5118020-03 Operating System

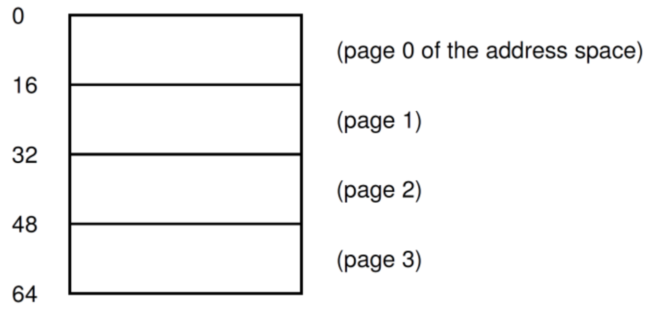
Paging

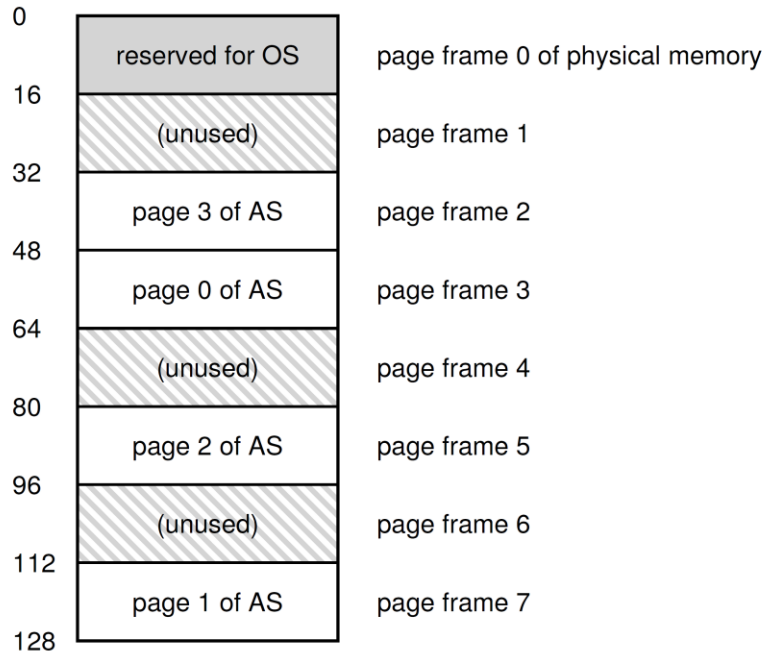
OSTEP Chapters 18 and 19

Shin Hong

Paging

- Paging is to allocate memory to a process as a set of small fixed-size pieces
  - Divide an address space into **pages** and a physical memory space into **frames** such that a page and a frame have the same size
  - Limit both internal and external fragmentation
    - it is empirically found that first-fit loses 33% of memory due to external fragmentation
  - Ex. 16-byte pages with 64 bytes address space and 128 bytes physical memory

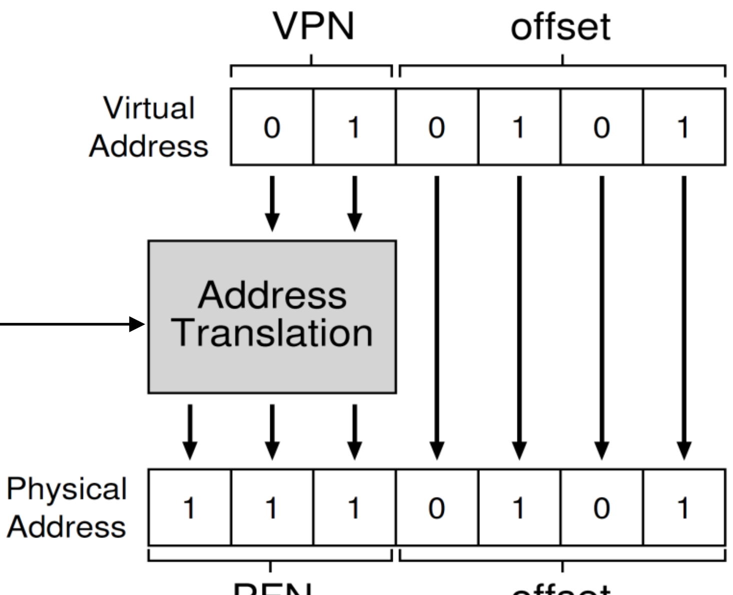




Page Table and Address Translation

- A page table maintains to which frame a page is assigned
  - as per-process page table, or as inverted page table
- In an address translation, a virtual address is split it into a VPN and an offset, and then the VPN is replaced with the corresponding PFN
  - ex. 16-byte page, 64-byte address space & 128-byte physical memory

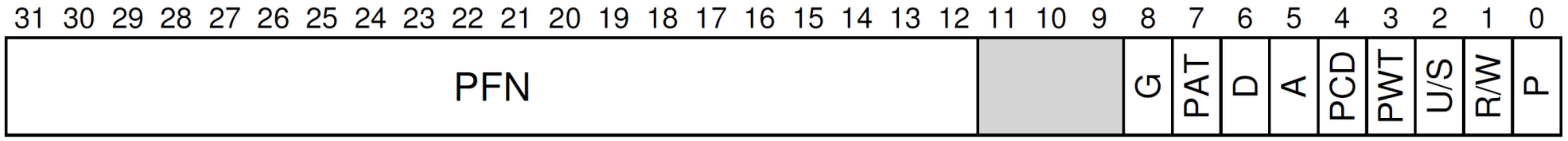
VPN	PFN
00	011
01	111
10	101
11	010



Per-process Page Table Entry

- Physical Frame Number (PFN)
- valid bit: whether the page is given to the process (i.e., in use)
- present bit: whether the page is in physical memory or on disk
- permission bits: read-only, read-write, super mode-only, etc.
- dirty bit: whether the page was modified since it was brought into memory
- reference bit: whether the page has been recently accessed

- Example of 32-bit x86



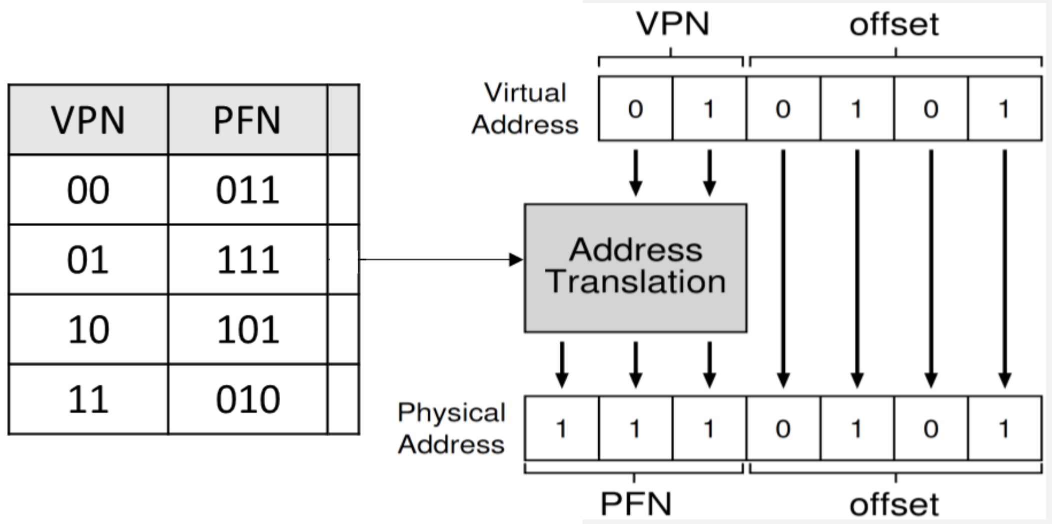
Accessing Memory with Paging

```
// Extract the VPN from the virtual address
VPN = (VirtualAddress & VPN_MASK) >> SHIFT

// Form the address of the page-table entry (PTE)
PTEAddr = PTBR + (VPN * sizeof(PTE))

// Fetch the PTE
PTE = AccessMemory(PTEAddr)

// Check if process can access the page
if (PTE.Valid == False)
    RaiseException(SEGMENTATION_FAULT)
else if (CanAccess(PTE.ProtectBits) == False)
    RaiseException(PROTECTION_FAULT)
else
    // Access is OK: form physical address and fetch it
    offset   = VirtualAddress & OFFSET_MASK
    PhysAddr = (PTE.PFN << PFN_SHIFT) | offset
    Register = AccessMemory(PhysAddr)
```

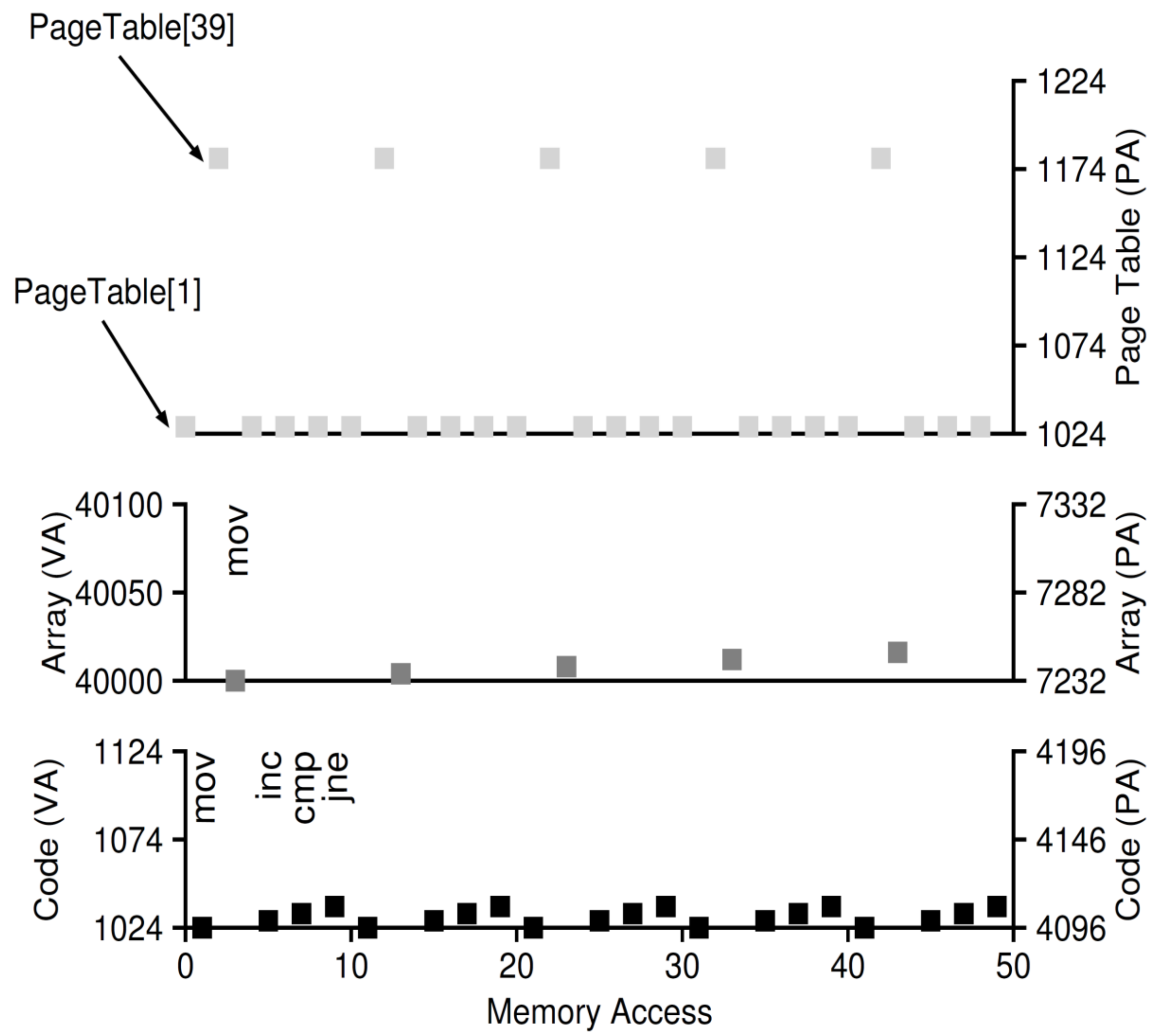


VPN	PFN
00	011
01	111
10	101
11	010

Ex. Memory Trace

```
int array[1000];
...
for (i = 0; i < 1000; i++)
    array[i] = 0;
```

```
1024 movl $0x0,(%edi,%eax,4)
1028 incl %eax
1032 cmpl $0x03e8,%eax
1036 jne  0x1024
```

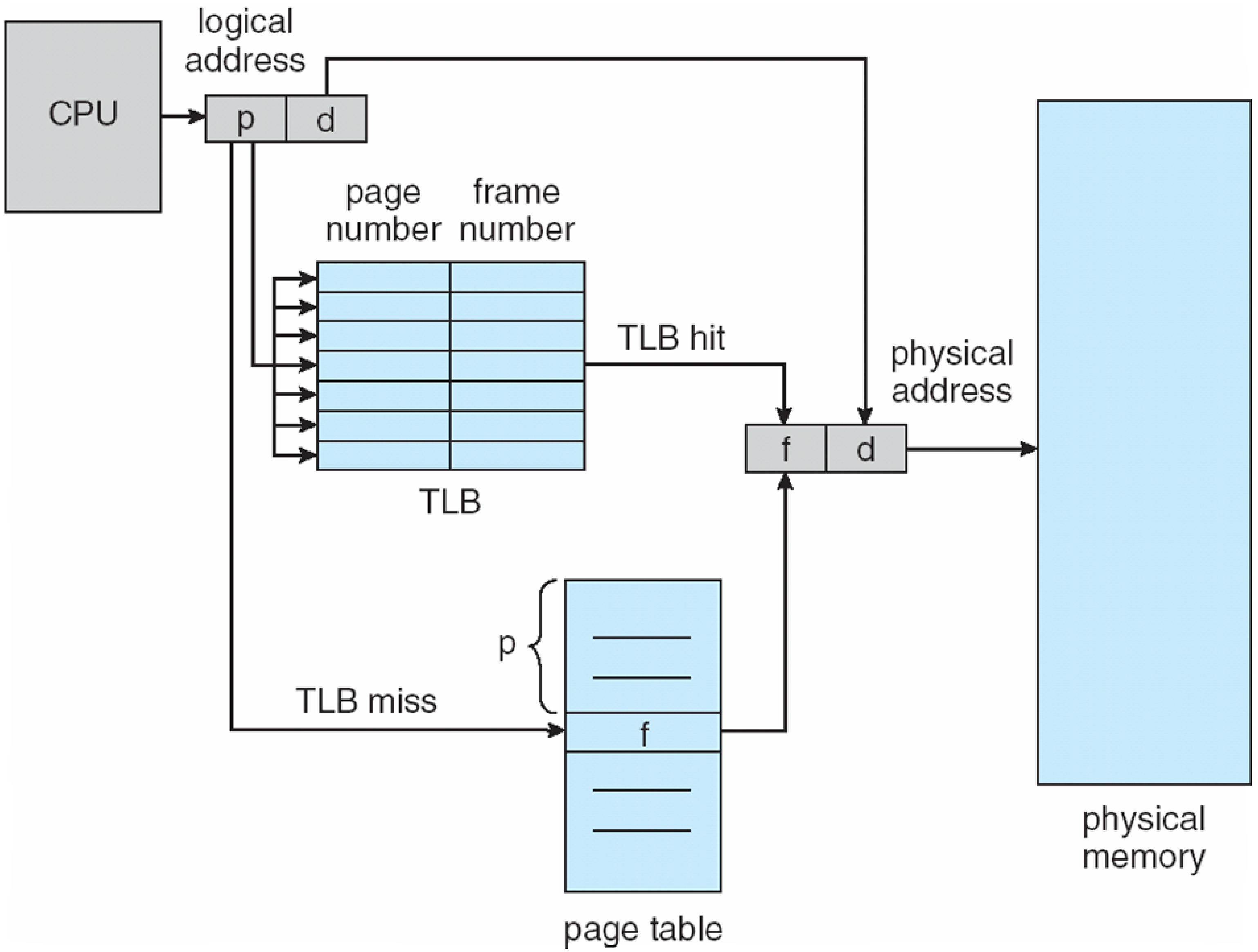


Page Table Size

- page tables take a large amount of memory
  - example
    - 4 KB page/frame in a 32-bit address space and a 4 GB physical mem.
    - 20-bits VPN, thus 2^{20} entries in a page table
    - assume that 4 bytes is needed for each entry, thus, 4 MB (=2^{22} bytes) for a page table
    - 400 MB for 100 processes

- a page table should be resided in a main memory since MMU cannot accommodate a whole page table at a time
  - every memory access incurs at least one extra memory access for address translation

Paging Hardware with TLB

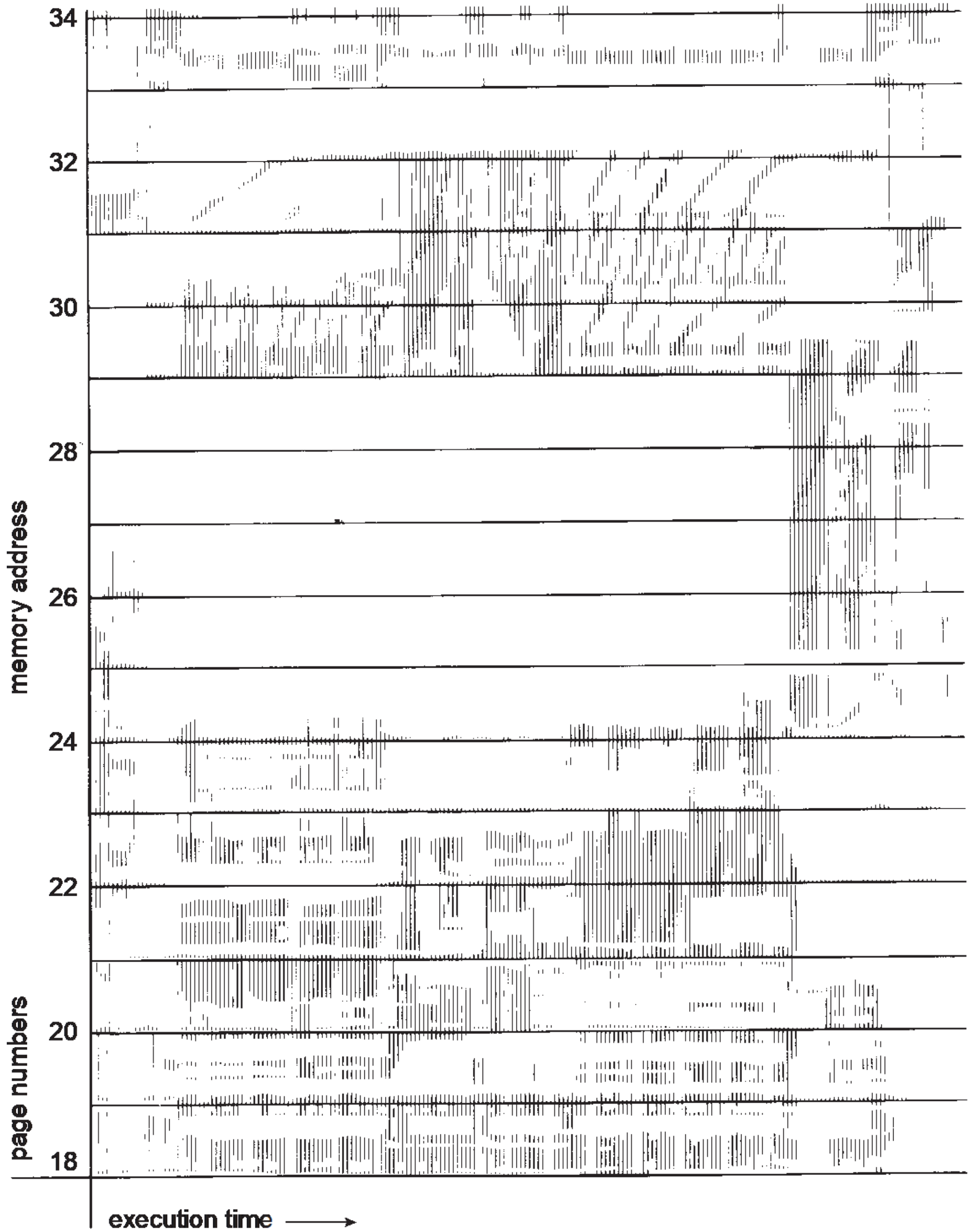


Translation-lookaside Buffer (TLB)

- A TLB is a part of MMU working as a cache of a page table
  - upon a memory access request, the computer architecture first checks the TLB to see if it has page translation information for the corresponding VPN
  - address-translation cache

- Steps for translating a virtual address with TLB
  - extract the VPN from a virtual address
  - check if the entry for the VPN is found in the TLB
    - if there exists, get the PFN from the TLB (i.e., TLB hit)
    - otherwise (i.e., TLB miss)
      - reference the page table to get the PFN
      - update the TLB with the VPN and the PFN
      - repeat from the beginning

Memory Reference Pattern



Locality

- A TLB can save memory accesses because an application program tends to make memory accesses with **temporal locality** and **spatial locality**
  - **temporal locality**: if a program accessed a memory at address *x*, it will likely access *x* again in near future.
  - **spatial locality**: if a program accesses a memory at address *x*, it will likely access *x* + *d* in near future (where *d* is a small number)

- Trade off between TLB size and TLB access speed
  - The bigger the size of a TLB is, the higher the hit ratio is
  - The bigger the size of a TLB is, the longer the look-up time takes

Handling TLB Miss

- Hardware-managed TLB
  - there must be a special register to point to the page table
  - the structure of a page table must be fixed

- Software-managed TLB
  - the architecture raises an exception in a middle of an instruction execution if a TLB miss occurs
  - the OS finds proper information from page tables and updates TLB
  - after the trap handler execution, the architecture retries the instruction

TLB Entry

- A TLB of an architecture typically has 32, 64 or 128 entries
  - parallel search

- A TLB entry typically has the following attributes
  - VPN
  - PFN
  - valid bit
  - protection bit: read-only or read-write
  - dirty bit

Managing TLB at Context Switching

- The existing TLB entries must not be used after a context switching

- Approaches

1. Flush all entries
    - flushing can be made by turning off the valid bit of every entry
    - a series of TLB misses will happen right after a context switch

2. Have a process identifier in a TLB entry
    - address space identifier (ASID)
    - multiple processes can readily share a TLB

TLB Entry Replacement Policies

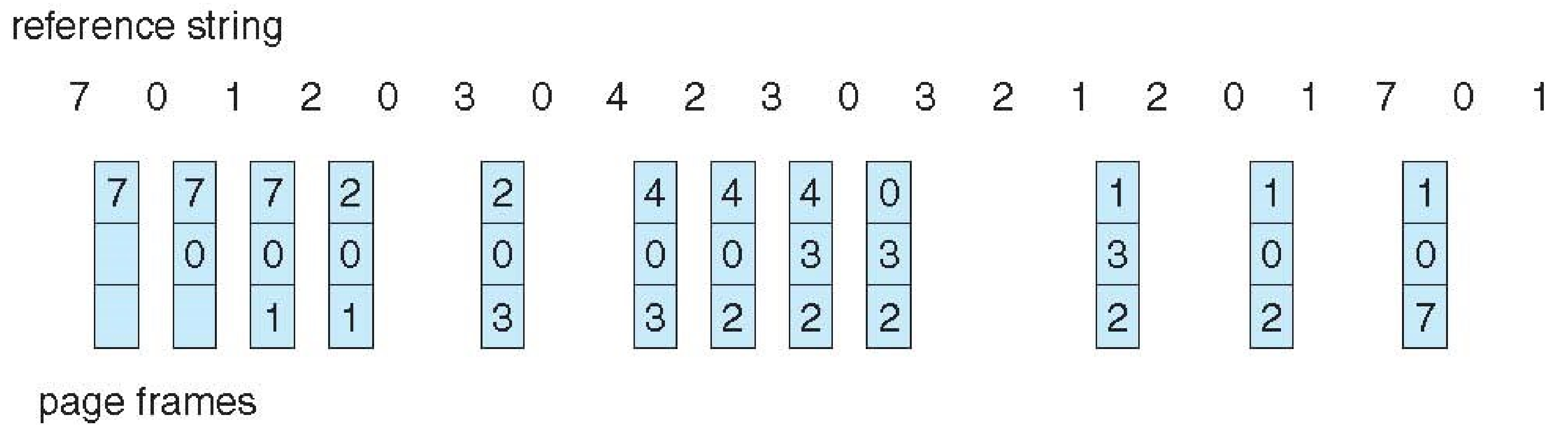
- Which entry to evict from a full TLB when a new entry must be installed?
  - decision should be made to minimize a TLB miss rate
  - a typical case of the cache replacement problem

- Approaches

1. evict the least-recently-used (LRU) one
2. evict a random entry

Least Recently Used (LRU) Algorithm

- Use past knowledge to predict future
- Associate time of last use with each page
- Choose one that has not been used in the most amount of time


reference string
7 0 1 2 0 3 0 4 2 3 0 3 2 1 2 0 1 7 0 1
7 | 7 0 | 7 0 1 | 2 0 1 | 2 0 3 | 4 0 3 | 4 0 2 | 4 3 2 | 0 3 2 | 1 3 2 | 1 0 2 | 1 0 7
page frames