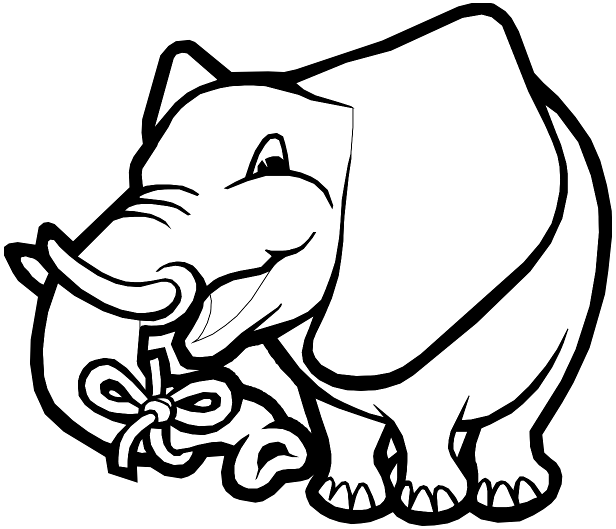
CS 520, *Operating Systems Concepts*

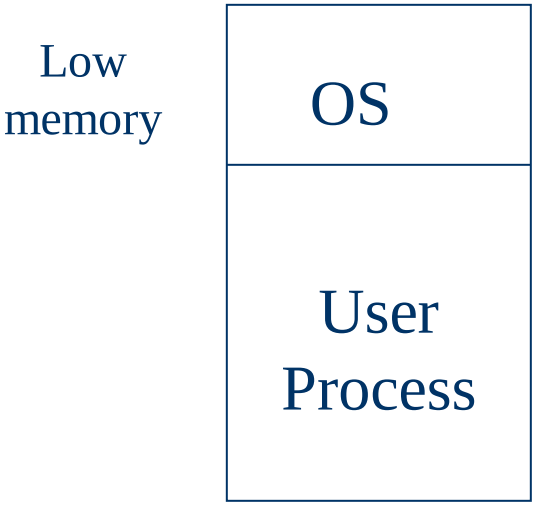
Lecture 6

Memory Management



Where is the Process?

- In older systems (up until 1960), there was a place in memory for *one* process. The Operating System loaded it and ran it

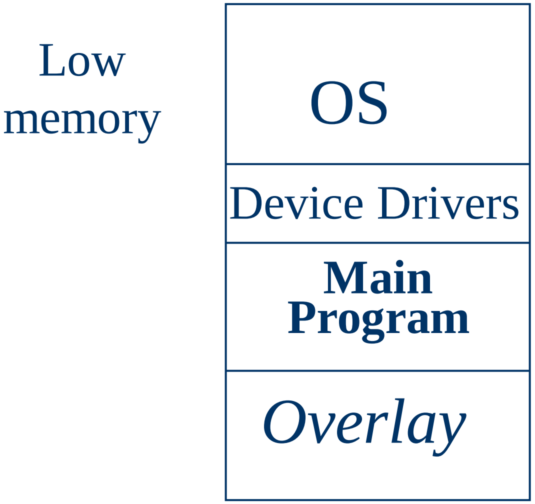


Problems:

1. Low CPU Utilization
2. A need to program device drivers in each process for the devices it uses
3. Inability to split a program into manageable independent concurrent pieces
4. Effectively, inability to support more than one interactive user

Overlays

- The situation was improved by introducing the *overlays* brought in and out of *backing store* (disk or drum). Overlays were effectively independent modules.



Low memory

OS

Device Drivers

Main Program

Overlay

Problems solved:

1. Low CPU Utilization (**not solved)**
2. A need to program device drivers in each process for the devices it uses **(solved)**
3. Inability to split a program into manageable independent concurrent pieces **(solved)**
4. Inability to support more than one interactive user **(not solved)**

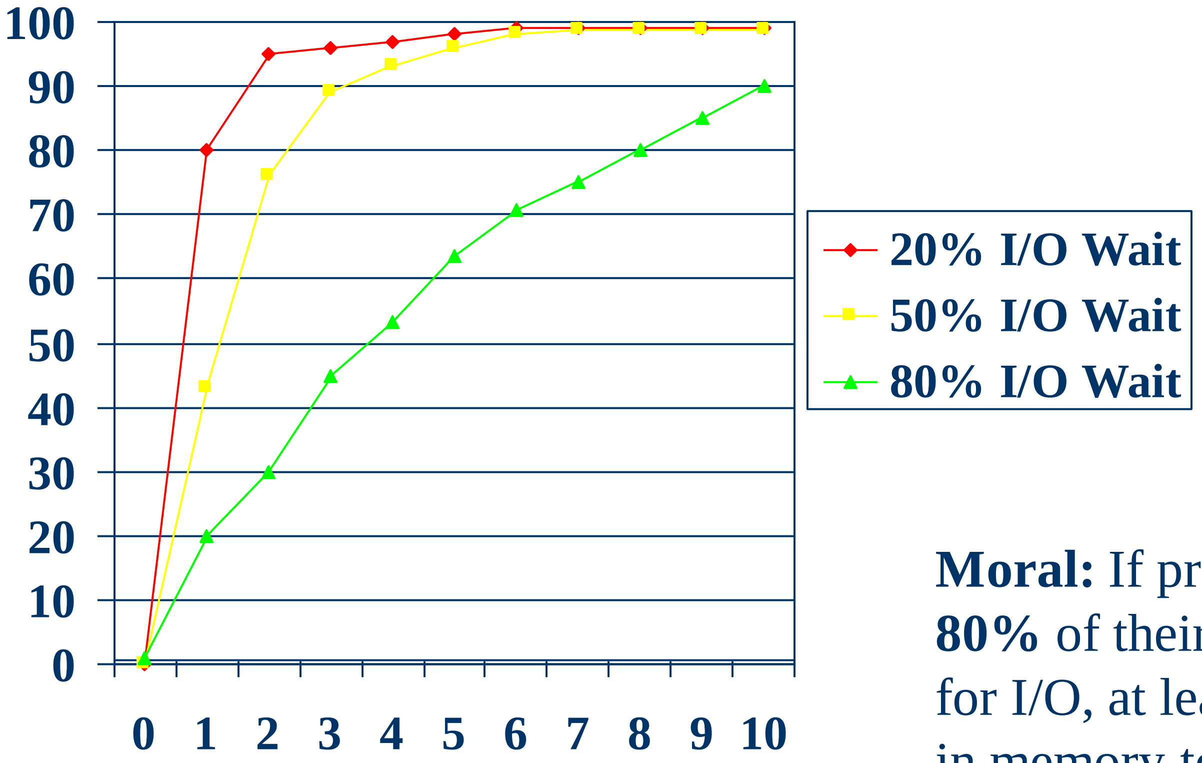
Modeling Multiprogramming

- If p is the fraction of the time the process waits for I/O, then the probability that n concurrent processes are waiting for I/O is p^n, and thus the CPU utilization is $1 - p^n$.

(Of course, we assume that the processes are independent.)

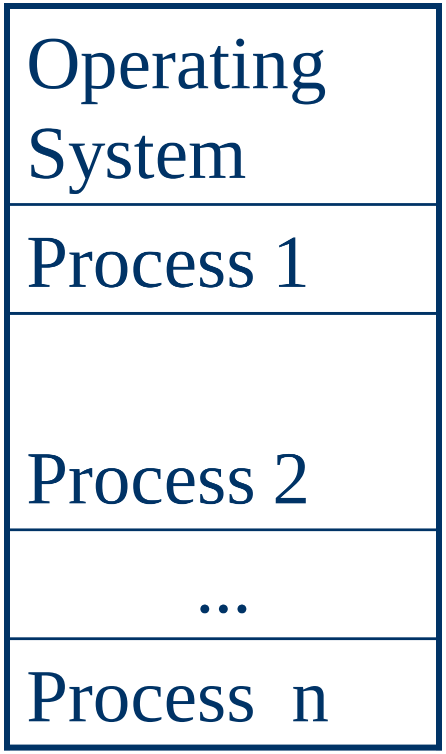
Definition: *n* is called the degree of multiprogramming

CPU Utlilization As a Function of the Degree of Multiprogramming



Moral: If processes spend **80%** of their time waiting for I/O, at least **10** must be in memory to get the CPU waste below **10%**.

Multiprogramming



Operating System

Process 1

Process 2

...

Process n

For multiprogramming, the code and data of several processes must be resident in memory.

With that, there should be enough memory to execute at least a subset of processes; the rest of the processes can be temporarily stored on disk and then brought back into memory, while other processes get stored. This mechanism is called *swapping*.

Notes on Swapping

- A process can be swapped out *only* when its memory does not need to be accessed by the OS (to finish an I/O, for example) or any other process
- The time to write and read the process to the disk is long, so the context switch penalty for *synchronous* swapping is high (this can be mitigated by *asynchronous* swapping)
- If the process code is not being modified during its execution (i.e., the code is *reentrant*), only the process data need to be saved, but both the code and data need to be read back
- *UNIX* starts using swapping when the processes start requesting memory beyond a configurable threshold

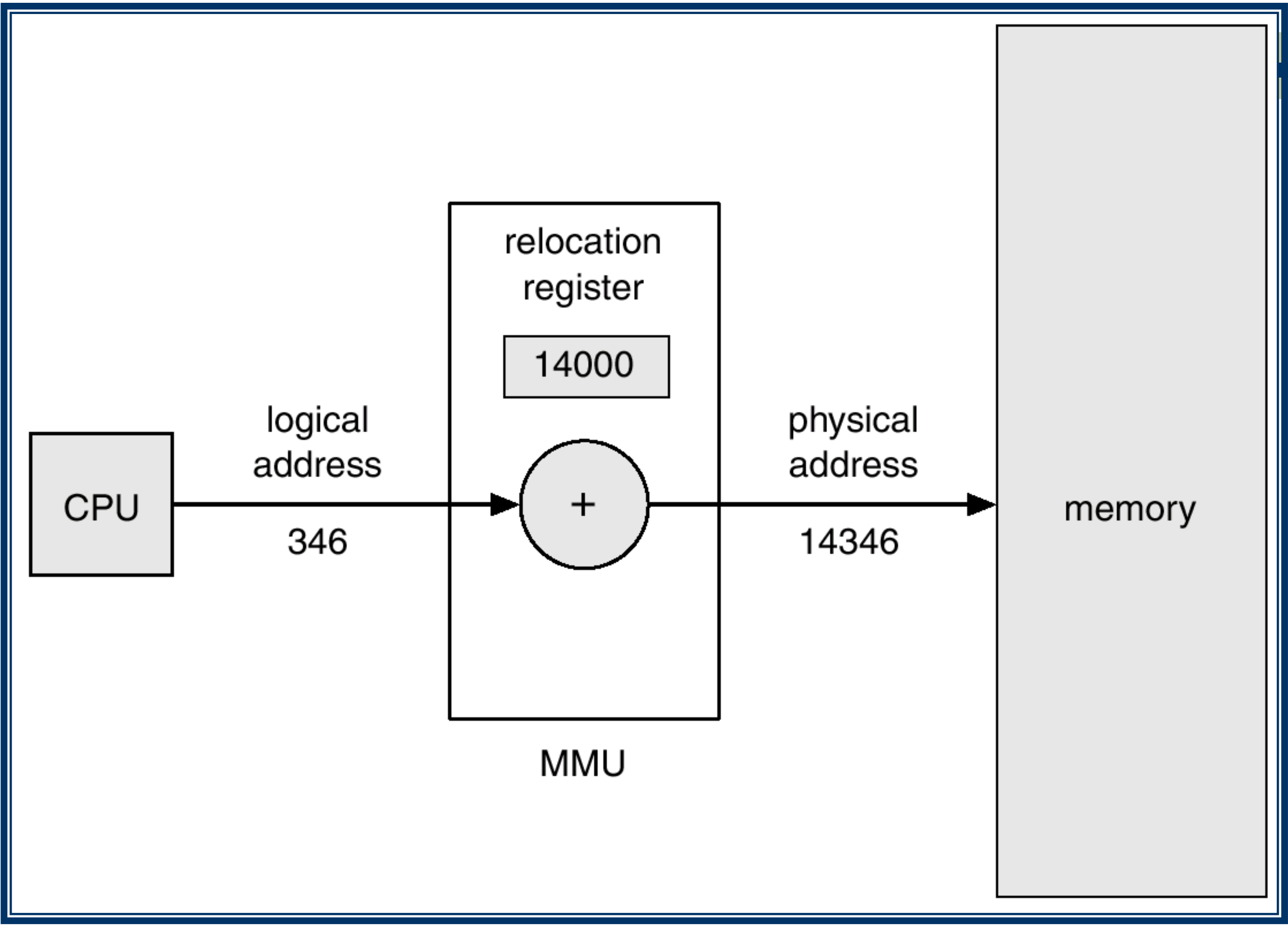
But Does That Not Mean That The Processes Should Be Able to Execute Anywhere in the Memory?

- Exactly! Therefore,
  - ⇒ The process code must be written, in general, with no *absolute* addressing
  - ⇒ The variables must be bound to relocatable, *relative* addresses (like *#base* + *43*)
  - ⇒ The actual binding may be unfinished until the load (execution) time
  - ⇒ Care should be taken so that a process is denied access to other process's space

Binding of Instructions and Data to Memory

- **Compile time**: If memory location is known, the absolute code can be generated; the code must be recompiled if the starting location changes
- **Load time**: Relocatable code must be generated if a 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. Here, hardware support is needed for address maps (e.g., *base* and *limit* registers).

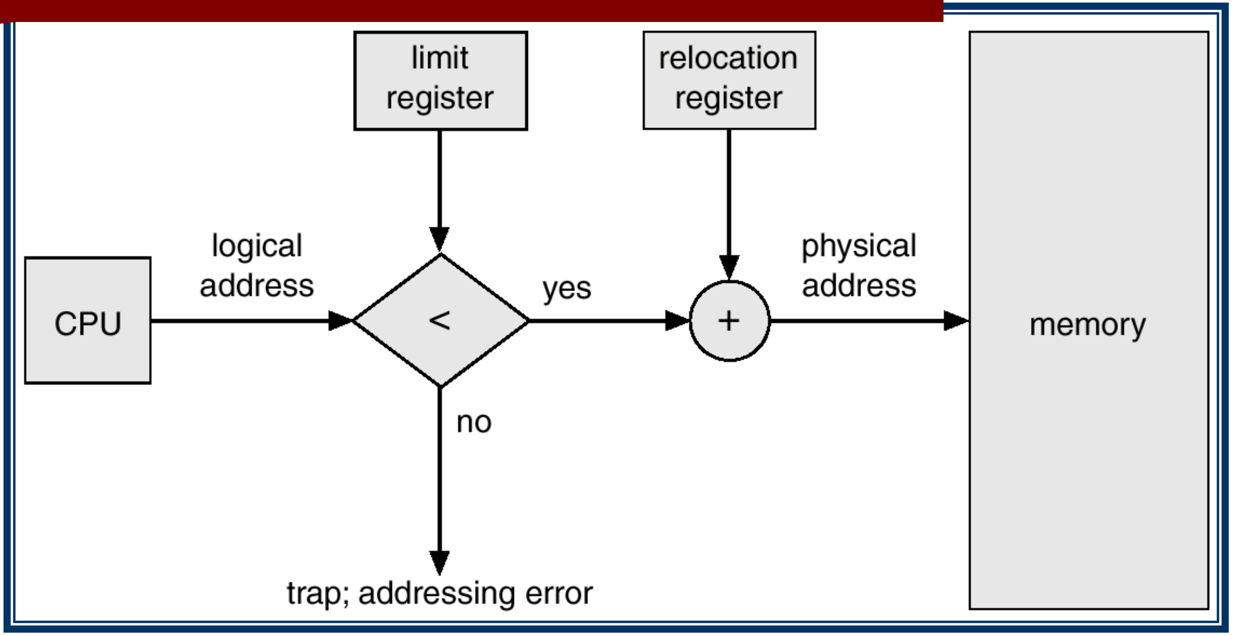
Dynamic relocation using a base (relocation) register [from the Book]



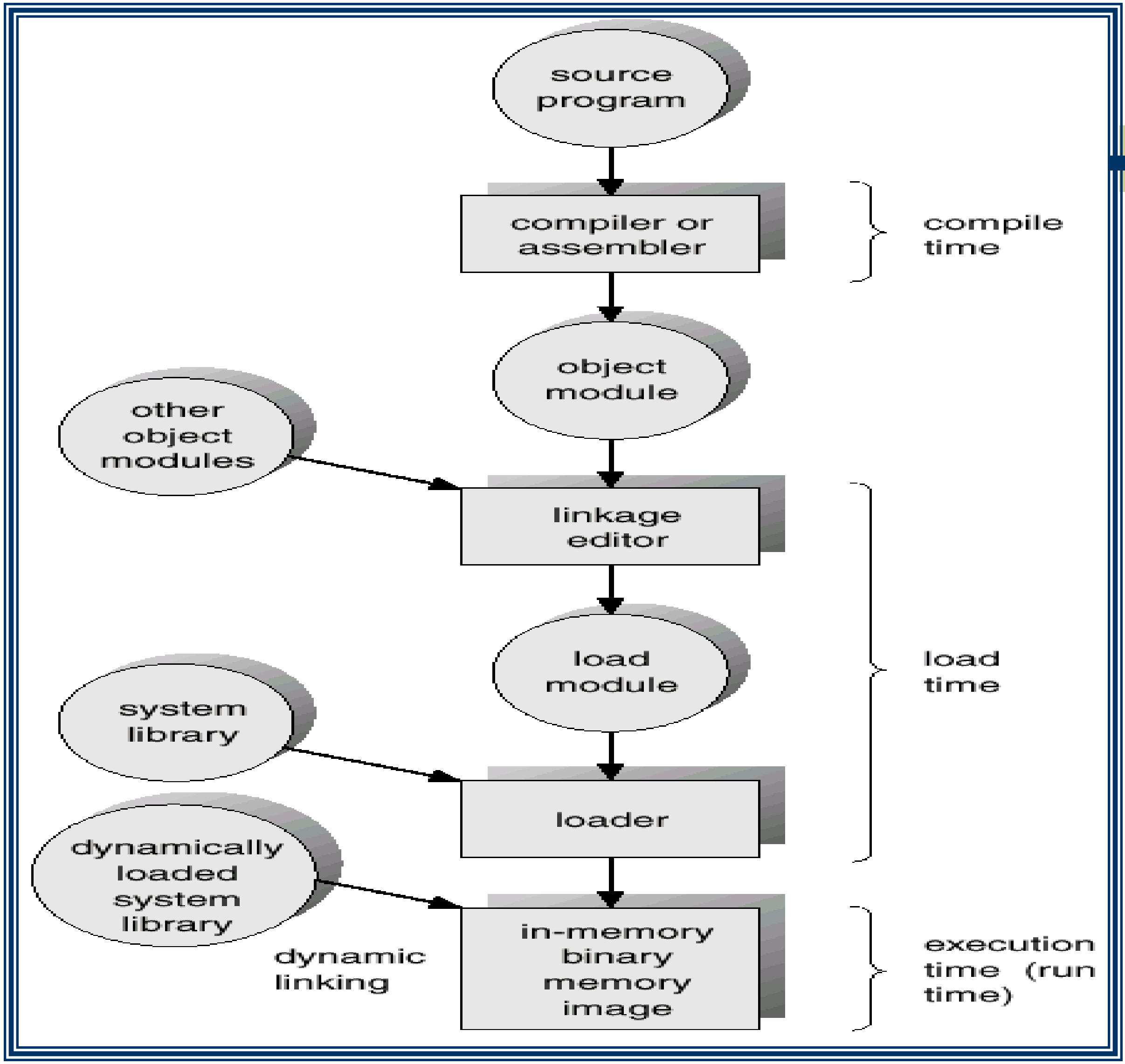
Hardware Support for Relocation and Limit Registers [from the Book]

“In multiuser systems, it is undesirable to let processes read and write memory belonging to other users.”

A. Tanenbaum



Processing of a User Program [from the Book]



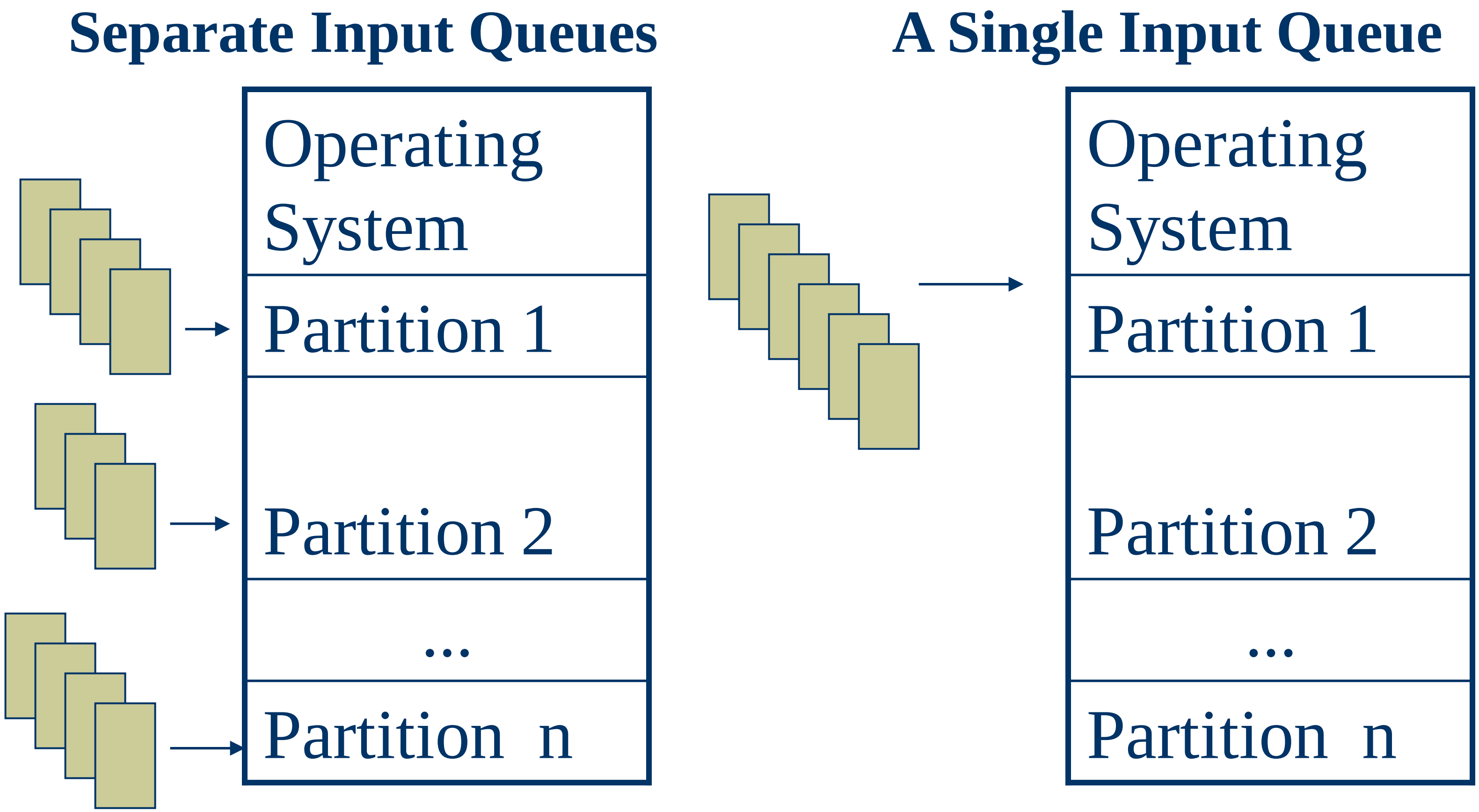
Dynamic Loading

- With dynamic loading, a subroutine is not loaded until it is called
- This affords better memory space utilization; unused routines are never loaded
- This technique is useful when large amounts of code are needed to handle infrequently occurring cases (like error recovery; *The 90/10 Rule*: 90% of the code executes only 10% of the time)
- This technique is particularly useful for maintaining libraries

How Should the Memory Be Organized for Multiprocessing?

- This can be done with *fixed partitions* or *variable partitions*
  - With fixed partitions, the user memory is divided in a configurable number of slots of configurable sizes
  - With variable partitions, the number and size of the partitions vary throughout the life of the system

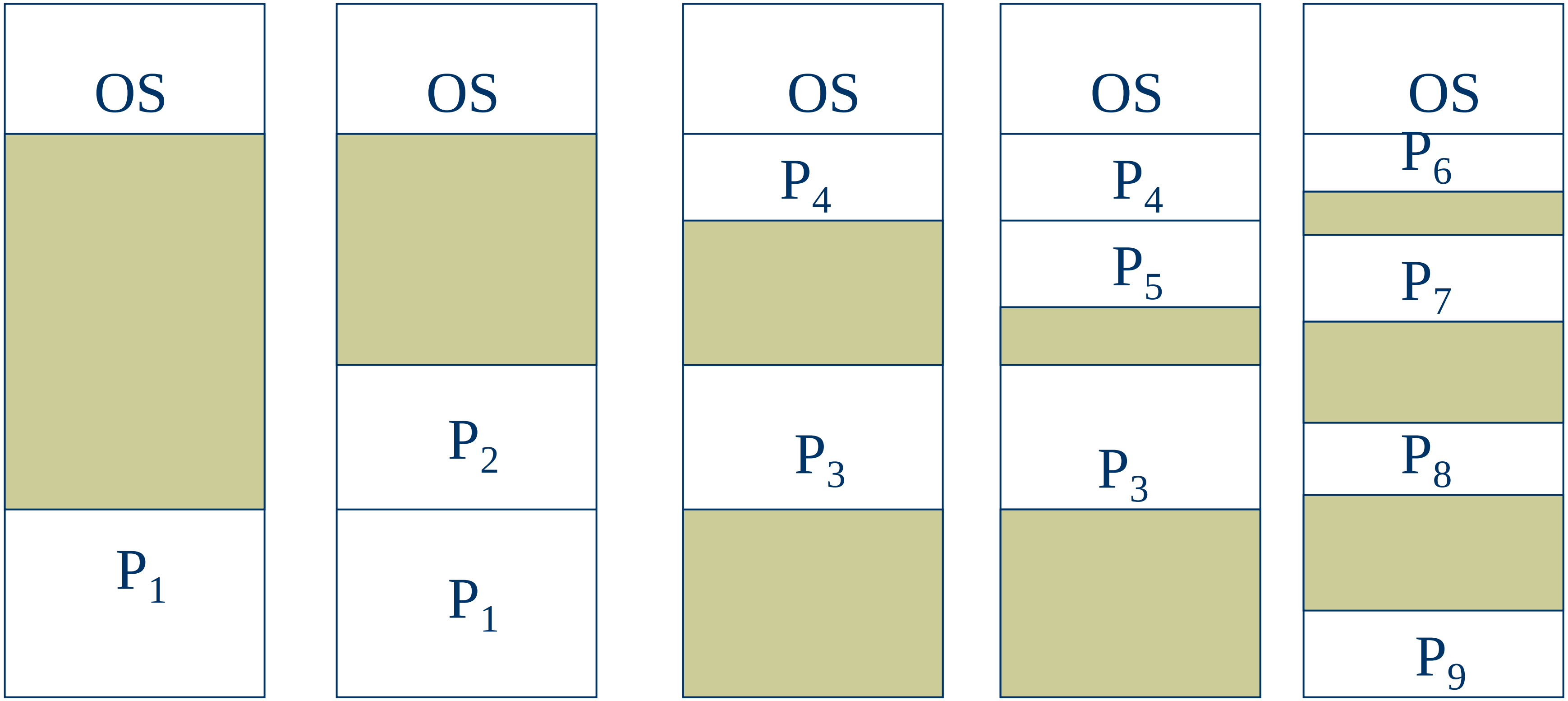
Multiprogramming with Fixed Partitions



Variable Partitions

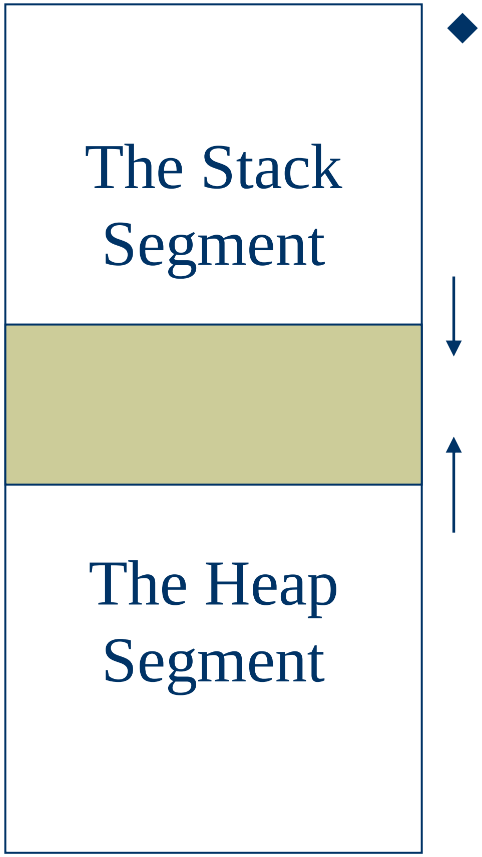
- Fixed partitions suit well the batch system
- With time sharing, there are typically more users than there is the memory to hold the processes
- Even though the swapping technique can be used with the fixed partitions, it works best when the size and number of partitions can vary dynamically, depending on the needs of the processes

Memory Allocation in the Case of Variable Partitions



Dynamic Space Allocation

A Process



- In modern processes,
  - *Text* (code) is generally read-only (does not need to be swapped)
  - Global (*heap*) memory *segment* is allocated dynamically as is the *stack* segment
  - Heap and stack memory grow toward each other

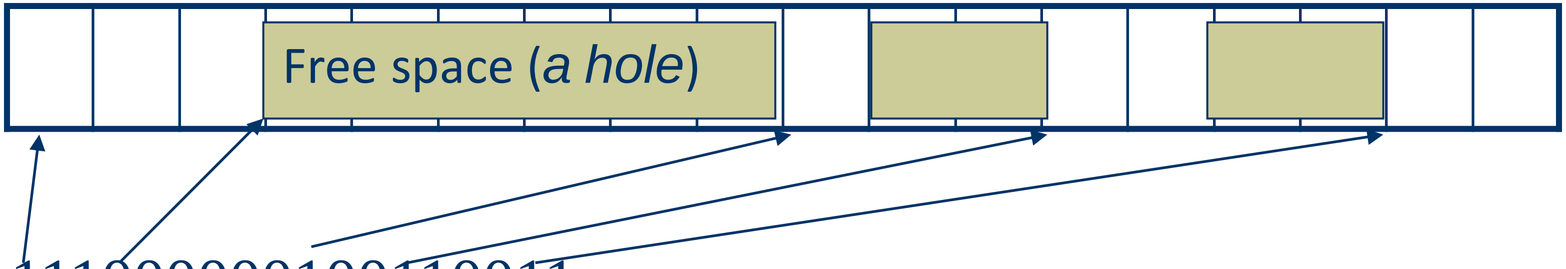
Three Ways to Keep Track of the Use of Memory

1. Bit Maps
2. Linked Lists
3. Buddy Systems

1) Bit Maps

- Memory is divided into *allocation units* (from several bytes to several kilobytes each)
- The bit map is a *characteristic function* of the allocation—a corresponding to the unit bit is *zero*, if the unit is free, or *one*, if it is occupied

Memory divided into allocation units



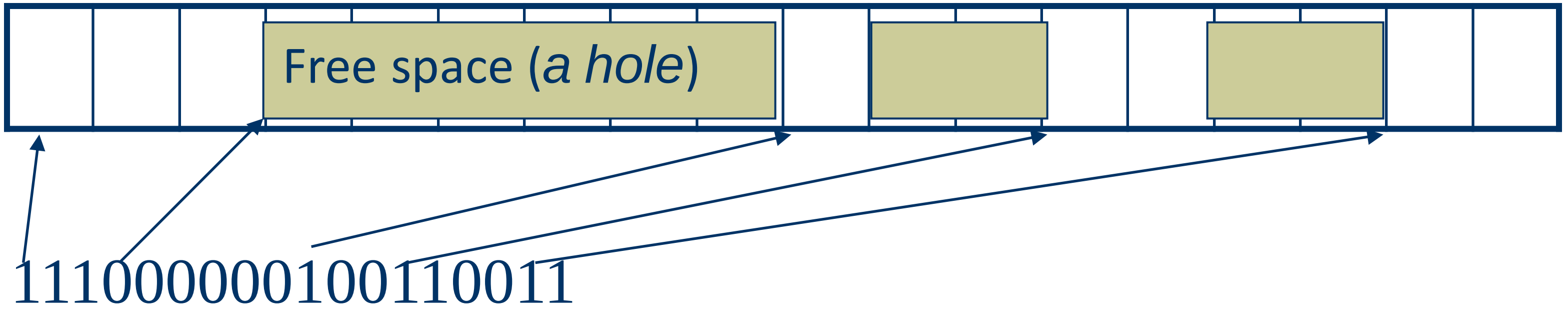
111000000100110011

A bit map

1) Bit Maps (cont.)

- *Advantage:* For *n* units of memory only *n* bits are needed for housekeeping (which works especially well with large units)
- *Disadvantage:* Searching for a hole of a given size and updating the bit map are slow processes

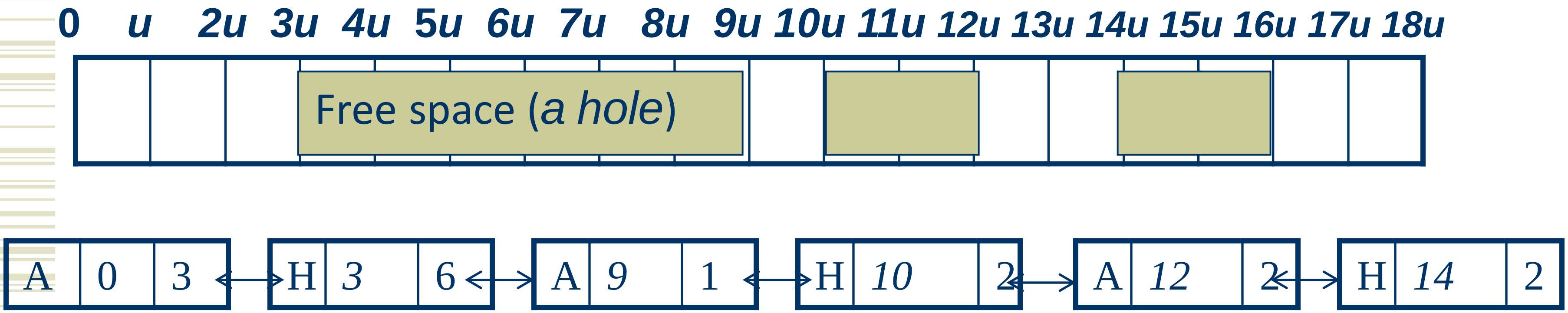
Memory divided into allocation units



A bit map

2) Linked Lists

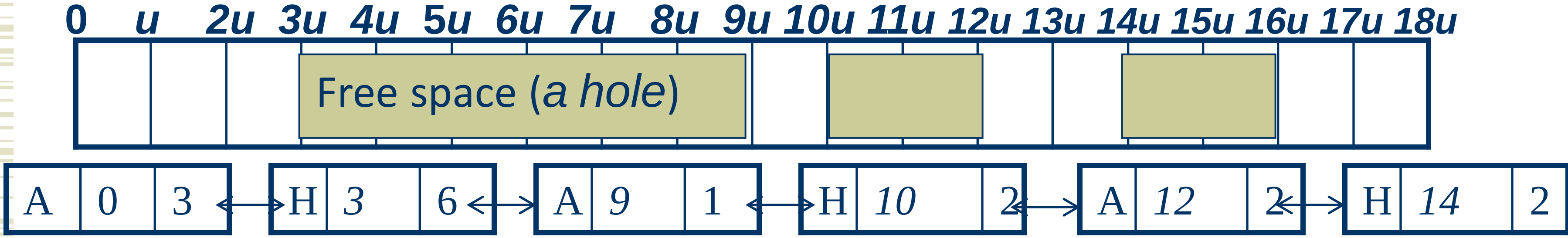
- Each item specifies whether the respective segment of memory is ***allocated*** (in which case it has a pointer to its process in the process table) ***or is a hole***; the ***starting address***; the ***size*** in units
- The items here are ***sorted by address*** and ***doubly-linked***, which makes list management simple



A linked list of memory partitioning sorted by address

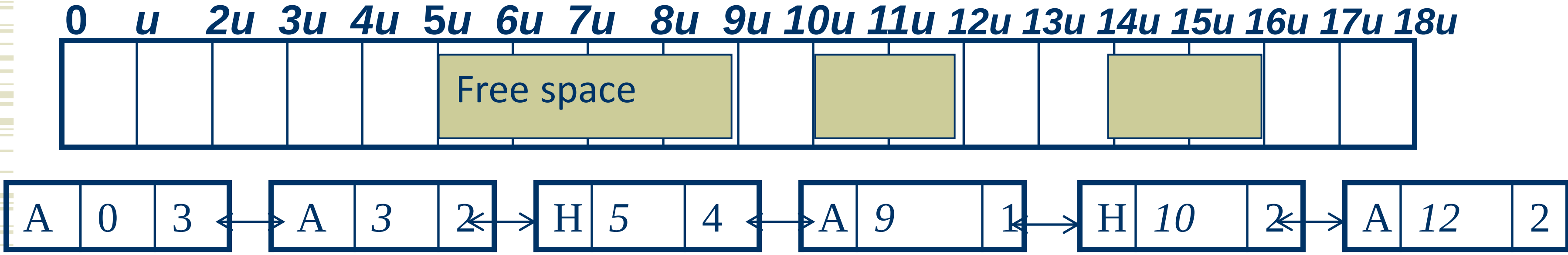
2) Linked Lists (cont.)
Allocating Memory

- To allocate memory, an H-item whose size is not smaller than that sought by the process is taken:


0 u 2u 3u 4u 5u 6u 7u 8u 9u 10u 11u 12u 13u 14u 15u 16u 17u 18u
Free space (*a hole*)

A | 0 | 3 ↔ H | *3* | 6 ↔ A | *9* | 1 ↔ H | *10* | 2 ↔ A | *12* | 2 ↔ H | *14* | 2

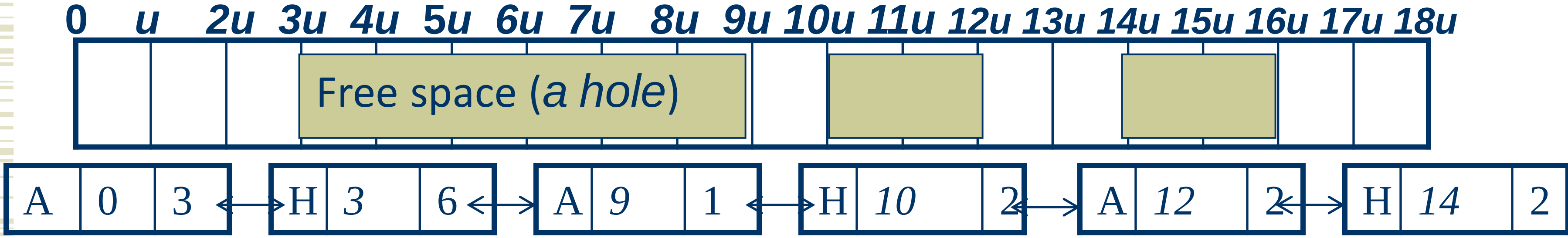
Request: allocate 2 units to process *P*


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

A | 0 | 3 ↔ A | 3 | 2 ↔ H | *5* | 4 ↔ A | *9* | 1 ↔ H | *10* | 2 ↔ A | *12* | 2

2) Linked Lists (cont.) *Releasing Memory*

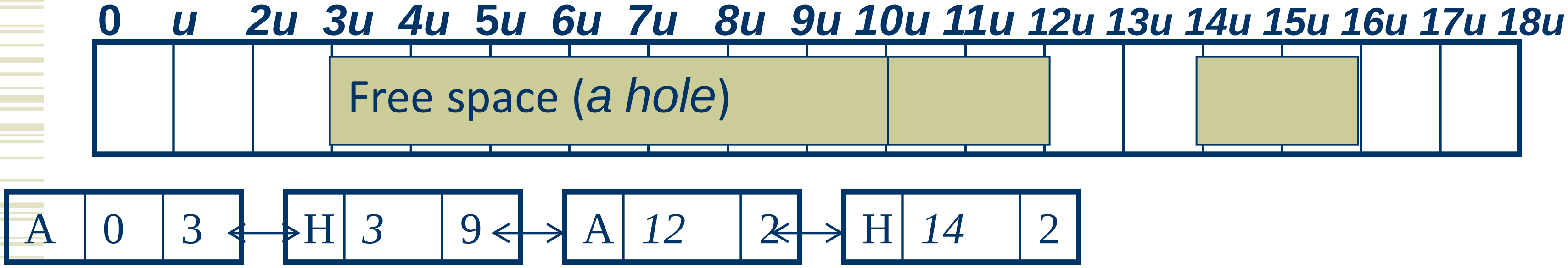
- To release the memory, the A-item is either just turned into an H-item, or—if it follows or precedes an H-item—the space is consolidated



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

Free space (*a hole*)

A | 0 | 3 ↔ H | *3* | 6 ↔ A | *9* | 1 ↔ H | *10* | 2 ↔ A | *12* | 2 ↔ H | *14* | 2



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

Free space (*a hole*)

A | 0 | 3 ↔ H | *3* | 9 ↔ A | *12* | 2 ↔ H | *14* | 2

2) Linked Lists (cont.)
Memory Allocation Algorithms

- *First Fit*: find the first hole that is big enough
  - *Next Fit* is a variation, in which the holes are examined in the *round robin* fashion
- *Best Fit*: find the *smallest* adequate hole (by searching the whole list)
- *Worst Fit*: find the *largest* adequate hole (by searching the whole list)

2) Linked Lists (cont.)
Algorithms' pros and cons

- *First Fit* and *Next Fit* are the fastest algorithms, but they leave fragmentation uncertain
- *Best Fit* guarantees that it would leave the smallest holes, but then these would be too small to use in the future, so the memory could be wasted
- *Worst Fit* guarantees that it would leave the largest holes, which may have a better chance to be used up later

2) Linked Lists (cont.) *Modifications to Algorithms*

- If separate lists are maintained for allocated blocks and holes, the algorithms work faster on allocation (why?) but slower on deallocation (why?). Again, this thing about the free lunch...
- The list of holes can be then sorted by size to further speed up the Best Fit and Worst Fit algorithms, while further slowing down the deallocation (why?). (What will happen to the First Fit algorithm in this case?)
- Furthermore, when the list of holes is distinct, a small decrease in the overhead can be implemented: a hole's own space may be used to store the size and the pointer to the next hole.

3) Buddy System

D. Knuth and *K. Knowlton* developed an algorithm that speeds up deallocation :

- The memory manager maintains lists of free blocks of respective sizes 1, 2, 4, ..., 2^n bytes, up to the size of the memory. (With 1M of memory, 21 such lists are needed, ranging from 1 byte to 1 megabyte.)
- Initially, all the memory is free, so there is one list containing a single block of 2^n bytes, where n is $\log_2$(*Memory_Size*); the rest of the lists are empty.
- When a request for ***k*** bytes comes, $\mathbf{r = \lfloor \log_2 \mathit{k} \rfloor + 1}$ is computed. At that point, the original 2^n block is broken into two *buddies*—one starting at 0, and the other starting at 2^{n-1}. The former block is again broken in two, and so on until we obtain the block of size $\mathbf{2^r}$, which is given to the requesting process. Meanwhile the lists of blocks with the sizes of powers of 2 from *r* to *n-1*are created.

N.B.: The remainder of the $\mathbf{2^r}$ block (that is $\boldsymbol{2^r - k}$) is ***not*** divided among the appropriate lists—it is simply left unused.

3) Buddy System (allocation example)

- Suppose, there are 64 bytes of free memory, and 7 bytes are requested. The resulting lists are
  - 32-byte blocks: one, starting at byte 32;
  - 16-byte blocks: one, starting at byte 16;
  - 8-byte blocks: one, starting at byte 8;
  - 4-byte blocks: <empty>;
  - 2-byte blocks: <empty>;
  - 1-byte block: <empty>.
- Now, what will happen when a request for 5 bytes arrives? And then for 13 bytes? And then for 12?

3) Buddy System (deallocation)

- Every time the block is returned, the algorithm puts it in the appropriate list and sees if the contiguous space can be consolidated into an item of the higher order
- The advantage of the algorithm is that when a block of size 2^k is freed, only the 2^k-list of holes needs to be searched on a subject of consolidation
- The disadvantage of the algorithm is that it causes *internal* fragmentation due to rounding

A Buddy System Example

Memory

0 | 128 K | 256 K | 384 K | 512 K | 640 K | 768 K | 896 K | 1,024K

<table>
<tr><td></td><td colspan="16">1,024 K</td></tr>
<tr><td>A: +70K</td><td colspan="2">A 128</td><td colspan="2">128</td><td colspan="4">256</td><td colspan="8">512</td></tr>
<tr><td>B: +35K</td><td colspan="2">A 128</td><td>B 64</td><td>64</td><td colspan="4">256</td><td colspan="8">512</td></tr>
<tr><td>C: +80K</td><td colspan="2">A 128</td><td>B 64</td><td>64</td><td colspan="2">C 128</td><td colspan="2">128</td><td colspan="8">512</td></tr>
<tr><td>A: -70K</td><td colspan="2">128</td><td>B 64</td><td>64</td><td colspan="2">C 128</td><td colspan="2">128</td><td colspan="8">512</td></tr>
<tr><td>D: +60K</td><td colspan="2">128</td><td>B 64</td><td>D 64</td><td colspan="2">C 128</td><td colspan="2">128</td><td colspan="8">512</td></tr>
<tr><td>B: -35K</td><td colspan="2">128</td><td>64</td><td>D 64</td><td colspan="2">C 128</td><td colspan="2">128</td><td colspan="8">512</td></tr>
<tr><td>D: -60K</td><td colspan="4">256</td><td colspan="2">C 128</td><td colspan="2">128</td><td colspan="8">512</td></tr>
<tr><td>C: -80K</td><td colspan="16">1,024 K</td></tr>
</table>

Fragmentation

- In addition to internal fragmentation, of which not much can be done without improving the Buddy System, there can be *external* fragmentation (also called *checkerboarding*)—which we noticed in the “fit” algorithms
- External fragmentation can be eliminated by *compaction*—the process of moving the processes and holes in the opposite direction (by swapping out all processes) and then reallocating the memory
- For obvious reasons, compaction is quite expensive in terms of time

The 50% Rule

- Statistical analysis shows that, averaged over time, there must be half as many holes as allocated blocks. In other words, if the mean number of processes is *n*, the mean number of holes is *n/2*
- This result is known as the *50% Rule*
- It demonstrates fundamental asymmetry between the process blocks and holes—the adjacent holes can be merged, but the process blocks can not

The *Unused Memory* Rule

- Let
  - *f* be the fraction of the memory occupied by holes
  - *a* be the average size of an allocated block within *n* such blocks and *h* be the average size of a hole among the *n/2* holes; let *k* = *h/a*
- With the total memory of *m* bytes, the *n/2* holes occupy *m – na* bytes, so

$$\frac{n}{2}ka = m - na,$$

and thus

$$m = na(1 + \frac{k}{2})$$

Then

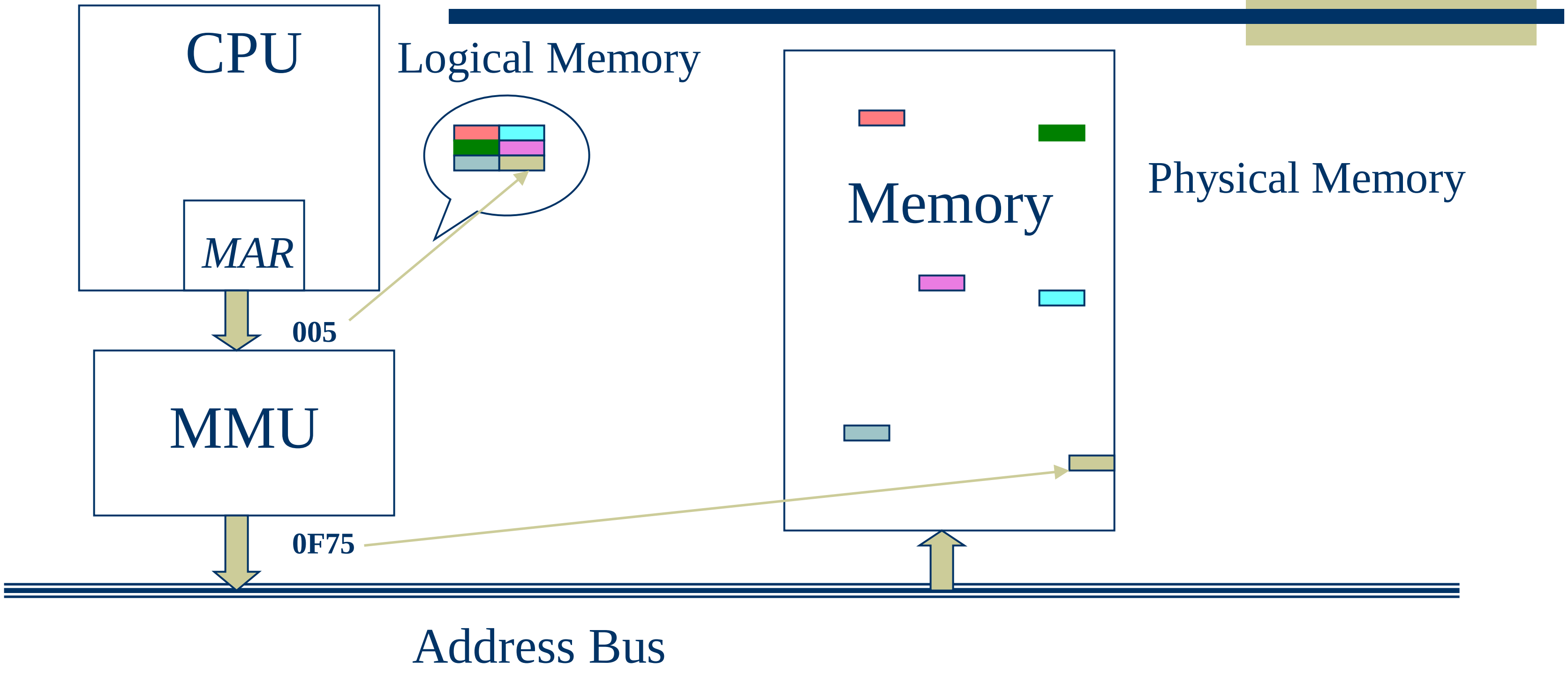
$$f = \frac{\frac{n}{2}ka}{m} = \frac{k}{k+2}.$$

Hence, if holes are half as large as allocated blocks, 20% of the memory is wasted in holes. If we reduce *k* to ¼, the waste will drop to near 11%. (The best fit algorithm helps reduce *k*.)

Paging

- We observed a problem caused by fragmentation: little pieces of memory worthless by themselves, add up to a substantial memory volume
- **What can we do about that?**
- Well, with a little help from the hardware, a memory that would look contiguous to a process, could actually be composed of fragmented small pieces (called *pages*), which can be scattered anywhere within the available memory

Memory Reference Translation with the *Memory Management Unit (MMU)*



More Definitions

- The address space is divided into *pages*
- The actual memory is divided into *page frames* of the same size
- The logical address is a pair <*page number*, *offset*>
- The MMU, among other things, translates the *logical* page number into the *actual (physical)* frame number; the offset is untouched by the translation
- When page frames are allocated, there is no external fragmentation, but there may be (and often is) internal fragmentation

The Page Table

- The page table stores the translation function

0	7
1	5
2	3
3	9
4	6
5	0
6	4

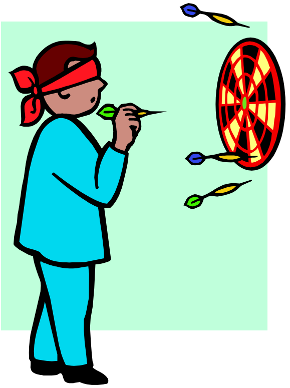
When a process asks for contiguous *n* pages, the translation table entries are filled one by one

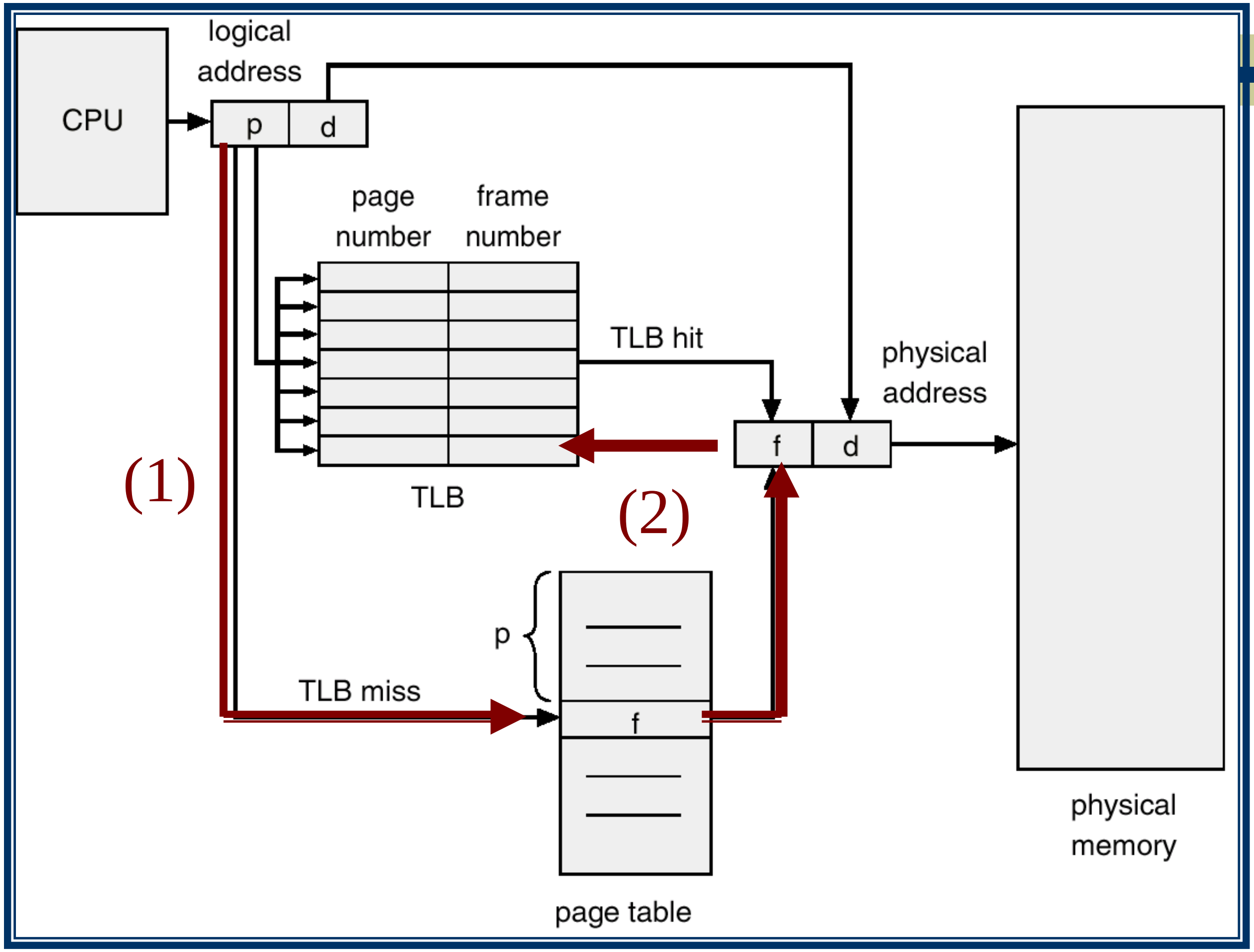
Real-Time Page Translation

- The pointer to the process's page translation table is kept in its process table entry
- With plain vanilla translation, two memory lookups are needed—one for the table, and one for the real reference—this is too expensive
- A special high-speed *associative* memory device called *Translation Look-Aside Buffer (TLB)* is used to make things faster, at least until there is a *miss* (as in *hit* or *miss*)

Paging Hardware With TLB

From the Book (but augmented!):





Notes on TLB Operation

- The *associative* memory allows a parallel search *O(1)* of **all** TLB entries
- Every TLB entry contains a corresponding page table entry *in its entirety* (more on this, later)
- If there is a miss, then
  1. The page table is consulted
  2. The result is put into TLC replacing some other entry

The Page Table Revisited

- The page table also stores protection information

0	7	r	v
1	5	rw	v
2	3	xr	v
3	9	xr	v
4	6		i
5	0		i
6	4	x	v

A page can be *read-only*, or *read-write*, or *execute*, *execute-read*, or...

As not all memory is being used (consider *stack* memory), a page may even be *valid* or *invalid*

The Size of the Page Table

- With a 32-bit logical address space and even a large page size 4KB, a page table will have near 1,000,000 entries! With a modest requirement of 4 bytes per entry, that adds 4MB of storage overhead just for a page table (of one process!)
- To deal with this problem, three techniques are used
  - Hierarchical (multi-level) paging,
  - Hashing, and
  - Inversion

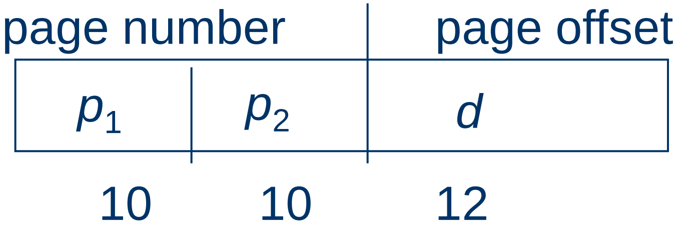
Multilevel Paging

- Here, the page tables are paged themselves!
- A tree structure maintains the path to the right page
- Two-level paging is used more often (on 32-bit address machines) than others, but at least a three-level paging is needed for 64-bit machines (Why?)

Two-Level Paging Example

- A logical address (on 32-bit machine with 4K page size) is divided into:
  - a page number consisting of 20 bits.
  - a page offset consisting of 12 bits.
- Since the page table is paged, the page number is further divided into:
  - a 10-bit page number p_1.
  - a 10-bit page offset p_2.

The Book:

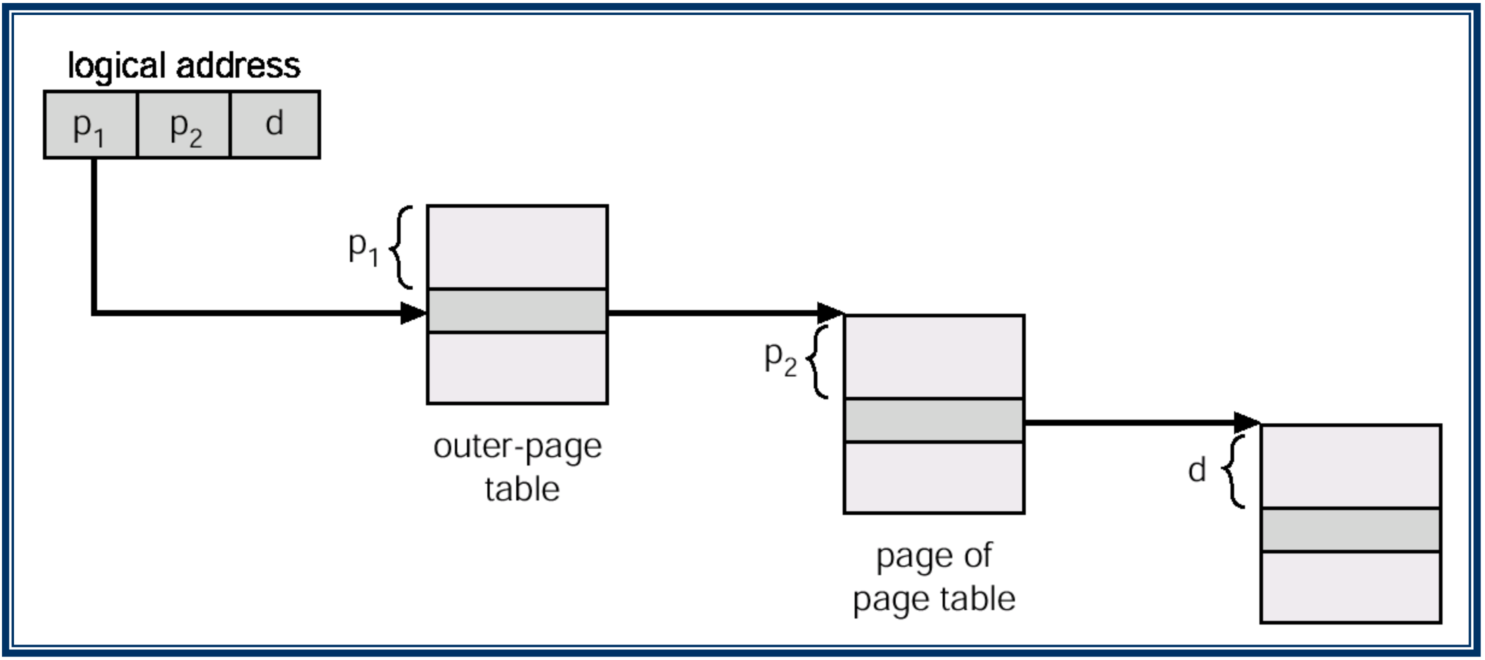


page number		page offset
p_1	p_2	d
10	10	12

where p_1 is the index into the outer page table, and p_2 is the displacement within the page of the outer page table.

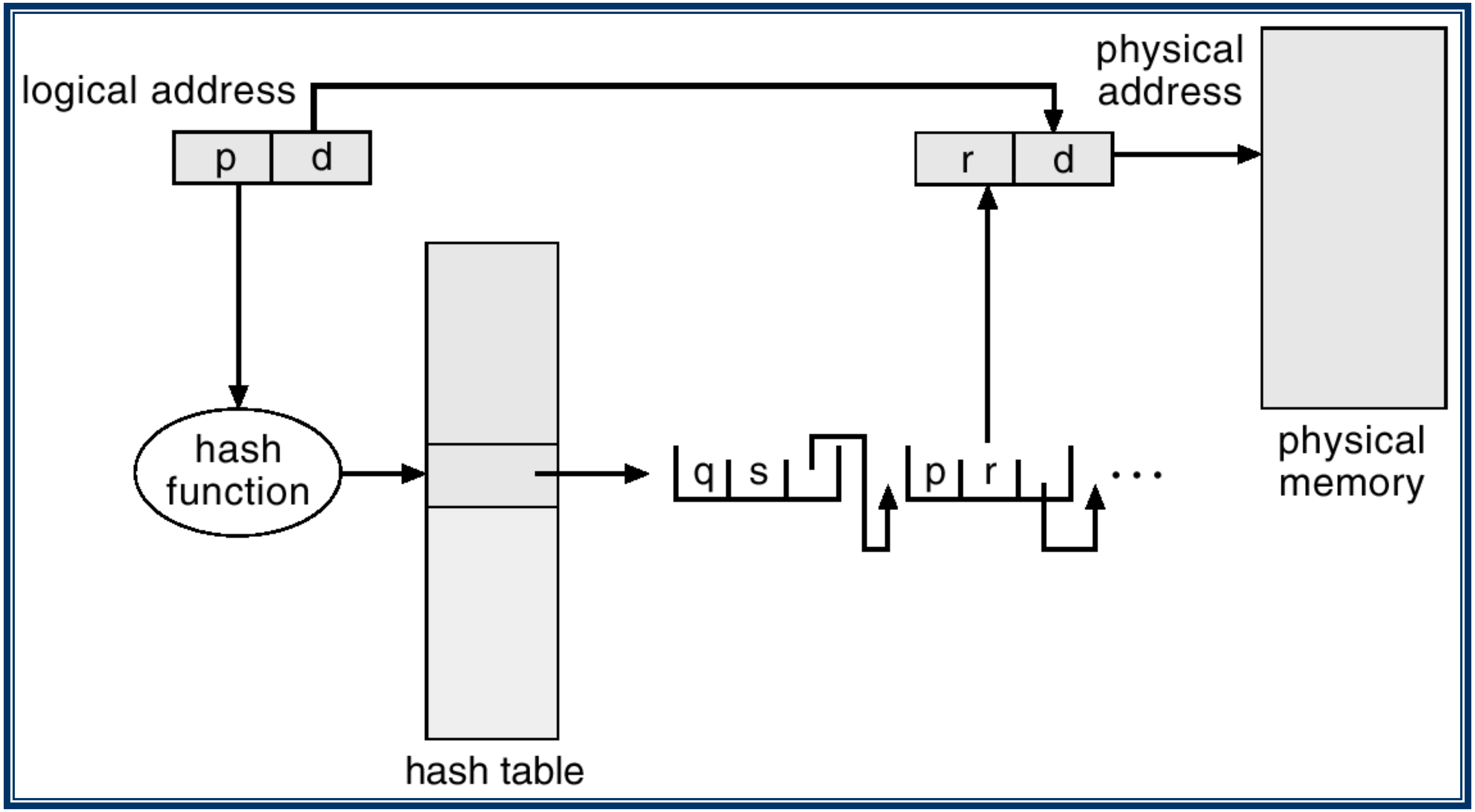
Address-Translation Scheme

The Book:



Hashed Page Tables

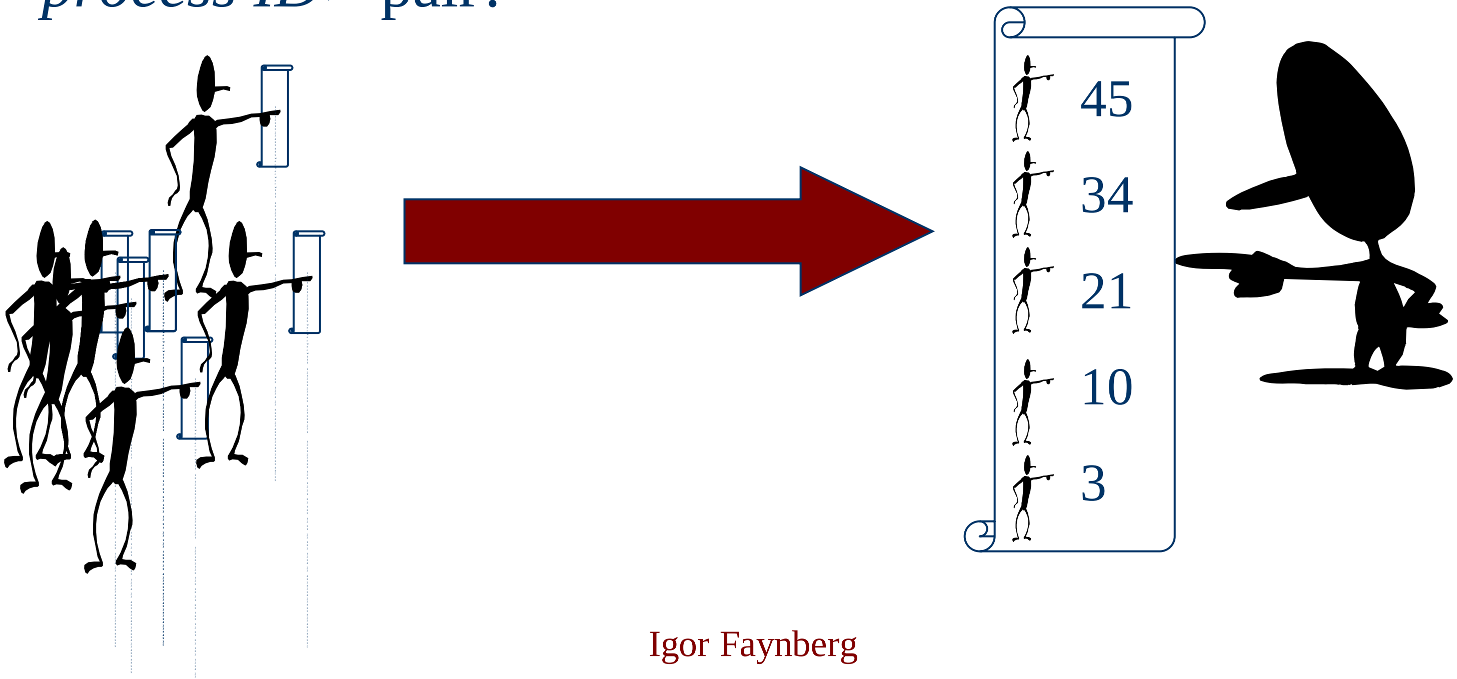
The Book:



Inverted Page Tables

The Book:

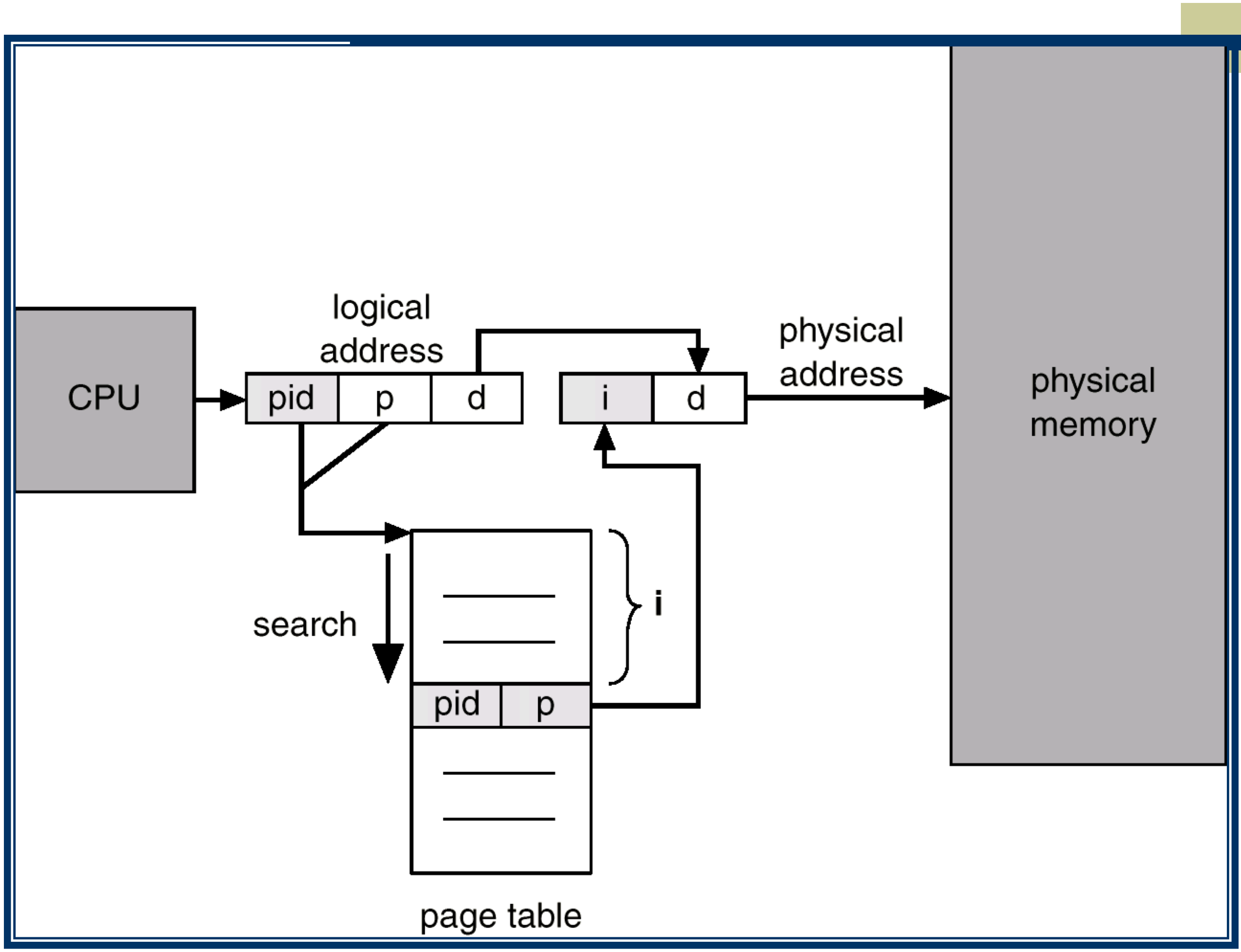
- *The idea:* Instead of maintaining a (large) page table for each process, why not actually keep (in ***one*** table) the pointers from real memory pages to the *<logical page, process ID>* pair?



Some considerations

- The inverted page table contains an entry for each *frame* (rather than each logical page).
- Thus the table occupies a **fixed** fraction of memory. (The size is proportional to that physical memory, not the virtual address space.)
- The full table containing the forward mapping is typically stored on disk.

Inverted Page Table Architecture (from the Book)



Big problem:

Sharing memory!

Page look-up

- The inverted page table is organized as a hash table
- The hash is to locate the string (PID, page number)
- If there is no match, the collision resolution technique (rehash, search, etc.) is employed

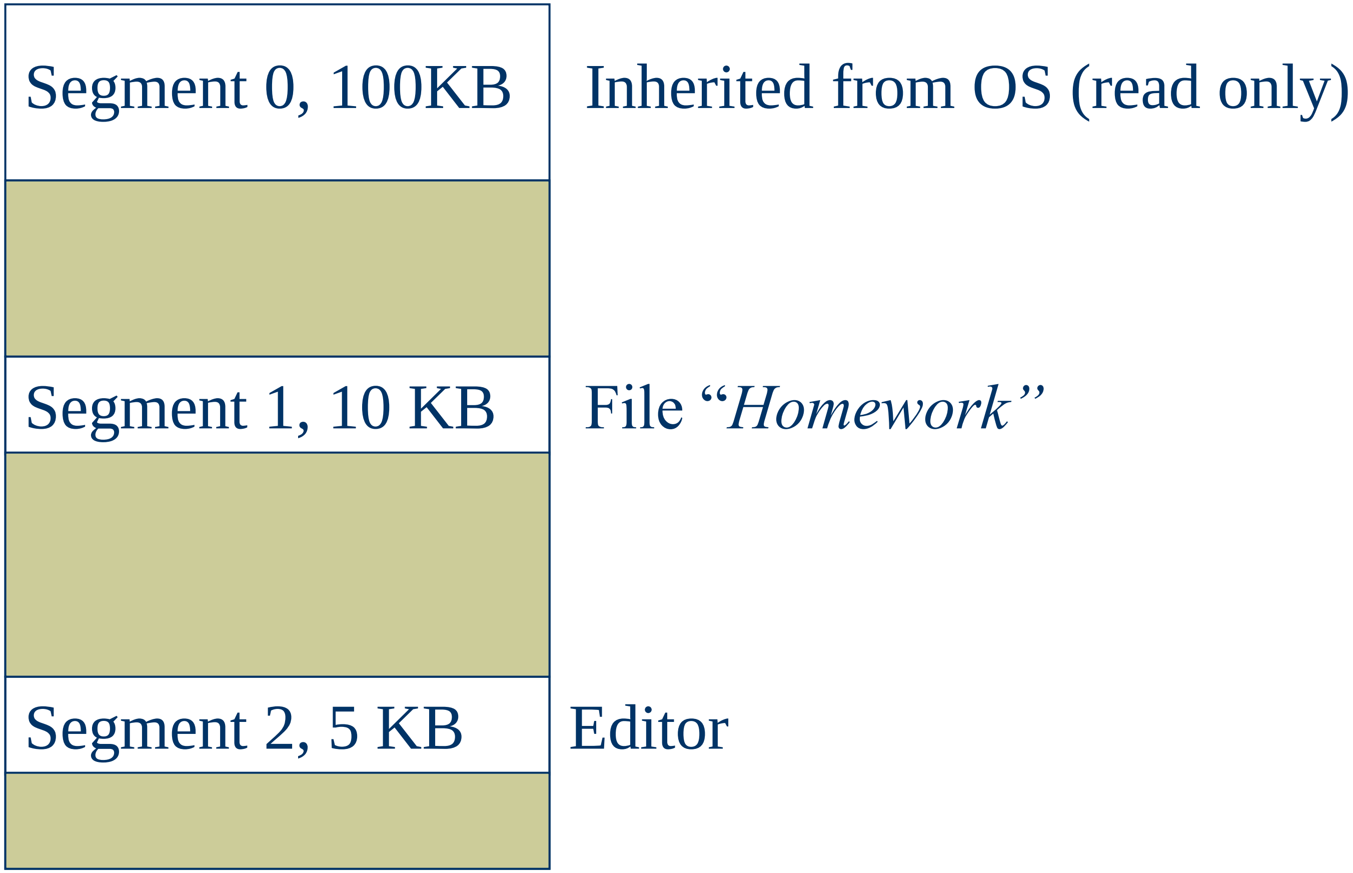
Segmentation

- This is how we think about the code: *global data*, *local data* (generally on the stack), the *code* (further seen as a set of modules and initialization)
- These logical items make programs. It is actually useful to look at them these way when they are in memory!

Segmentation (cont.)

- Each segment in a process space is recognized by a number, and it has its own addressing space from *0* to *segment_size-1*.
- Segments can be protected as well as shared (like a common code segment) for reading only
- In some systems (e.g., MUSS) a programmer wrote the code, indicating directly in which segment each piece of it belonged and where in memory each segment was to be placed; all files were dealt with as real memory data—*open* file meant creating a memory segment and reading the file in!

Segmentation Example



Segment 0, 100KB

Inherited from OS (read only)

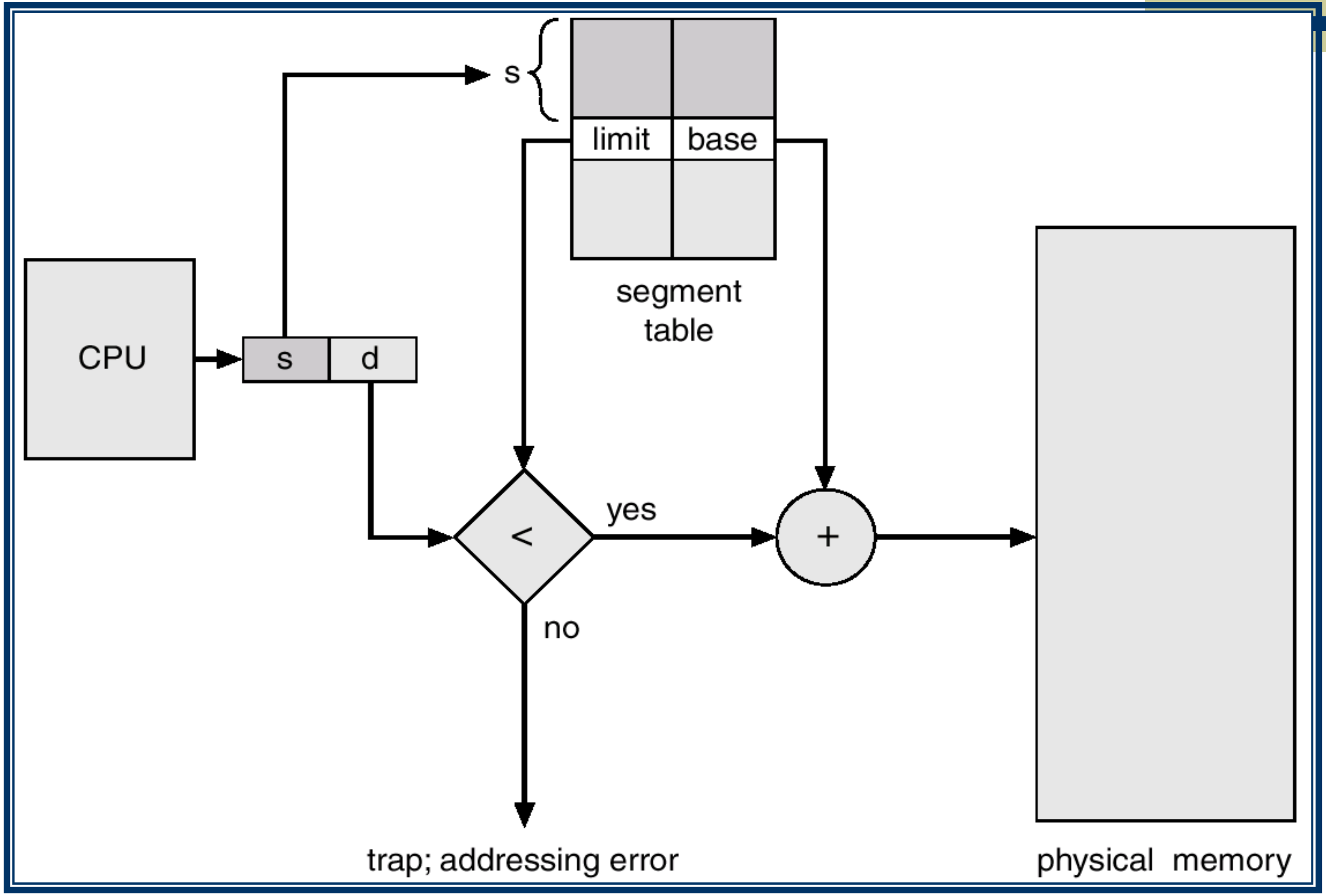
Segment 1, 10 KB

File “*Homework*”

Segment 2, 5 KB

Editor

Translation of the Segment Address



Segmentation and Paging

- The advantage of the segmentation technique is that it reflects an extremely useful view of the memory as a collection of objects, whose purposes, availability, and use are different, and which thus can be administered with these differences in mind
- The advantages of the paging technique (discussed so far) are in that it
  - Decreases memory fragmentation (and actually reduces it to internal fragmentation only)
  - Supports *virtual* addressing
- Starting from *Multics*, segmentation and paging have been often combined so as to take both sets of advantages