Operating Systems

Memory Management

Operating Systems

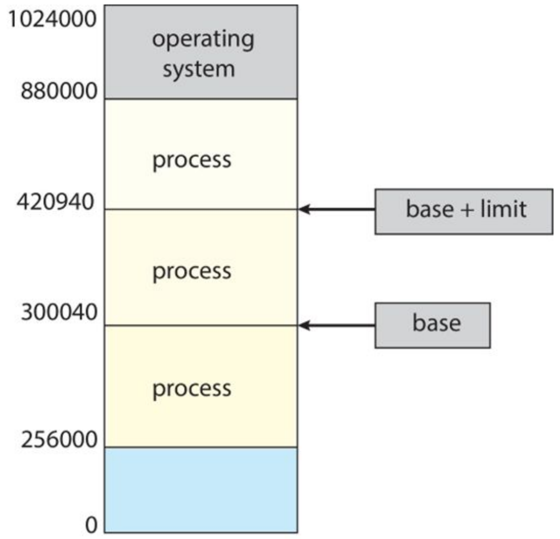
Main Memory

Background

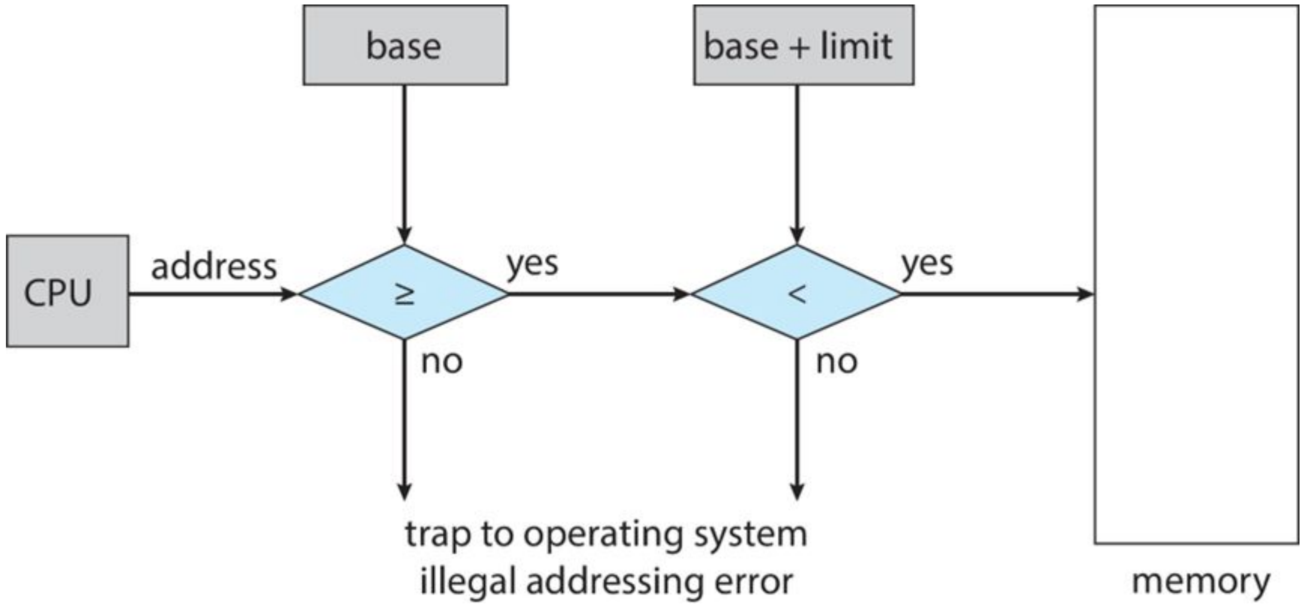
- ❑ Program must be brought (from disk) into memory and placed within a process for it to be run
- ❑ Main memory and registers are only storage CPU can access directly
- ❑ Memory unit only sees a stream of addresses + read requests, or address + data and write requests
- ❑ Register access in one CPU clock (or less)
- ❑ Main memory can take many cycles, causing a **stall,** since it does not have the data required to complete the instruction that it is executing
- ❑ **Cache** sits between main memory and CPU registers for fast access
- ❑ Protection of memory required to ensure correct operation

Base and Limit Registers

- A pair of **base** and **limit registers** define the logical address space
- CPU must check every memory access generated in user mode to be sure it is between base and limit for that user



Hardware Address Protection



Operating Systems

Address Space

Logical vs. Physical Address Space

- The concept of a logical address space that is bound to a separate **physical address space** is central to proper memory management
    - **Logical address** – generated by the CPU; also referred to as **virtual address**
    - **Physical address** – address seen by the memory unit
- Logical and physical addresses are the same in compile-time and load-time address-binding schemes; logical (virtual) and physical addresses differ in execution-time address-binding scheme
- **Logical address space** is the set of all logical addresses generated by a program
- **Physical address space** is the set of all physical addresses generated by a program

Operating Systems

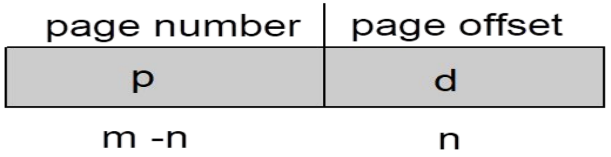
Paging

Paging

- Physical address space of a process can be noncontiguous; process is allocated physical memory whenever the latter is available
  - Avoids external fragmentation
  - Avoids problem of varying sized memory chunks
- Divide physical memory into fixed-sized blocks called **frames**
  - Size is power of 2, between 512 bytes and 16 Mbytes
- Divide logical memory into blocks of same size called **pages**
- Keep track of all free frames
- To run a program of size ***N*** pages, need to find ***N*** free frames and load program
- Set up a **page table** to translate logical to physical addresses
- Backing store likewise split into pages
- Still have Internal fragmentation

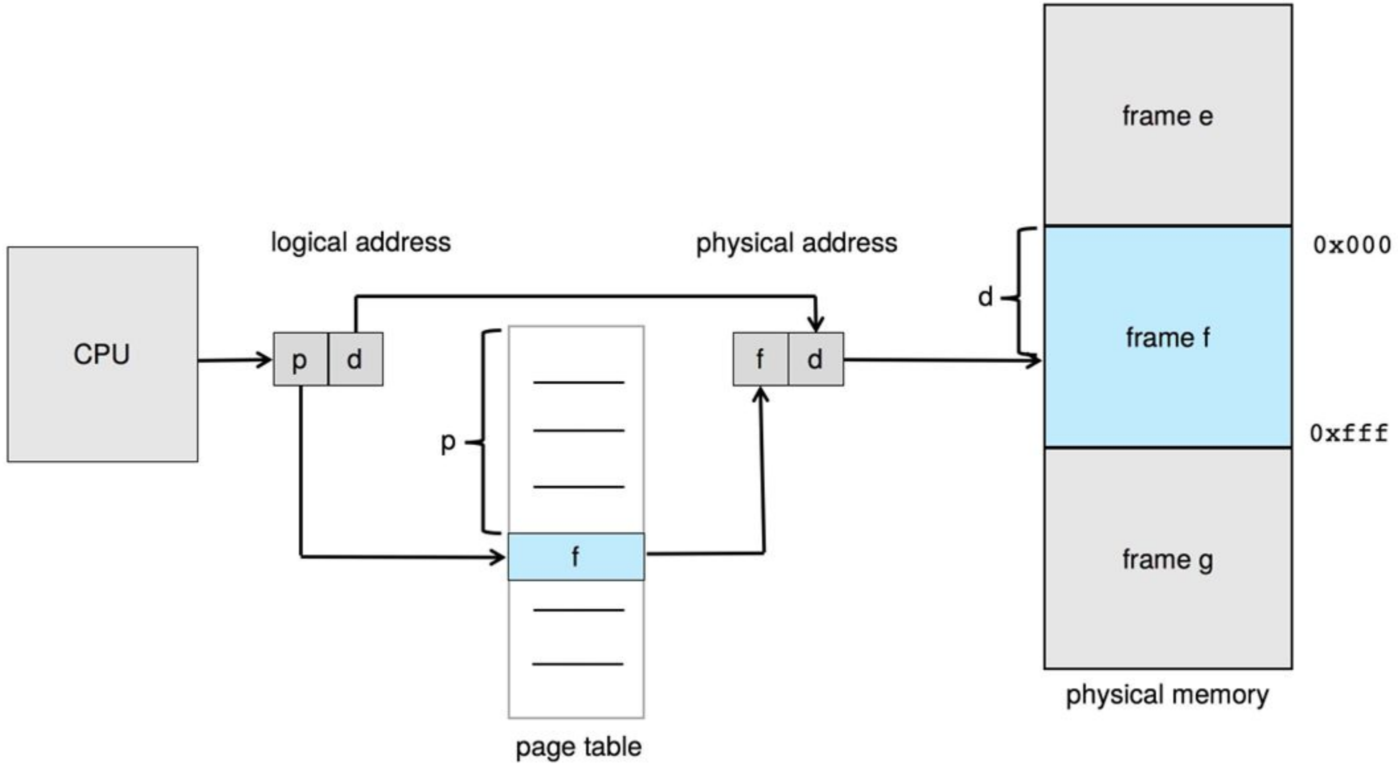
Address Translation Scheme

- Address generated by CPU is divided into:
    - **Page number** (***p***) – used as an index into a **page table** which contains base address of each page in physical memory
    - **Page offset** (***d***) – combined with base address to define the physical memory address that is sent to the memory unit

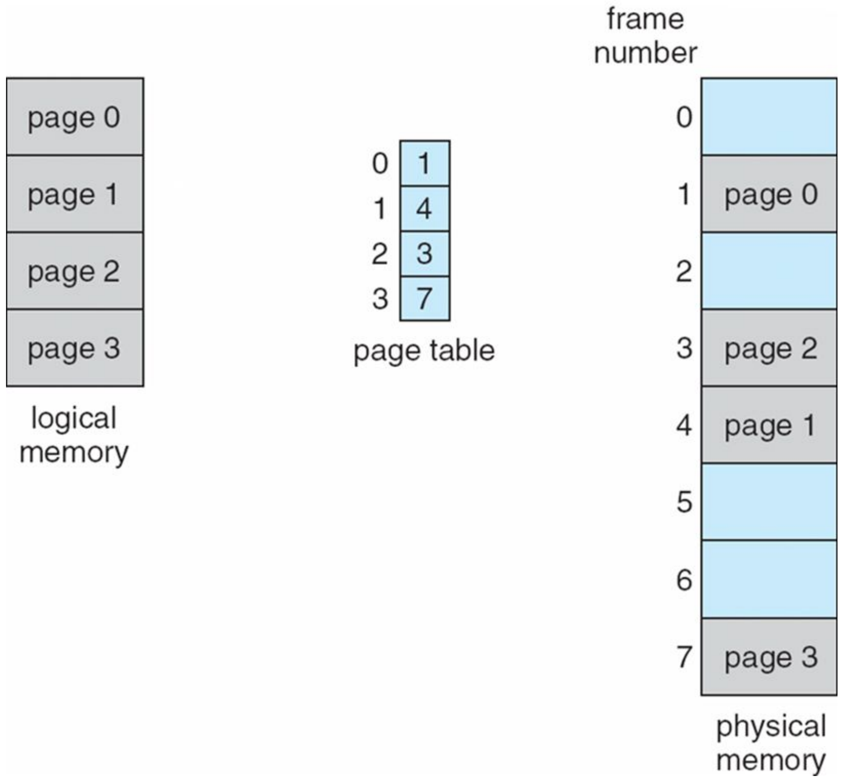


    - For given logical address space 2^m and page size 2^n

Paging Hardware



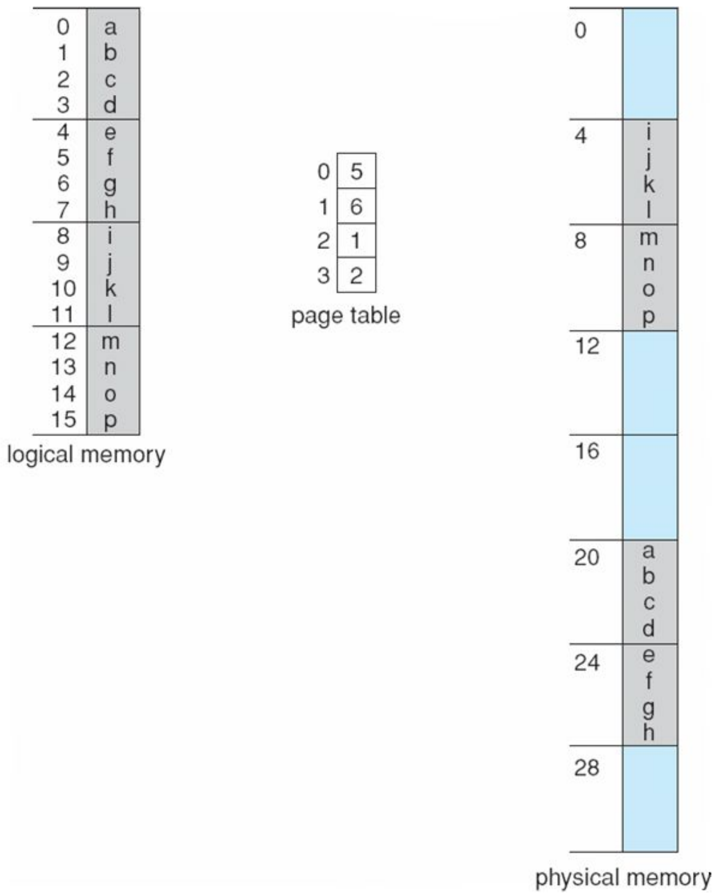
Paging Model of Logical and Physical Memory



Paging Example

n=2 and m=4

32-byte memory and 4-byte pages



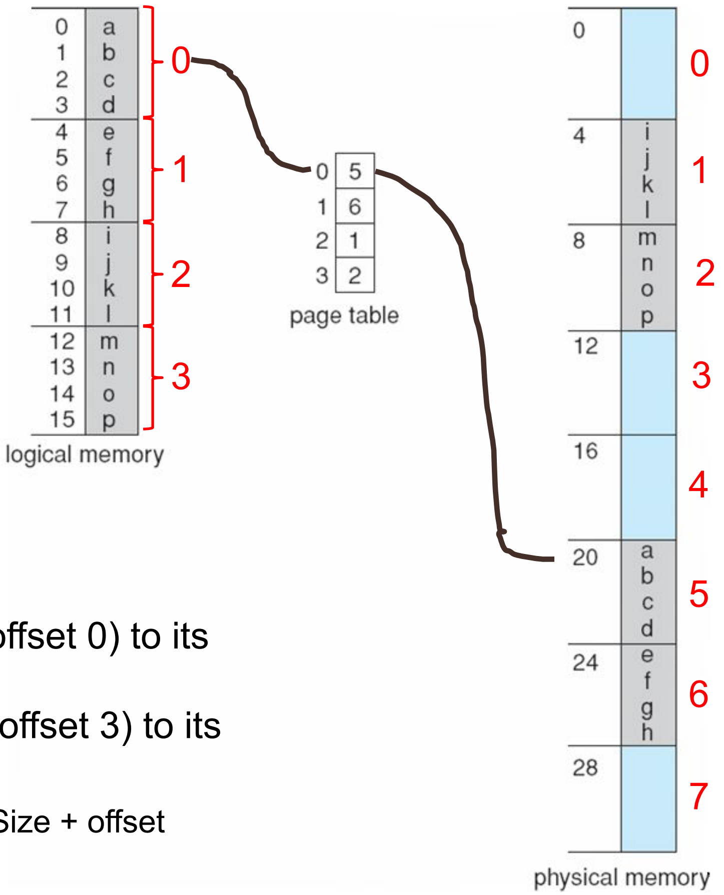
32/4 = 8 frames

Paging Example

- m is used to determine the number of logical addresses, $2^m = 16$
- n indicates the offset within each logical address

n=2 and m=4

Physical memory → 32-byte memory and 4-byte pages



- Map logical address 0 (page 0, offset 0) to its corresponding physical address
- Map Logical address 3 (page 0, offset 3) to its corresponding physical address

frameNumber * frameSize + offset

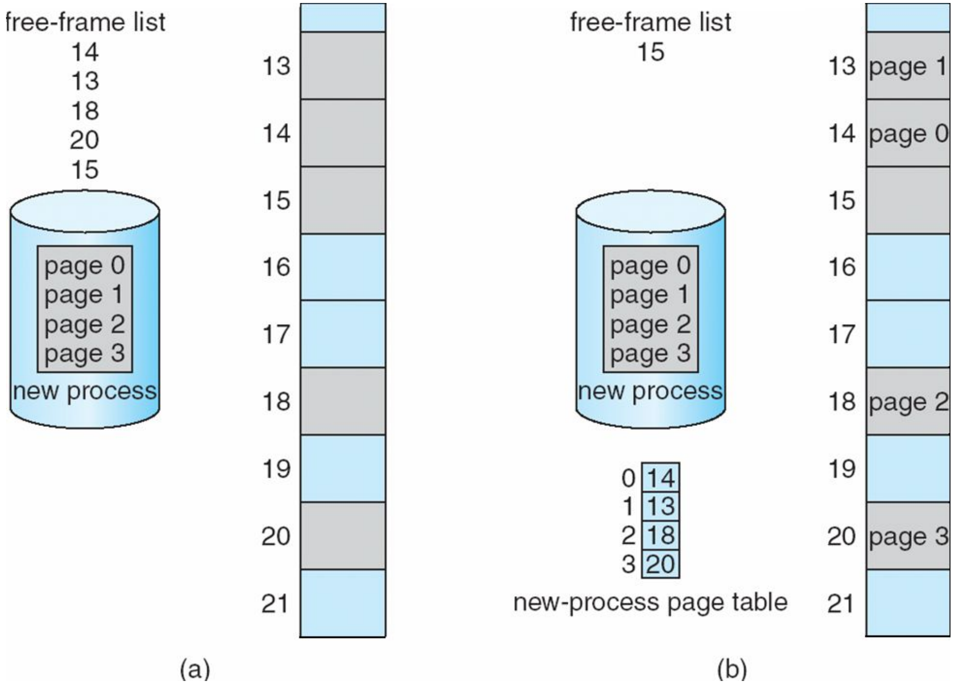
Size of Page

- Calculating internal fragmentation
  - Page size = 2,048 bytes
  - Process size = 72,766 bytes
  - 35 pages + 1,086 bytes
  - Internal fragmentation of 2,048 - 1,086 = 962 bytes
  - Worst case fragmentation = 1 frame – 1 byte
  - On average fragmentation = 1 / 2 frame size
  - So small frame sizes desirable?
  - But each page table entry takes memory to track
  - Page sizes growing over time
    - Solaris supports two page sizes – 8 KB and 4 MB

- Process view and physical memory now very different

- By implementation process can only access its own memory

Free Frames



Before allocation

After allocation

Operating Systems

Implementation of Page Table

Implementation of Page Table

- Page table is kept in main memory
- **Page-table base register** (**PTBR**) points to the page table
- **Page-table length register** (**PTLR**) indicates size of the page table
- In this scheme every data/instruction access requires two memory accesses
  - One for the page table and one for the data / instruction
- The two memory access problem can be solved by the use of a special fast-lookup hardware cache called **associative memory** or **translation lookaside buffers** (**TLBs**)

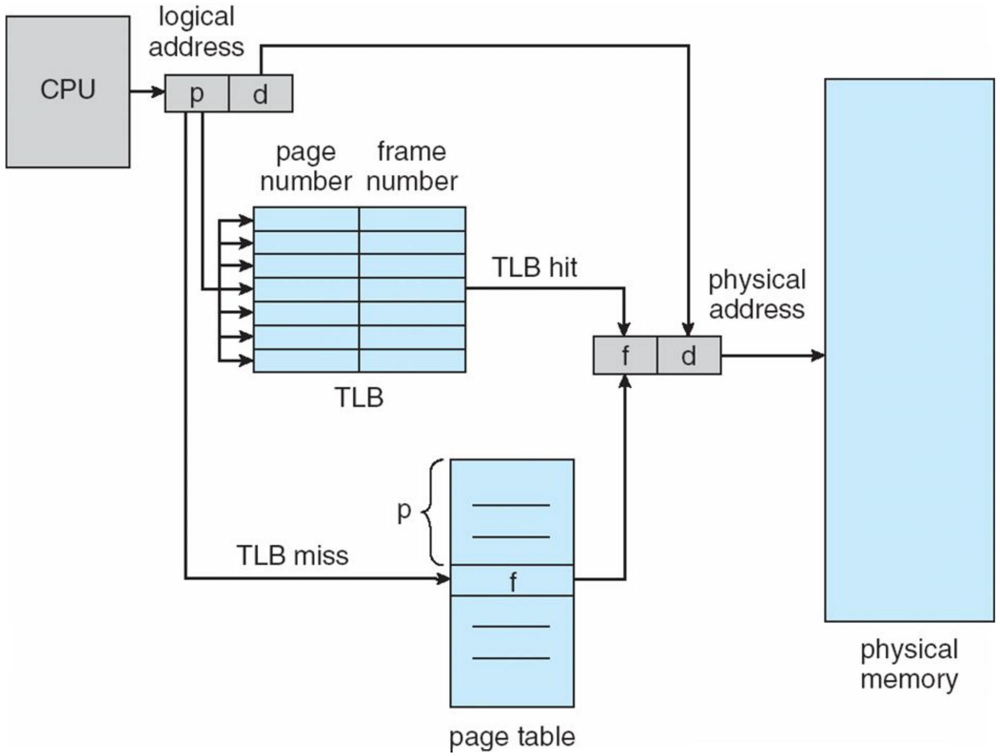
Associative Memory

- Associative memory – parallel search

Page #	Frame #

- Address translation (p, d)
  - If p is in associative register, get frame # out
  - Otherwise get frame # from page table in memory

Paging Hardware With TLB



Effective Access Time

- Associative Lookup = ε time unit
  - Can be < 10% of memory access time

- Hit ratio = α
  - Hit ratio – percentage of times that a page number is found in the associative registers;

- **Effective Access Time** (**EAT**):

- Consider α = 80%, ε = 20ns for TLB search, 100ns for memory access
  - EAT = 0.80 x 120 + 0.20 x 220 = 140ns

- Consider more realistic hit ratio -> α = 99%, ε = 20ns for TLB search, 100ns for memory access
  - EAT = 0.99 x 120 + 0.01 x 220 = 121ns

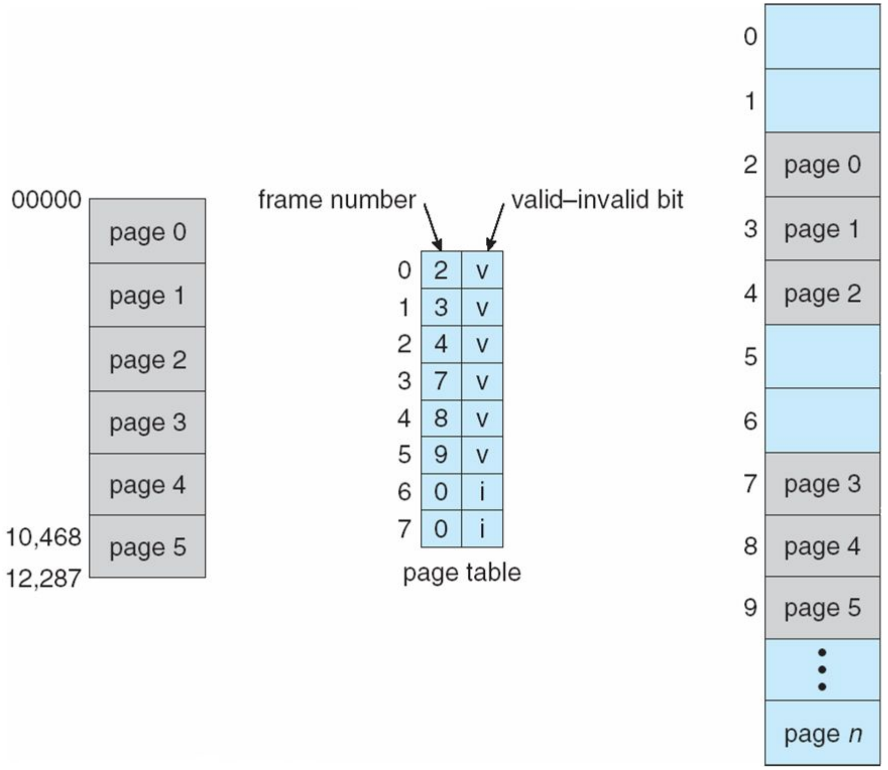
Memory Protection

- Memory protection implemented by associating protection bit with each frame to indicate if read-only or read-write access is allowed
    - Can also add more bits to indicate page execute-only, and so on

- **Valid-invalid** bit attached to each entry in the page table:
    - "valid" indicates that the associated page is in the process' logical address space, and is thus a legal page
    - "invalid" indicates that the page is not in the process' logical address space
    - Or use **page-table length register** (**PTLR**)

- Any violations result in a trap to the kernel

Valid (v) or Invalid (i) Bit In A Page Table



Operating Systems

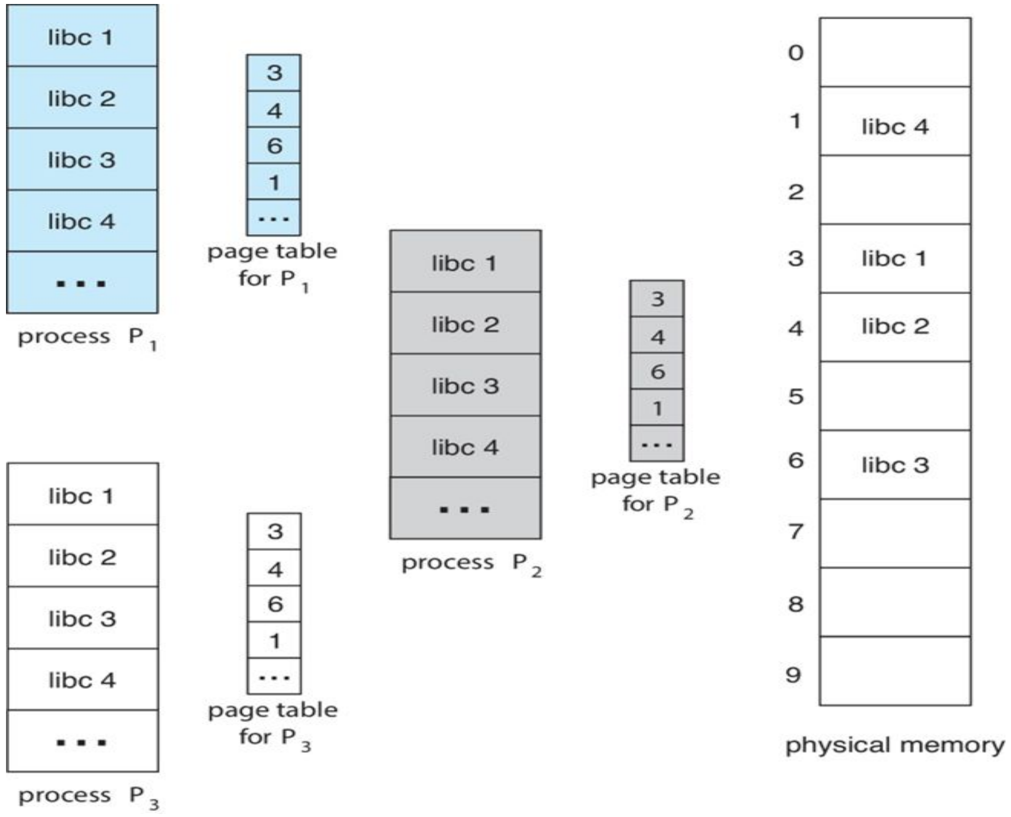
Shared Pages

Shared Pages

- **Shared code**
  - One copy of read-only (**reentrant**) code shared among processes (i.e., text editors, compilers, window systems)
  - Similar to multiple threads sharing the same process space
  - Also useful for inter-process communication if sharing of read-write pages is allowed

- **Private code and data**
  - Each process keeps a separate copy of the code and data
  - The pages for the private code and data can appear anywhere in the logical address space

Shared Pages Example



Operating Systems

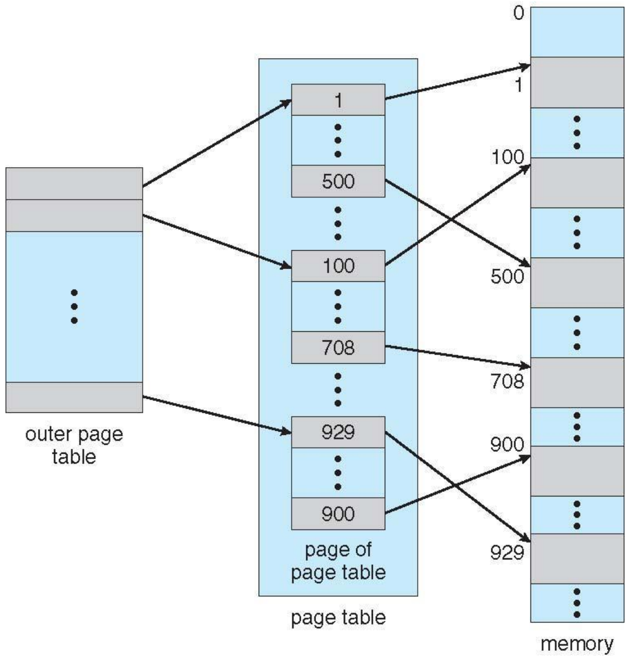
Hierarchical Page Table

Structure of the Page Table

- Memory structures for paging can get huge using straightforward methods
  - Consider a 32-bit logical address space as on modern computers
  - Page size of 4 KB (2^{12})
  - Page table would have 1 million entries (2^{32} / 2^{12})
  - If each entry is 4 Bytes -> 4 MB of physical address space / memory for page table alone
    - That amount of memory used to cost a lot
    - Don’t want to allocate that contiguously in main memory

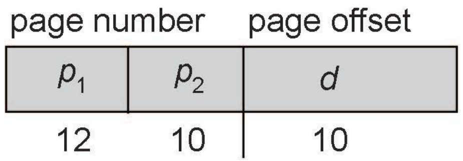
Hierarchical Page Tables

- Break up the logical address space into multiple page tables
- A simple technique is a two-level page table
- We then page the page table



Two-Level Paging Example

- A logical address (on 32-bit machine with 1KB page size) is divided into:
  - a page number consisting of 22 bits
  - a page offset consisting of 10 bits
- Since the page table is paged, the page number is further divided into:
  - a 12-bit page number
  - a 10-bit page offset
- Thus, a logical address is as follows:



- where p_1 is an index into the outer page table, and p_2 is the displacement within the page of the inner page table
- Known as **forward-mapped page table**

Operating Systems

Virtual Memory

Background

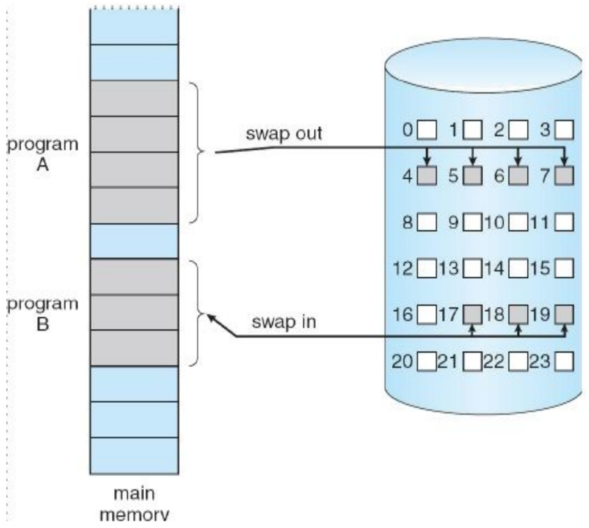
- ❑ The term “virtual memory” refers to something which appears to be present but actually it is not.
- ❑ The virtual memory technique allows users to use more memory for a program than the real memory of a computer.
- ❑ Virtual memory is a **concept** that we use when we have processes that exceed the main memory.
- ❑ When computer runs out of physical memory, it writes its requirement to the hard disc in a swap file as “virtual memory”.

Demand Paging

❑ Bring a page into memory only when it is needed
- Less I/O needed
- Less memory needed
- Faster response
- More users

❑ Disadvantage: Page fault interrupt

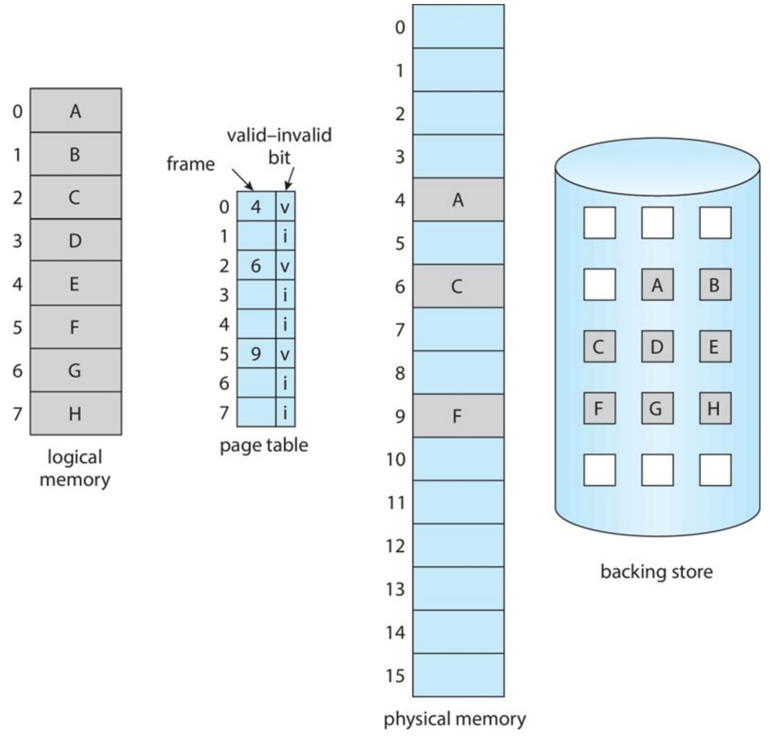
❑ Required hardware support:
- Page Table with valid-invalid bit
- Secondary memory



Demand Paging in OS

Valid-Invalid Bit

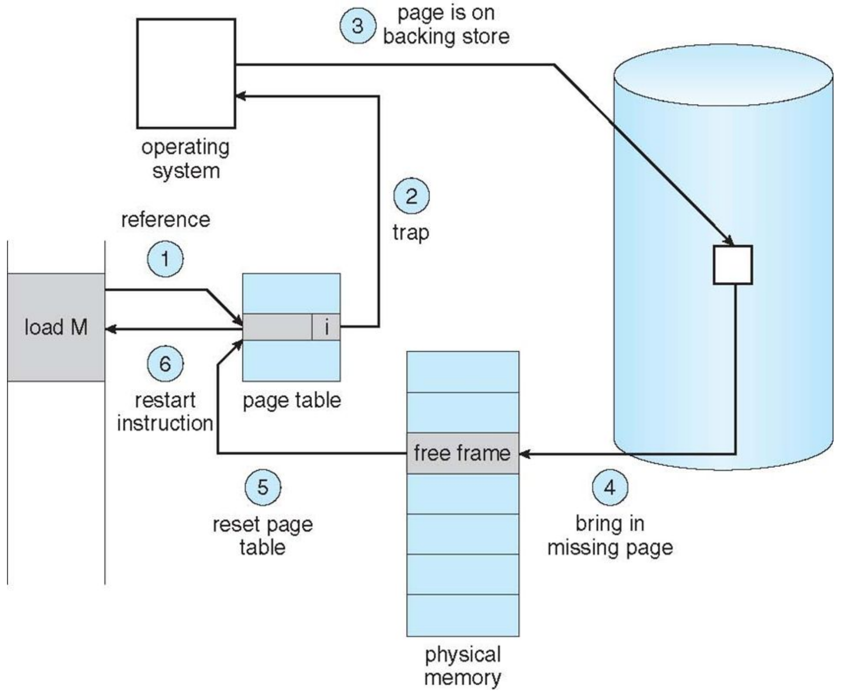
- ❑ An extra bit in the page table which indicates the existence of the page in the main memory.
- ❑ Attempt to access page
- ❑ If page is valid (in memory) then continue processing instruction as normal.
- ❑ If page is invalid then a page-fault trap / page-fault interrupt occurs.
- ❑ Page is needed ⇒ reference to it
  - Invalid reference ⇒ abort
  - Not-in-memory ⇒ bring to memory



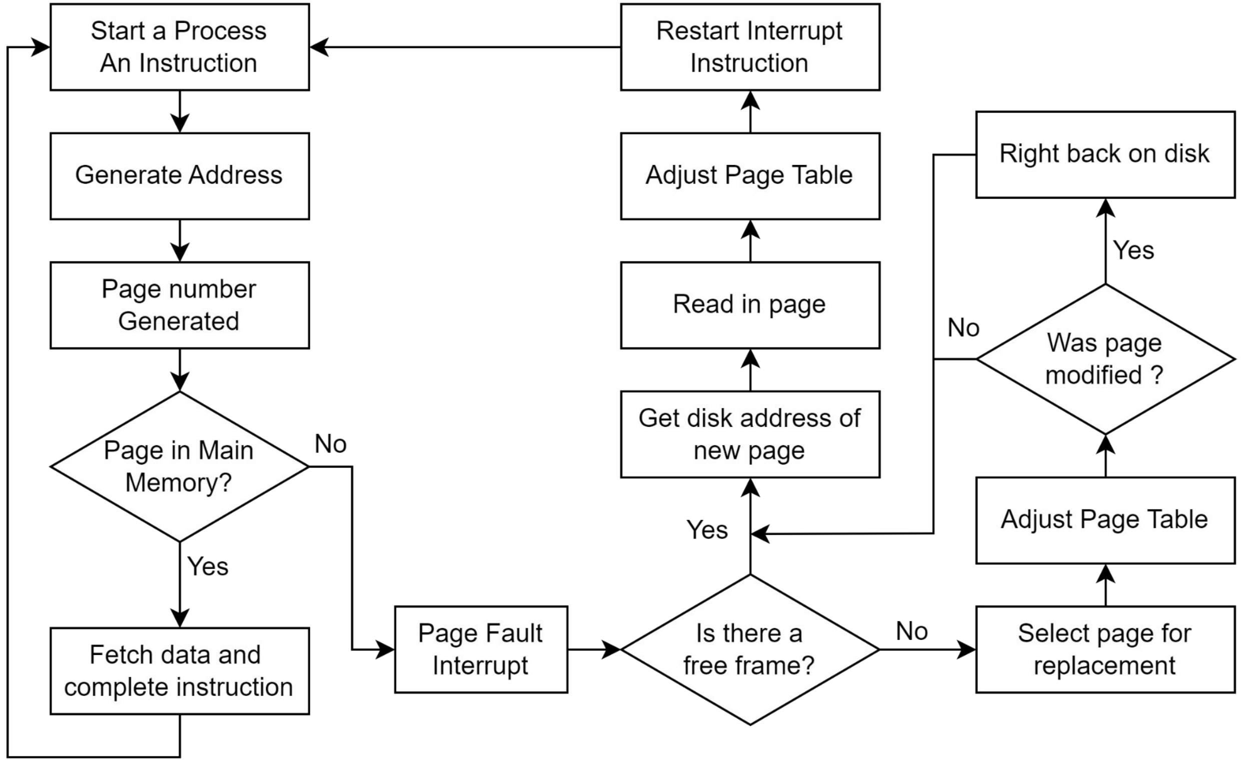
Page Fault

If there is ever a reference to a page, first reference will trap to OS ⇒ **page fault**

1. OS looks at another table to decide:
   - Invalid reference ⇒ abort.
   - Just not in memory.
2. Find empty/ free frame.
3. Load page from disk into frame.
4. Reset tables, validation bit = 1.
5. Restart instruction that caused page fault



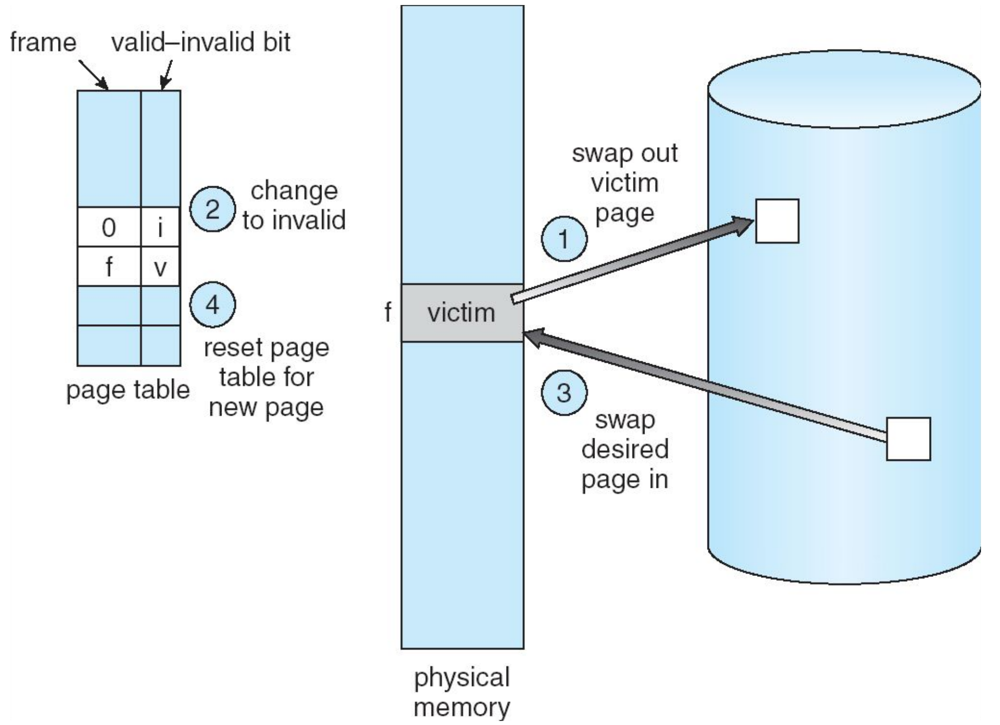
Demand Paging Flowchart



Page Replacement

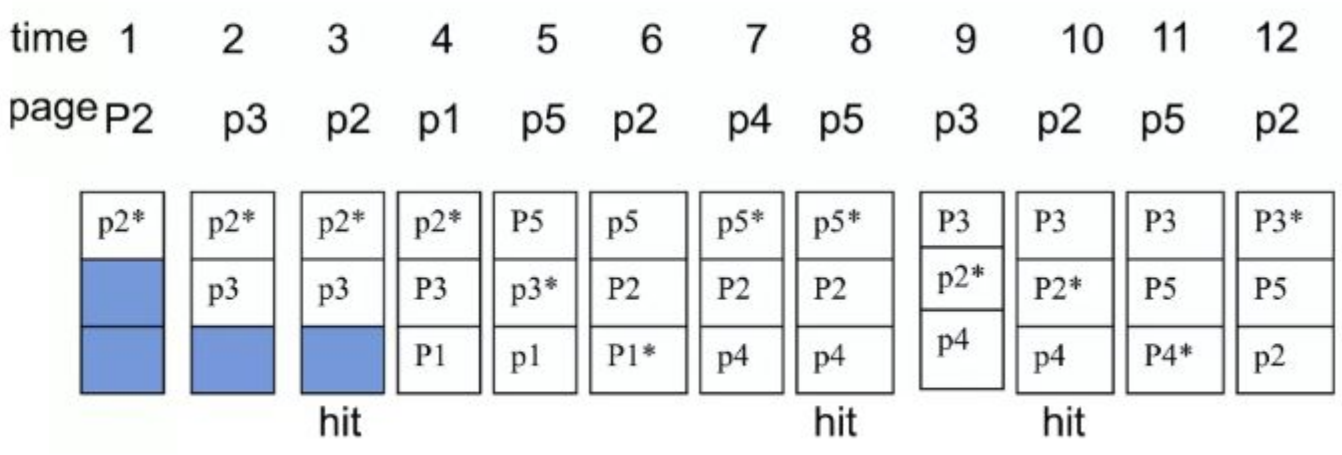
Page Replacement Algorithms:

- ❑ FIFO (First In First Out)
- ❑ LRU (Least Recently Used)
- ❑ OPT (Optimal)



FIFO (First In First Out)

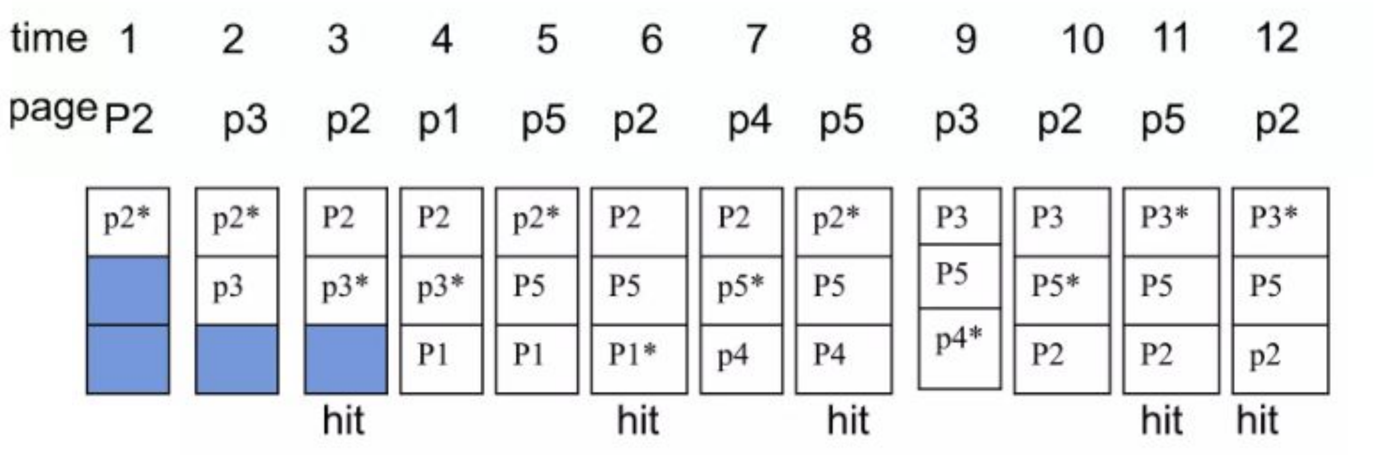
- Selects the page for replacement that has been in the memory for the longest amount of time



time	1	2	3	4	5	6	7	8	9	10	11	12
page	P2	p3	p2	p1	p5	p2	p4	p5	p3	p2	p5	p2
	p2*	p2*	p2*	p2*	P5	p5	p5*	p5*	P3	P3	P3	P3*
	🟦	p3	p3	P3	p3*	P2	P2	P2	p2*	P2*	P5	P5
	🟦	🟦	🟦	P1	p1	P1*	p4	p4	p4	p4	P4*	p2
			hit					hit		hit		

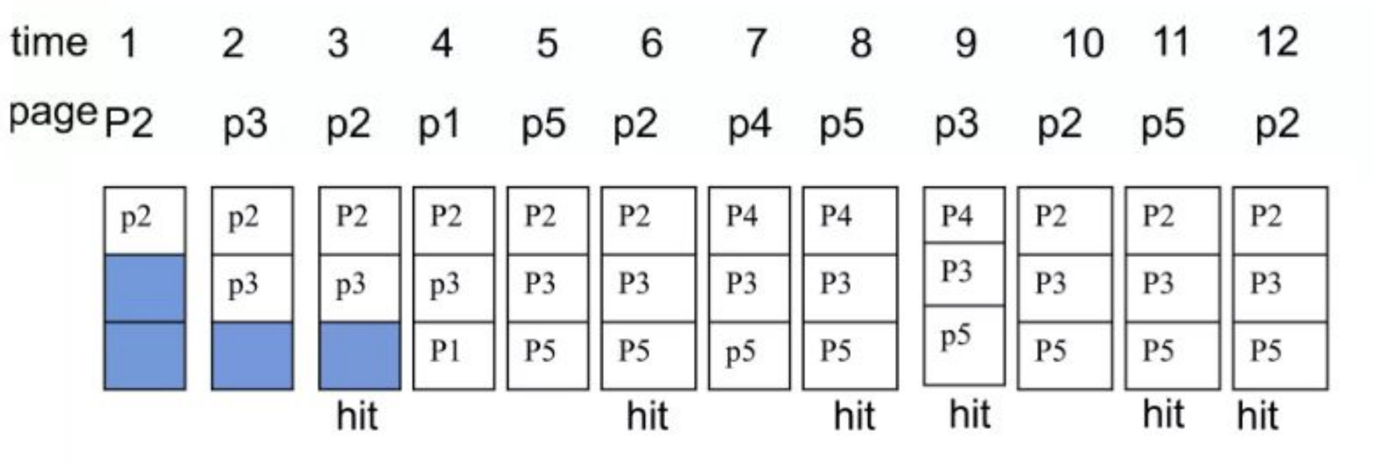
LRU (Least Recently Used)

- ❑ Replace the least recently used page in the past
- ❑ Can be implemented by associating a counter with every page that is in main memory



Optimal

- Replace the page which is not used in longest dimension of time in future



time	1	2	3	4	5	6	7	8	9	10	11	12
page	P2	p3	p2	p1	p5	p2	p4	p5	p3	p2	p5	p2
	p2	p2	P2	P2	P2	P2	P4	P4	P4	P2	P2	P2
	🟦	p3	p3	p3	P3	P3	P3	P3	P3	P3	P3	P3
	🟦	🟦	🟦	P1	P5	P5	p5	P5	p5	P5	P5	P5
			hit			hit		hit	hit		hit	hit