Memory Management
Ch. 3

Memory Hierarchy

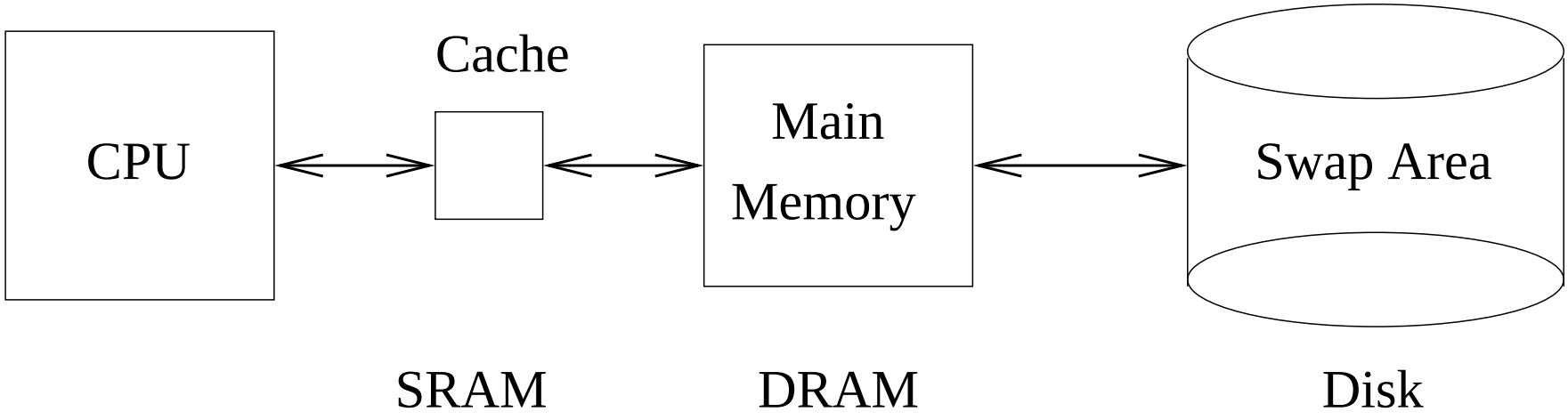
Cache RAM Disk

Compromise between speed and cost.

Hardware manages the cache.

OS has to manage disk.
Memory Manager

Memory Hierarchy

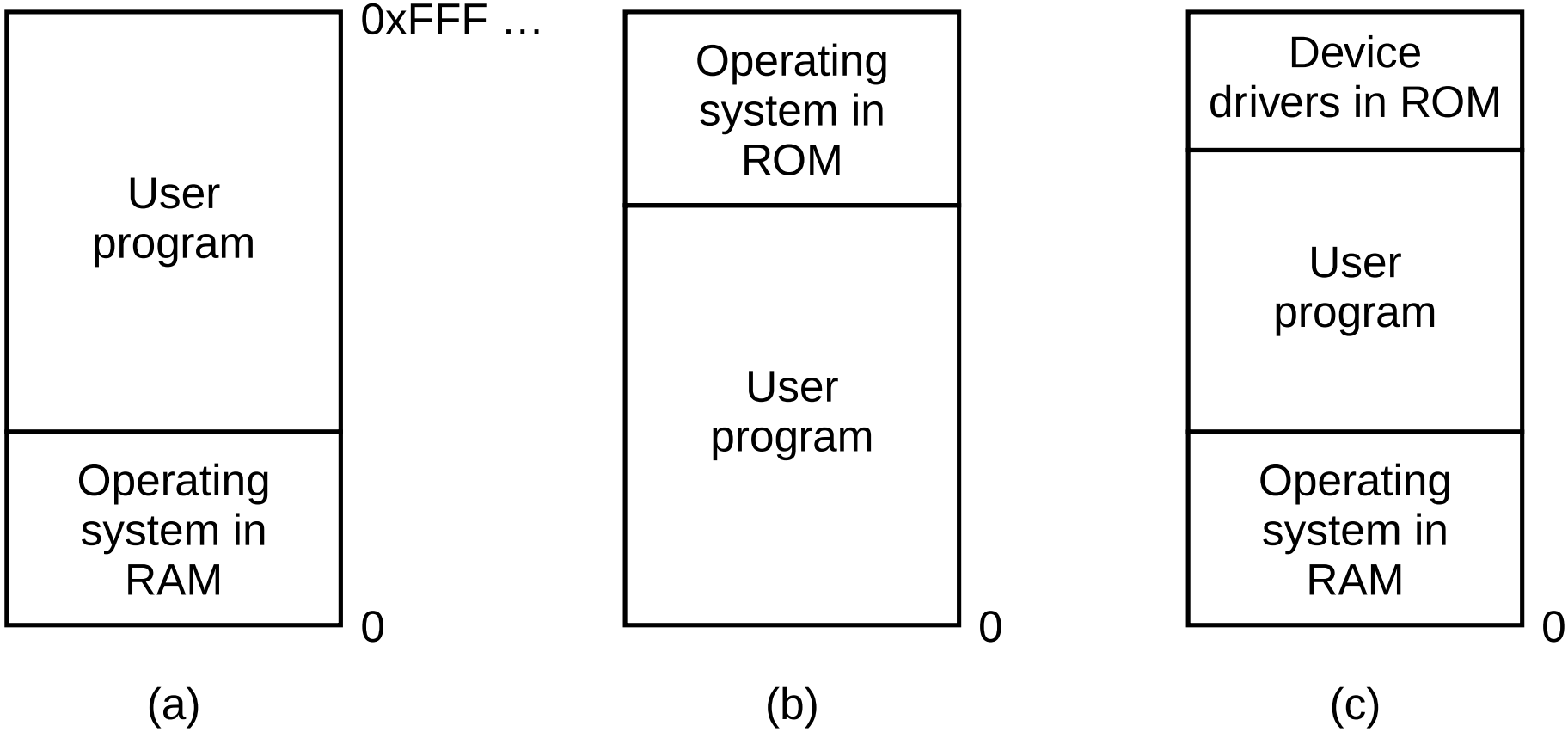


Two Classes

Move back and forth between memory and disk.

Don't

One Program at a Time

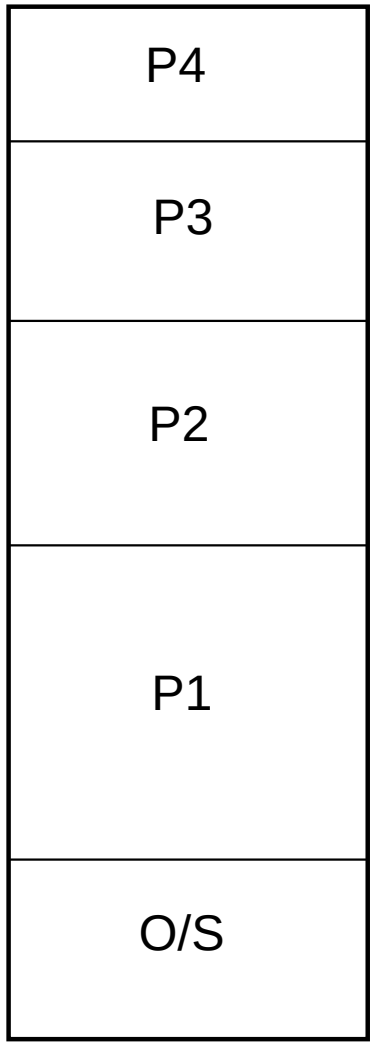


(a) Old Main/Mini

(b) Palm/Embedded

(c) DOS

Multi-Programming: Fixed Partitions



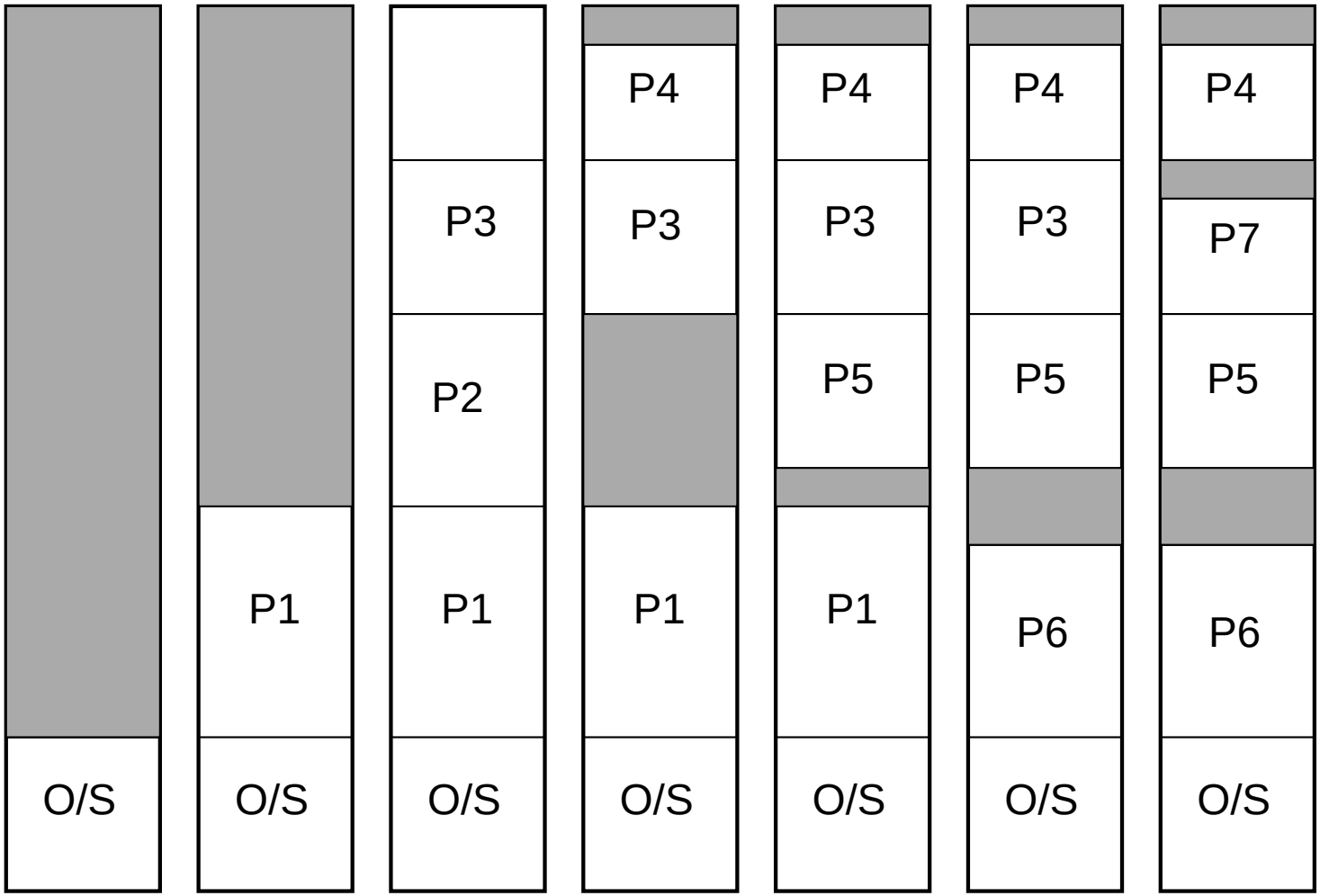
Set up fixed partitions at boot time.

Place each job into the largest partition in which it will fit.

Wasted space in each partition.

Small jobs when large partition free: Waste time or space.

Multi-Programming: Variable Partitions



Multi-Programming: Variable Partitions

Place jobs in available space as they arrive.

Creates Fragments

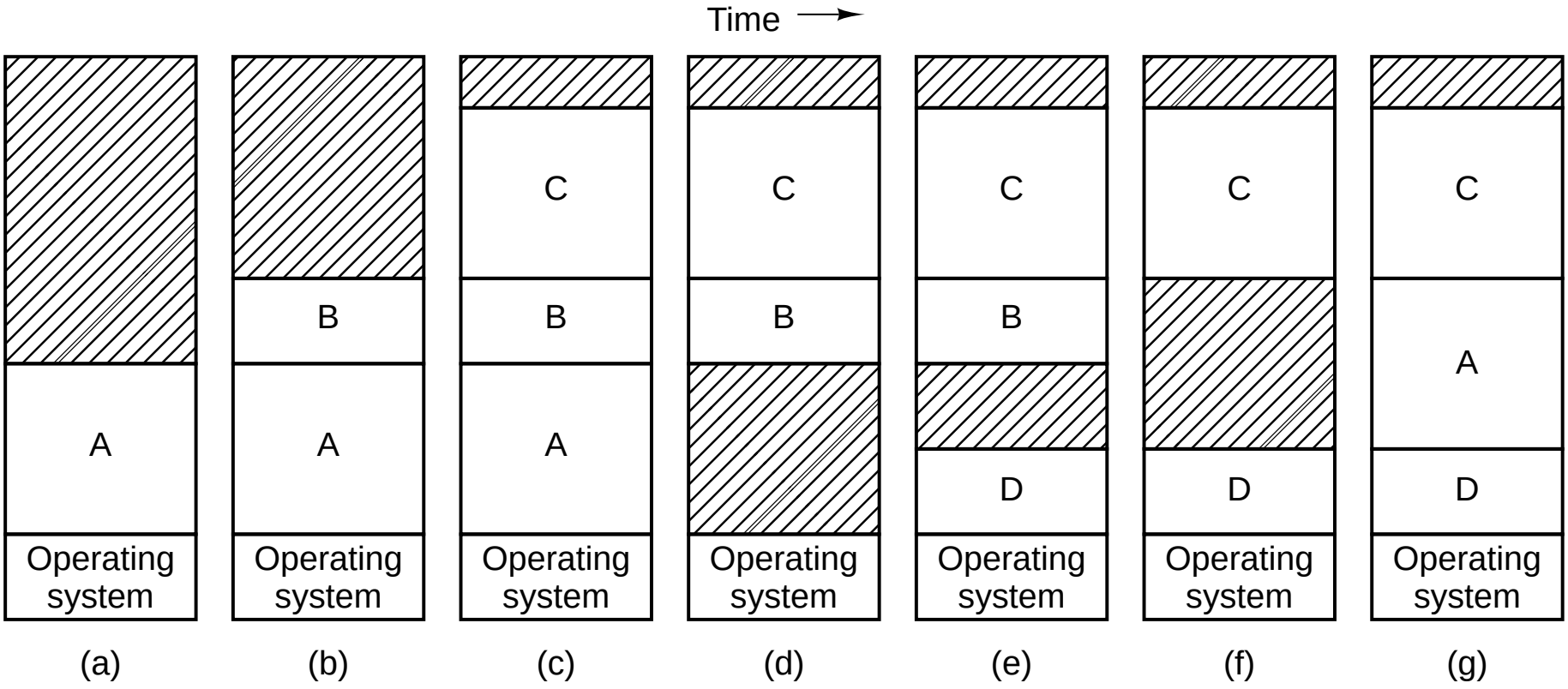
Can't fill an empty spot with a larger job
Exact fit very unlikely
Almost always something smaller

Creates small, useless empty slots.

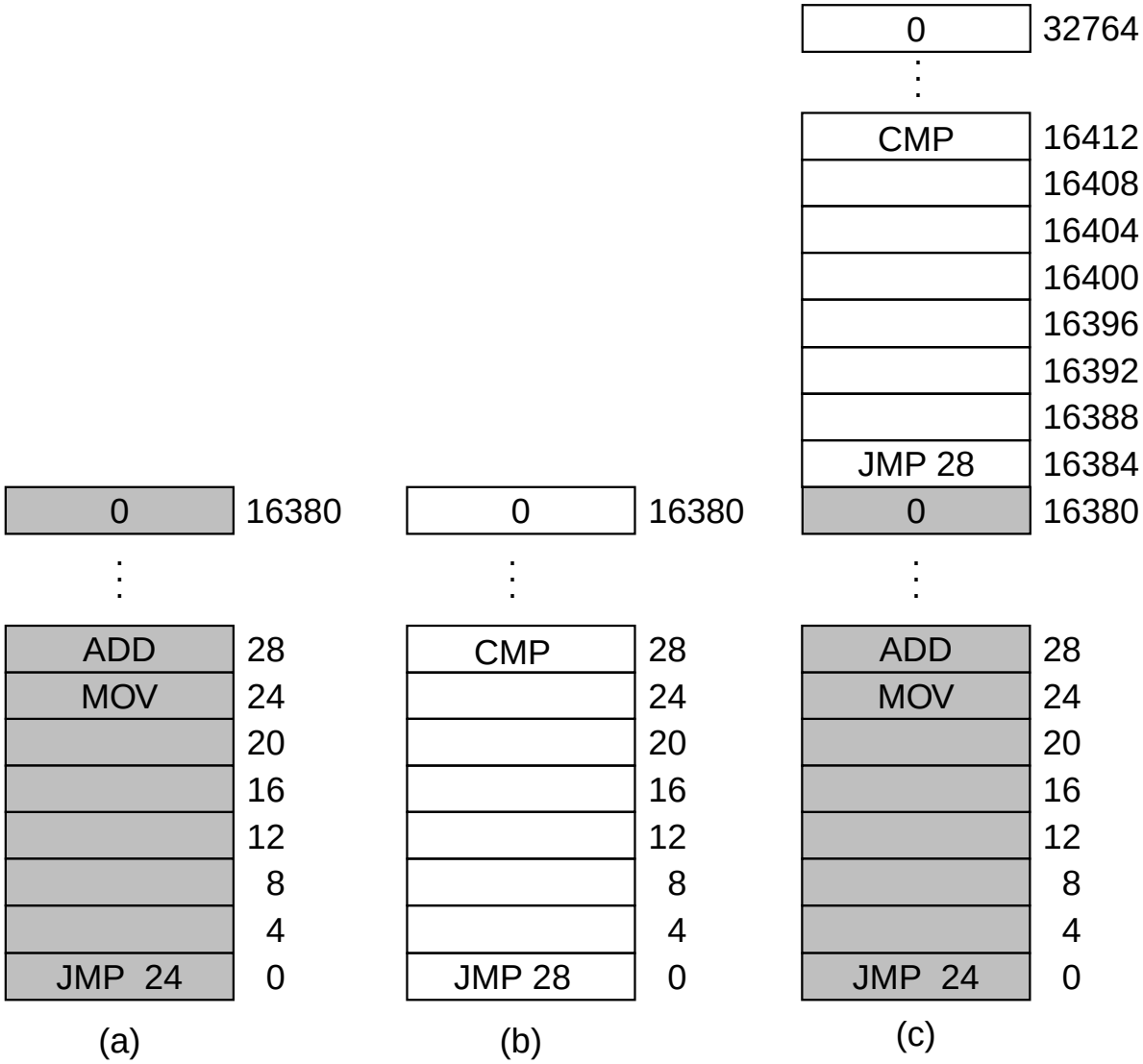
Compacting is expensive.

Swapping

A Job may be removed from memory and returned later.



Consider Two At Once



Relocation

Compiler assumes all programs start at address zero.

Add a base value to each address to
adjust to the actual memory location.

Relocation Required

A program may be have to run at different memory locations at different times.

Relocation at Start of Execution

The program may be placed into a partition anywhere in memory.

Relocation During Execution

Compacting to eliminate fragments.

Swapping out and back in.

Relocation Methods

Relocation at Start of Execution

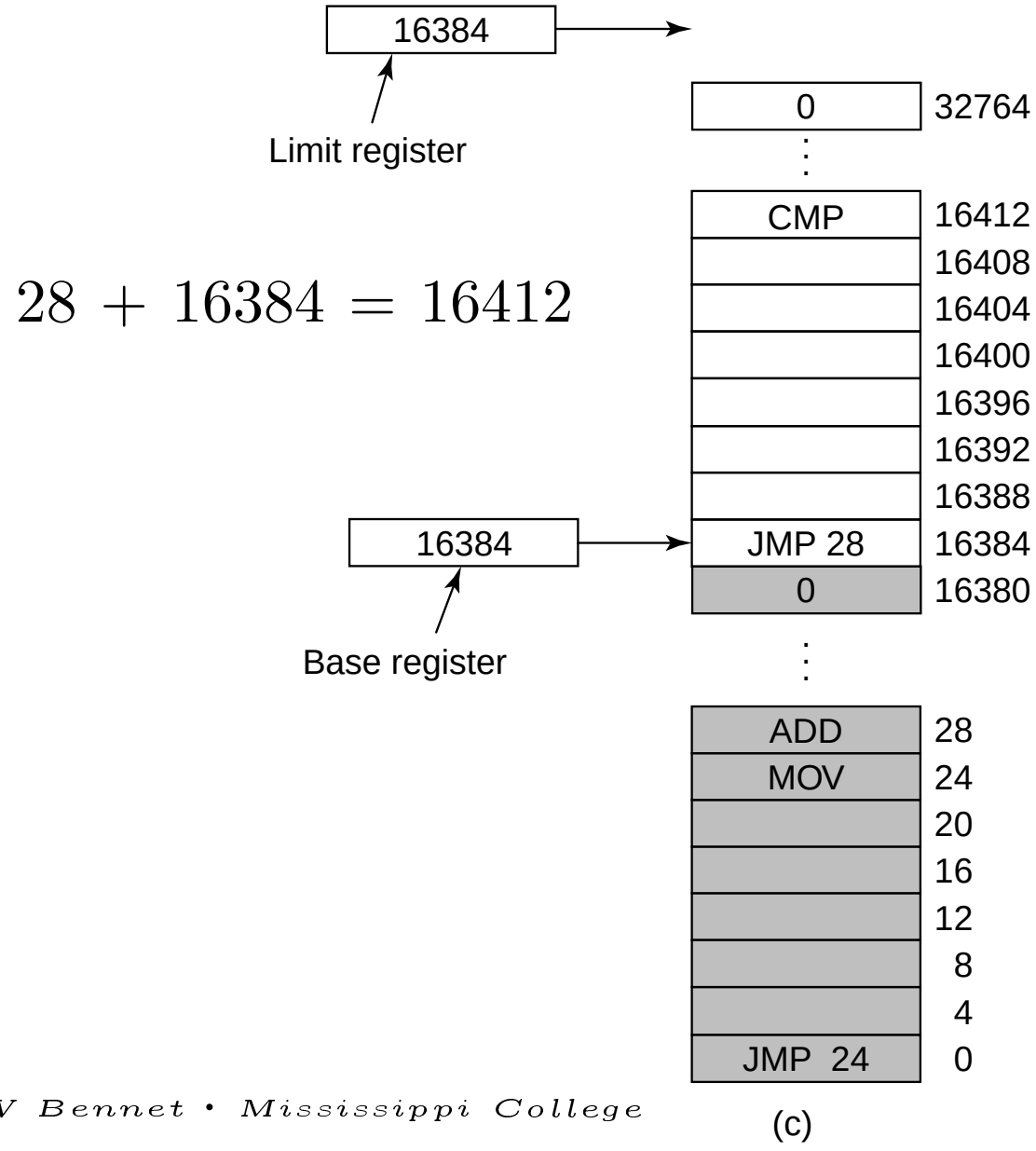
O/S can add the base when copying the executable from disk into memory.

Executable file must contain a relocation table to identify which values are pointers.

Relocation During Execution

Hardware adds base value to each address produced by the program.

Base And Limit Registers



$$28 + 16384 = 16412$$

Address Space

Programs operate in an *address space.*

The set of addresses used to address memory.

Usually differ in some way from the hardware addresses.

Hardware maps addresses and limits
programs to their own space.

Mapping and protection.

Overlays

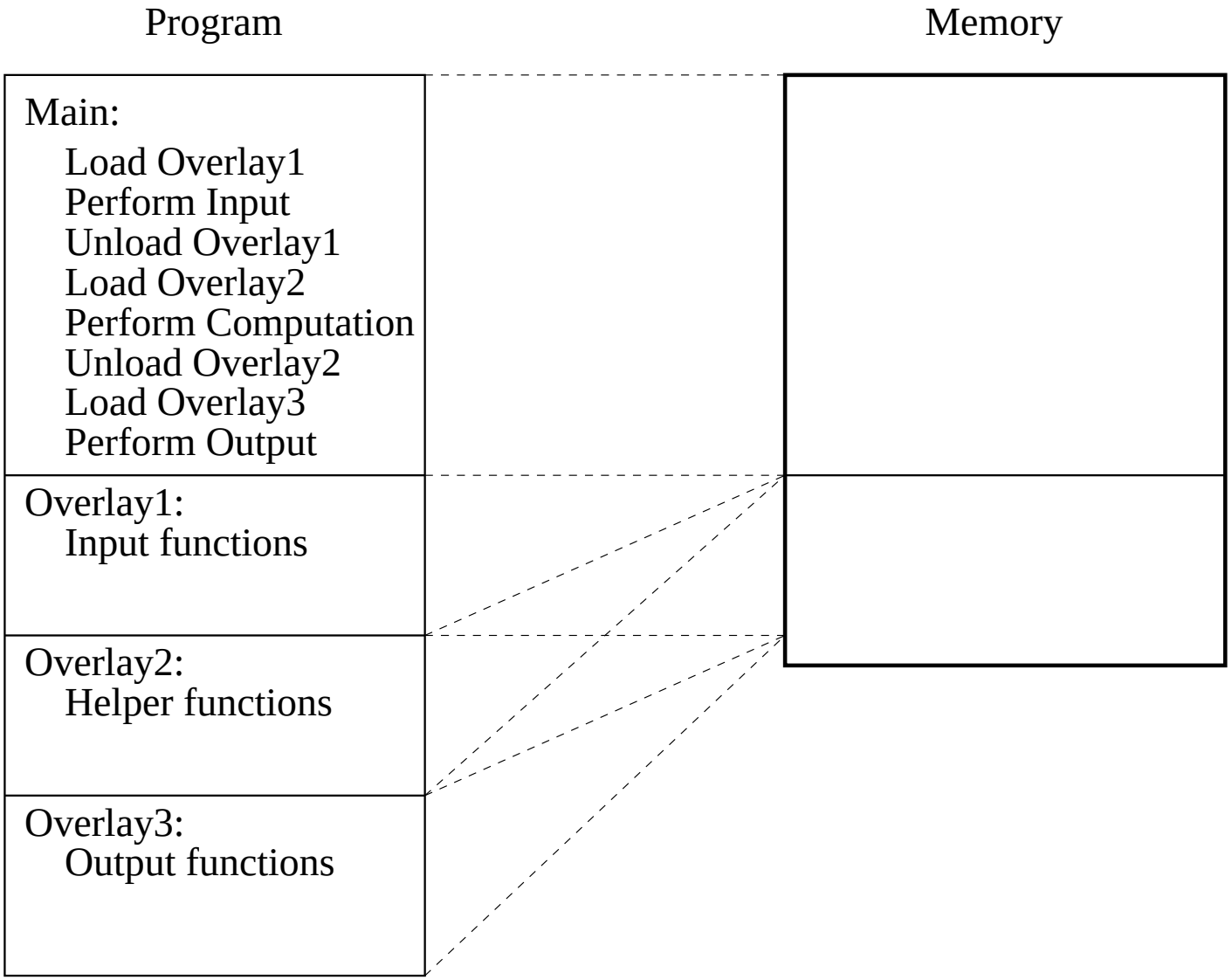
Allows programs larger than memory.

Programmer breaks his program into regions.

Regions are moved in and out under program request.

This is an obsolete technique

Overlays


Program
Memory
Main:
Load Overlay1
Perform Input
Unload Overlay1
Load Overlay2
Perform Computation
Unload Overlay2
Load Overlay3
Perform Output
Overlay1:
Input functions
Overlay2:
Helper functions
Overlay3:
Output functions

Paged Virtual Memory

Avoids fragmentation by dividing a
program into fixed-size blocks.

Automates the overlay technique.
Programs appear to have more memory than they really do.

Virtual Memory

Divide programs up into fixed-size pages.

Divide memory up into fixed size page frames.

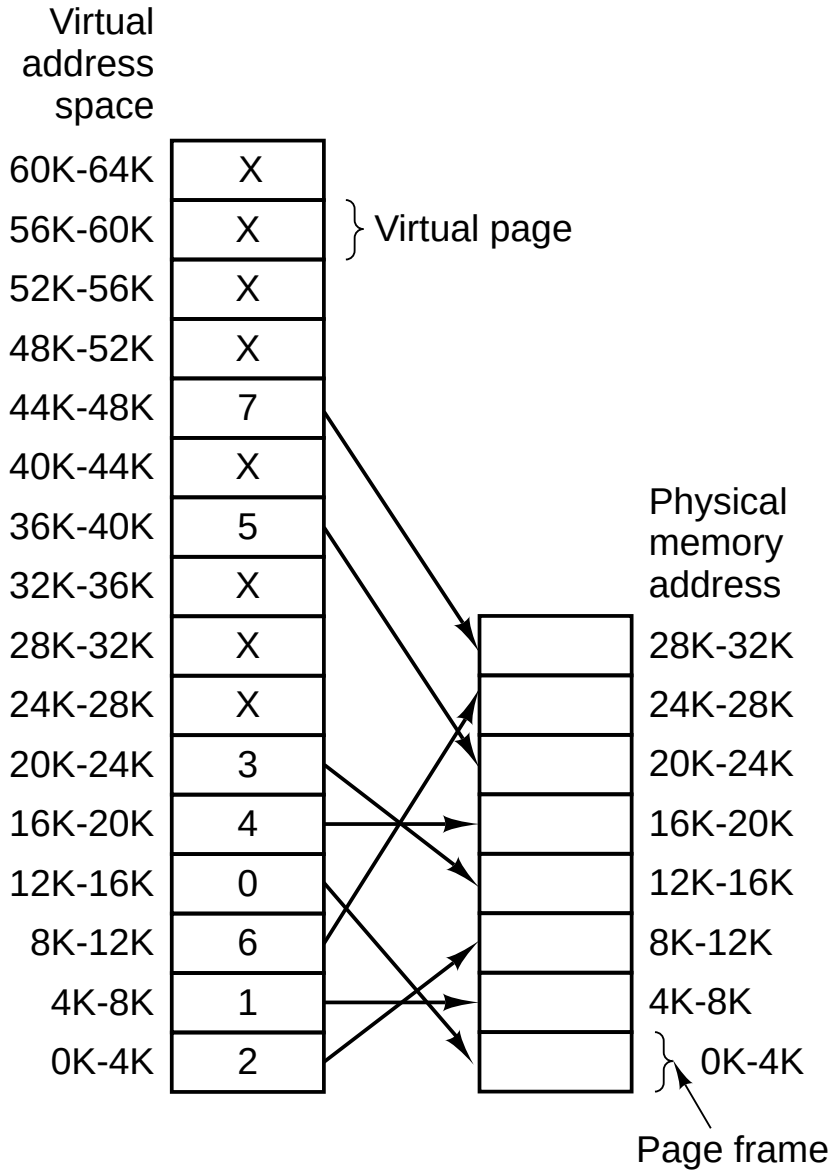
Put the pages in whatever frames are free.

Use a *page table* to map the programs *virtual addresses* to *real addresses.*

Pages that don't fit in memory
stay on disk and are brought in on demand.

Generally, each process will have its own page table.

Virtual Memory



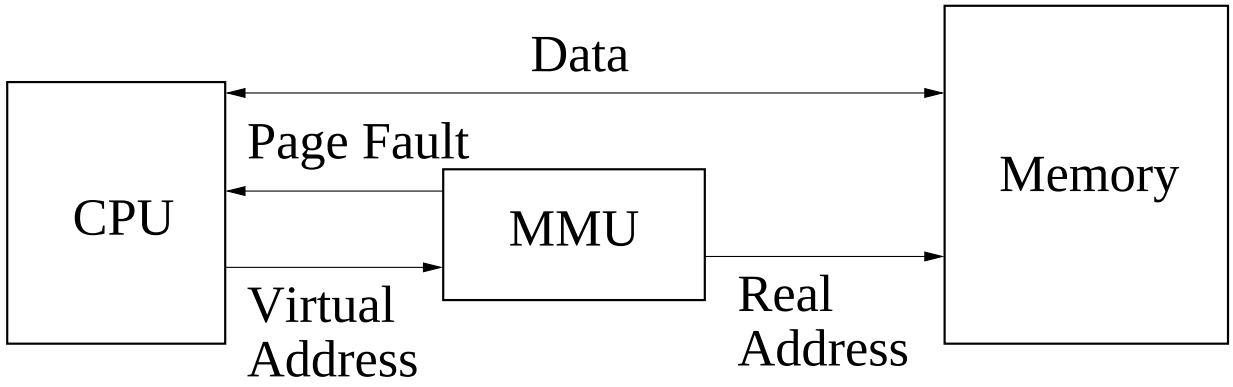
On A Memory Reference

The hardware translates the address using the page table.

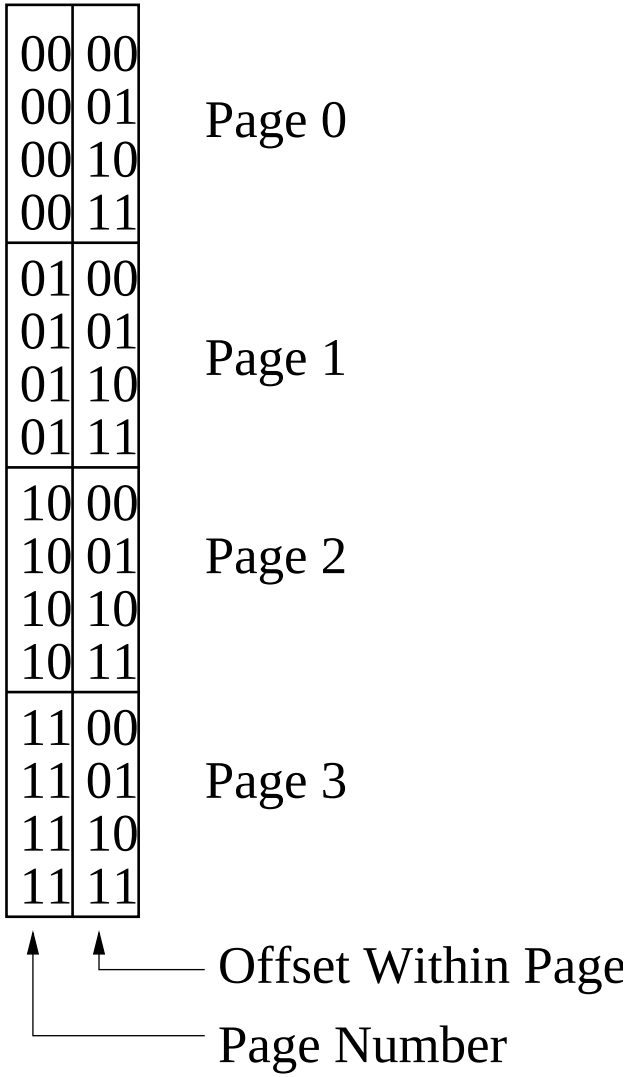
If the page is in memory, the hardware address is sent to the memory unit.

If not, trap to the O/S: *Page fault*

If a large proportion of memory references produce page faults, the system will run very slowly. This is called *thrashing*.



Pages and Address Bits

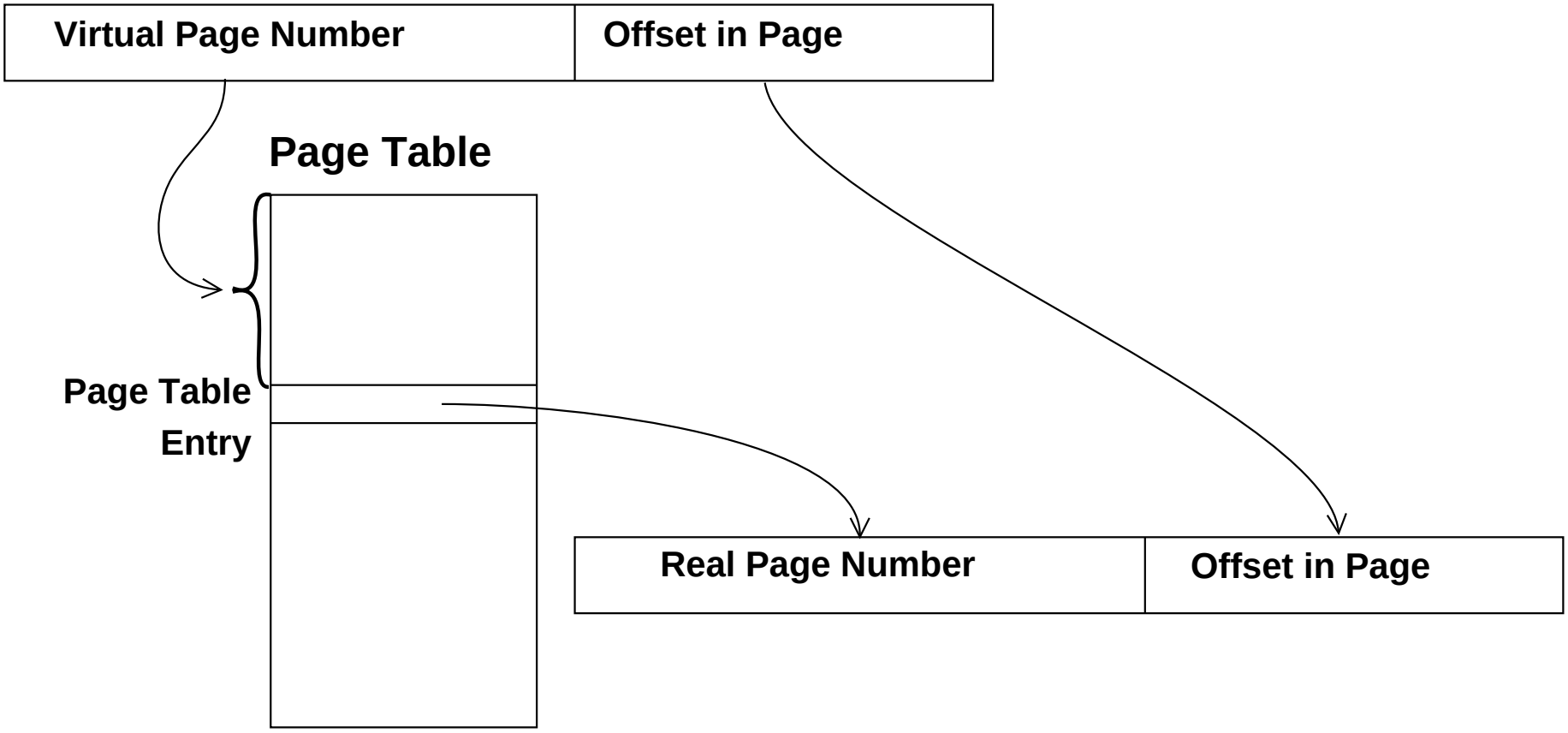


Four four-byte pages.

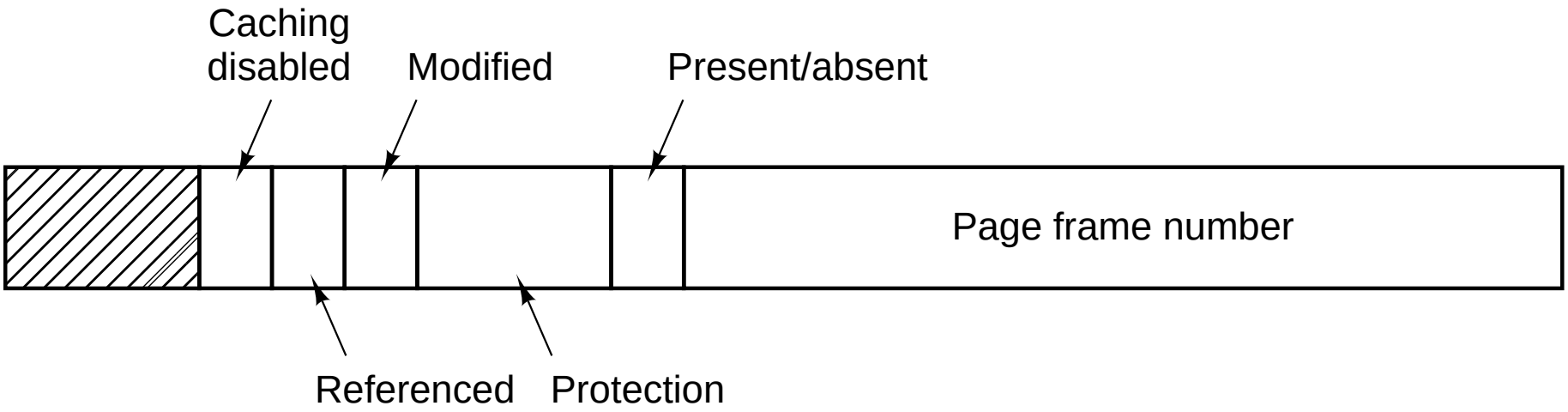
First part of address is the page number.

Offset carries into page number
at each page boundary.

Address Translation



Page Table Entry Structure



Where Is The Page Table?

In the MMU: The table is too big.

In memory: Memory is too slow.

Compromise: Translation Lookaside Buffer (TLB)

The Page table is in memory.

The TLB is a hardware cache of Page Table Entries (PTEs).

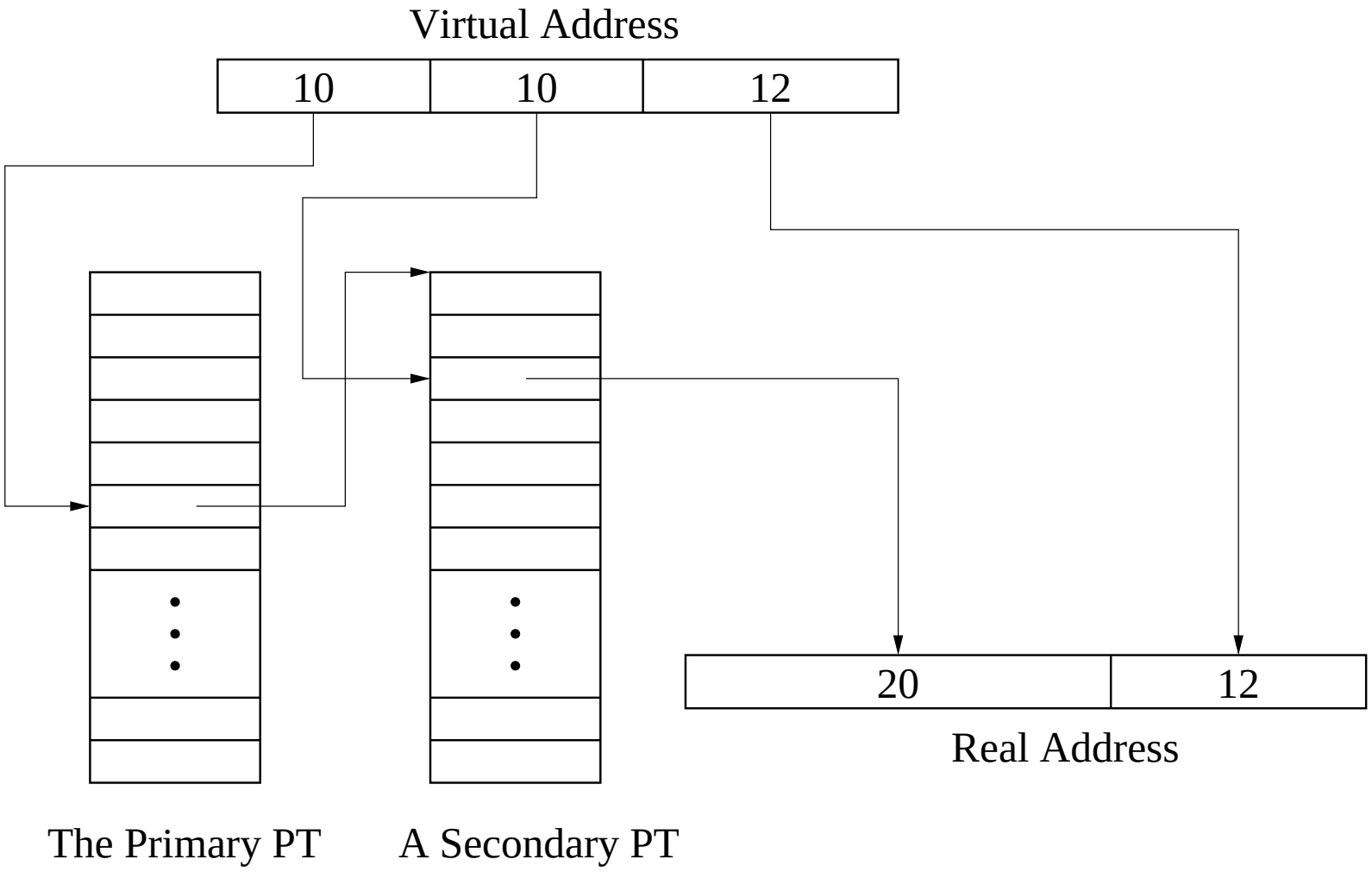
Most lookups hit the TLB.

TLB traditionally managed by hardware.
Newer architectures move this to the O/S.

TLB

Valid	Virtual page	Modified	Protection	Page frame
1	140	1	RW	31
1	20	0	R X	38
1	130	1	RW	29
1	129	1	RW	62
1	19	0	R X	50
1	21	0	R X	45
1	860	1	RW	14
1	861	1	RW	75

Multi-Level Page Table



Multi-Level Page Table

Each table is smaller.

Secondary page tables may be swapped out.
Only primary must stay in memory.

A memory reference may generate two faults.

The 386 class CPUs use a two-level.

Some newer designs use three levels.

Inverted Page Table

Keep an entry for each page frame,
not for each virtual page.

Much savings when VM much larger than real memory.

Can no longer just index with the address to find the entry.

Most lookups hit the TLB.

O/S keeps a hash table for ones that don't.

Page Replacement

On a page fault, the O/S must bring the page into memory.

If all real pages are in use, one must be replaced.

The policy used to choose the victim is the *page replacement algorithm.*

The goal is to minimize page faults.

Optimal Replacement

Choose the page which remains unwanted for the longest time.

Impossible to implement.
Requires prediction of the future.

Used as a comparison for practical methods.

Optimal Replacement Example

Virtual Page References:

1 2 3 4 5 4 6 3 6 2 3 7 1 2 5 3 6 5 6

Real Memory:

Not Recently Used

Depends on a referenced and modified bits in the PTE.
These are set on modification or reference.

Periodically, all referenced bits are cleared.

Four classes (RM):

- 00: Not refed, not mod.
- 01: Not refed, modified.
- 10: Referenced, not mod.
- 11: Referenced, modified.

Choose a page at random from the
lowest-numbered non-empty class.

FIFO/Second Chance

Plain FIFO: Choose the page which has been in memory the longest.

Keep a queue: Add new pages to the end; evict pages at the head.

May throw out popular pages.

Second chance algorithm

Before eviction, check the reference bit.

If it is one, clear it, move the page to the tail, and check the next one.

FIFO/Second Chance Example

Virtual Page References:

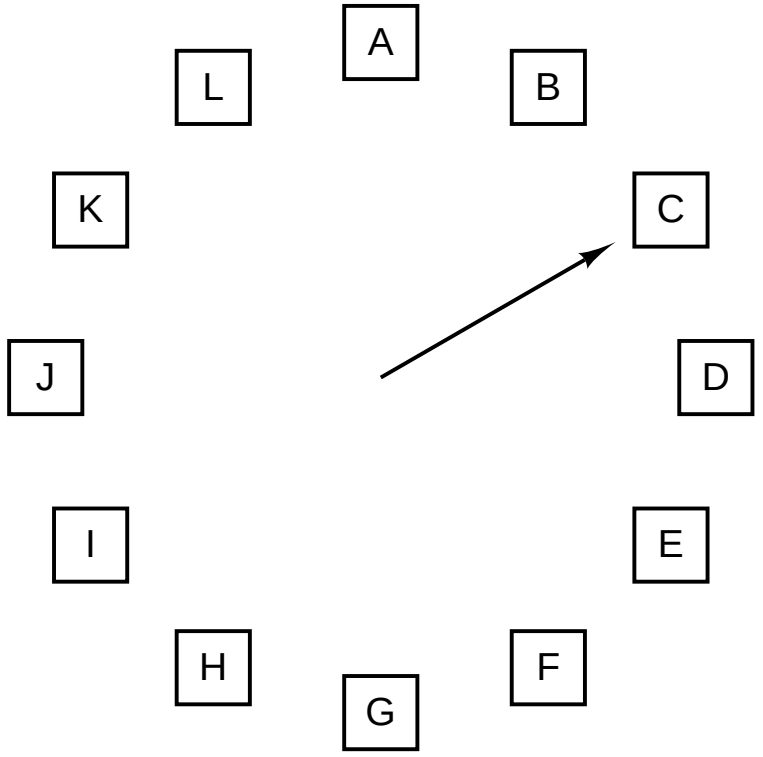
1 2 3 4 5 4 6 3 6 2 3 7 1 2 5 3 6 5 6

Real Memory:

Clock Algorithm

A more efficient version of second-chance.

The pages are arranged in a circular linked list. There is a pointer *Current* to the current page frame.



When a page fault occurs,
the page the hand is
pointing to is inspected.
The action taken depends
on the R bit:

R = 0: Evict the page

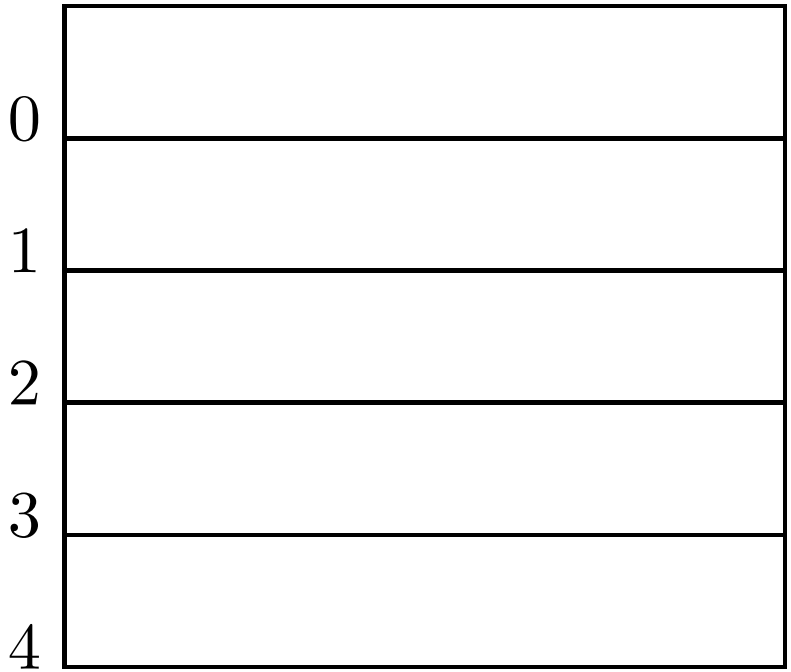
R = 1: Clear R and advance hand

Clock Example

Virtual Page References:

1 2 3 4 5 4 6 3 6 2 3 7 1 2 5 3 6 5 6

Real Memory:



LRU

Choose the page which has remained unused for the longest time.

Best possible algorithm.

Expensive to implement.

Can be done with specialized hardware.

LRU Example

Virtual Page References:

1 2 3 4 5 4 6 3 6 2 3 7 1 2 5 3 6 5 6

Real Memory:

Page Aging

Each page has a counter which keeps its *age*.

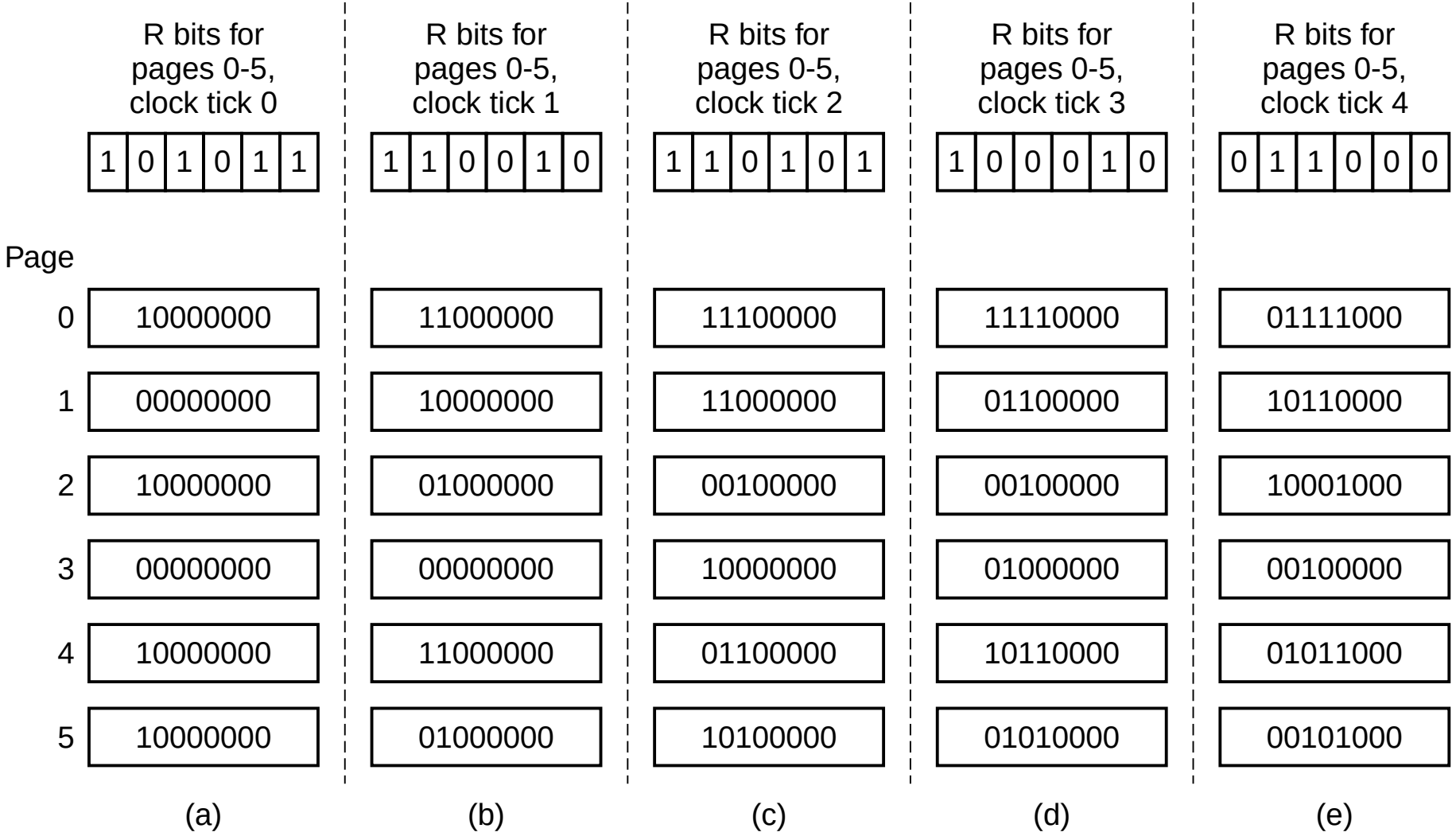
Each PTE has a referenced bit.

At Each clock tick:

- Shift each age right.
- Set the high bit for each page which has its referenced bit on.
- Clear all referenced bits.

Evict the page with the smallest age.

Aging Example



LRU and Aging

Aging does not distinguish *when* a
reference occurs during a clock period.

Aging does not distinguish pages old enough to have zero age.
Zero is reached when the number of
clocks is the number of bits.

Working Set Theory

For any process, $W(k, t)$ is set of pages at time t that the process has referenced during the period k.

k is theoretically a number of memory references.

In practice, a time period is often used.

Time is counted only when the process is running.

Working Set

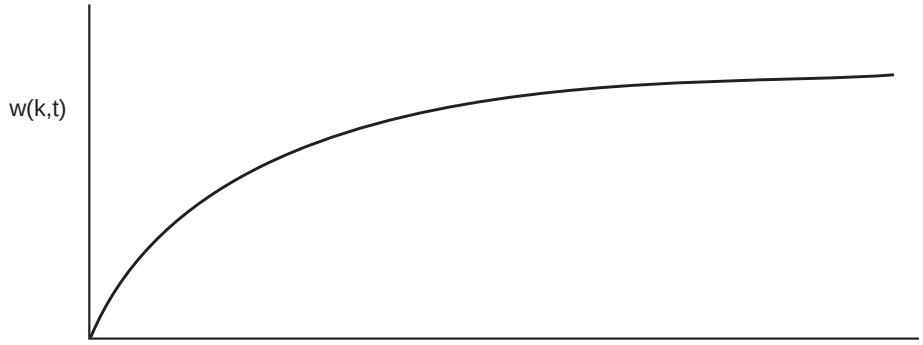
References back from t:
18, 24, 17, 24, 15, 17, 17, 18, 24, 18, 17, 24, 23, 18, 15, 24

$$W(2,t) = \{18, 24\}$$

$$W(3,t) = W(t,4) = \{17, 18, 24\}$$

$$W(5,t) = \ldots = W(t,12) = \{15, 17, 18, 24\}$$

$$W(13,t) = \ldots = W(t,16) = \{15, 17, 18, 23, 24\}$$



Using The Working Set

Choose some arbitrary amount of time to
be the size of the WS.
Called τ

Idea is to evict a page which is not in the WS.

Too impractical to compute the exact working set.

Basic Support

We still have referenced and modified bits set by hardware.

We have a virtual time for each process.
A counter which runs while the process is on the CPU.

We have periodic timer interrupts used to
clear the referenced bits.

Basic Algorithm

Each PTE has a field for the last time referenced.

On each page fault, scan the whole table.

For each referenced PTE, set the last time referenced to the process virtual time.

For un-referenced PTEs, the age is the current time less the last referenced field.

Pages older than τ are listed for removal; the first is replaced.

If no page is older then τ, the oldest is replaced.

If all pages are referenced, one is chosen at random.

WSClock Algorithm

Scan the PTEs in a circular list with a current location.

- If $R = 1$, clear and proceed.
- If $R = 0 \wedge age > \tau \wedge M = 0$, evict.
- If $R = 0 \wedge age > \tau \wedge M = 1$, schedule write and proceed.
- If all around with some writes, wait for one to finish.
- If all around with no writes, take the current page.

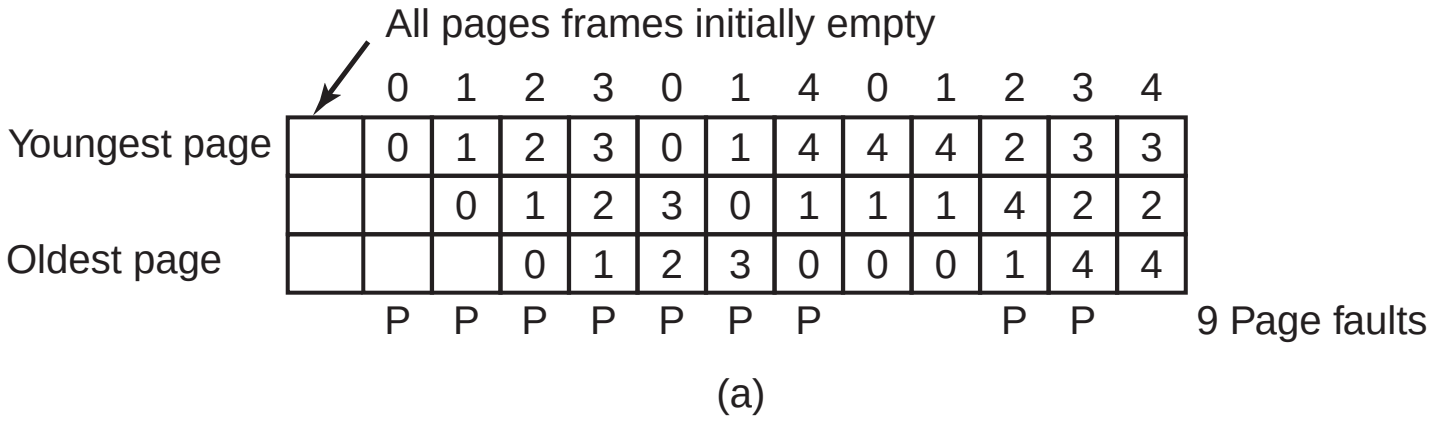
May limit the number of writes scheduled.

Summary

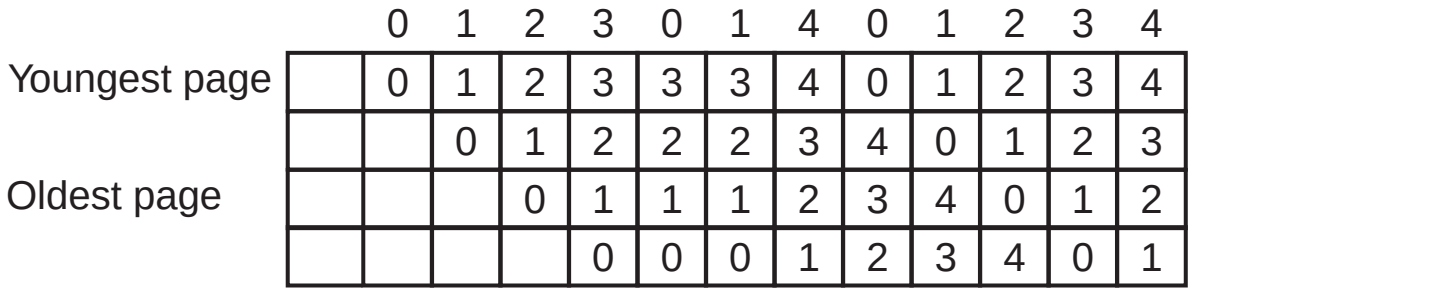
Algorithm	Comment
Optimal	Not implementable, but useful as a benchmark
NRU (Not Recently Used)	Very crude
FIFO (First-In, First-Out)	Might throw out important pages
Second chance	Big improvement over FIFO
Clock	Realistic
LRU (Least Recently Used)	Excellent, but difficult to implement exactly
NFU (Not Frequently Used)	Fairly crude approximation to LRU
Aging	Efficient algorithm that approximates LRU well
Working set	Somewhat expensive to implement
WSClock	Good efficient algorithm

Belady's Anomaly

More memory, more faults.



(a)



(b)

Stack Algorithms

Stack algorithms do not display Belady's Anomaly.

$$M(m, r) \subseteq M(m + 1, r)$$

At r through the execution, the pages kept in m pages of real memory is a subset of those kept in $m + 1$ pages.

LRU is a stack algorithm.

Stack Algorithms

By definition,

if $p \in M(m, r)$
then $p \in M(m + 1, r)$

So, if any page is in memory when the memory size is m, it is also in memory when the size is $m + 1$.

So adding memory cannot create a new page fault.

Local v. Global

When process P faults we can:

Select the "best" page from among all of them.
global

Select the "best" page from those owned by P.
local

Local means that P owns its frames.
Global means they are shared.

Global allows processes' working set to change.

Most replacement algorithms can use either
Not working-set

Page Fault Frequency (PFF)

Local should use some policy to adjust the number of pages each process owns.

Measure the time frequency of page faults for each P.
Gives a sense if it owns enough page frames.

Add frames when PFF is high
Remove frames when PFF is low

Load Control

Can have too much load for the existing memory.
If total working set exceeds memory, you will thrash.

If PFF is high for some processes and low for none.

Swap something out.

Swapping may also take account of I/O behavior to keep CPU busy.

Page Size

May be fixed or limited by hardware.

Larger pages increase internal fragmentation.
Half the last page.

Larger pages may increase memory requirements.
A whole page must be brought in, even if the program doesn't need it all.

Smaller pages mean larger page tables.

Smaller pages usually take more time per byte to transfer to disk.

4K is common 8K increasing popularity

Balance Page Size and Table Size

$$s = \text{Process size}$$
$$p = \text{Page size}$$
$$e = \text{PTE size}$$

$$overhead = \frac{se}{p} + \frac{p}{2}$$

Set first derivative equal to zero to find minimum:

$$p = \sqrt{2se}$$

If $s = 1MB$, $e = 8$, then $p = 4K$

Separate Instruction and Data Spaces

Separate instruction and data memories.

One is chosen based on type of purpose of fetch.

Separate page tables.

Processes Running The Same Program

Several processes may run the same program.

The same pages can be mapped into each process.

May be possible to map at the first level.

Need to keep track of shared pages in case one is swapped out.

After a Unix Fork

Both processes have the same process image.
Generally share the pages.

Each page is shared until a process changes one.
Separate copies are then made: Copy On Write.

Invisible to processes.

When a process forks, its pages are set read-only and shared.

If either process writes a page, it traps.

O/S makes separate copies of each page and restarts.

Only pages actually written must be copied.

Sharing Pages for Communication

Visible to Processes

Processes may be able to name portions of address space.

They can agree to share these regions.
The O/S maps the pages into both tables.

Cleaning Policy

Helps to have a pool of pages which are ready to evict.
Avoids waiting for the decision when a page is needed.

A *paging daemon* periodically wakes up and runs the selection algorithm to find pages to evict.

When a dirty page is chosen to evict, the write-back is started.
Likely to be clean whenever needed.

Doomed pages retain their content until needed.

If referenced before being replaced, no page-in is needed.

OS Tasks

Allocate page table at process start.
In memory when process is not swapped out.

When the process is started on the CPU, the MMU must be set to use the page table.

Service page faults.

Free the page table when the program exits.

Page Faults

CPU traps and starts the OS.

Vector saves registers and calls a function in the OS.

Find which virtual page was referenced.
OS register, or find the saved PC and analyze it.

If the reference was illegal, the process is terminated.
Otherwise, it is suspended until the page is brought in.

Choose a page frame
Perhaps from the list maintained by the paging daemon.

Page Faults, Cont.

If the frame is dirty, schedule the I/O.

Schedule an I/O to bring the page in from disk.

Update the page table.

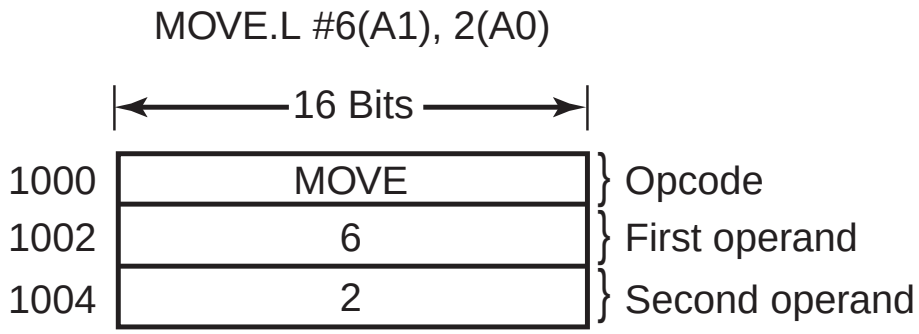
Faulting process is marked ready.

When scheduled, registers are reloaded.
Faulting instruction is retried.

Restarting The Instruction

A single instruction involves several addresses.
Its own, and perhaps several arguments.

Any of these might cause the fault.



If the instruction bytes are fetched separately, the PC may be part way through the instruction.
May be hard to know where the start of the instruction is.

Hardware may provide special registers to help out the O/S.

Computer Organization Note

The Pentium has most of the difficult features mentioned on the last page.

Note that the MIPS design studied in Computer Organization has fewer.

All instructions are four bytes and start on an address which is a multiple of four:
No instruction can cross a page boundary.

At most one memory reference per instruction.

Locking Pages

If a an I/O operation is active, the page containing the I/O buffer had best not be reallocated.
Read might complete into the new page. Ooops

Such frame must be *pinned* in memory until the I/O completes.

Not a problem if O/S restricts I/O to its own buffers.

Backing Store

Allocate a swap area on disk.

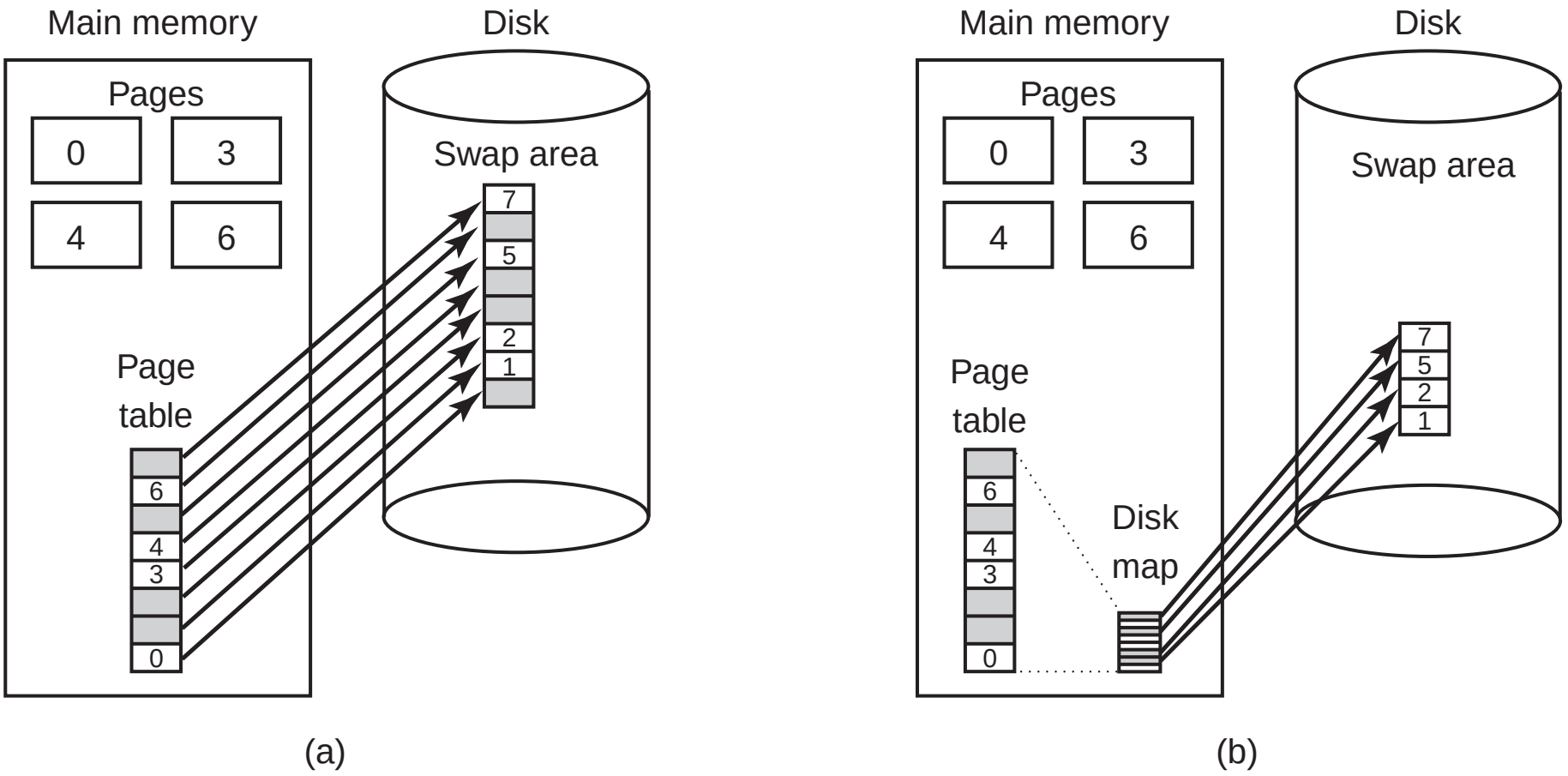
May create images of the virtual memory on the disk.
Must manage regions much like partitions in memory.

Each page table has a disk location,
with page table offset by its number.

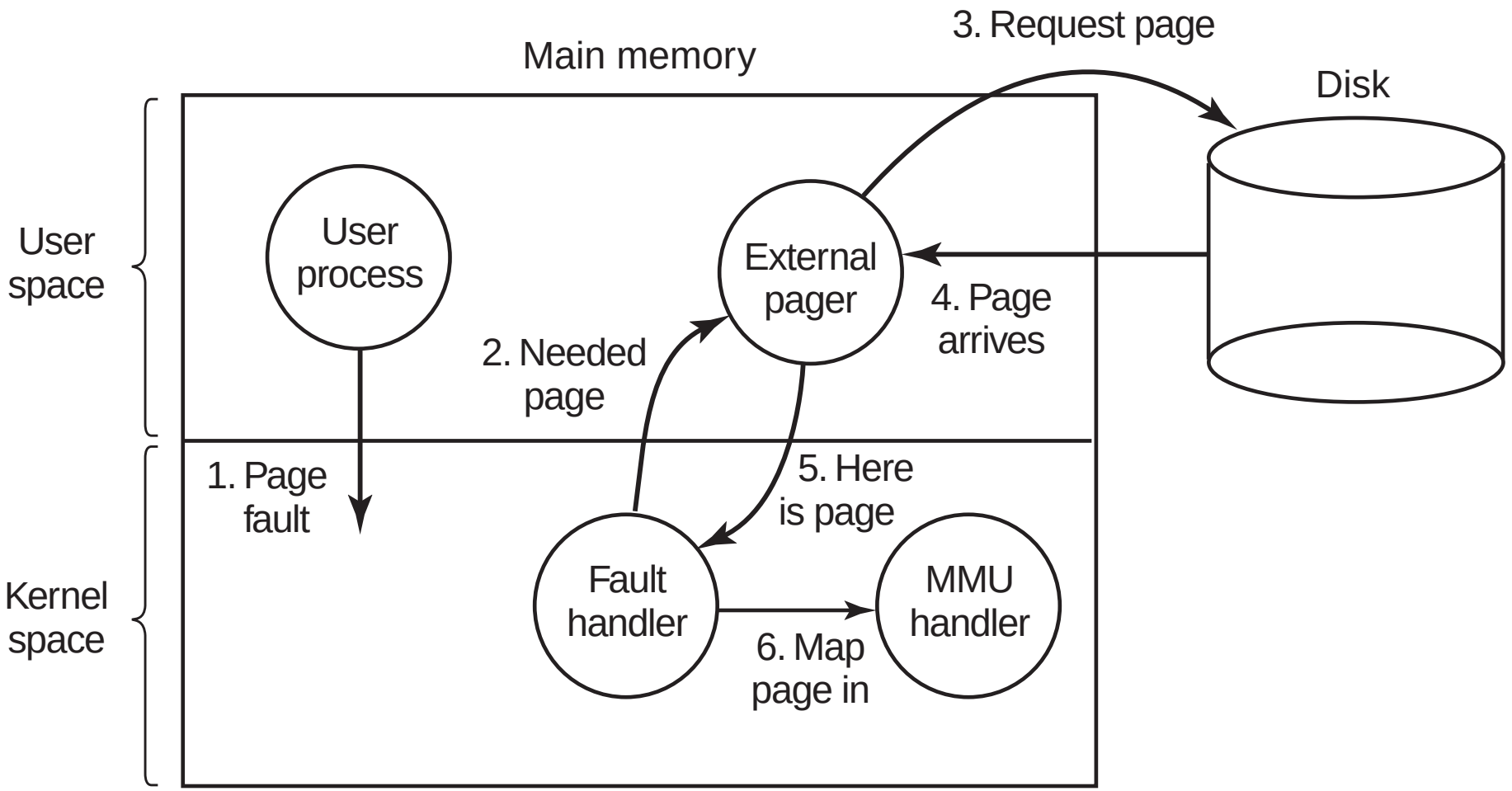
May allocate page images on disk as needed.

Each page not in memory has it own random location on disk.

Backing Store



User-Level Paging



Segmentation

Addresses are two-part.
Segment number and offset.

Programmer (or compiler) is aware of segments.

A segment table gives the memory location of the segment.

Segments are of various sizes.

Each segment is something meaningful to the program.
A data structure, library, function, etc.

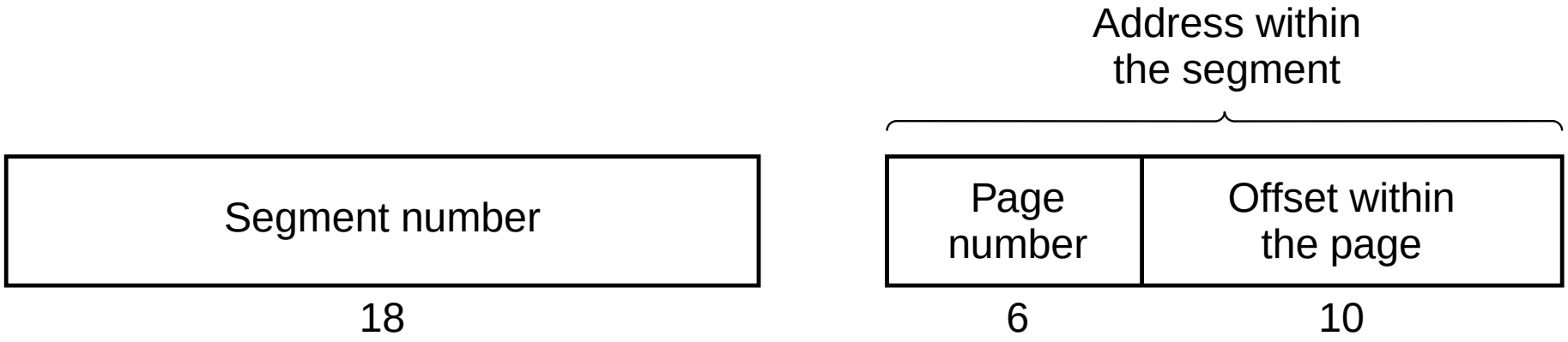
Permissions implemented at the segment level.
More meaningful than at the page level.

Segmentation and Paging

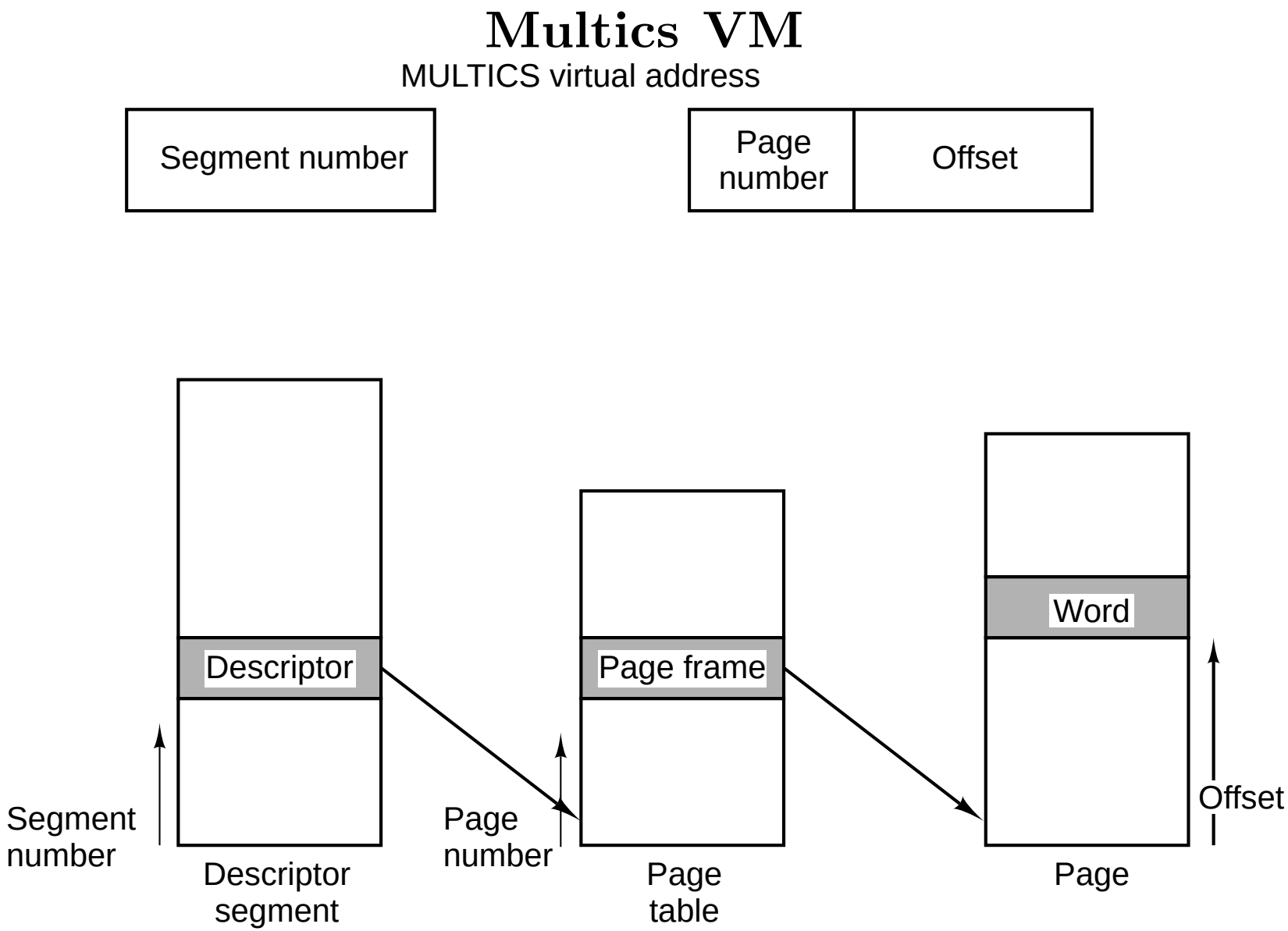
Generally implemented with paging.
Otherwise, you have to manage segments in memory.

The segment table denotes a page table.

The page table maps the segment offset to a real page.



Multics VM

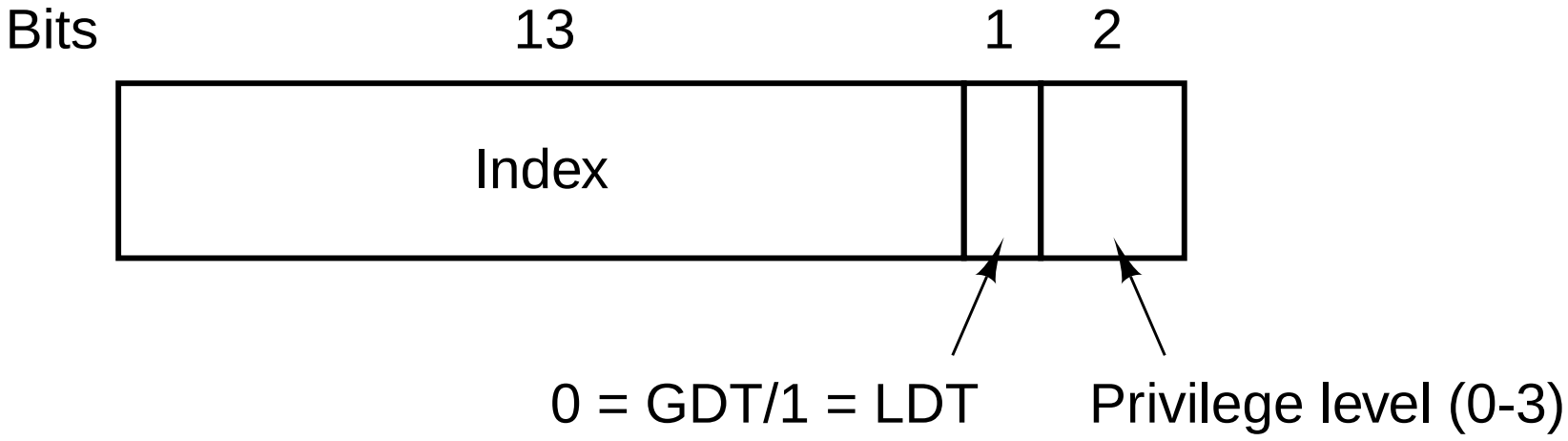

MULTICS virtual address
Segment number
Page number
Offset
Descriptor
Word
Page frame
Segment number
Page number
Offset
Descriptor segment
Page table
Page

Pentium Segmentation

Current segment is selected by CS, DS, SS, or other segment register.

Register may be selected by an instruction prefix, or by default based on the memory operation.

Segment register contains a 16-bit value with these fields.

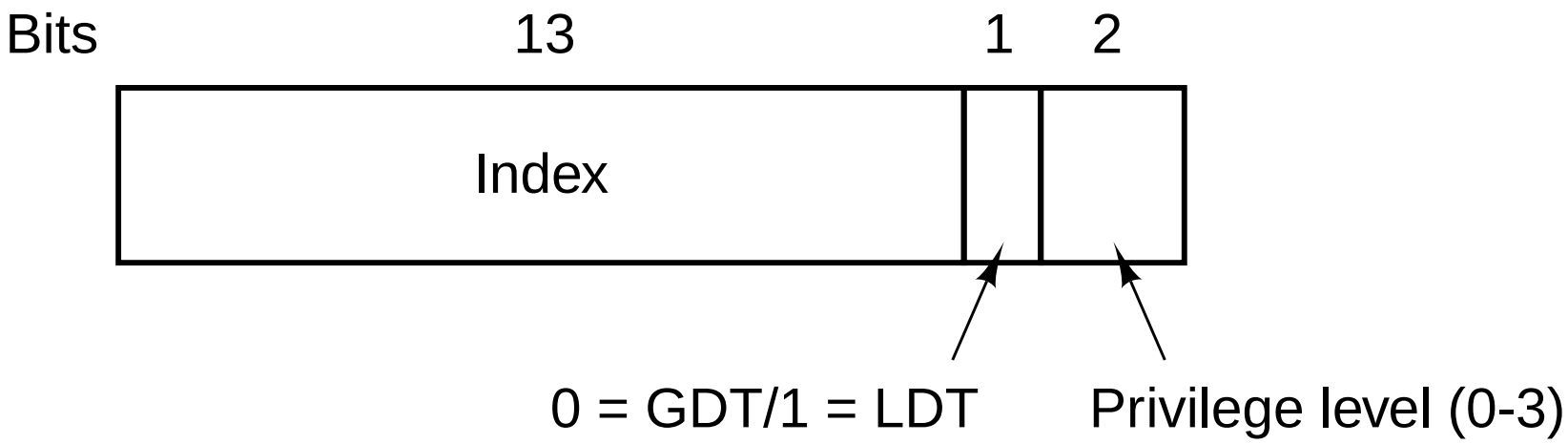


Descriptors

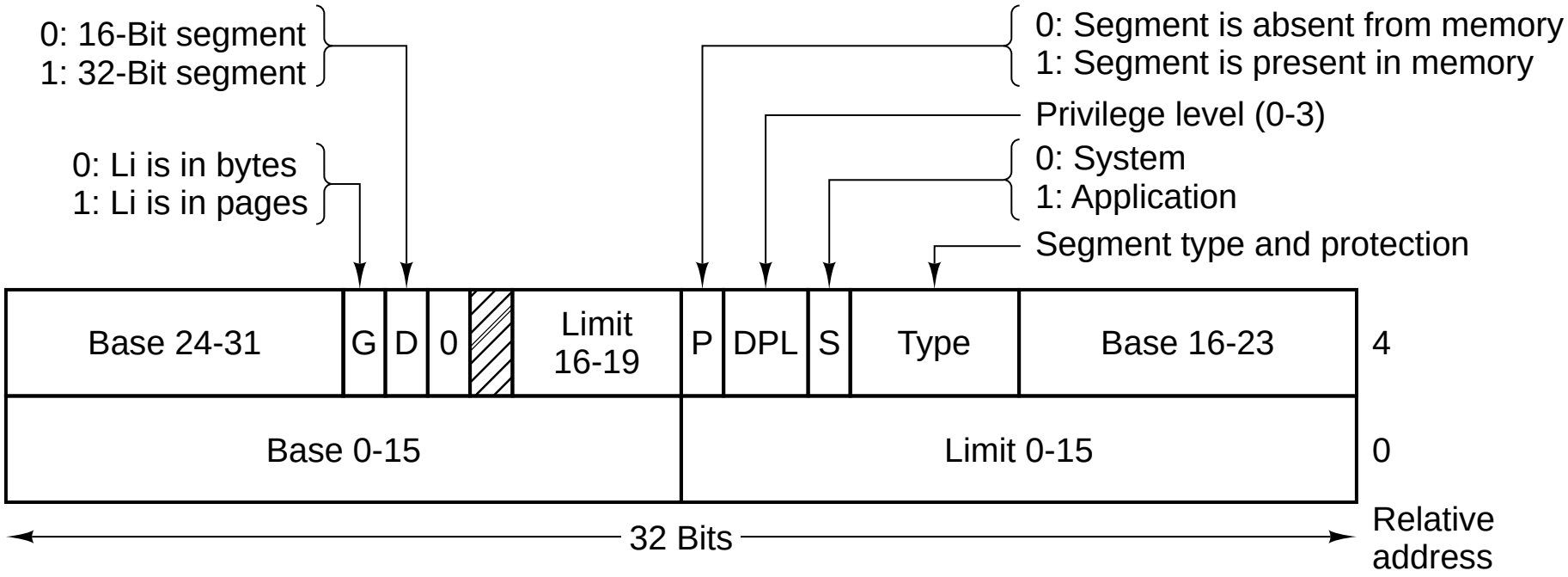
Descriptors denote a descriptor by index into one of two tables.

Global Descriptor Table
Local Descriptor table

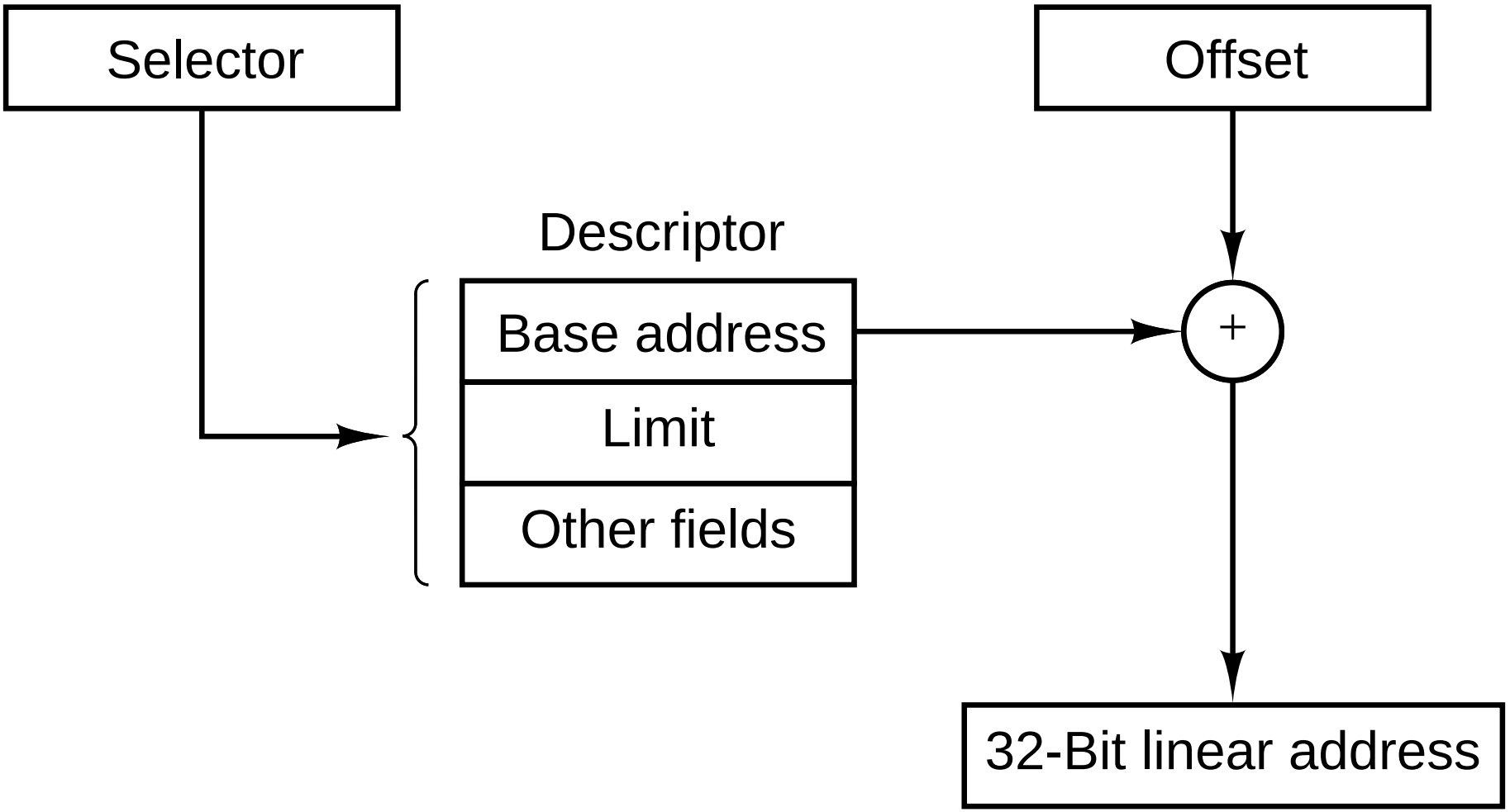
Location and size denoted by the gdtr and ldtr control registers.



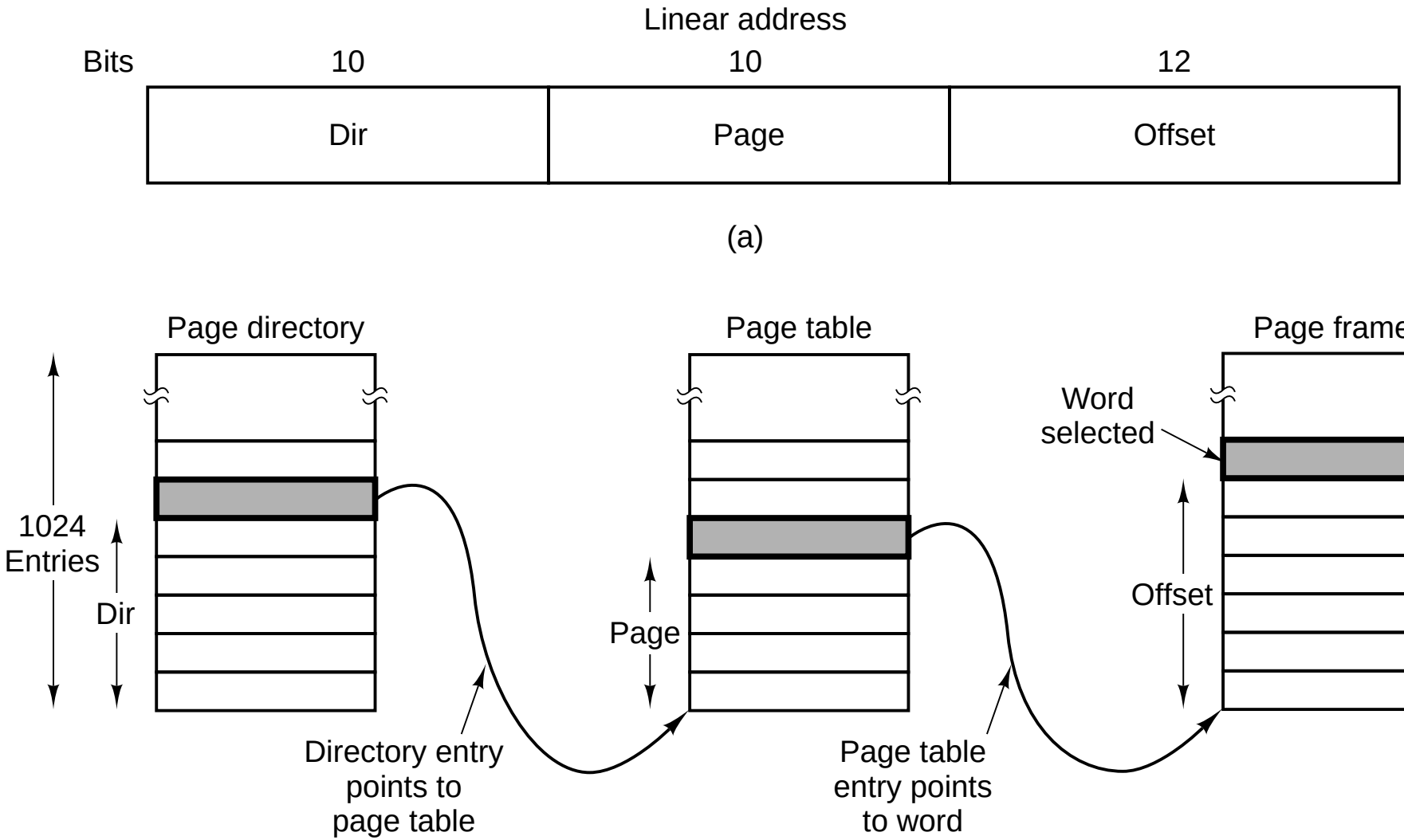
Descriptors (Descriptor Table Entries)



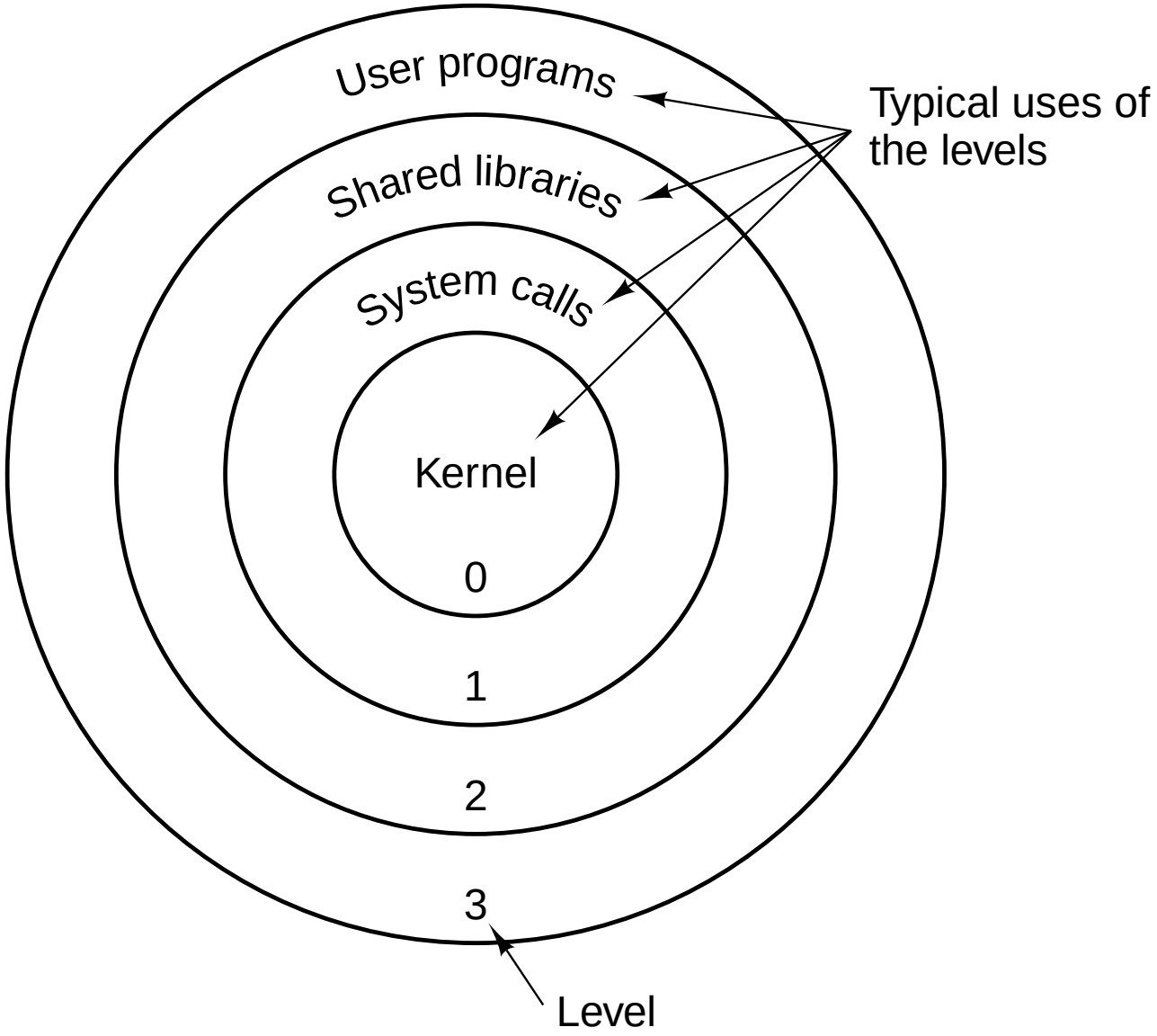
Mapping An Address



More Mapping



Protection Levels



Sources

Tanenbaum, *Modern Operating Systems*
(Course textbook.)

Understanding the Linux Kernel by Bovet and Cesati.
Contains a good discussion of Pentium segmentation.