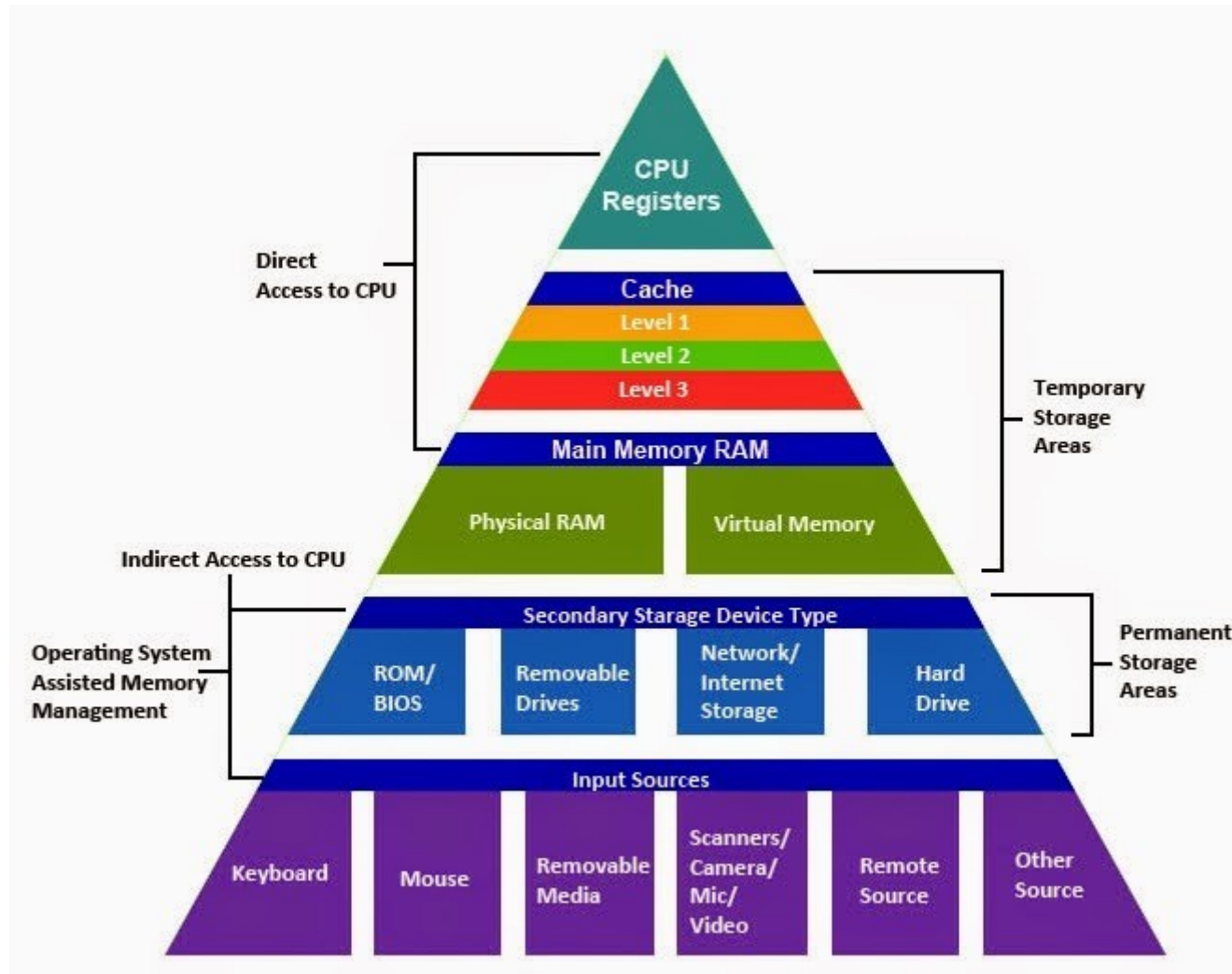
MOS4E

Computer Memory Hierarchy



https://en.wikipedia.org/wiki/Memory_hierarchy

Memory Hierarchy



Cache	Hit cost	Size
1st level cache / first level TLB	1 ns	64 KB
2nd level cache / second level TLB	4 ns	256 KB
3rd level cache	12 ns	2 MB
Memory (DRAM)	100 ns	10 GB
Data center memory (cooperative caching)	100 μ s	100 TB
Local non-volatile flash memory	100 μ s	100 GB
Local disk	10 ms	1 TB
Data center disk	10 ms	100 PB
Remote data center disk	200 ms	1 XB

T. Anderson, M. Dahlin. *Operating Systems. Principles and Practice (2E)*. 2014

Memory Manager

The part of the operating system that manages (part of) the memory hierarchy is called the **memory manager**.

Its job is to efficiently manage memory:

keep track of which parts of memory are **in use**,

allocate memory to processes when they need it, and

deallocate it when they are done.

Since managing the lowest level of cache memory is normally done by the hardware, the focus of this chapter will be on the programmer's model of main memory and how it can be managed.

The abstractions for, and the management of, permanent storage – the disk – are the subject of the next chapter.

We will first look at the simplest possible schemes and then gradually progress to more and more elaborate ones.

3.1 No Memory Abstraction

**Early mainframe computers (before 1960),
early minicomputers (before 1970),
early personal computers (before 1980),
embedded and smart card systems (till now).**

Every program simply **saw the physical memory.**

MOV REGISTER1, 1000



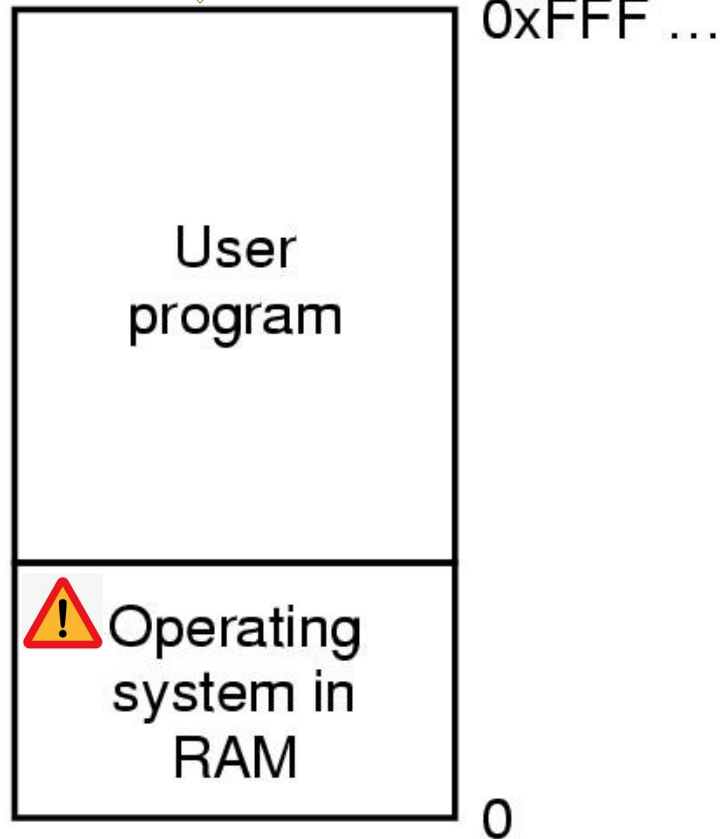
from the physical memory location 1000 to REGISTER1

Thus, the model of memory presented to the programmer was simply physical memory, a set of addresses from 0 to some maximum, each address corresponding to a cell containing some number of bits, commonly eight.

Under these conditions it was not possible to have two running programs in memory at the same time. If the first program wrote a new value to, say, location 2000, this would erase whatever value the second program was storing there. Nothing would work and both programs would crash almost immediately.

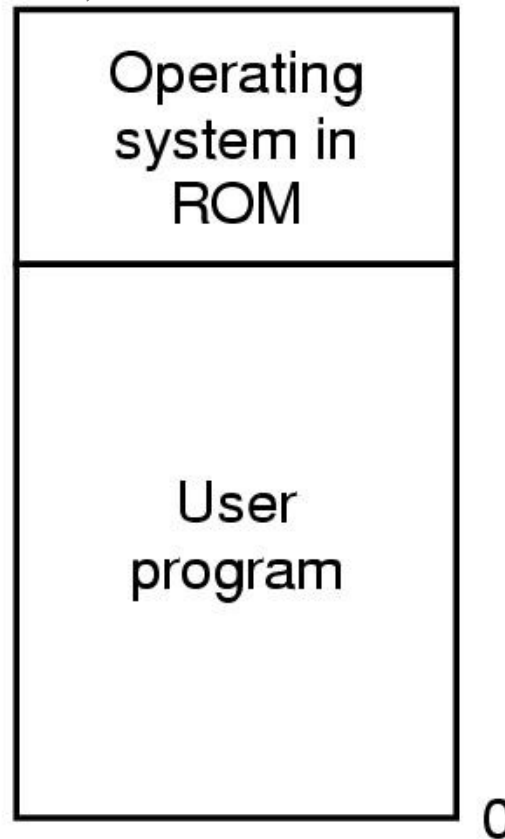
No Memory Abstraction

formerly used on mainframes
and minicomputers



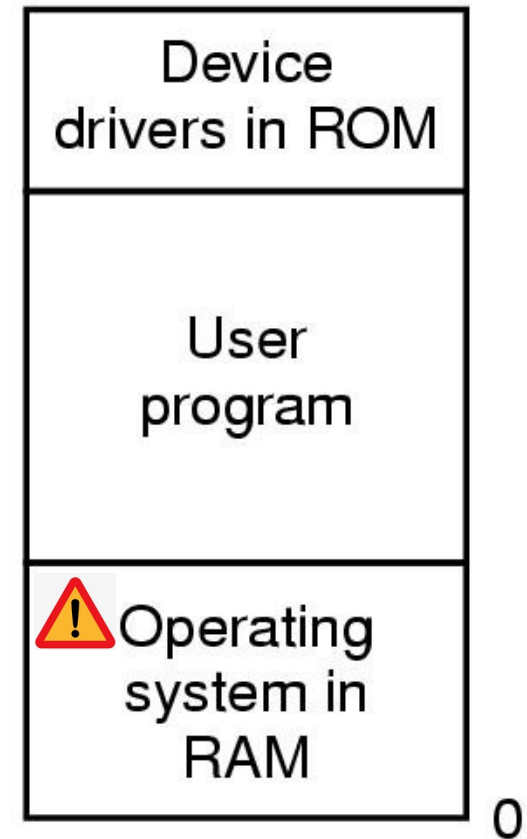
(a)

used on embedded
systems



(b)

used by early
personal computers



(c)

Figure 3-1. Three simple ways of organizing memory with an operating system and **one user process.**

When the system is organized in this way, generally only one process at a time can be running. As soon as the user types a command, the operating system copies the requested program from disk to memory and executes it.

When the process finishes, the operating system displays a prompt character and waits for a user new command.

When the operating system receives the command, it loads a new program into memory, overwriting the first one.

Altair 8800 microcomputer

**Altair 8800 Computer
with 8-inch floppy disk
system**

Developer MITS

Manufacturer MITS

Release date
January 1975;
45 years ago

Introductory price
Kit price 439 US\$
(\$2100 in 2019)

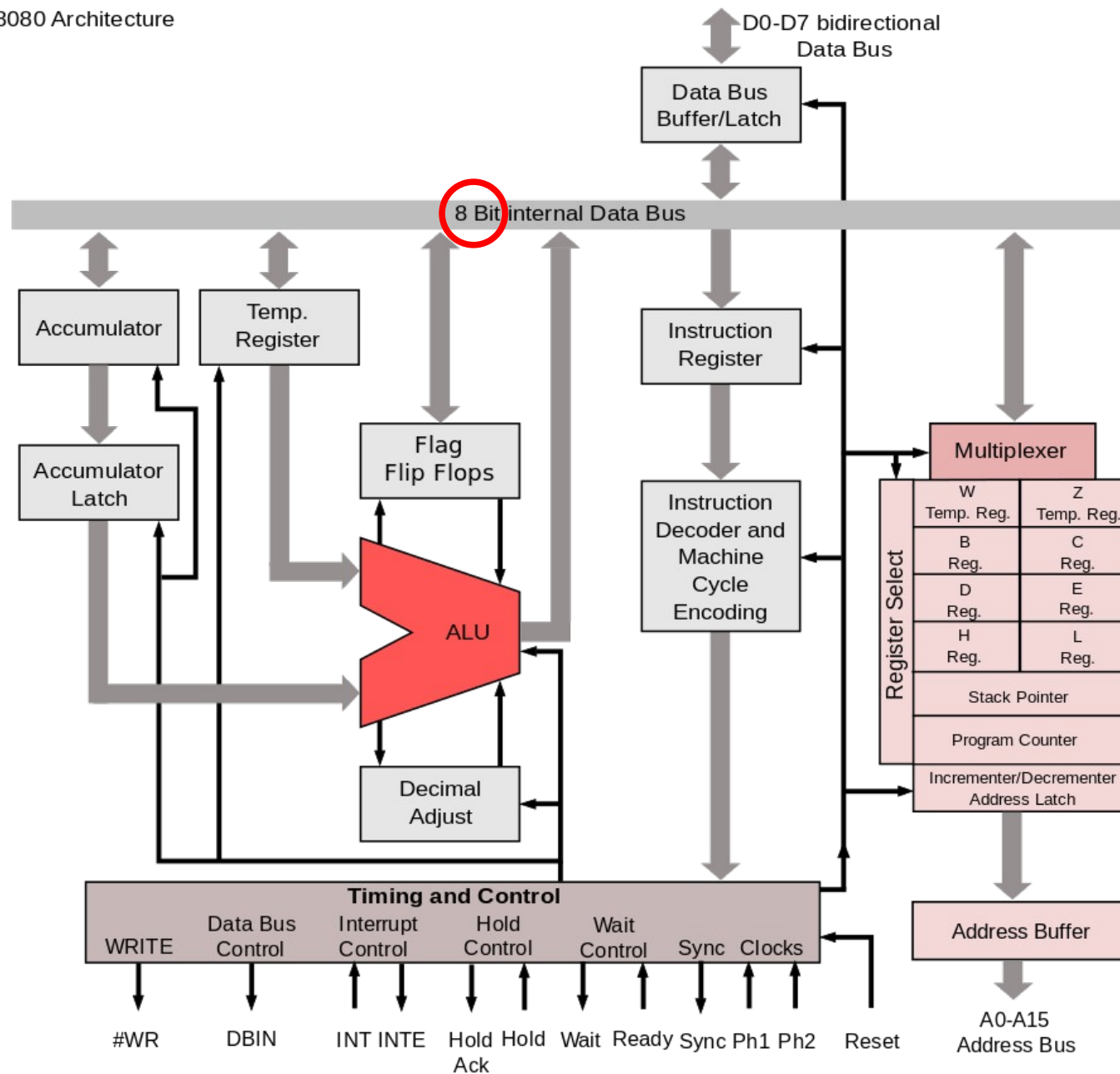
Assembled
621 US\$
(\$3000 in 2019)

CPU
Intel 8080 @ 2 MHz

256 bytes +
1K, 4K memory boards



https://en.wikipedia.org/wiki/Altair_8800



16 bits: $2^{16} = 64K$

https://en.wikipedia.org/wiki/Intel_8080

Gary Kildall

**“Gary did make a difference.
He was a genius and a gentleman.
A rare combination.
Gary did a lot of money but he was
driven by an honest desire to create
new ideas that could expand the
human potential.” (Gary Kildall Special)**



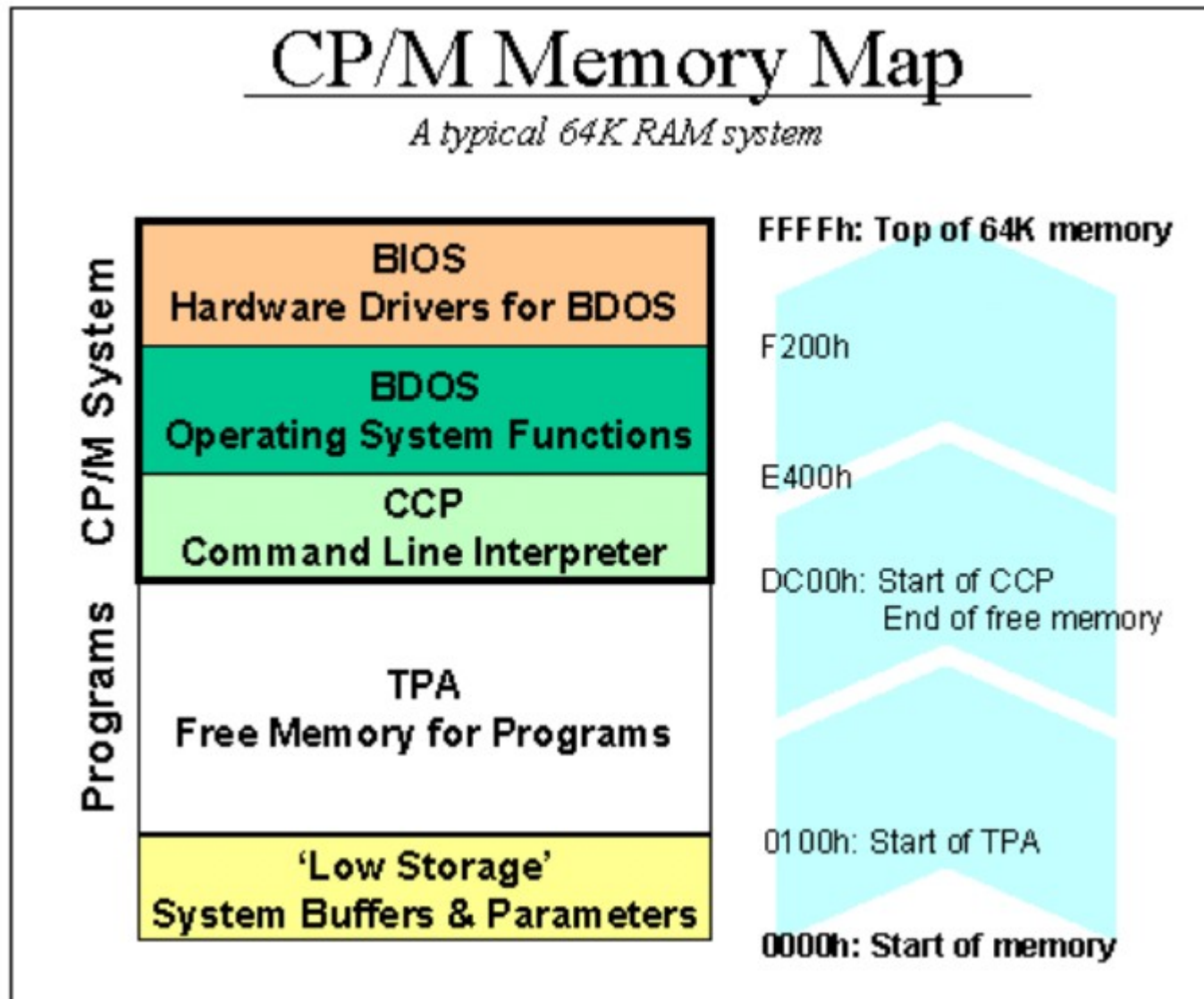
PL/M – the first high-level programming language for microprocessors.

Operating systems:

CP/M (1974), MP/M (1979), CP/M-86 (1981).

The concept of a BIOS.

$$2^{16} = 64 \text{ KB}$$



```
CP/M Plus for MSX2 rel. 818787  
(c) 1982 Digital Research, (c) 1987 RVS Datentechnik
```

```
AD-COLOR 15,4,4,12
```

```
AD-SET [CREATE=ON, UPDATE=ON]
```

```
Label for drive A:
```

Directory Label	Parade Read	Stamp Create	Stamp Access	Stamp Update
A:CPM3DOS .	off	on	off	on

```
AD-SHOW A:
```

```
A: RM, Space: 80k
```

```
AD-LANGUAGE 2
```

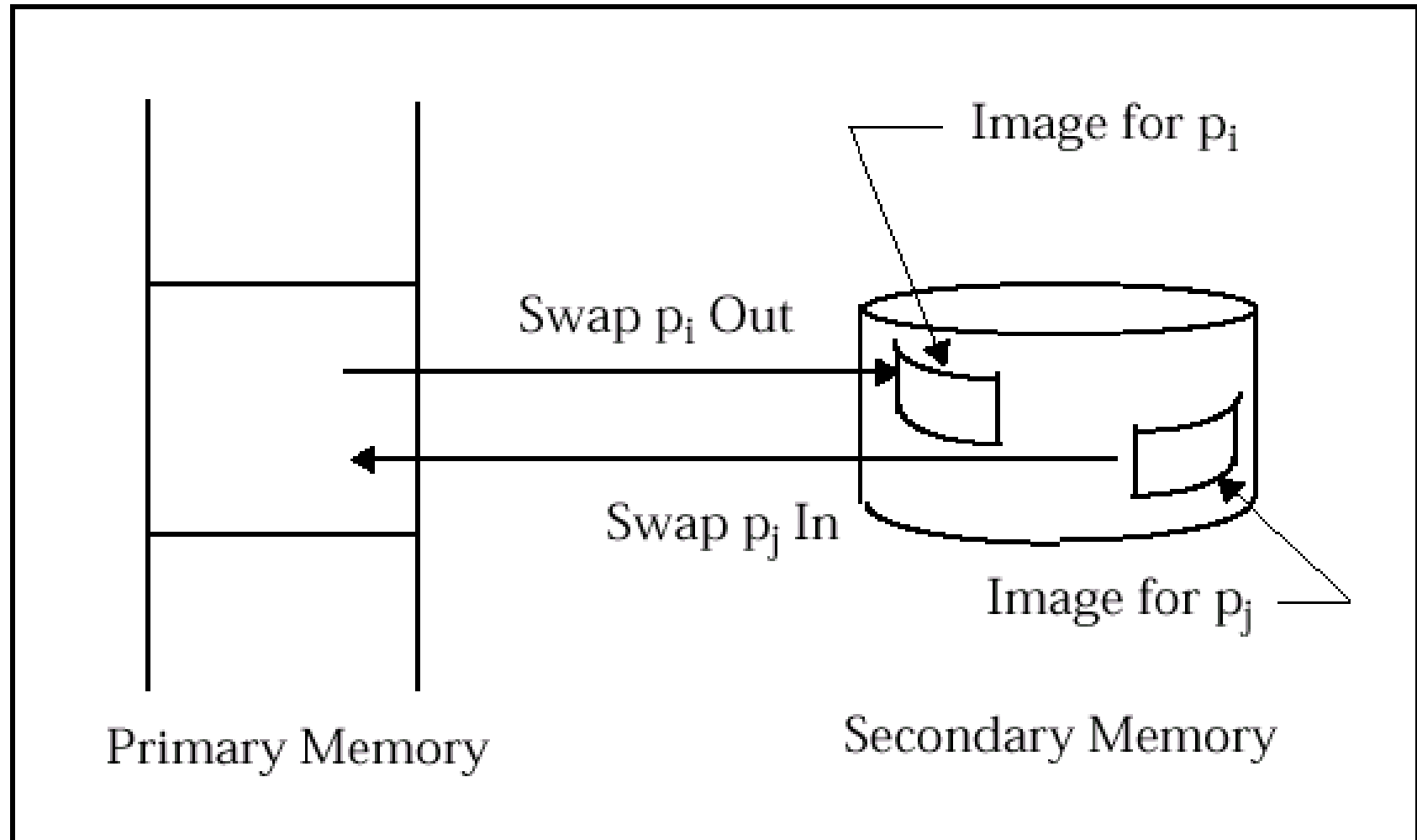
```
[...]  
/* C P / M   B A S I C   I / O   S Y S T E M   ( B I O S )  
                C O P Y R I G H T   ( C )   G A R Y   A .   K I L D A L L  
                        J U N E ,   1 9 7 5                               */  
[...]  
/*   B A S I C   D I S K   O P E R A T I N G   S Y S T E M   ( B D O S )  
                C O P Y R I G H T   ( C )   G A R Y   A .   K I L D A L L  
                        J U N E ,   1 9 7 5                               */  
[...]
```


Running Multiple Programs Without a Memory Abstraction

However, even with no memory abstraction, it is possible to run multiple programs at the same time. What the operating system has to do is save the entire contents of memory to a disk file, then bring in and run the next program.

As long as there is only one program at a time in memory, there are no conflicts.

This concept (swapping**) will be discussed below.**



GN, OSAMP2E, Fig. 11-16

Memory Protection

Protection keys:

A memory protection key (MPK) mechanism divides physical memory into blocks of a particular size (e.g., 4 KiB), each of which has an associated numerical value called a protection key.

Each process also has a protection key value associated with it. On a memory access the hardware checks that the current process's protection key matches the value associated with the memory block being accessed; if not, an exception occurs. This mechanism was introduced in the System/360 architecture.

https://en.wikipedia.org/wiki/Memory_protection#Protection_keys

Memory Protection

IBM S/360 System Calls

Mark Smotherman. Last updated October 2009

Introduction



Two execution modes: user, supervisor. Mode is recorded in PSW bit 15.

```
+-----+-----+-----+-----+-----+-----+-----+-----+
|sys mask| mkey |a|m|w|p| interrupt code |il|cc|pmsk| instruction address |
+-----+-----+-----+-----+-----+-----+-----+-----+
0         8   12       16               32 34 36   40
```

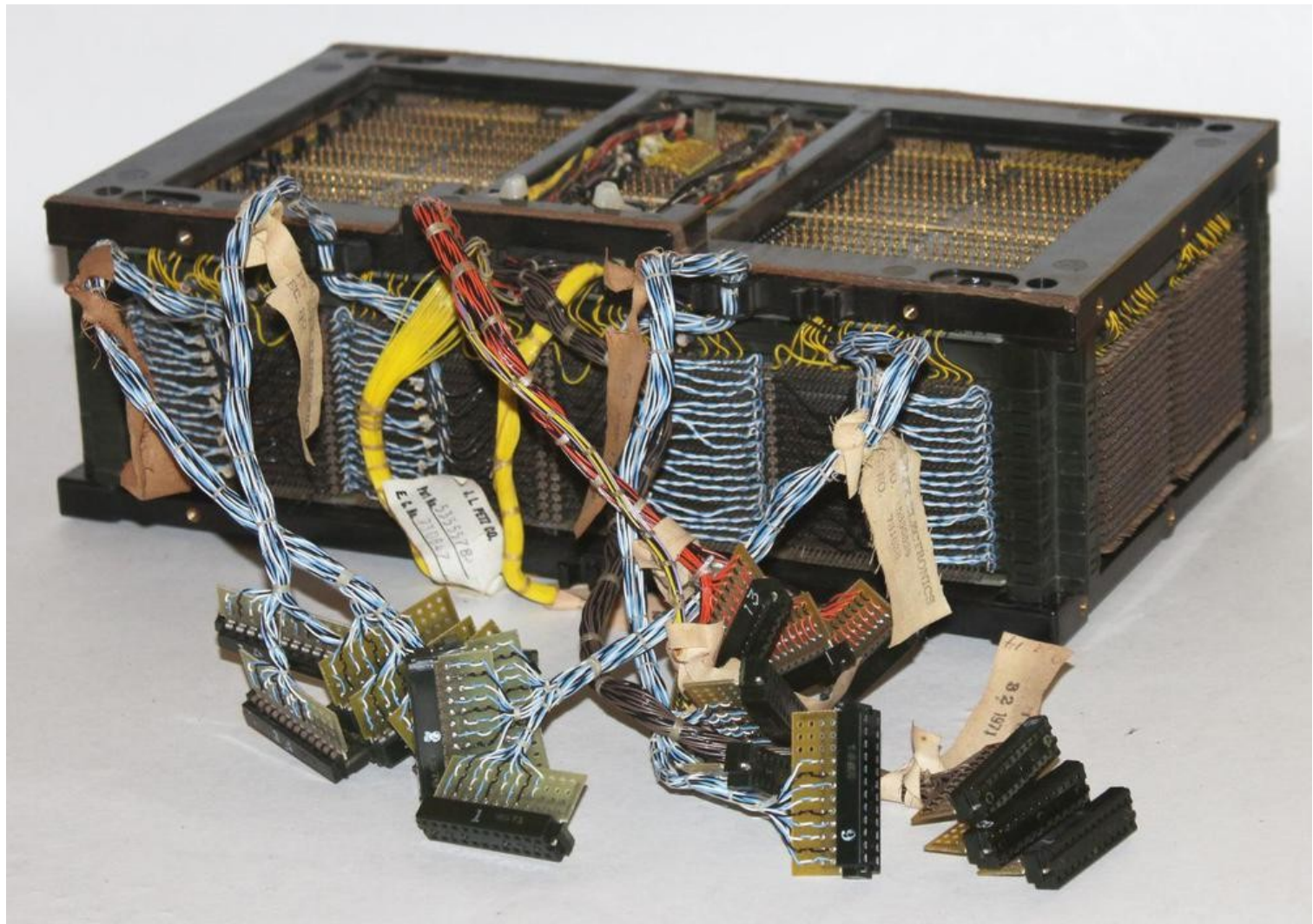
**Without swapping, but
with the addition of
some special
hardware.**

```
0: 7  system mask (channels 0-6, timer + interrupt key + external signal)
8:11 memory protection key
12   ASCII/EBCDIC
13   machine check mask (enable/disable machine check interrupts)
14   running/wait state
15   problem/supervisor state
16:31 interruption code (program error encoding: 1 = invalid operation, etc.)
32:33 ILC (instruction length code)
34:35 CC (condition code)
36:39 program mask (fixed-point, decimal, and exponent overflow; significance)
40:63 instruction address (24 bits on S/360)
```

Memory protection was provided by having a five-bit key assigned to each 2 KB block of main memory. The four high bits of this key must match the protection key in bits 8-11 of the current PSW. (A protection key of zero in the PSW is a special case for OS access - all blocks can be accessed. Thus there can be at most 15 problem state programs in main memory at any one time.) The low bit in the memory block key indicates whether only stores are checked (value = 0) or both fetches and stores are checked (value = 1).

<https://people.cs.clemson.edu/~mark/syscall/s360.html>

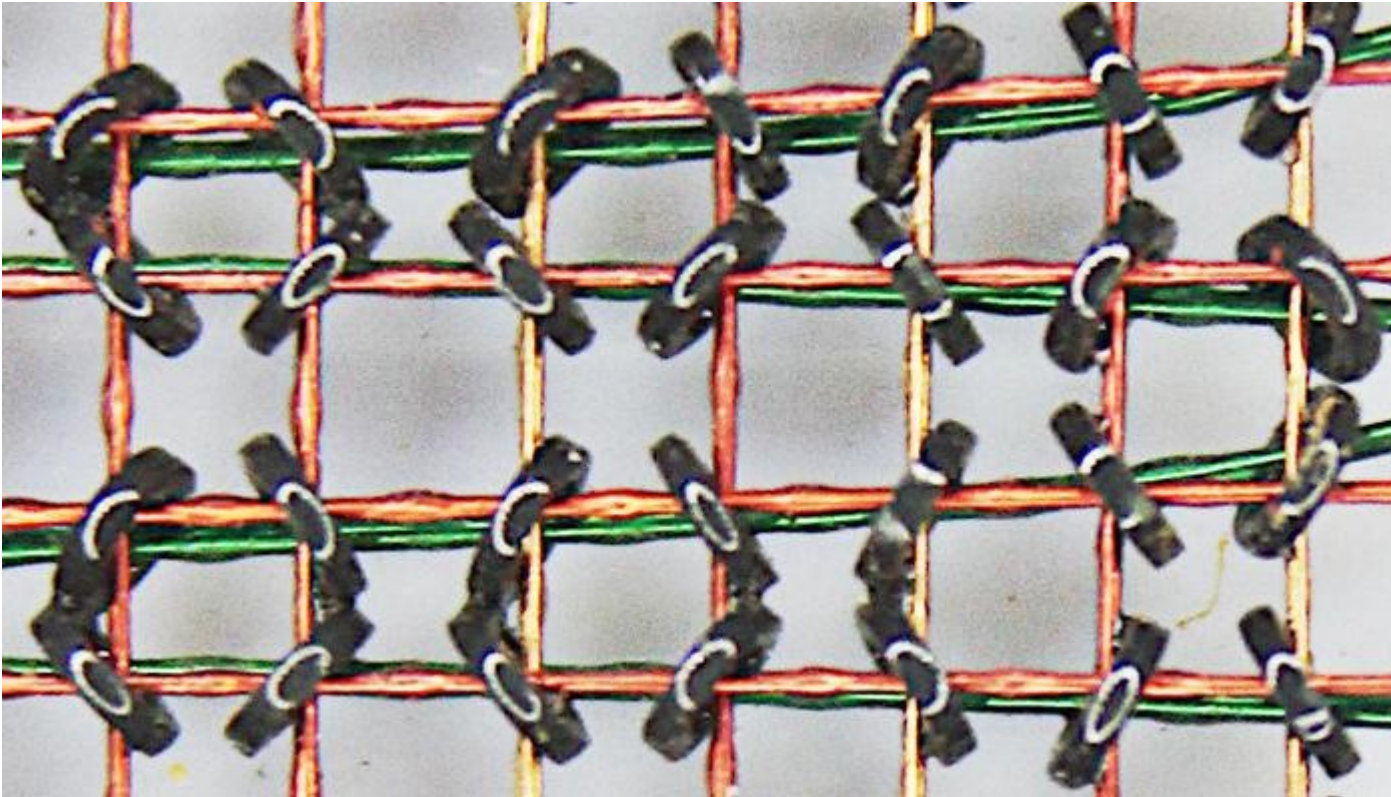
IBM S/360 Core Memory



Depending on the particular computer, 128K weighed 575, 610, or 750 pounds (261, 277 o 340 kg).

<http://www.righto.com/2019/04/a-look-at-ibm-s360-core-memory-in-1960s.html>

IBM S/360 Core Memory



Closeup of an IBM 360 Model 50 core plane. There are three wires through each core. The X and Y wires select one core from the grid. The green wire is the sense/inhibit line. These cores are 30 mils in diameter (.8mm).

<http://www.righto.com/2019/04/a-look-at-ibm-s360-core-memory-in-1960s.html>

How core memory worked

Core memory was the dominant form of computer storage from the 1950s until it was replaced by semiconductor memory chips in the early 1970s. Core memory was built from tiny ferrite rings called *cores*, storing one bit in each core. Each core stored a bit by being magnetized either clockwise or counterclockwise. A core was magnetized by sending a pulse of current through a wire threaded through the core. The magnetization could be reversed by sending a pulse in the opposite direction. Thus, each core could store a 0 or 1.

To read the value of a core, a current pulse flipped the core to the 0 state. If the core was in the 1 state previously, the changing magnetic field created a voltage in a sense wire. But if the core was already in the 0 state, the magnetic field wouldn't change and the sense line wouldn't pick up a voltage. Thus, the value of the bit in the core could be read by resetting the core to 0 and testing the tiny voltage on the sense wire. (An important side effect was that the process of reading a core erased its value so it needed to be rewritten.)

<http://www.righto.com/2019/04/a-look-at-ibm-s360-core-memory-in-1960s.html>

Running Multiple Programs Without a Memory Abstraction

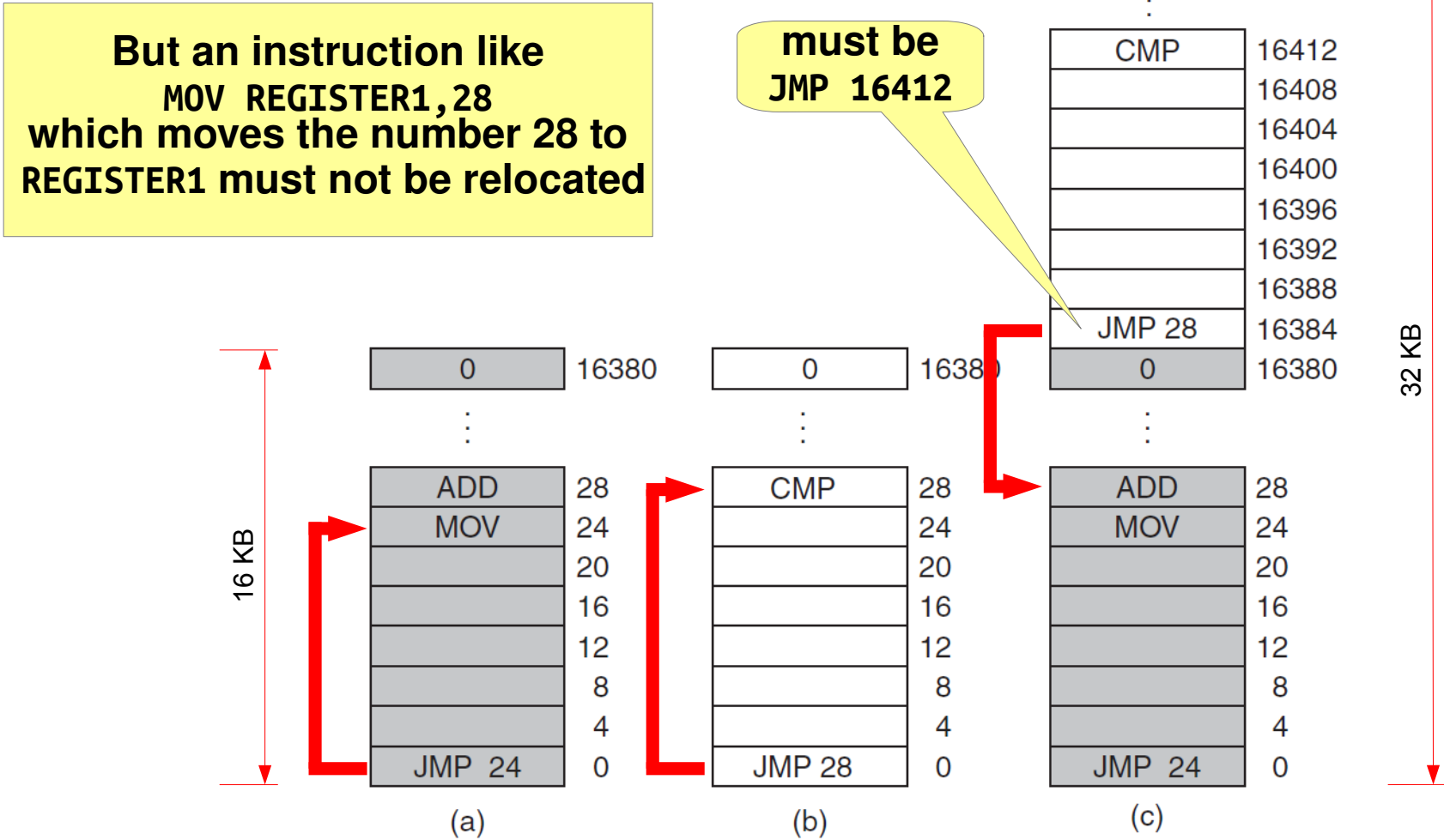


Figure 3-2. Illustration of the relocation problem. (a) A 16-KB program. (b) Another 16-KB program. (c) The two programs loaded consecutively into memory.

MOS4E

Running Multiple Programs Without a Memory Abstraction

The core problem here is that the two programs both reference **absolute physical memory**. That is not what we want at all. What we want is that each program can reference a **private set of addresses local to it**.

What the IBM 360 did as a stop-gap solution was modify the second program on the fly as it loaded it into memory using a technique known as **static relocation**.

Running Multiple Programs Without a Memory Abstraction

It worked like this. When a program was loaded at address 16,384, the constant 16,384 was added to every program address during the load process (so “JMP 28” became “JMP 16,412”, etc.).

While this mechanism works if done right, it is not a very general solution and slows down loading.

Furthermore, it requires **extra information in all executable programs** to indicate which words contain (relocatable) addresses and which do not.

MS Windows executable file headers MZ and PE

	0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f	
00000000h:	4D	5A	50	00	02	00	00	00	04	00	0F	00	FF	FF	00	00	; MZ.....ÿÿ..
00000010h:	B8	00	00	00	00	00	00	00	40	00	1A	00	00	00	00	00	;@.....
00000020h:	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	;
00000030h:	00	00	00	00	00	00	00	00	00	00	00	00	00	01	00	00	; e_lfanew contains offset of PE header (NB bytes in reverse order: 00 00 01 00)
00000040h:	BA	10	00	0E	1F	B4	09	CD	21	B8	01	4C	CD	21	90	90	; °....'.í!..Lí![]
00000050h:	54	68	69	73	20	70	72	6F	67	72	61	6D	20	6D	75	73	; This program mus
00000060h:	74	20	62	65	20	72	75	6E	20	75	6E	64	65	72	20	57	; t be run under W
00000070h:	69	6E	33	32	0D	0A	24	37	00	00	00	00	00	00	00	00	; in32..\$7.....
00000080h:	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	;
00000090h:	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	;
000000a0h:	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	;
000000b0h:	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	;
000000c0h:	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	;
000000d0h:	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	;
000000e0h:	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	;
000000f0h:	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	;
00000100h:	50	45	00	00	4C	01	08	00	19	5E	42	2A	00	00	00	00	; PE..L....^B*.... PE signature at start of PE header
00000110h:	00	00	00	00	E0	00	8E	81	0B	01	02	19	00	A0	02	00	;à.Ž[] ..

<https://marcoramilli.blogspot.com/2010/12/windows-pe-header.html>

The DOS MZ executable header contains relocation information, which allows multiple segments to be loaded at arbitrary memory addresses.

https://en.wikipedia.org/wiki/DOS_MZ_executable

PE files normally do not contain position-independent code. Instead they are compiled to a preferred base address, and all addresses emitted by the compiler/linker are fixed ahead of time. If a PE file cannot be loaded at its preferred address (because it's already taken by something else), the operating system will rebase it.

This involves recalculating every absolute address and modifying the code to use the new values. The loader does this by comparing the preferred and actual load addresses, and calculating a delta value. This is then added to the preferred address to come up with the new address of the memory location.

https://en.wikipedia.org/wiki/Portable_Executable

Base relocations are stored in a list and added, as needed, to an existing memory location. The resulting code is now private to the process and no longer shareable, so many of the memory saving benefits of DLLs are lost in this scenario. It also slows down loading of the module significantly.

For this reason rebasing is to be avoided wherever possible, and the DLLs shipped by Microsoft have base addresses pre-computed so as not to overlap. In the no rebase case PE therefore has the advantage of very efficient code, but in the presence of rebasing the memory usage hit can be expensive. This contrasts with ELF which uses fully position-independent code and a global offset table, which trades off execution time in favor of lower memory usage.

https://en.wikipedia.org/wiki/Portable_Executable

Memory and Multiprogramming

Challenges:

Relocation

Memory protection

Allocation

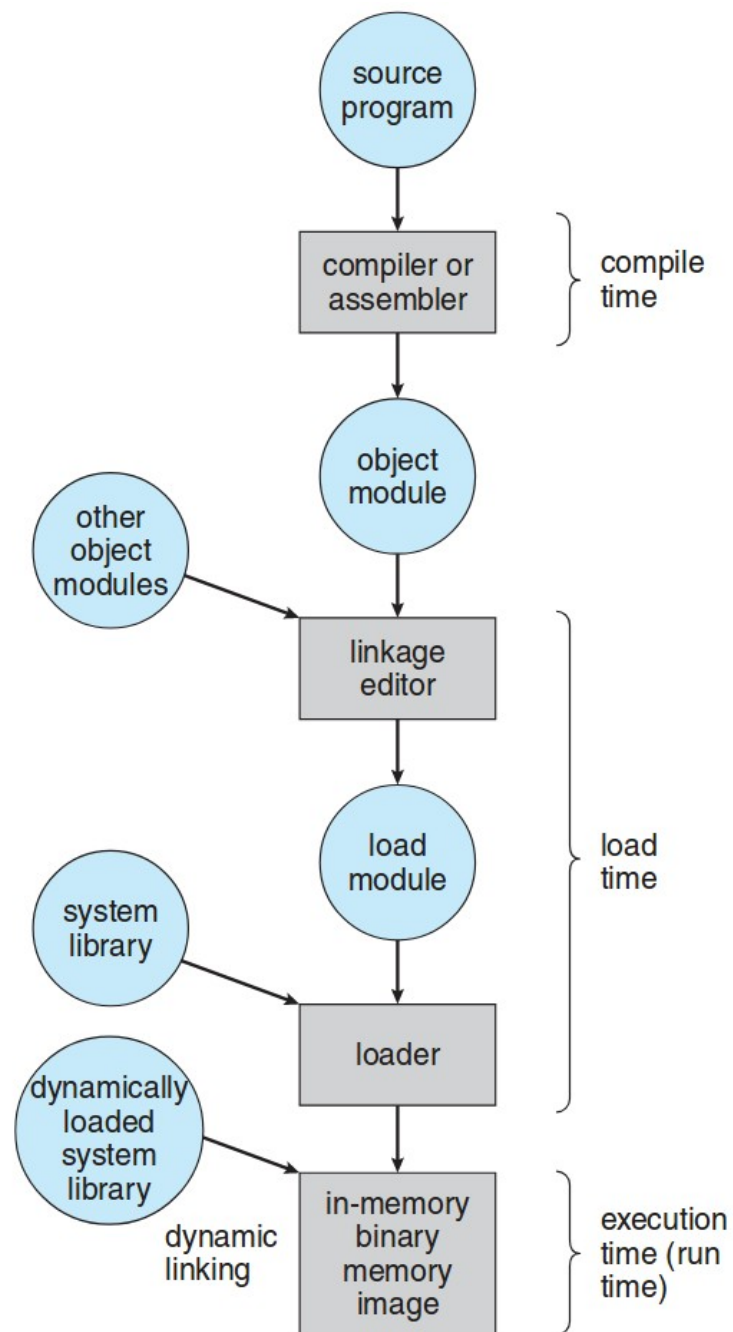


Figure 8.3 Multistep processing of a user program.

OSC9E

The Address Binding Procedure (1/5)

Figure 11.3
A Sample Code Segment:

```
static int gVar;  
...  
int proc_a(int arg) {  
    ...  
    gVar = 7;  
    put_record(gVar);  
    ...  
}
```


The Address Binding Procedure (2/5)

Figure 11.4

The Relocatable Object module:

Relative Address	Generated Code	
0000		
...		
0008	entry	proc_a
...		
0036	[Space for gVar variable]	
...		
0220	load	=7, R1
0224	store	R1, 0036
0228	push	0036
0232	call	'put_record'
...		
0400	External reference table	
...		
0404	'put_record'	0232
...		

OSAMP2E

The Address Binding Procedure (3/5)

Figure 11.4

The Relocatable Object module:

Relative Address	Generated Code
...	
0500	External definition table
...	
0540	' <i>proc_a</i> ' 0008
...	
0600	(optional symbol table)
...	
0799	(last location in the module)
...	

The Address Binding Procedure (4/5)

Figure 11.5

The Absolute Program:

Relative Address	Generated Code
0000	(Other modules)
...	
1008	entry proc_a
...	
1036	[Space for gVar variable]
...	
1220	load =7, R1
1224	store R1, 1036
1228	push 1036
1232	call 2334
...	
1399	(End of proc_a)
...	(Other modules)
2334	entry put_record
...	
2670	(optional symbol table)
...	
2999	(last location in the module)

OSAMP2E

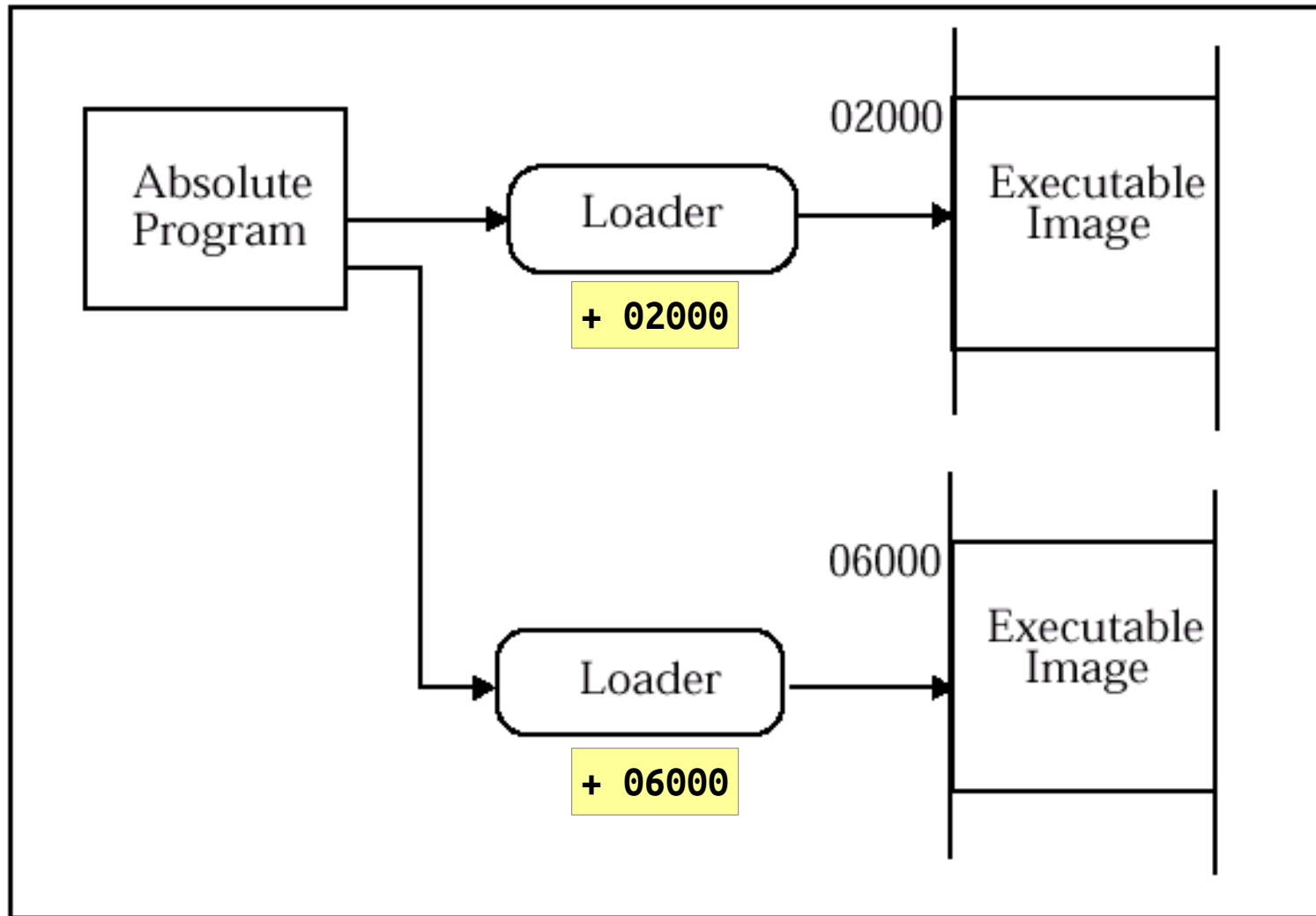
The Address Binding Procedure (5/5)

Figure 11.6
The Program loaded
at Location 4000:

Relative Address	Generated Code
0000	(Other process's programs)
4000	(Other modules)
...	
5008	entry proc_a
...	
5036	[Space for gVar variable]
...	
5220	load =7, R1
5224	store R1, 5036
5228	push 5036
5232	call 6334
...	
5399	(End of proc_a)
...	(Other modules)
6334	entry put_record
...	
6670	(optional symbol table)
...	
6999	(last location in the module)

OSAMP2E

Static Relocation



OSAMP2E, Fig. 11.11

3.2 A memory abstraction: address space

An **address space** is the set of addresses that a process can use to address memory.

Each process has its own address space, independent of those belonging to other processes.

The concept of an address space is very general and occurs in many contexts. Consider telephone numbers. The address space for telephone numbers runs from 000,000,000 to 999,999,999.

IPv4 addresses are 32-bit numbers, so their address space runs from 0 to $2^{32} - 1$.

Address spaces do not have to be numeric. The set of *.com* Internet domains is also an address space. This address space consists of all the strings of length 2 to 63 characters that can be made using letters, numbers, and hyphens, followed by *.com*.

How to give each program its own address space?

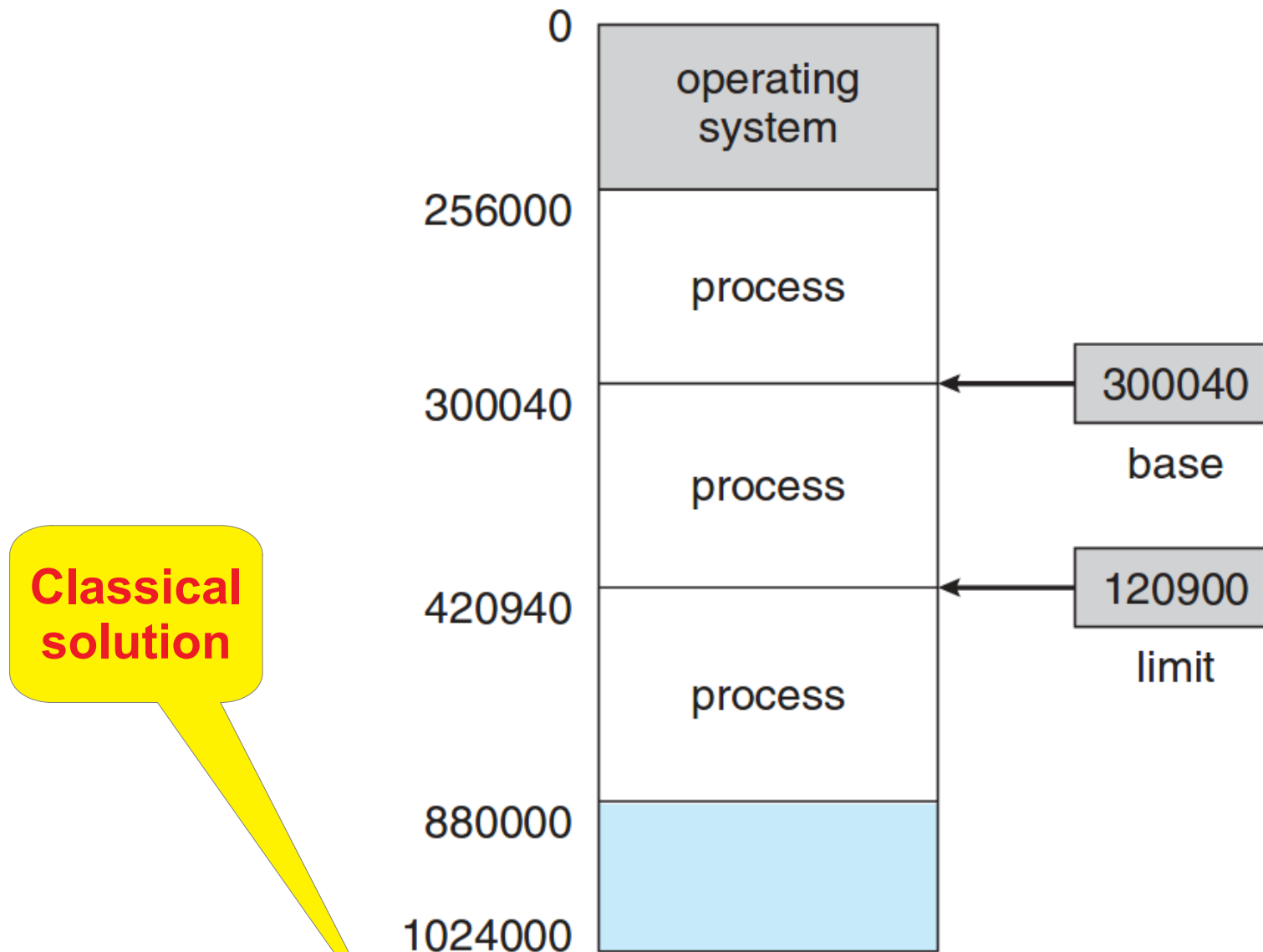
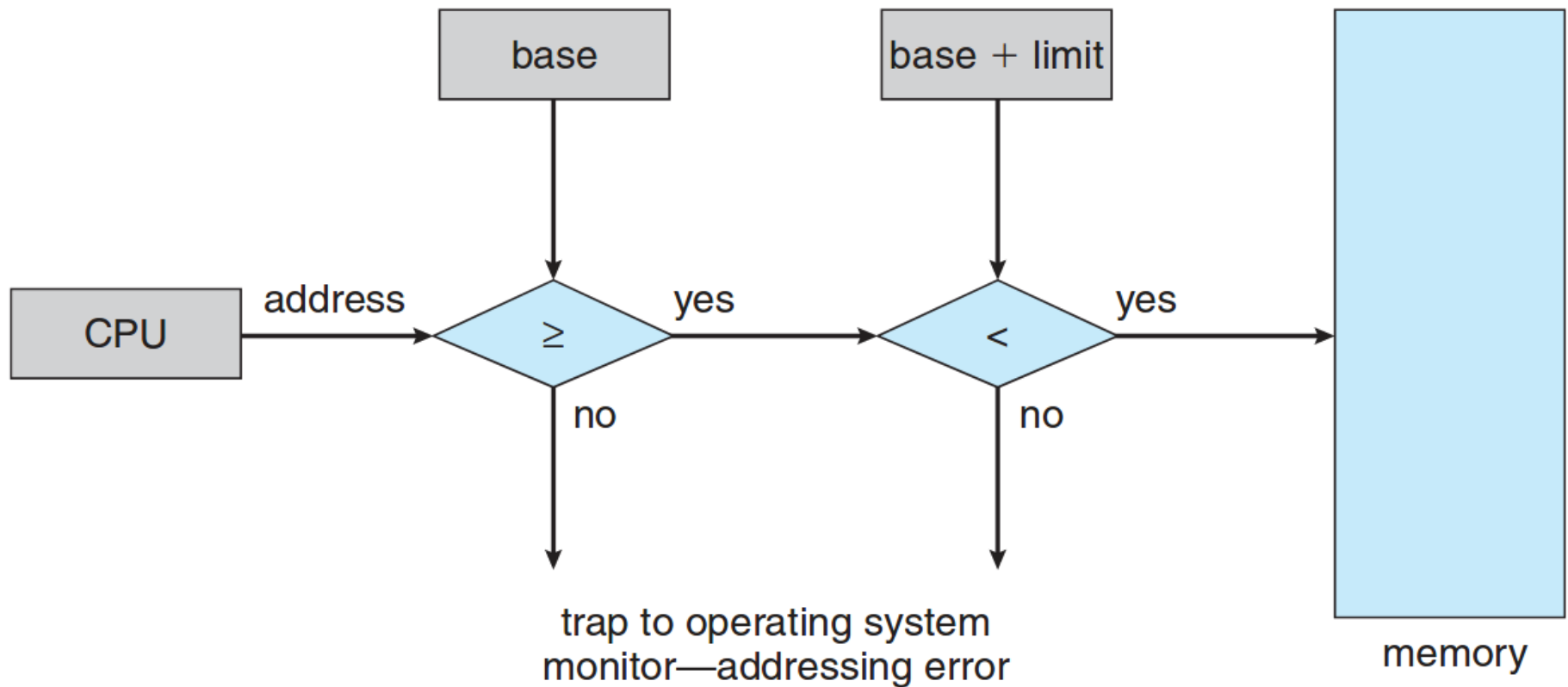


Figure 8.1 A base and a limit register define a logical address space.



OSC9E

Figure 8.2 Hardware address protection with base and limit registers.

Every time a process references memory, either to fetch an instruction or read or write a data word, the CPU hardware automatically adds the base value to the address generated by the process before sending the address out on the memory bus. Simultaneously, it checks whether the address offered is equal to or greater than the value in the limit register, in which case a fault is generated and the access is aborted.

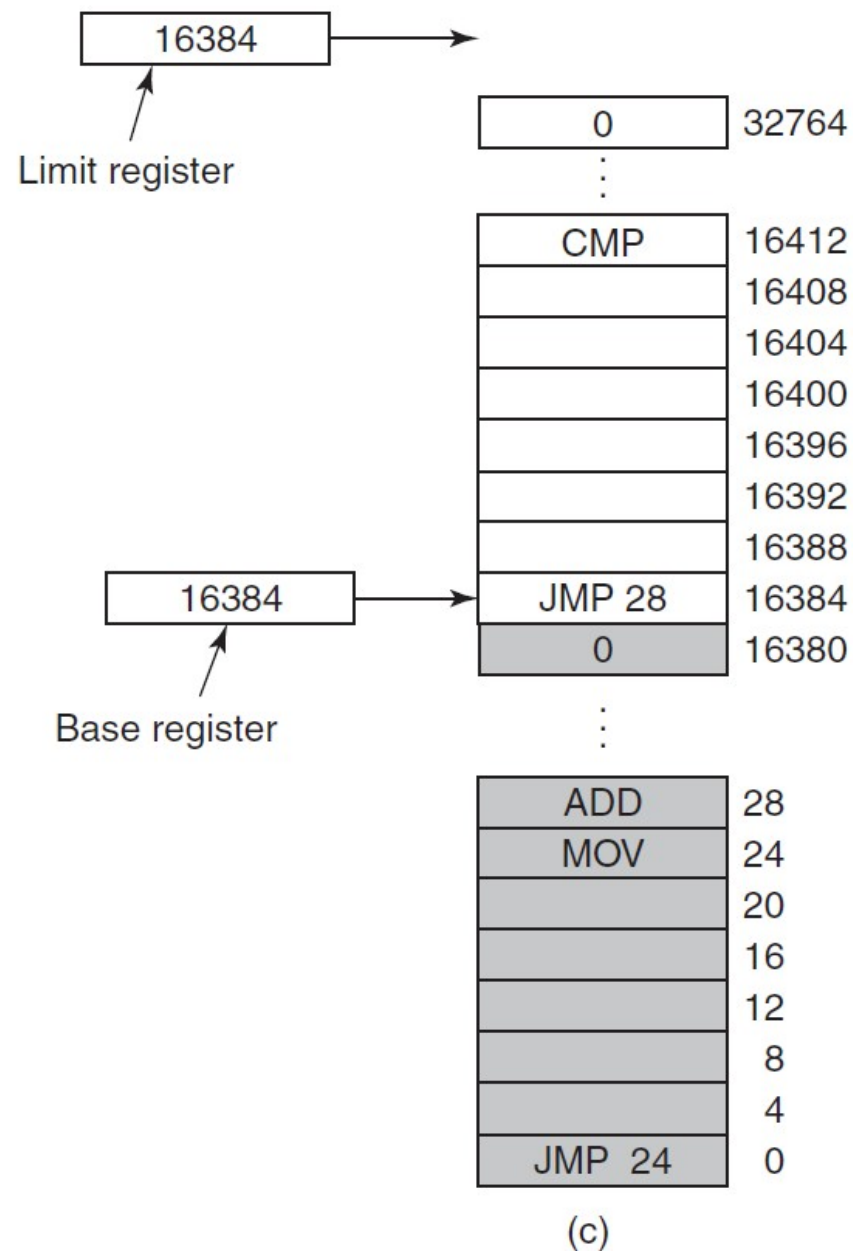
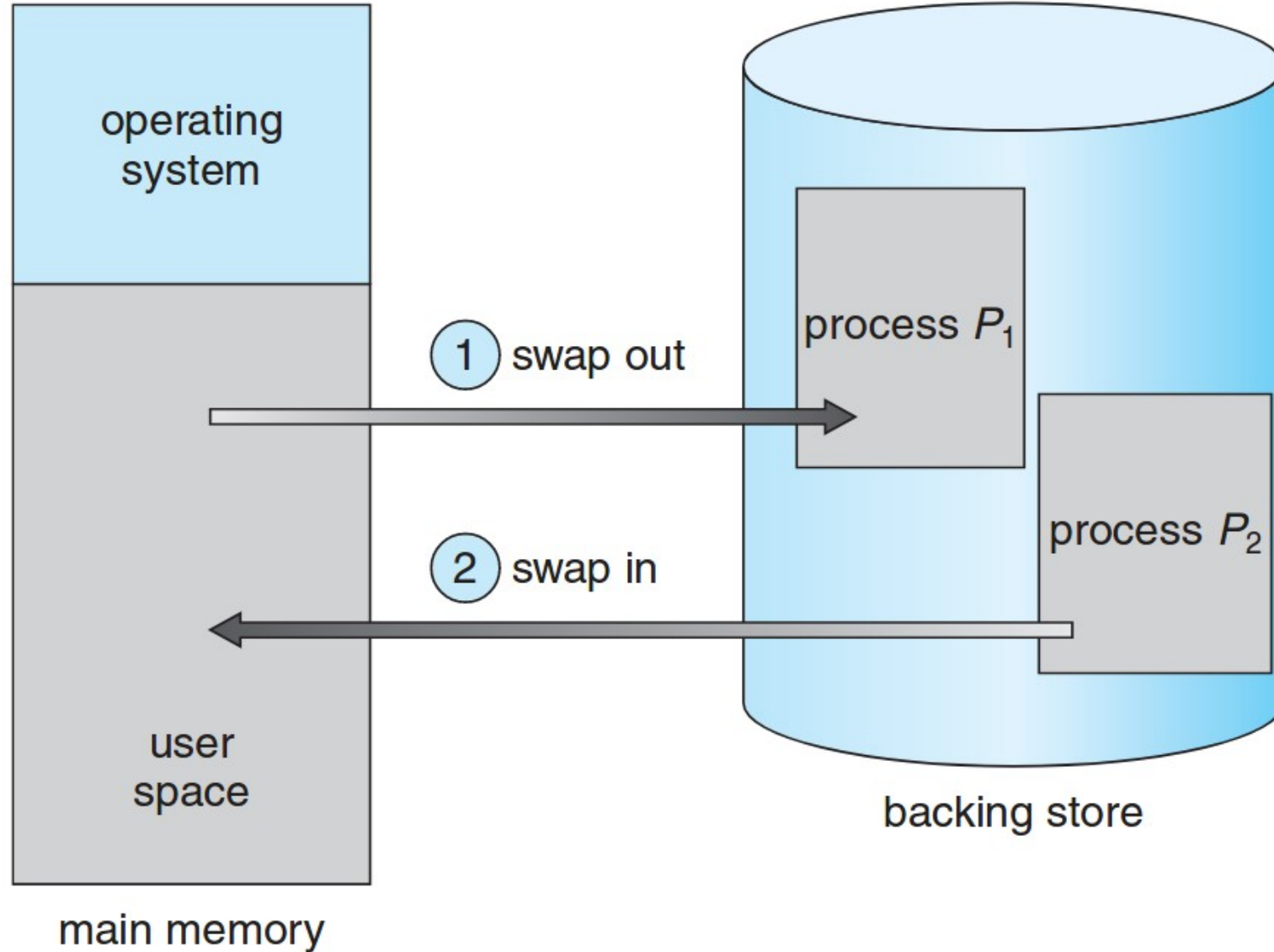


Figure 3-3. Base and limit registers can be used to give each process a separate address space.

3.2.2 Swapping



OSC9E

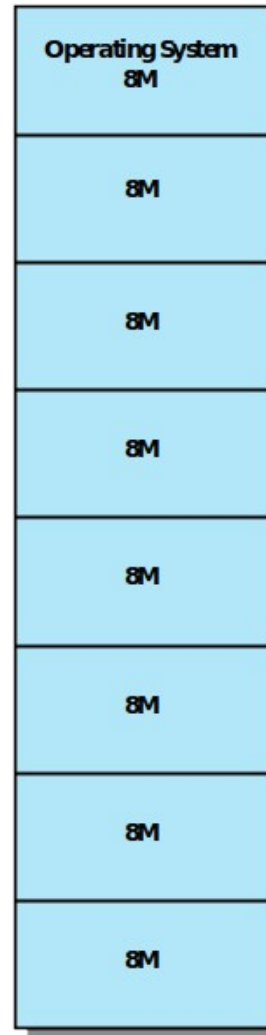
Figure 8.5 Swapping of two processes using a disk as a backing store.

Memory Partitioning: Fixed Partitioning

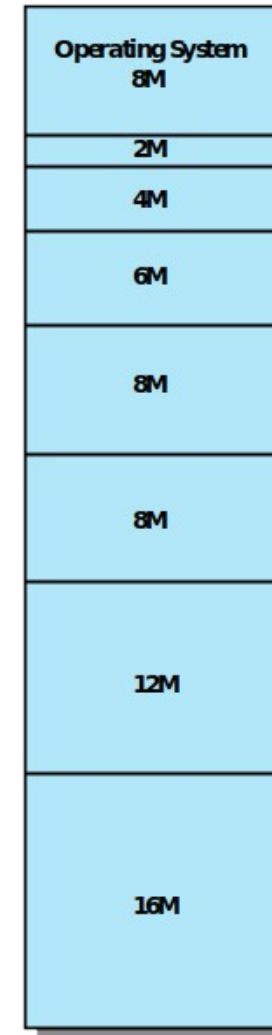
IBM OS/360 (MFT)

Every partition is used only by one process.

So we have the *internal fragmentation*.



(a) Equal-size partitions



(b) Unequal-size partitions

Figure 7.2 Example of Fixed Partitioning of a 64-Mbyte Memory

OSIDP9E

Memory Partitioning: Fixed Partitioning

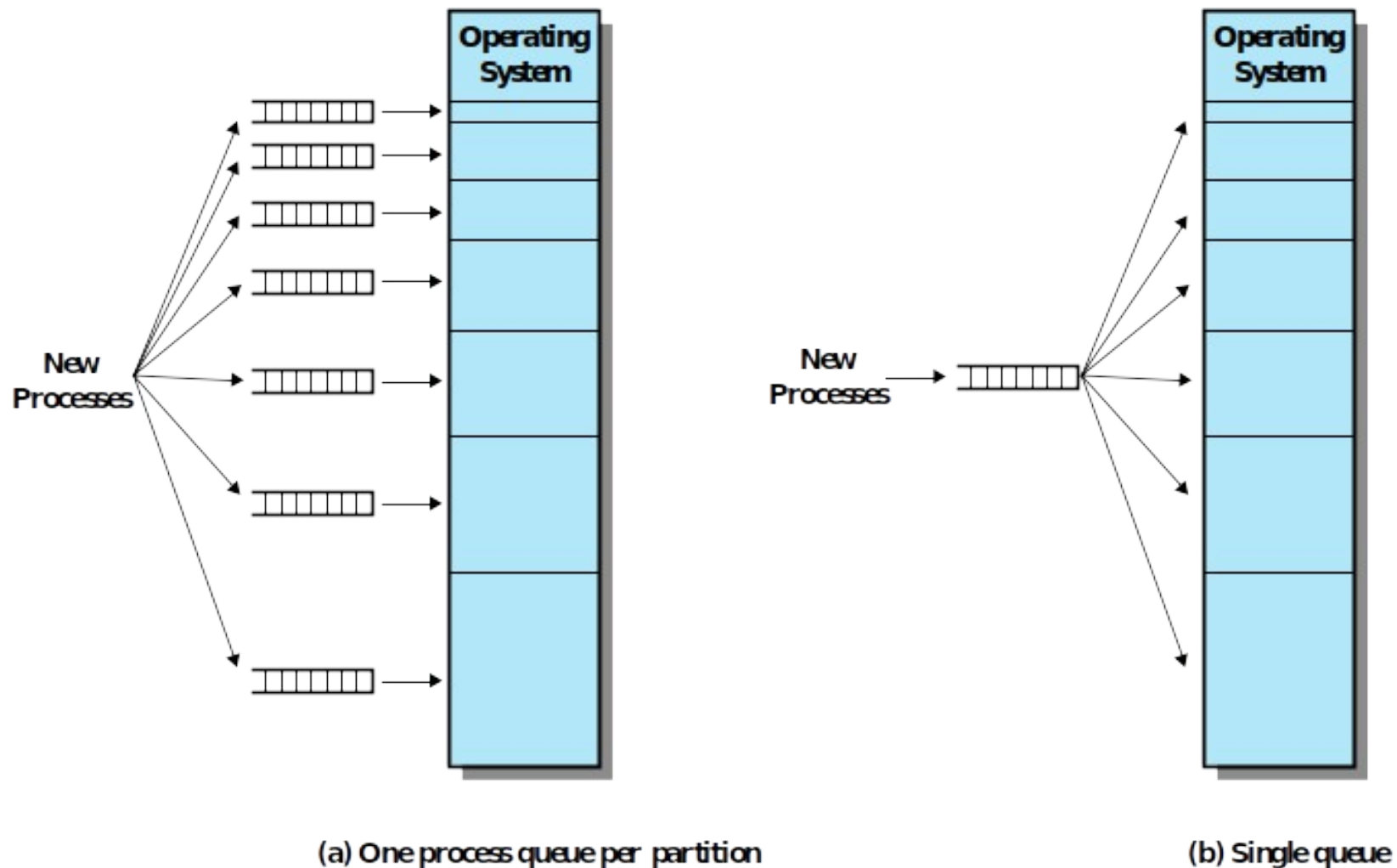


Figure 7.3 Memory Assignment for Fixed Partitioning

OSIDP9E

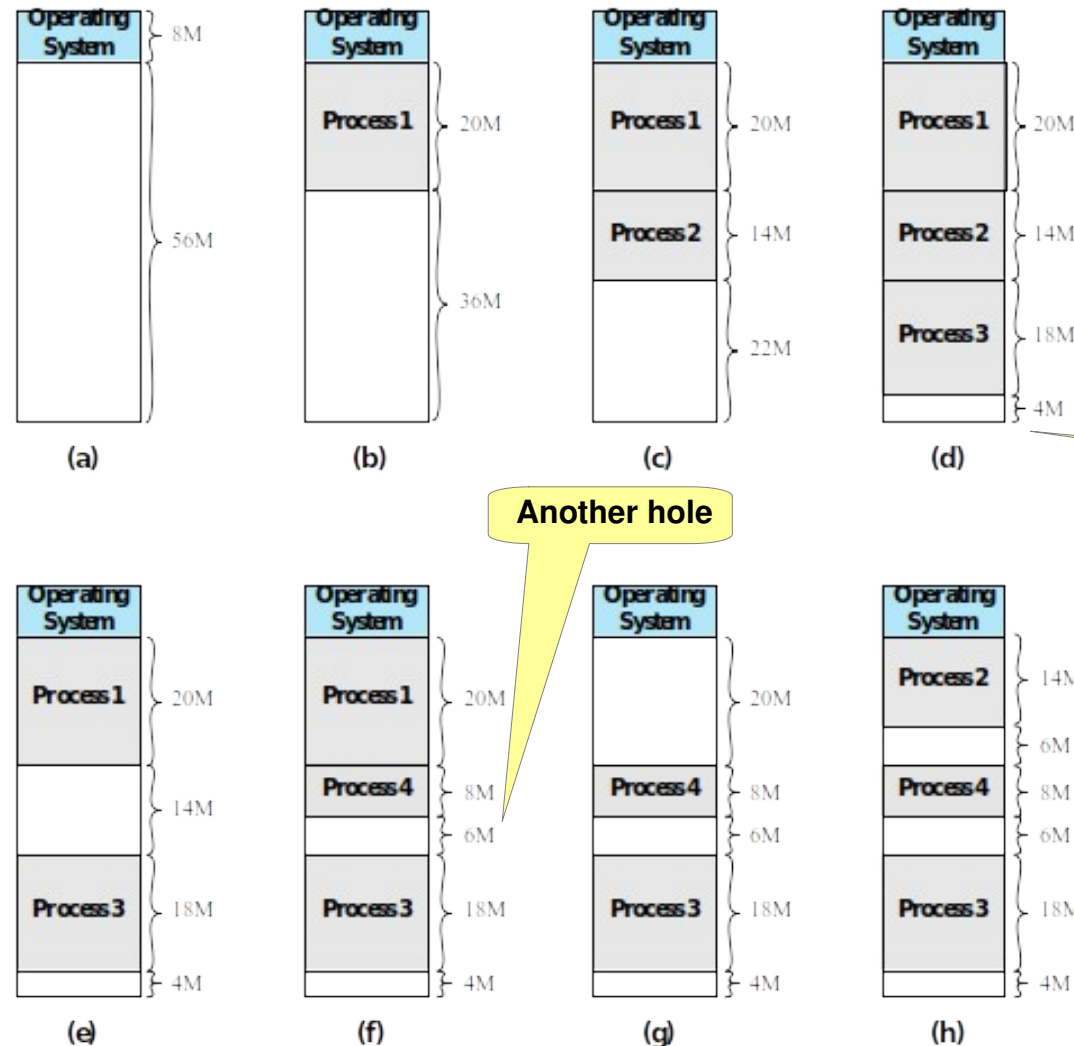
Memory Partitioning: Dynamic Partitioning

IBM OS/370 (MVT)

Every partition is of the process size.

But we have the *external fragmentation*.

Compaction is a time-consuming procedure and wasteful of processor time.



Process 2 is swapped out.

Process 4 is loaded. Process 2 is Ready-Suspended.

Process 1 is swapped out. Process 2 is loaded.

OSIDP9E

Figure 7.4 The Effect of Dynamic Partitioning

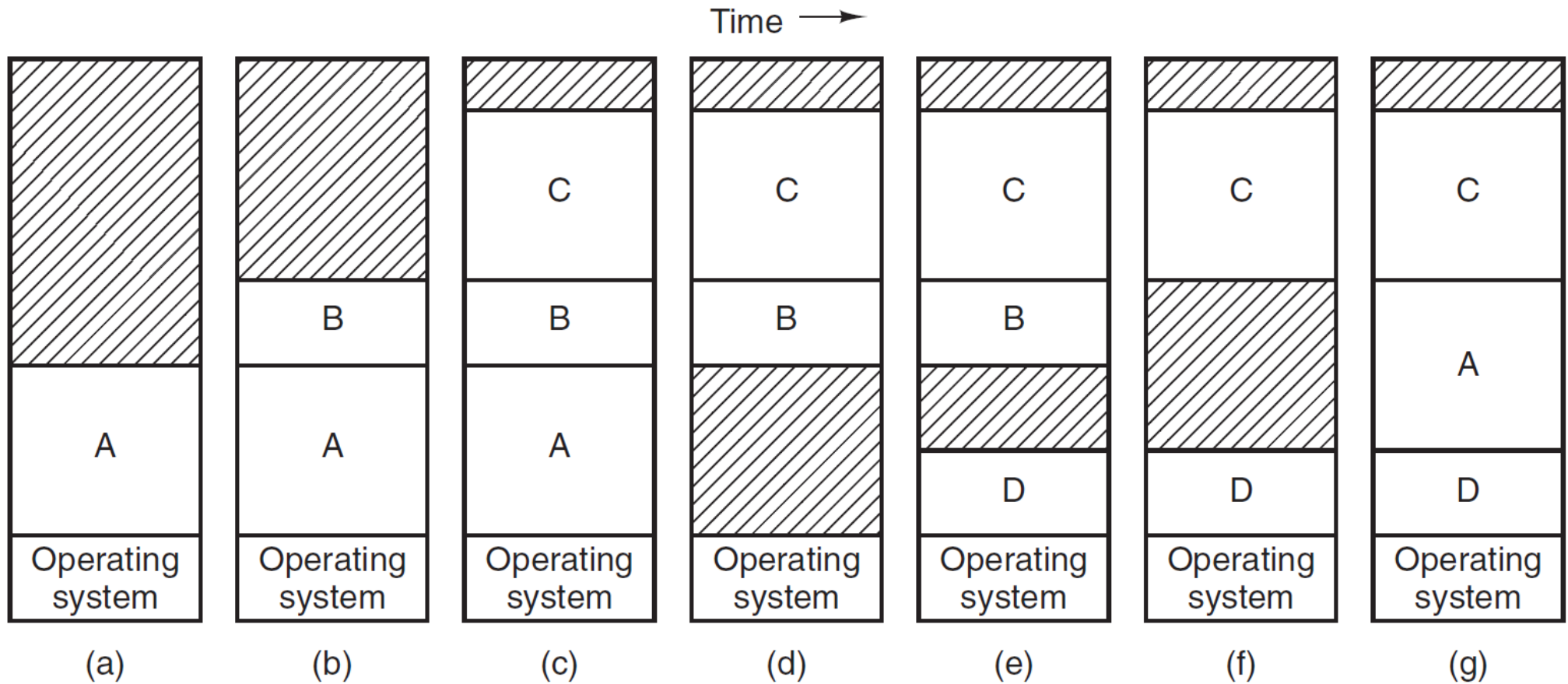


Figure 3-4. Memory allocation changes as processes come into memory and leave it. The shaded regions are unused memory.

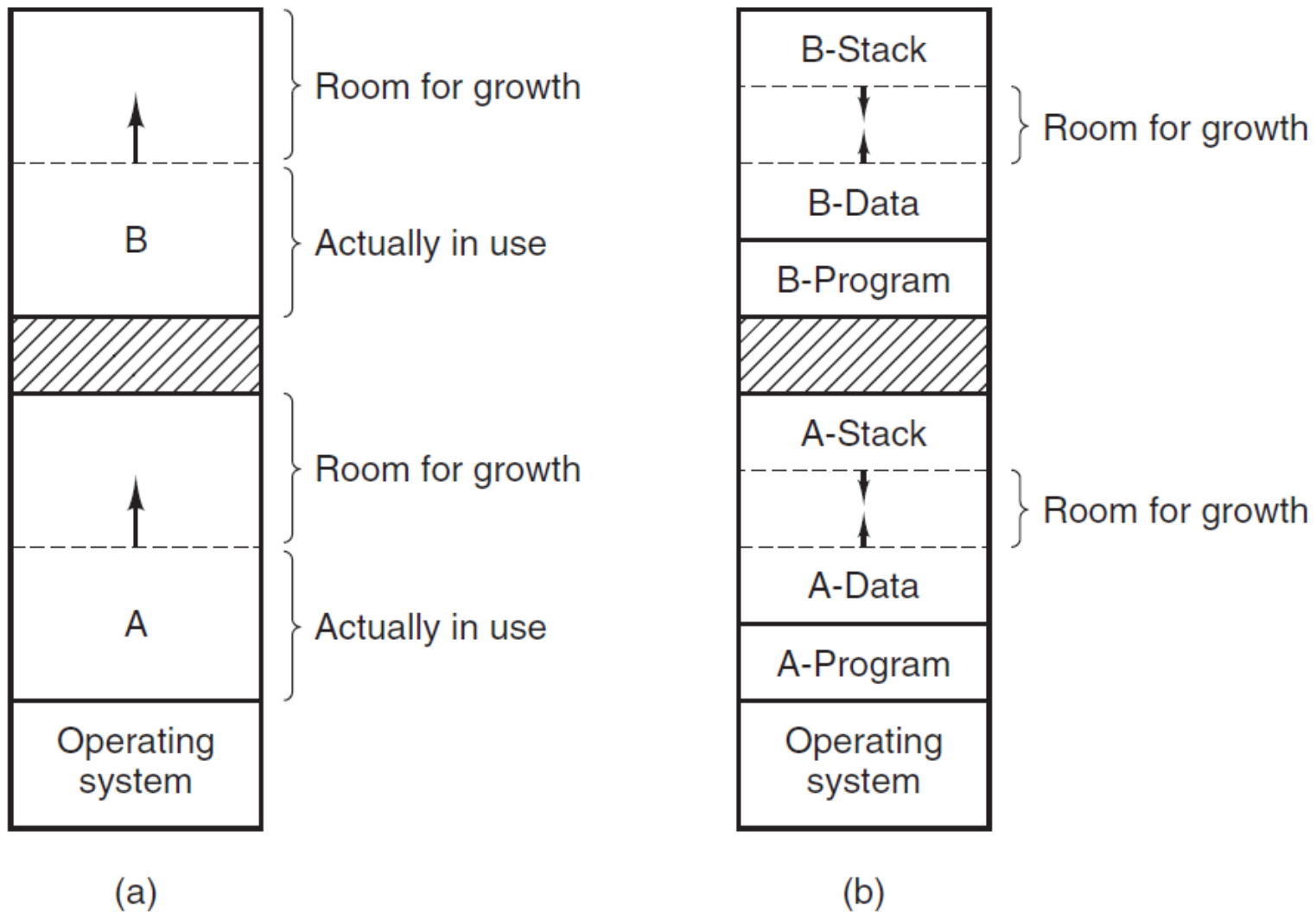


Figure 3-5. (a) Allocating space for a growing data segment. (b) Allocating space for a growing stack and a growing data segment.

3.2.3 Managing Free Memory: Bitmaps, Linked Lists

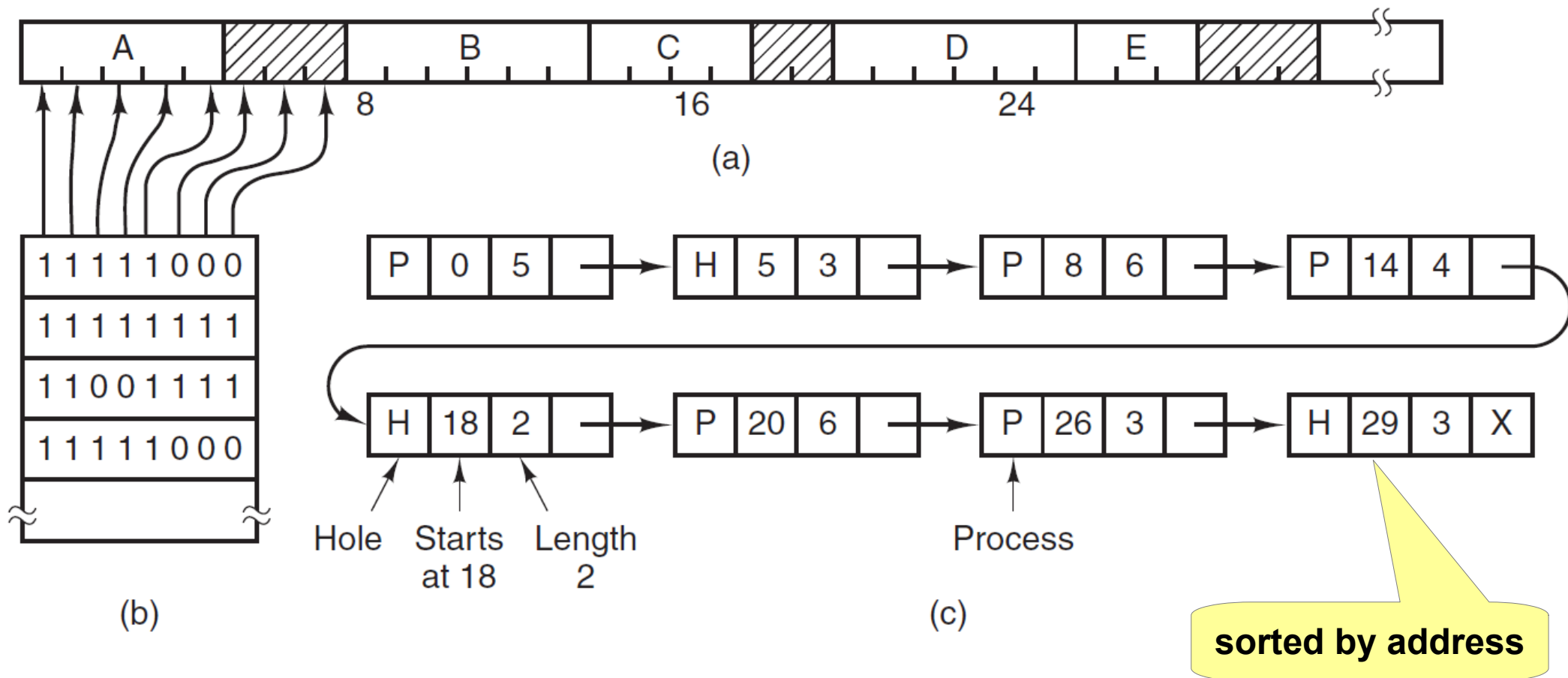


Figure 3-6. (a) A part of memory with five processes and three holes. The tick marks show the memory allocation units. The shaded regions (0 in the bitmap) are free. (b) The corresponding bitmap. (c) The same information as a list.

Managing Free Memory: with Linked Lists

Doble-linked list?



Figure 3-7. Four neighbor combinations for the terminating process, X.

Managing Free Memory: with Linked Lists

First Fit (sorted by address)
Last Fit
Next Fit (circular list)
Best Fit (sorted by size?)
Worst Fit (sorted by size?)

Speedup on allocation:
separate lists for processes
and holes.

Price: slowdown when
deallocating memory.

Place the hole list into
every hole itself.

Quick Fit
Buddy system

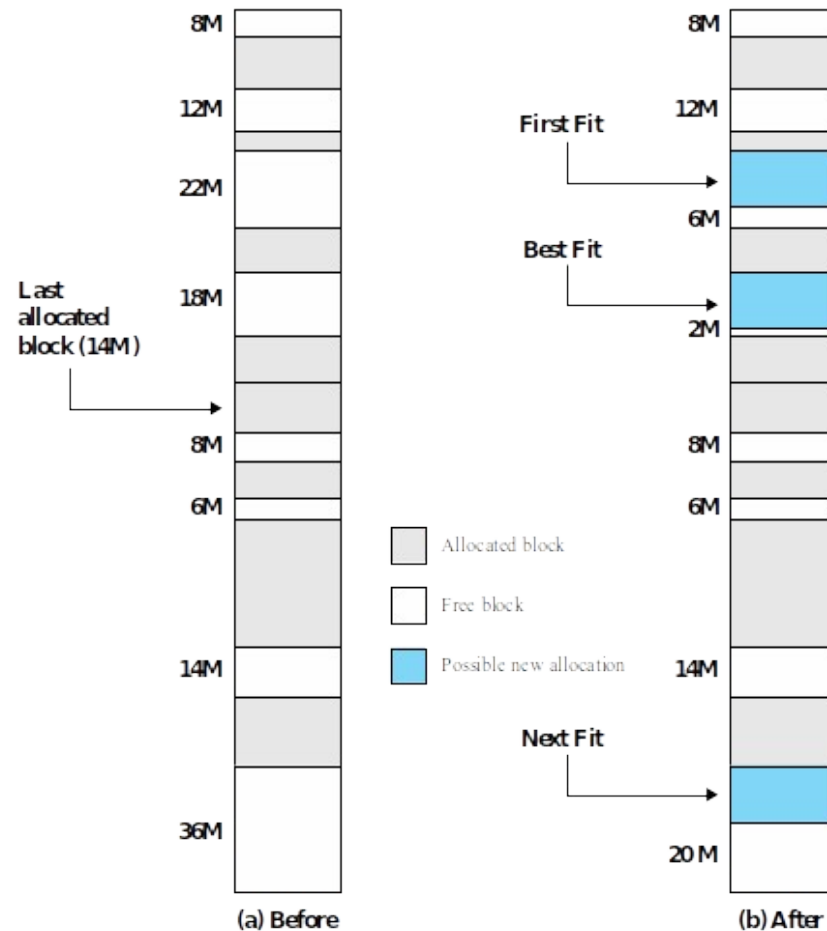
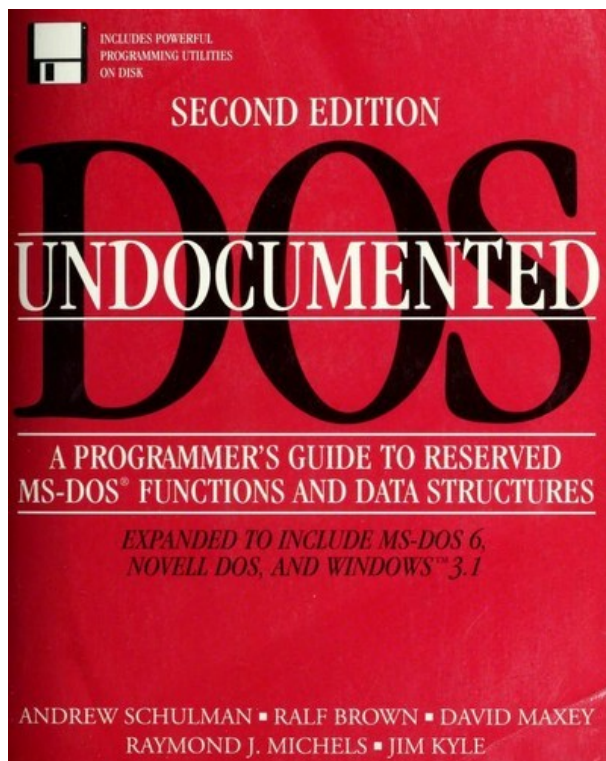


Figure 7.5 Example Memory Configuration before and after Allocation of 16-Mbyte Block

OSIDP9E



800 pages

CHAPTER 7 (continued)

Making Use of UMBs	347
The High Memory Area	349
How To Find the Start of the MCB Chain	350
How To Trace the MCB Chain	351
MCB Consistency Checks	354
A More Detailed UDMEM Program	356
Allocation Precautions	364
RAM Allocation Strategies	365
<i>First-fit Strategy</i>	366
<i>Best-fit Strategy</i>	366
<i>Last-fit Strategy</i>	366

Managing Free Memory: with Linked Lists

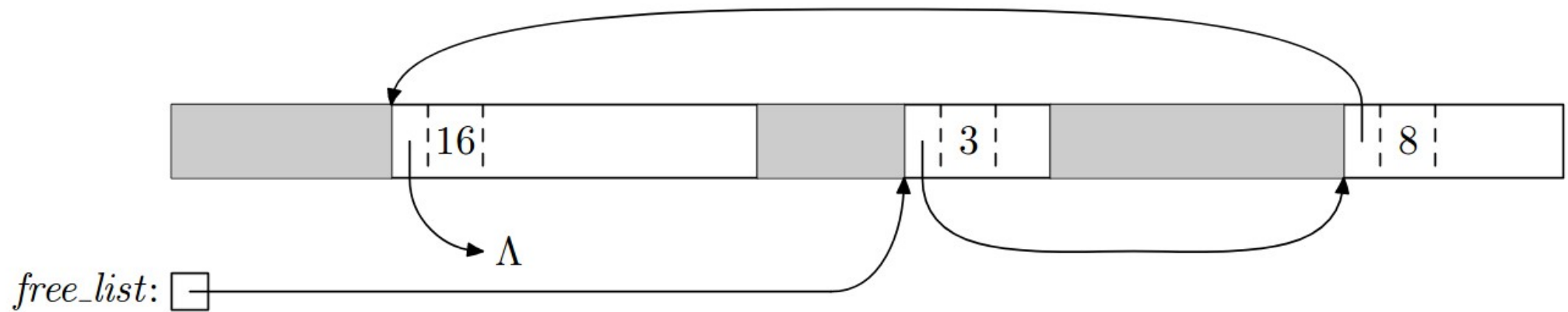


Figure 9-8: Free List Example

<https://www.cs.drexel.edu/~bls96/excerpt3.pdf>

Managing Free Memory: with Buddy System

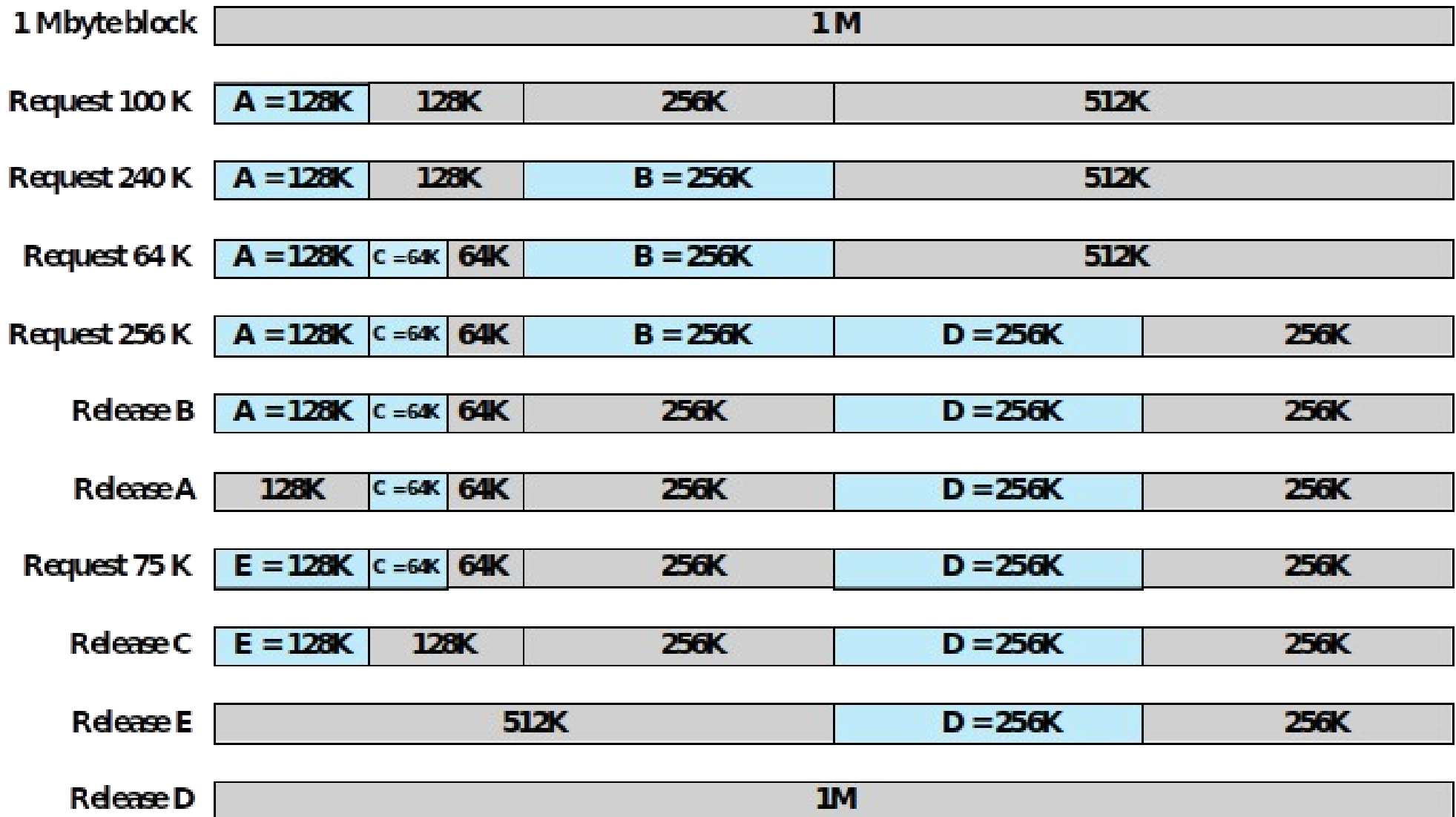


Figure 7.6 Example of Buddy System

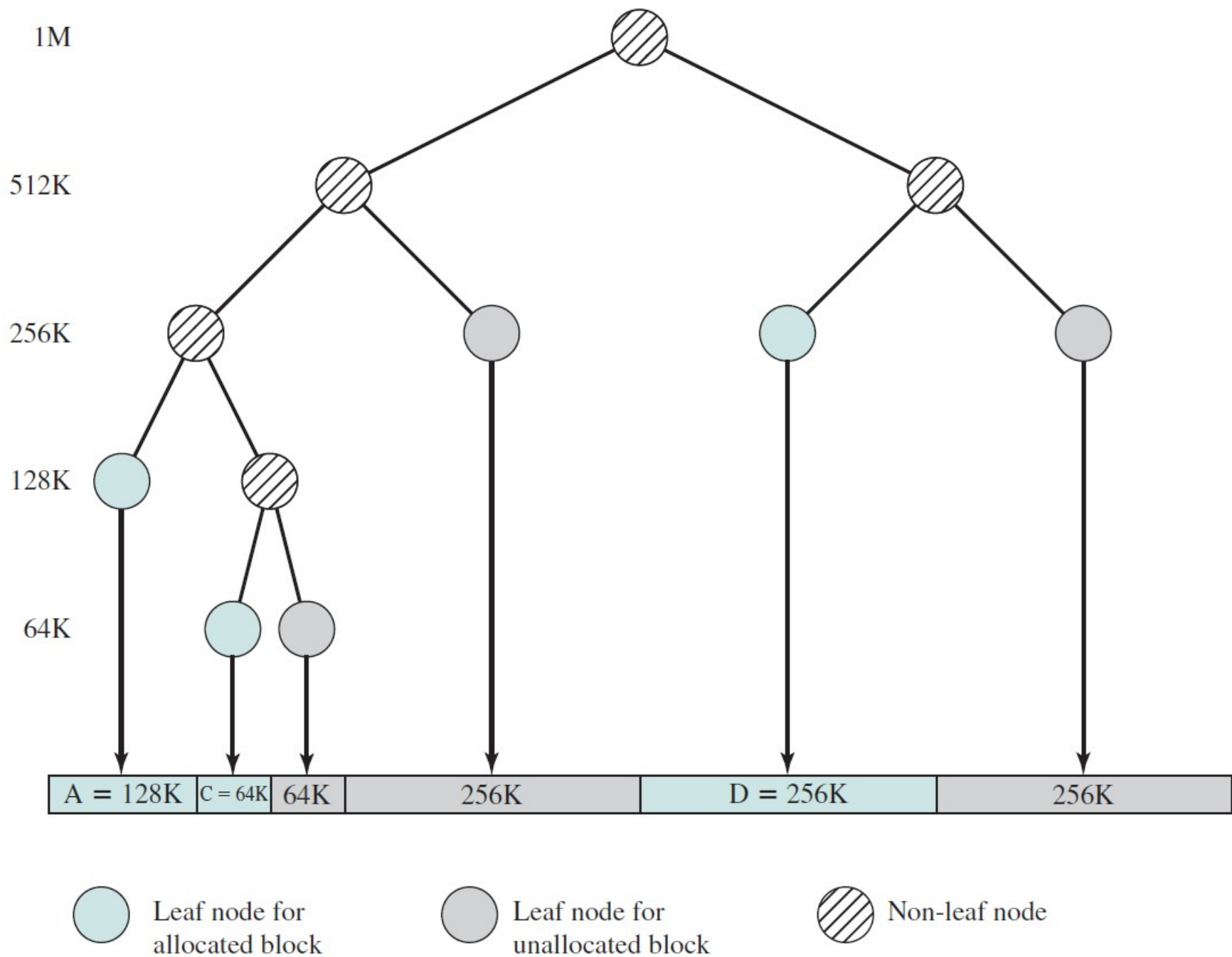
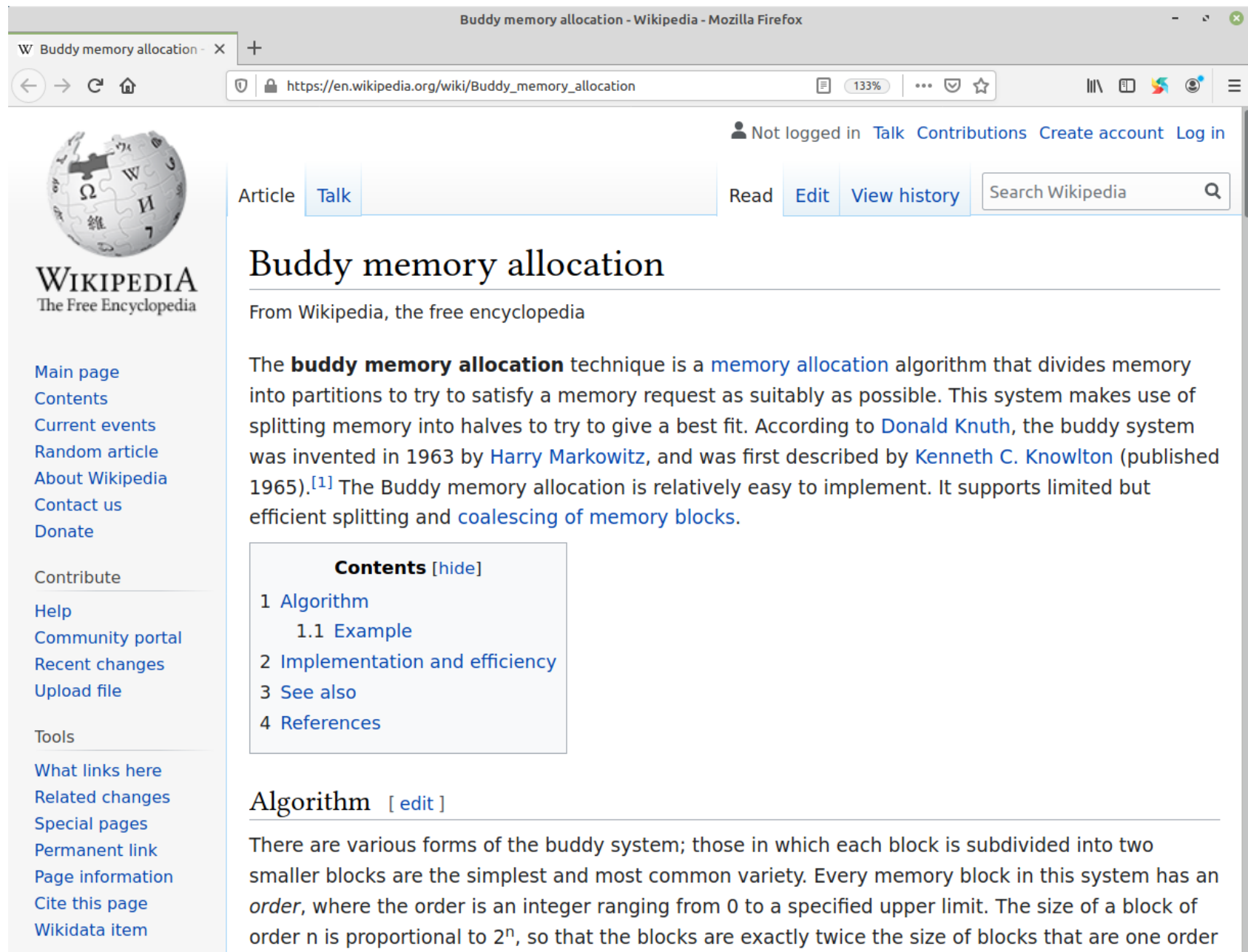


Figure 7.7 Tree Representation of Buddy System

OSIDP9E

Managing Free Memory: with Buddy System



The screenshot shows a web browser window titled "Buddy memory allocation - Wikipedia - Mozilla Firefox". The address bar displays the URL "https://en.wikipedia.org/wiki/Buddy_memory_allocation". The page content includes the Wikipedia logo, navigation links, and the article text. The article title is "Buddy memory allocation", and the subtitle is "From Wikipedia, the free encyclopedia". The main text describes the buddy memory allocation technique, mentioning its invention in 1963 by Harry Markowitz and its first description by Kenneth C. Knowlton in 1965. A table of contents is visible, listing sections: 1 Algorithm, 1.1 Example, 2 Implementation and efficiency, 3 See also, and 4 References. The "Algorithm" section is currently selected.

Wikipedia - The Free Encyclopedia

Main page
Contents
Current events
Random article
About Wikipedia
Contact us
Donate

Contribute

Help
Community portal
Recent changes
Upload file

Tools

What links here
Related changes
Special pages
Permanent link
Page information
Cite this page
Wikidata item

Article **Talk** Read Edit View history Search Wikipedia

Buddy memory allocation

From Wikipedia, the free encyclopedia

The **buddy memory allocation** technique is a [memory allocation](#) algorithm that divides memory into partitions to try to satisfy a memory request as suitably as possible. This system makes use of splitting memory into halves to try to give a best fit. According to [Donald Knuth](#), the buddy system was invented in 1963 by [Harry Markowitz](#), and was first described by [Kenneth C. Knowlton](#) (published 1965).^[1] The Buddy memory allocation is relatively easy to implement. It supports limited but efficient splitting and [coalescing of memory blocks](#).

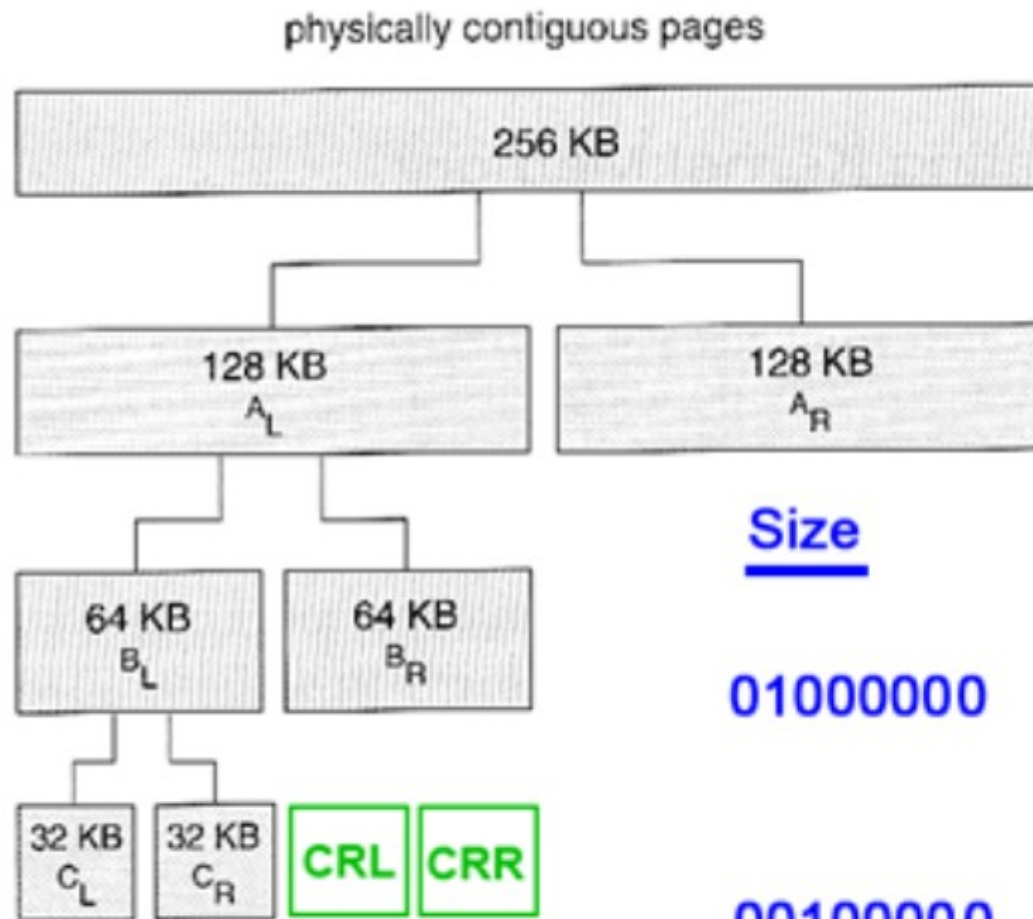
Contents [\[hide\]](#)

- Algorithm
 - Example
- Implementation and efficiency
- See also
- References

Algorithm [\[edit \]](#)

There are various forms of the buddy system; those in which each block is subdivided into two smaller blocks are the simplest and most common variety. Every memory block in this system has an *order*, where the order is an integer ranging from 0 to a specified upper limit. The size of a block of order n is proportional to 2^n , so that the blocks are exactly twice the size of blocks that are one order

Managing Free Memory: with Buddy System



Buddy system allocation.

Buddy Addresses

00000000

00000000 10000000

Size

01000000

00000000 01000000

00100000

00000000 00100000

0100000

0110000

<http://www.expertsmind.com/questions/describe-the-buddy-system-of-memory-allocation-3019462.aspx>

Managing Free Memory: with Buddy System

762

CASE STUDY 1: UNIX, LINUX, AND ANDROID

CHAP. 10

The basic idea for managing a chunk of memory is as follows. Initially memory consists of a single contiguous piece, 64 pages in the simple example of Fig. 10-17(a). When a request for memory comes in, it is first rounded up to a power of 2, say eight pages. The full memory chunk is then divided in half, as shown in (b). Since each of these pieces is still too large, the lower piece is divided in half again (c) and again (d). Now we have a chunk of the correct size, so it is allocated to the caller, as shown shaded in (d).

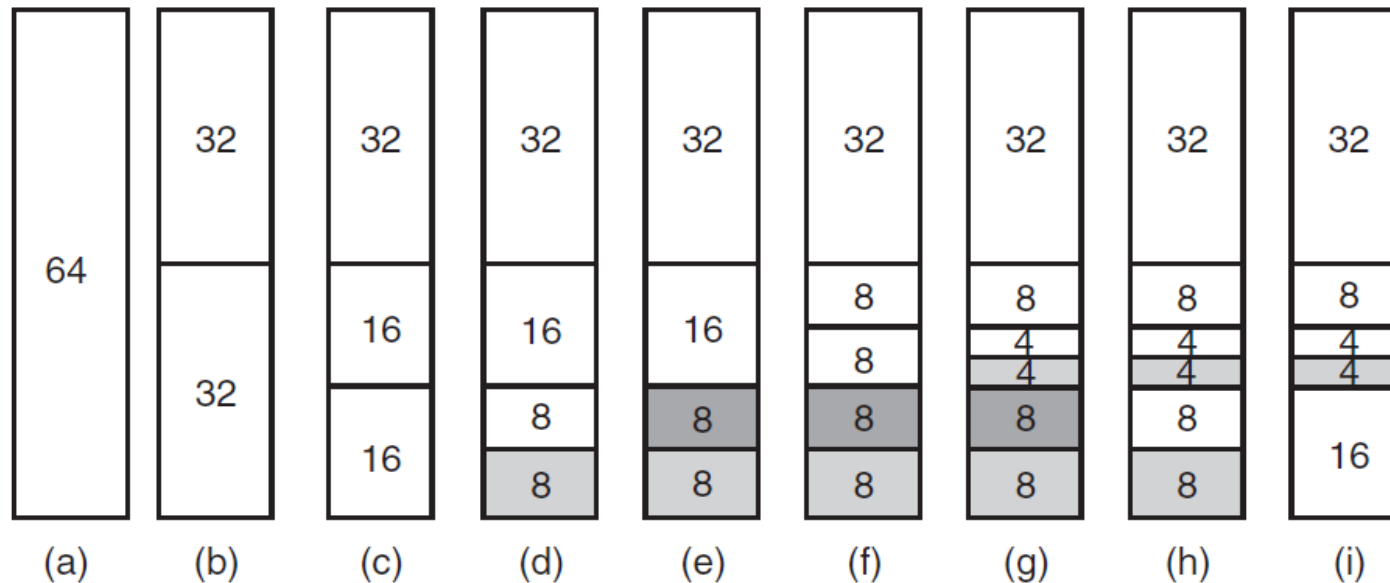


Figure 10-17. Operation of the buddy algorithm.

MOS4E

Problems

4. Consider a swapping system in which memory consists of the following hole sizes in memory order: 10 MB, 4 MB, 20 MB, 18 MB, 7 MB, 9 MB, 12 MB, and 15 MB. Which hole is taken for successive segment requests of
- (a) 12 MB
 - (b) 10 MB
 - (c) 9 MB
- for first fit? Now repeat the question for best fit, worst fit, and next fit.