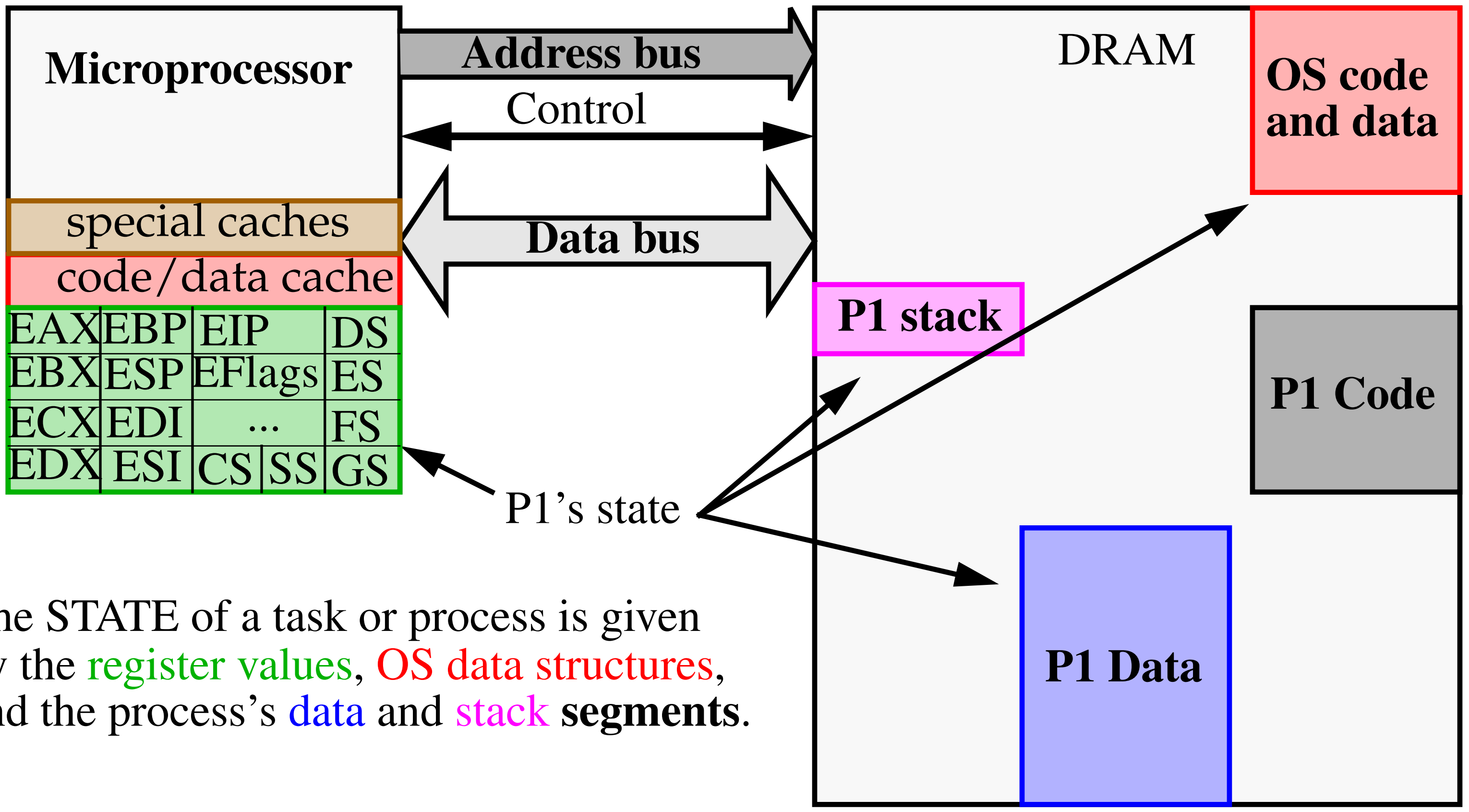
Processes and Tasks

What comprises the state of a running program (a process or task)?



The STATE of a task or process is given by the register values, OS data structures, and the process's data and stack **segments**.

If a second process, **P2**, is to be created and run (not shown), then the state of **P1** must be saved so it can be later resumed with no side-effects.

Since only one copy of the registers exist, they must be saved in memory.

We'll see there is hardware support for doing this on the Pentium later.

Memory Hierarchy

For now, let's focus on the organization and management of memory.

Ideally, programmers would like a fast, infinitely large nonvolatile memory.

In reality, computers have a memory hierarchy:

Cache (SRAMS): Small (KBytes), expensive, volatile and very fast ($<$ 5ns).

Main Memory (DRAM): Larger (MBytes), medium-priced, volatile and medium-speed ($<$80ns).

Disk: GBytes, low-priced, non-volatile and slow (ms).

Therefore, the OS is charged with managing these limited resources and creating the illusion of a fast, infinitely large main memory.

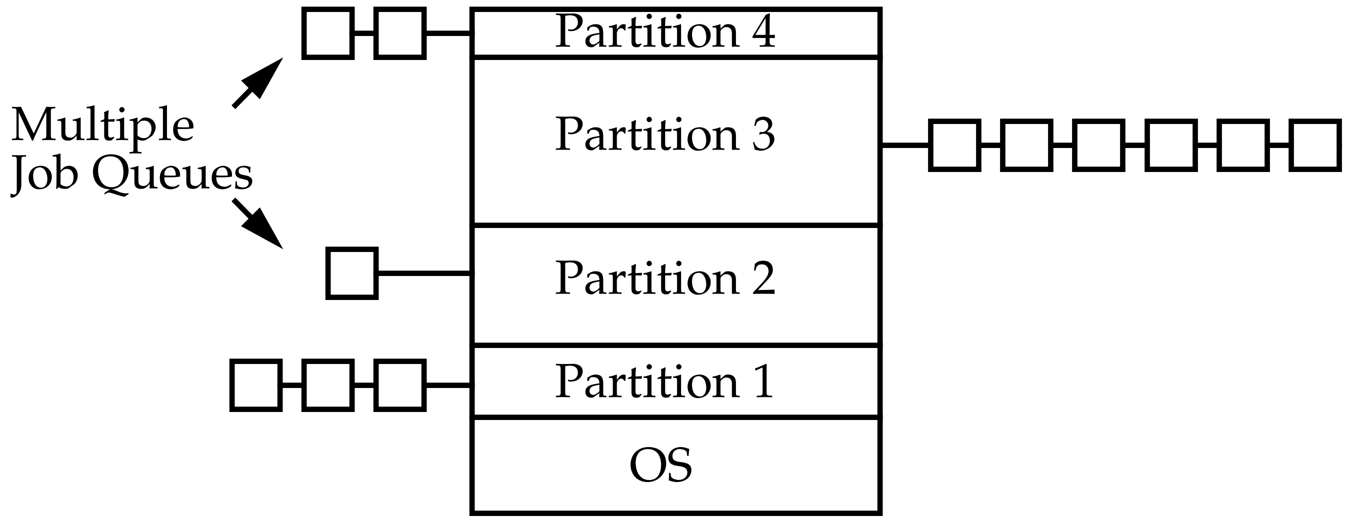
The Memory Manager portion of the OS:

- Tracks memory usage.
- Allocates/Deallocates memory.
- Implements virtual memory.

Simple Memory Management

In a multiprogramming environment, a simple memory management scheme is to divide up memory into n (possibly unequal) fixed-sized partitions.

These partitions are defined at system start-up and can be used to store all the segments of the process (e.g., *code*, *data* and *stack*).

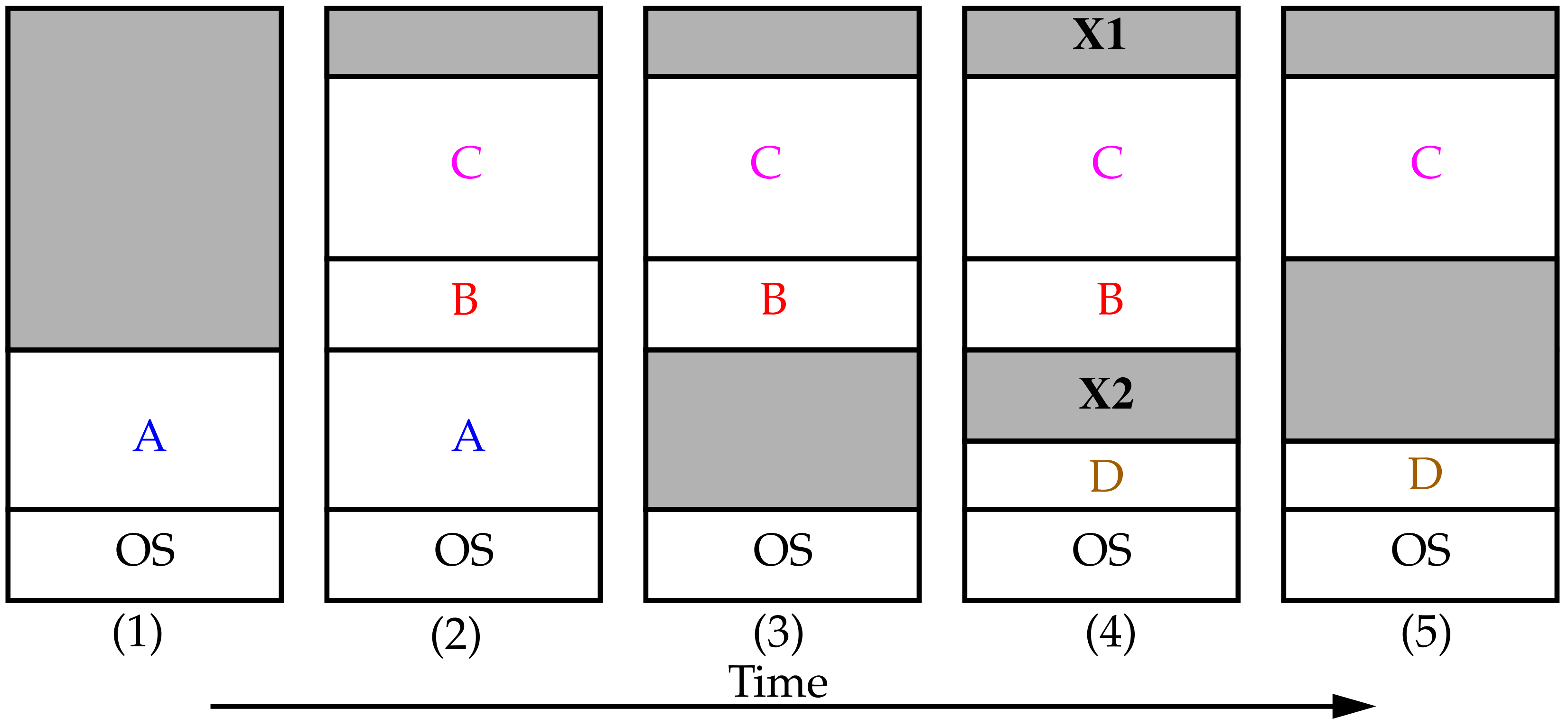


Advantage: it's simple to implement.

However, it utilizes memory poorly. Also, in time sharing systems, queueing up jobs in this manner leads to unacceptable response time for user processes.

Variable-Sized Partitions

In a *variable-sized* partition scheme, the number, location and size of memory partitions vary dynamically:



(1) Initially, process *A* is in memory.

(2) Then *B* and *C* are created.

(3) *A* terminates.

(4) *D* is created, *B* terminates.

Variable-Sized Partitions

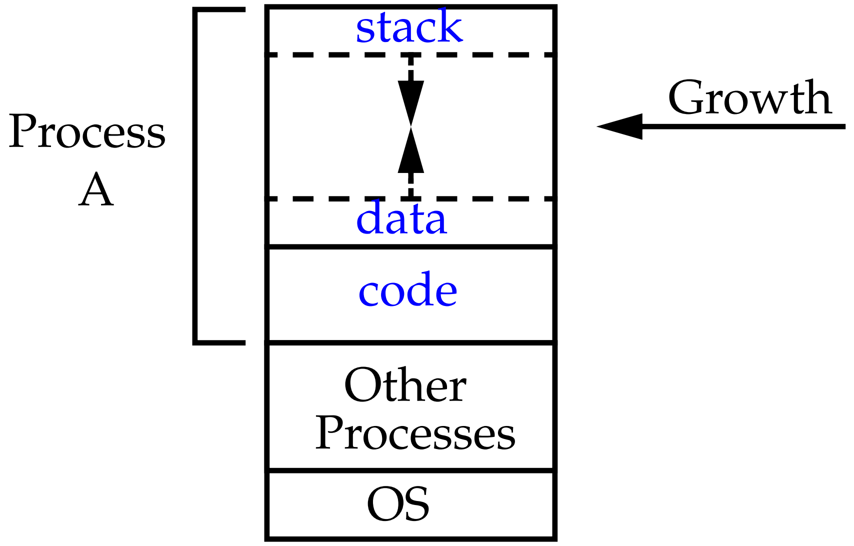
Problem: Dynamic partition size improves memory utilization but complicates allocation and deallocation by creating holes (**external fragmentation**).

This may prevent a process from running that could otherwise run if the holes were merged, e.g., combining *X1* and *X2* in previous slide.

Memory compaction is a solution but is rarely used because of the CPU time involved.

Also, the size of a process's data segments can change dynamically, e.g. *malloc()*.

If a process does not have room to grow, it needs to be moved or killed.



Implementing Memory on the Hard Drive

The hard disk can be used to allow more processes to run than would normally fit in main memory.

For example, when a process blocks for I/O (e.g. keyboard input), it can be *swapped* out to disk, allowing other processes to run.

The movement of whole processes to and from disk is called **swapping**.

The disk can be used to implement a second scheme, **virtual memory**.

Virtual memory allows processes to run even when their total size (code, data and stack) exceeds the amount of physical memory (installed DRAM).

This is very common, for example, in microprocessors with 32-bit address spaces.

If an OS supports **virtual memory**, it allows for the execution of processes that are only *partially* present in main memory.

OS keeps the parts of the process that are currently in use in main memory and the rest of the process on disk.

Virtual Memory

When a new portion of the process is needed, the OS swaps out older "*not recently used*" memory to disk.

Virtual memory also works in a multiprogrammed system.

- Main memory stores bits and pieces of many processes.
- A process blocks whenever it requires a portion of itself that is on disk, much in the same way it blocks to do I/O.
- The OS schedules another process to run until the referenced portion is fetched from disk.

But swapping out portions of memory that vary in size is not efficient.
External fragmentation is still a problem (it reduces memory utilization).

Two concepts:

- **Segmentation:** Allows the OS to "share" code and enforce meaningful constraints on the memory used by a process, e.g. no execution of data.
- **Paging:** Allows the OS to efficiently manage physical memory, and makes it easier to implement virtual memory.

Paging and Virtual Memory

So how does paging work?

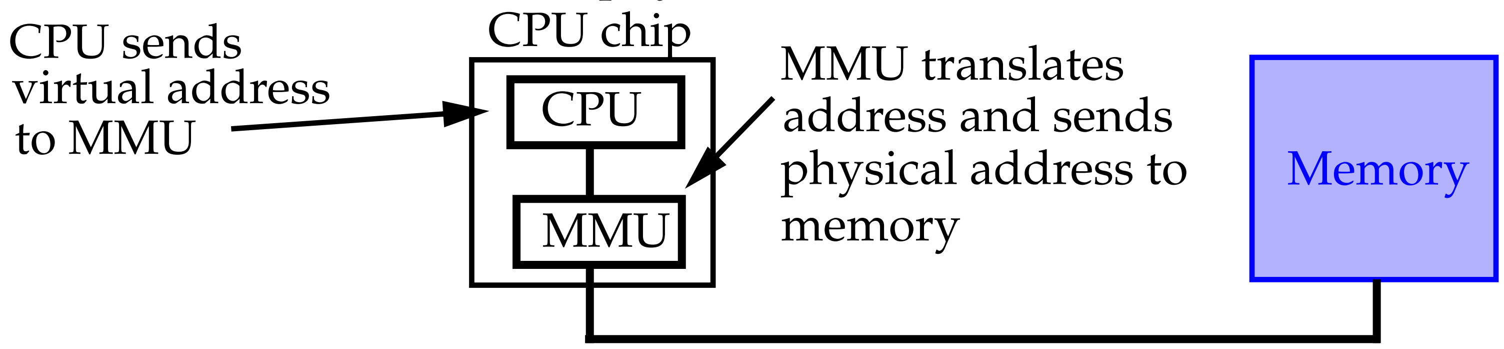
We will refer to addresses which appear on the address bus of main memory as a *physical addresses*.

Processes generate *virtual addresses*, e.g., MOV EAX, [EBX]

Note, the value given in [EBX] can reference memory locations that exceed the size of physical memory.

(We can also start with *linear addresses*, which are *virtual addresses* translated through the segmentation system, to be discussed).

All virtual (or linear) addresses are sent to the **Memory Management Unit** (MMU) for translation to a physical address.


CPU sends virtual address to MMU
CPU chip
CPU
MMU
MMU translates address and sends physical address to memory
Memory

Paging and Virtual Memory

The virtual (and physical) address space is divided into **pages**.

Page size is architecture dependent but usually range between 512- 64K.

Corresponding units in physical memory are called **page frames**.

Pages and page frames are usually the same size.

Assume:

Page size is 4K.

Physical mem is 32K.

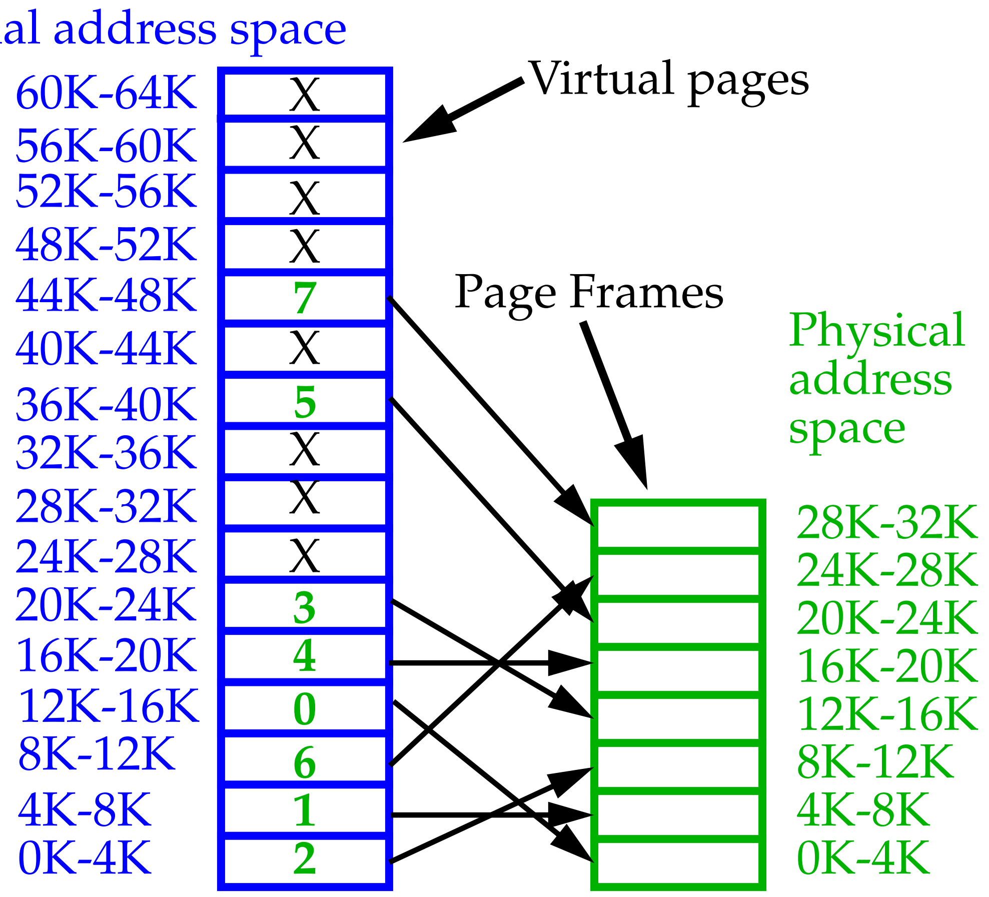
Virtual mem is 64K.

Therefore, there are 16 virtual pages. 8 page frames.

Assume the process issues the virtual address 0:

Paging translates it to physical address 8192 using the layout on right.

20500 is translated to physical address 12K + 20 = 12308.



Paging and Virtual Memory

Note that 8 virtual pages are not mapped into physical memory (indicated by an *X* on the previous slide).

A *present/absent* bit in the hardware indicates which virtual pages are mapped into physical RAM and which ones are not (out on disk).

What happens when a process issues an address to an unmapped page?

- MMU notes page is unmapped using present/absent bit.
- MMU causes CPU to **trap** to OS - page fault.
- OS selects a page frame to replace and saves its current contents to disk.
- OS fetches the page referenced and places it into the freed page frame.
- OS changes the mem map and restarts the instruction that caused the trap.

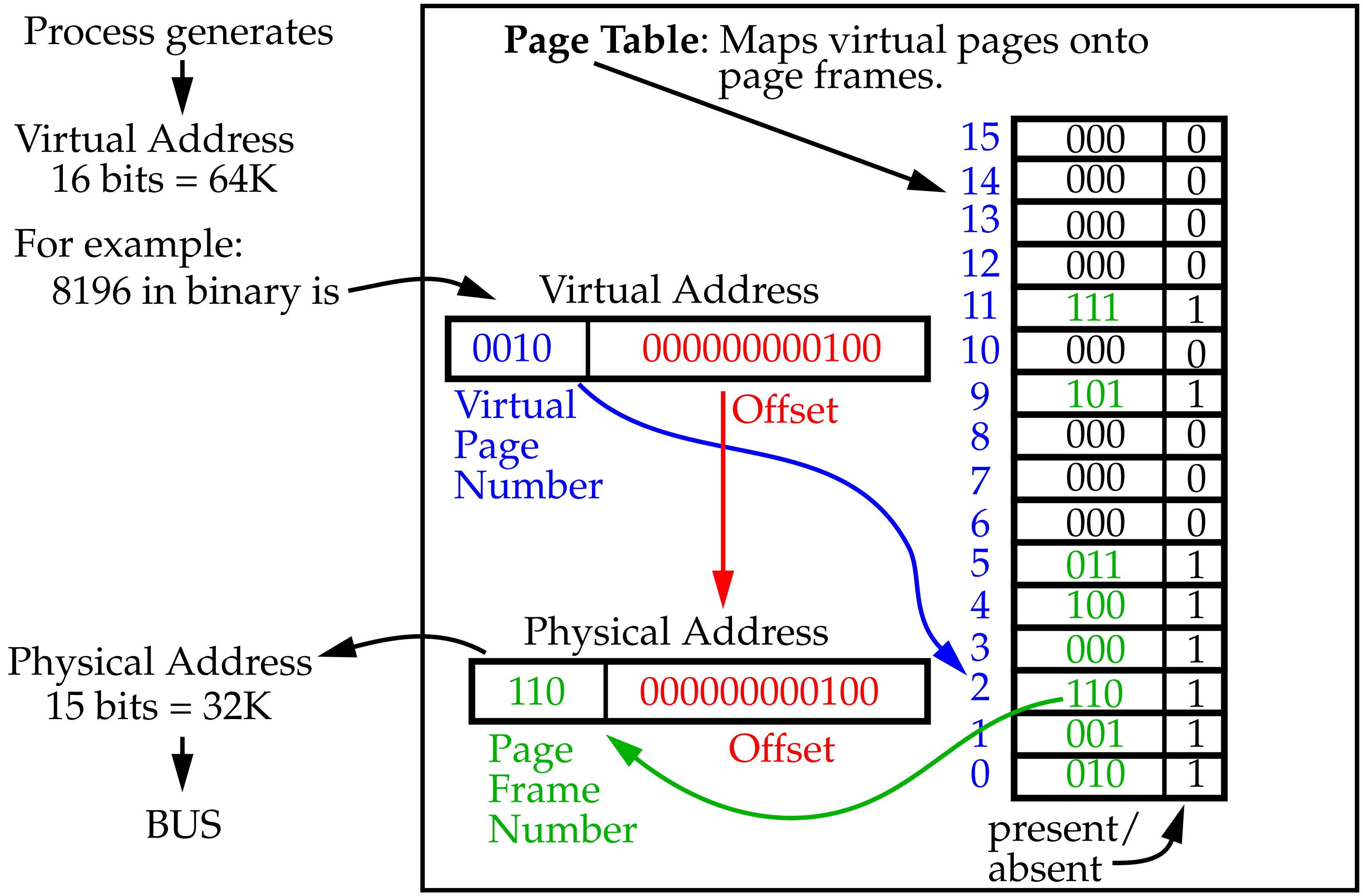
Paging allows the physical address space of a process to be *noncontiguous* !

This solves the *external fragmentation* problem (since any set of pages can be chosen as the address space of the process).

However, it generally doesn't allow 100% mem utilization, since the last page of a process may not be entirely used (**internal fragmentation**).

Paging and Virtual Memory

Addresses Translation by the MMU



15	000	0
14	000	0
13	000	0
12	000	0
11	111	1
10	000	0
9	101	1
8	000	0
7	000	0
6	000	0
5	011	1
4	100	1
3	000	1
2	110	1
1	001	1
0	010	1

Paging and Virtual Memory

Two important issues w.r.t the Page Table:

- *Size*:

  The Pentium uses 32-bit virtual addresses.

  With a 4K page size, a 32-bit address space has $2^{32}/2^{12} = 2^{20}$ or 1,048,576 virtual page numbers !

  If each page table entry occupies 4 bytes, that's 4MB of memory, just to store the page table.

  For 64-bit machines, there are 2^{52} virtual page numbers !!!

- *Performance*:

  The mapping from virtual-to-physical addresses must be done for *EVERY* memory reference.

  Every instruction fetch requires a memory reference.

  Many instructions have a memory operand.

  Therefore, the mapping must be extremely fast, a couple nanoseconds, otherwise it becomes the bottleneck.

Page Table Design Alternatives

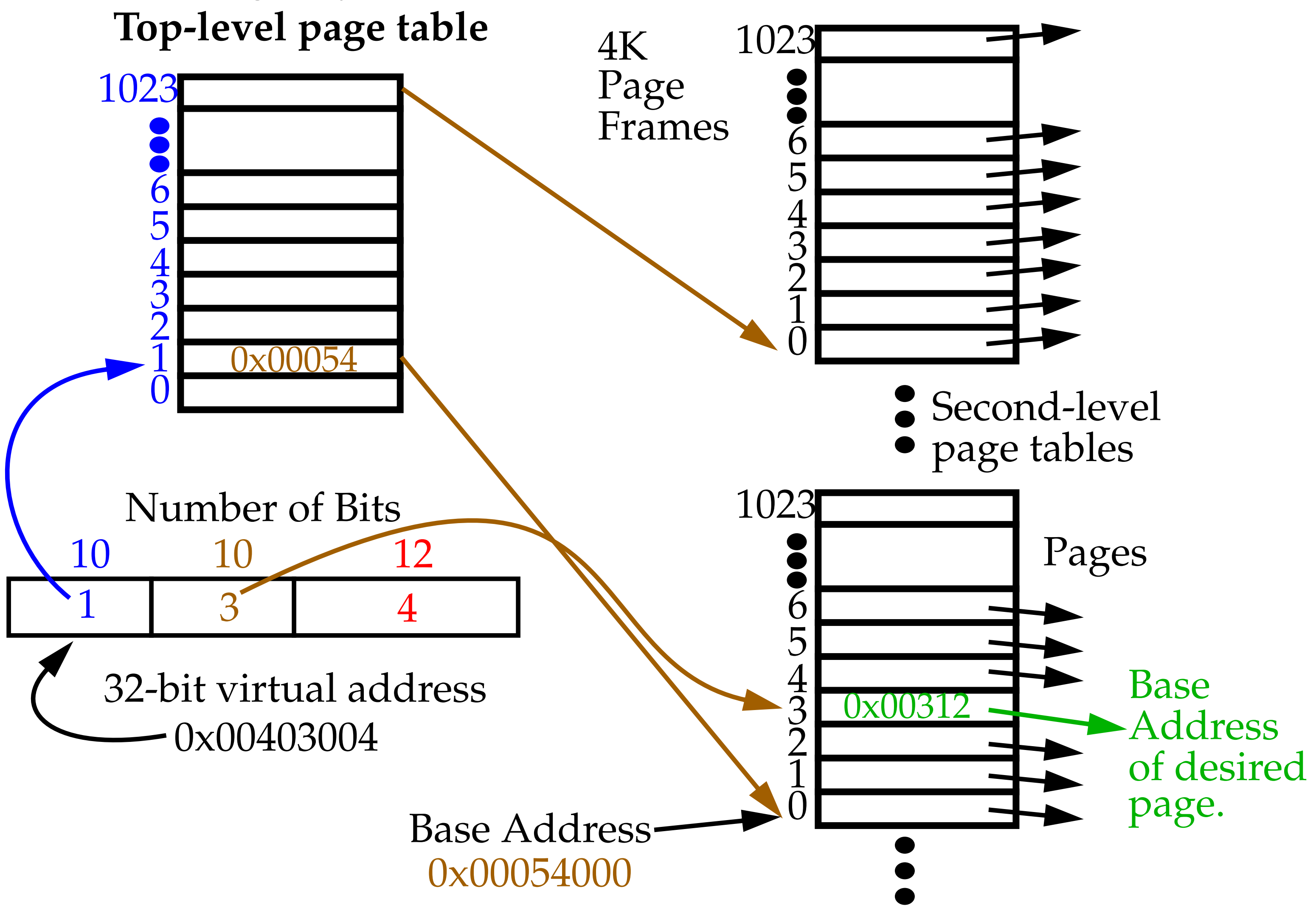
- Single page table stored in an array of fast hardware registers.
  OS loads registers from memory when a process is started.
  - Advantage: No memory references are needed for the page table.
  - Disadvantage: Context switches require the entire page table to be loaded.
    If it is large, this will be expensive.

- Page table kept entirely in main memory.
  Single register points to the start of the page table.
  - Advantage: Context switches only require updating the register pointer.
  - Disadvantage: One or more memory references are needed to read page table entries for each instruction.

Modern computers keep "frequently used" page table entries on chip in a cache (similar to first alternative above) and the others in main memory (similar to the second alternative).

Multilevel Page Tables

Instead of using only one level of indirection, use two.



Multilevel Page Tables

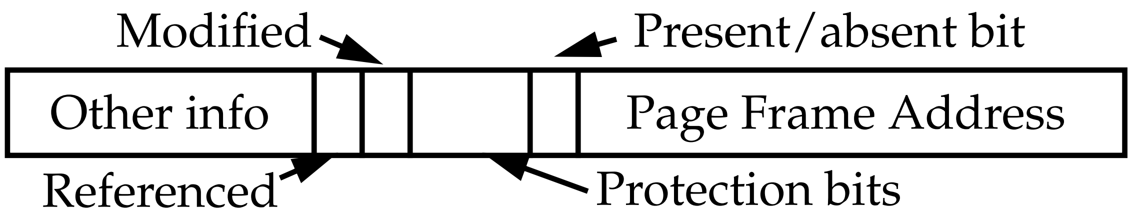
This addresses page table size problem since many of the second-level page tables need not be defined (and therefore stored in main memory).

Note that two page faults can occur for a single memory reference.

If the second-level page table is not in memory, a page fault occurs.

If the page that the second-level entry refers to is not in memory, another page fault occurs.

In general, **Page Frames** are machine dependent with the following info:



- *Page Frame address*: Most significant bits of physical memory address.
- *Present/Absent bit*: If 1, page is in memory, if 0, it is on disk.
- *Modified bit*: If set, page has been written to, e.g. it is ‘dirty’.
- *Referenced bit*: Used in the OS page replacement algorithm.
- *Protection bits*: Specifies if data in page can be read/written/executed.

Translation Lookaside Buffers (TLBs)

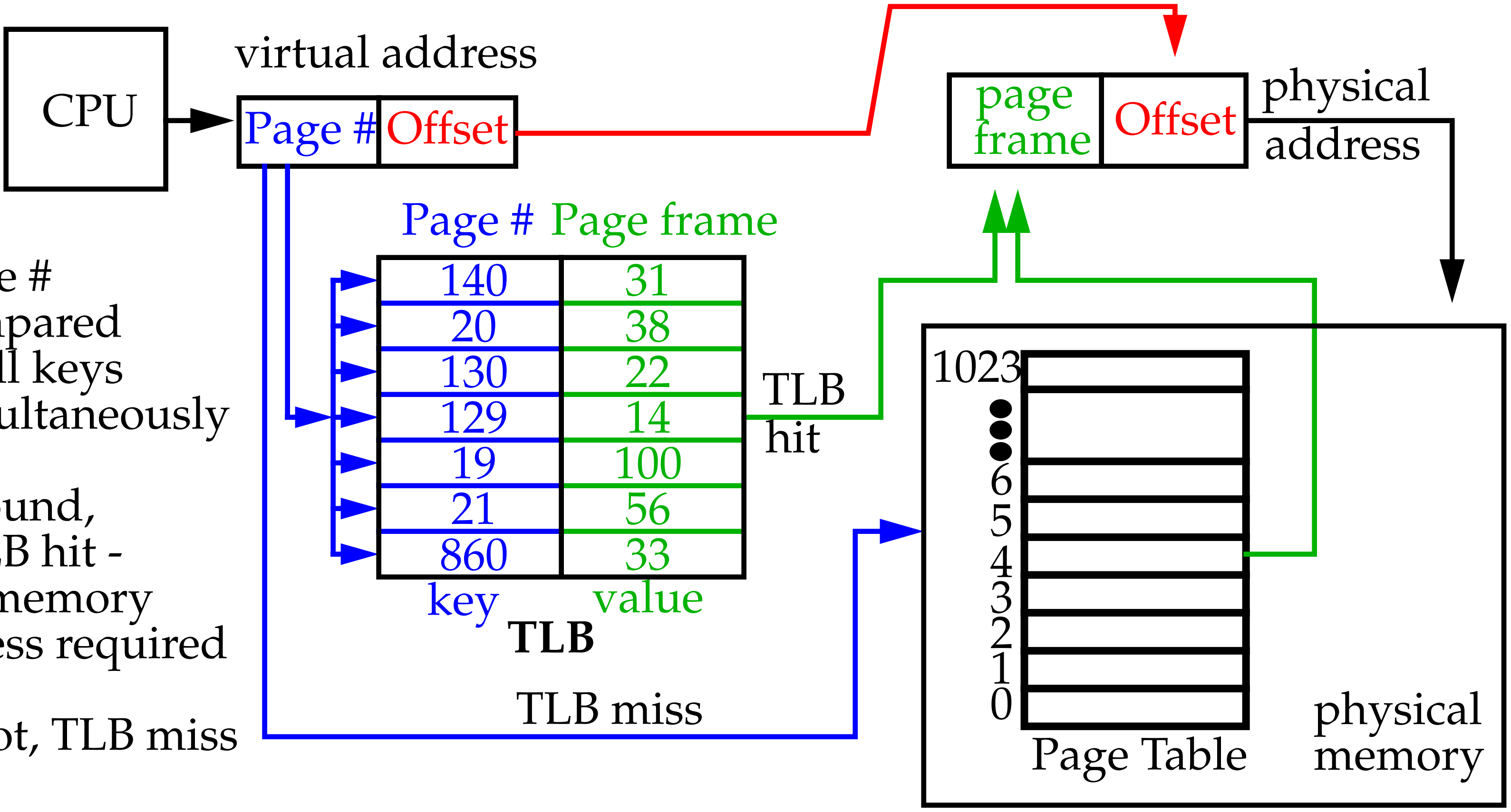
With two-level paging, one memory reference could require three memory accesses !

In order to reduce the number of times this occurs, a fast lookup table called a **TLB** is added as a hardware cache in the microprocessor.

Page # compared to all keys simultaneously

If found, - TLB hit - no memory access required

If not, TLB miss



Translation Lookaside Buffers (TLBs)

Number of TLB entries varies from 8 to 2048.

Typically around 64.

When a *TLB miss* occurs:

- A trap occurs and an OS routine handles the fault. The instruction is then restarted.
- The OS routine copies one (or more) page frame(s) from the page table in memory to one (or more) of the TLB entries.

Therefore, if page is referenced again soon, a *TLB hit* occurs eliminating the memory reference for the page frame.