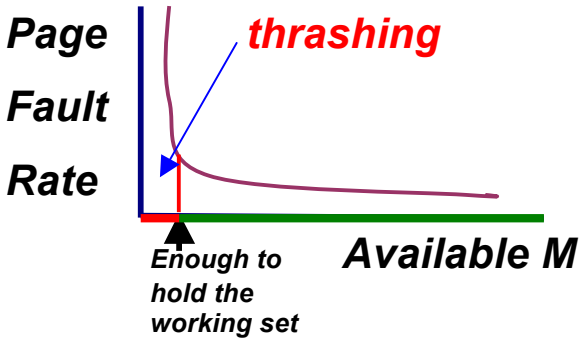
Virtual Memory Implementation:

Working Set

Consider a page reference string

0, 1, 2, 2, 1, 1, 2, 2, 1, 1, 2, 2, … 100,000 references

The size of the *working set* is 2 pages.

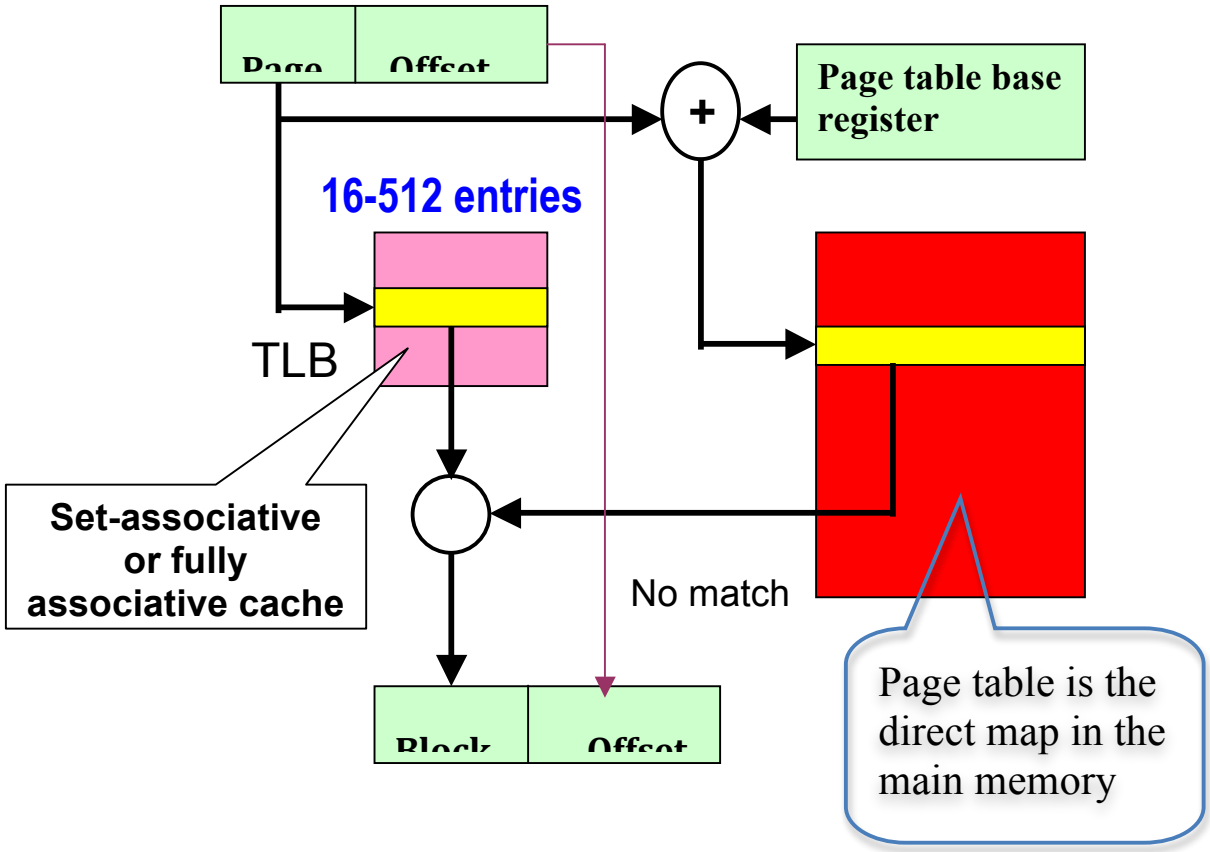


Always allocate enough memory to hold the working set of a program (Working Set Principle)

Disk cache

Modern computers allocate up a large fraction of the main memory as file cache. Similar principles apply to disk cache that drastically reduces the miss penalty.

Address Translation Using TLB



TLB is a **set-associative cache** that holds a partial page table. In case of a TLB hit, the block number is obtained from the TLB (fast mode). Otherwise (i.e. for TLB miss), the block number is obtained from the direct map of the page table in the main memory, and the TLB is updated.

Multi-level Address Translation

Example 1: The old story of VAX 11/780

30-bit virtual address (1 GB) per user

Page size = 512 bytes = 2^9

Maximum number of pages = 2^{21} i.e. 2 million

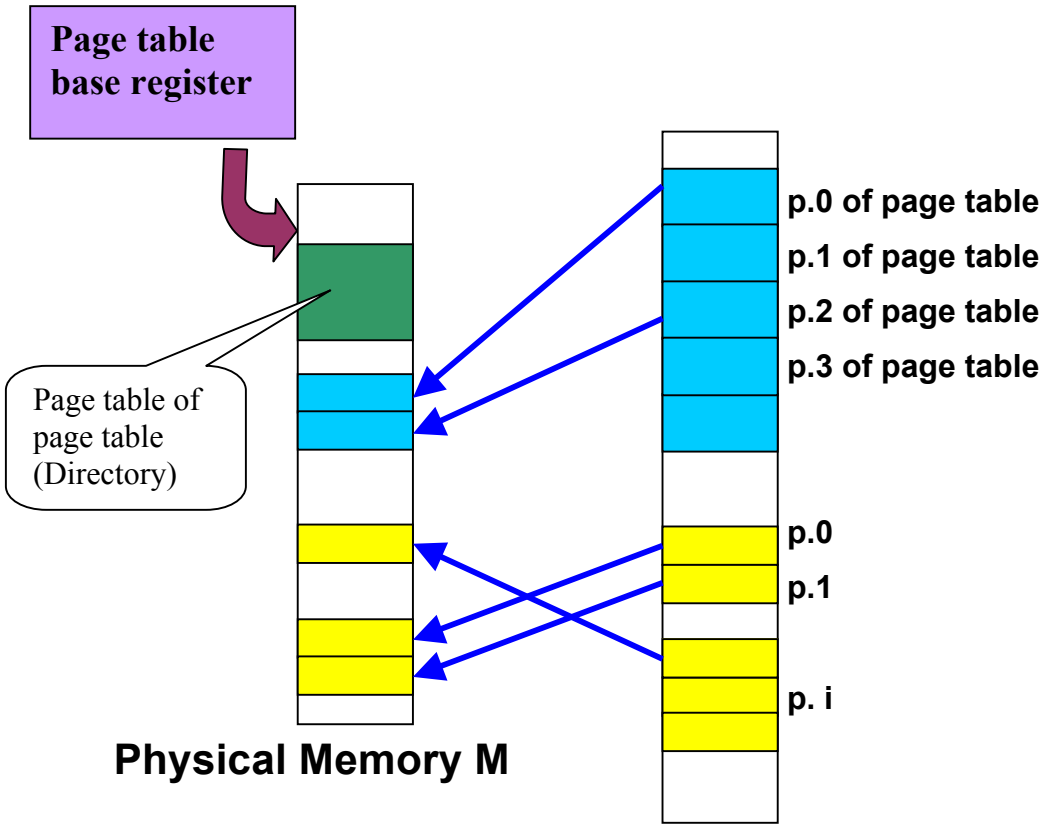
Needs 8 MB to store the page table. Too big!

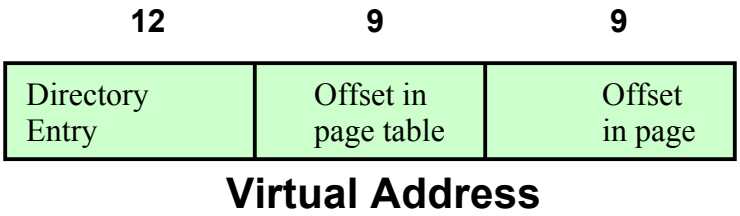
Solution?

Store the page table in Virtual Memory.

Thus, page table is also paged!

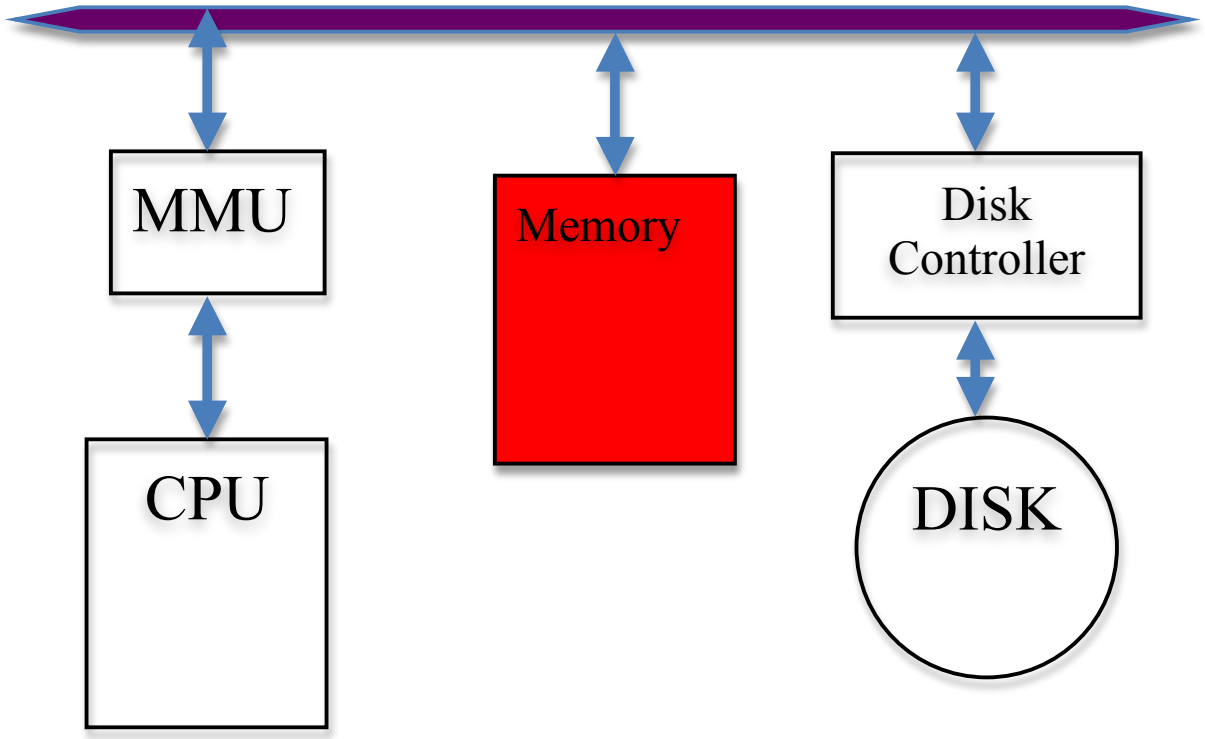
Example: Two-level address translation





Memory management Unit

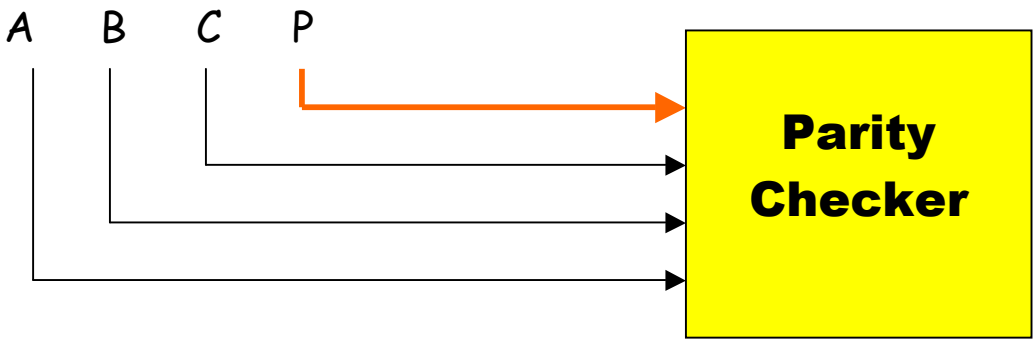
A **memory management unit** (**MMU**), is a hardware unit responsible for handling accesses to memory requested by the processor. Its functions include translation of virtual addresses virtual to physical addresses, memory protection and cache control.



Error Detection and Correction

Transmission or reading errors can corrupt data. If the extent of the damage is known, then error detection codes can **detect** if the data has been corrupted. Using special type of error-correction codes, we can even **correct** errors.

Parity Checking for single error detection



The parity-bit generator generates a P so that the number of 1's in the transmitted data is odd (or even). The **parity checker** checks this in the received data. How will you design (1) a parity-bit generator, and (2) a parity checker?

Hamming Code

Hamming code not only detects if there is an error, but also locates where the error is, and subsequently corrects it. Here is an example:

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

The 4-bit data (1 0 1 1) to be sent is placed in bit positions 7, 6, 5, 3 of a 7-bit frame. The remaining bit positions are reserved for error-correction bits, and their values are chosen so that there is **odd parity** for bit combinations

(4, 5, 6, 7),

(2, 3, 6, 7),

and (1, 3, 5, 7).

What is special about these sets of bits?

The receiver, upon receiving the 7-bit data, computes the parities of the above bit combinations, and reports the result using three bits b2, b1, b0 as follows:

Odd parity for bits 4, 5, 6, 7 $\Rightarrow$ b2 = 1 else b2 = 0.

Odd parity for bits 2, 3, 6, 7 $\Rightarrow$ b1 = 1 else b1 = 0.

Odd parity for bits 2, 3, 6, 7 $\Rightarrow$ b0 = 1 else b0 = 0.

If b2 b1 b0 = 000 then there is no error, otherwise, the decimal equivalent of b2 b1 b0 reports the position of the corrupt bit. To correct the error, the receiver flips that bit.

Question 1. The above implementation works for single errors only. Can you figure out why it works?

Question 2. Can you generalize it to 32-bit data?