Final Review 03

Operating Systems

Wenbo Shen

Acknowledgement: Some slides are based on Yajin Zhou@ZJU slides.

Summary

- Computer architecture
- OS introduction
- OS structures
- Processes
- IPC
- Thread
- Scheduling
- Synchronization
- Deadlock

Summary

- Memory – segmentation
- Memory – paging
- Virtual memory
- Virtual memory – Linux
- Mass storage
- IO
- FS interface
- FS implementation
- FS in practice

08: Deadlock

The Deadlock Problem

- **Deadlock**: a set of blocked processes each holding a resource and waiting to acquire a resource held by another process in the set
- Examples:
  - a system has 2 disk drives, P_1 and P_2 each hold one disk drive and each needs another one
  - semaphores A and B, initialized to 1

P_1	P_2
wait (A);	wait(B)
wait (B);	wait(A)

Deadlock in program

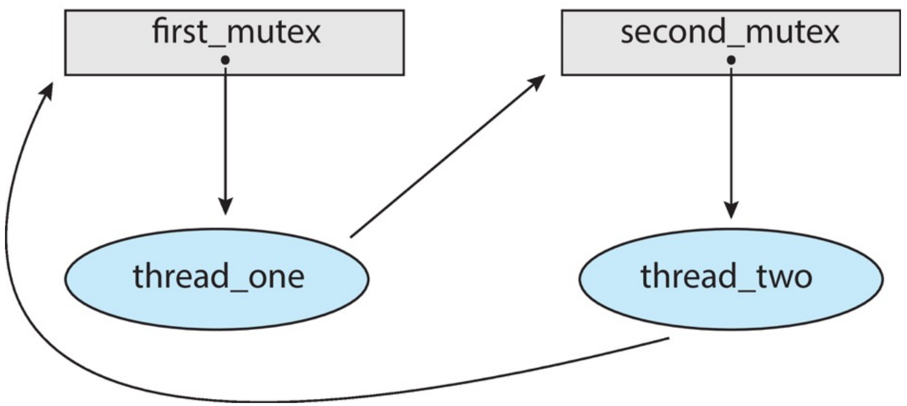
- Two mutex locks are created an initialized:

```
pthread_mutex_t first_mutex;
pthread_mutex_t second_mutex;

pthread_mutex_init(&first_mutex, NULL);
pthread_mutex_init(&second_mutex, NULL);
```

```
/* thread one runs in this function */
void *do_work_one(void *param){
    pthread_mutex_lock(&first_mutex);
    pthread_mutex_lock(&second_mutex);

    /* Do some work*/

    pthread_mutex_unlock(&second_mutex);
    pthread_mutex_unlock(&first_mutex);
    pthread_exit(0);
}

/* thread two runs in this function */
void *do_work_two(void *param){
    pthread_mutex_lock(&second_mutex);
    pthread_mutex_lock(&first_mutex);

    /* Do some work*/

    pthread_mutex_unlock(&first_mutex);
    pthread_mutex_unlock(&second_mutex);

    pthread_exit(0);
}
```

Deadlock in program

- Deadlock is possible if thread 1 acquires **first_mutex** and thread 2 acquires **second_mutex**. Thread 1 then waits for **second_mutex** and thread 2 waits for **first_mutex**.
- Can be illustrated with a **resource allocation graph**:



Four Conditions of Deadlock

- **Mutual exclusion**: only one process at a time can use a resource
- **Hold and wait**: a process holding at least one resource is waiting to acquire additional resources held by other processes
- **No preemption**: a resource can be released only voluntarily by the process holding it, after it has completed its task
- **Circular wait**: there exists a set of waiting processes $\{P_0, P_1, \ldots, P_n\}$
  - P_0 is waiting for a resource that is held by P_1
  - P_1 is waiting for a resource that is held by P_2 …
  - P_{n-1} is waiting for a resource that is held by P_n
  - P_n is waiting for a resource that is held by P_0

How to Handle Deadlocks

- Ensure that the system will never enter a deadlock state
  - **Prevention**
  - **Avoidance**
- Allow the system to enter a deadlock state and then recover - database
  - **Deadlock detection and recovery**:
- **Ignore the problem** and pretend deadlocks never occur in the system
  - Usually adopted by **mainstream operating systems**

Deadlock Prevention

- How to prevent **mutual exclusion**
  - Not required for sharable resources
  - Must hold for non-sharable resources
- How to prevent **hold and wait**
  - Whenever a process requests a resource, it doesn't hold any other resources
    - Require process to request ***all*** its resources before it begins execution
    - Allow process to request resources only when the process has none
  - Low resource utilization; starvation possible

Deadlock Prevention

- How to handle **no preemption**
  - if a process requests a resource not available
    - release all resources currently being held
    - preempted resources are added to the list of resources it waits for
    - process will be restarted only when it can get all waiting resources
- How to handle **circular wait**
  - **impose a total ordering of all resource types**
  - require that each process requests resources in an increasing order
  - **Many operating systems adopt this strategy for some locks.**

Deadlock Avoidance

- **Dead avoidance: require extra information about how resources are to be requested**
  - **Is this requirement practical**?
- Each process declares a **max** number of resources it may need
- Deadlock-avoidance algorithm ensure there can **never be a circular-wait condition**
- Resource-allocation state:
  - the number of **available** and **allocated** resources
  - the **maximum demands** of the processes

Deadlock Avoidance Algorithms

- Single instance of each resource type ➠ use **resource-allocation graph**
- Multiple instances of a resource type ➠ use the **banker's algorithm**
- Allocate based on safe state
  - Limit the order of resource application

Banker’s Algorithm

- Banker’s algorithm is for **multiple-instance resource deadlock avoidance**
  - each process must claim **maximum** use of each resource type **in advance**
  - when a process requests a resource it may have to wait
  - when a process gets all its resources it must release them in a finite amount of time

Data Structures for the Banker's Algorithm

- n processes, m types of resources
  - available: an array of length m, instances of available resource
    - available[j] = k: k instances of resource type R_j available
  - max: a n x m matrix
    - max [i,j] = k: process P_i may request at most k instances of resource R_j
  - allocation: n x m matrix
    - allocation[i,j] = k: P_i is currently allocated k instances of R_j
  - need: n x m matrix
    - need[i,j] = k: P_i may need k more instances of R_j to complete its task
    - **need [i,j] = max[i,j] – allocation [i,j]**

Banker's Algorithm: Example

- System state:
  - **5 processes** P_0 through P_4
  - **3 resource types**: A (10 instances), B (5 instances), and C (7 instances)
- Snapshot at time T_0:

	allocation	max	available
	A B C	A B C	A B C
P_0	0 1 0	7 5 3	3 3 2
P_1	2 0 0	3 2 2	
P_2	3 0 2	9 0 2	
P_3	2 1 1	2 2 2	
P_4	0 0 2	4 3 3	

Banker's Algorithm: **Safe State**

- Data structure to compute whether the system is in a safe state
  - use **work** (a vector of length *m*) to track **allocable resources**
    - **current available resources**
    - **unallocated** + **released by finished processes**
  - use **finish** (a vector of length n) to track whether process has finished
  - initialize: **work** = **available**, **finish[i]** = **false** for i = 0, 1, …, n- 1
- Algorithm:
  - find an i such that **finish[i] = false && need[i] ≤ work**
    - if no such i exists, go to step 3
  - **work = work + allocation[i]**, **finish[i] = true**, go to step 1
  - if finish[i] == true for all i, then the system is in a safe state

Bank's Algorithm: **Resource Allocation**

- Data structure: request vector for process P_i
  - **request**[j] = k then process P_i wants k instances of resource type R_j
- Algorithm:

1. if request[i]≤ need[i] go to step 2; otherwise, raise error condition (the process has exceeded its maximum claim)
2. if request[i] ≤ available, go to step 3; otherwise P_i must wait (not all resources are not available)
3. pretend to allocate requested resources to P_i by modifying the state:

   available = available – request[i]

   allocation[i] = allocation[i] + request[i]

   need[i] = need[i] – request[i]
4. use **previous algorithm** to test if it is a safe state, if so ➠ allocate the resources to P_i
5. if unsafe ➠ P_i must wait, and the old resource-allocation state is restored

Banker's Algorithm: Example

- System state:
  - **5 processes** P_0 through P_4
  - **3 resource types**: A (10 instances), B (5 instances), and C (7 instances)
- Snapshot at time T_0:

	allocation	max
	A B C	A B C
P_0	0 1 0	7 5 3
P_1	2 0 0	3 2 2
P_2	3 0 2	9 0 2
P_3	2 1 1	2 2 2
P_4	0 0 2	4 3 3

available
A B C
3 3 2

Banker's Algorithm: Example

- System state:
  - **5 processes** P_0 through P_4
  - **3 resource types**: A (10 instances), B (5 instances), and C (7 instances)
- need = max – allocation

	allocation	max	need
	A B C	A B C	A B C
P_0	0 1 0	7 5 3	7 4 3
P_1	2 0 0	3 2 2	1 2 2
P_2	3 0 2	9 0 2	6 0 0
P_3	2 1 1	2 2 2	0 1 1
P_4	0 0 2	4 3 3	4 3 1

available
A B C
3 3 2

Banker's Algorithm: Example

- System state:
  - **5 processes** P_0 through P_4
  - **3 resource types**: A (10 instances), B (5 instances), and C (7 instances)
- need = max – allocation

	allocation	max	need
	A B C	A B C	A B C
P_0	0 1 0	7 5 3	7 4 3
P_1	2 0 0	3 2 2	1 2 2
P_2	3 0 2	9 0 2	6 0 0
P_3	2 1 1	2 2 2	0 1 1
P_4	0 0 2	4 3 3	4 3 1

available
A B C
3 3 2

- First one can be either P1 or P3

Banker's Algorithm: Example

- What's the safe sequence

	allocation	max	need
	A B C	A B C	A B C
P_0	0 1 0	7 5 3	7 4 3
P_1	2 0 0	3 2 2	1 2 2
P_2	3 0 2	9 0 2	6 0 0
P_3	2 1 1	2 2 2	0 1 1
P_4	0 0 2	4 3 3	4 3 1

available
A B C
3 3 2

- finish[1] = true, needed[1] < work -> work = work + allocation = [5 3 2]
- finish[3] = true, needed[3]< work -> work = work + allocation = [7 4 3]
- finish[4] = true, needed[4] < work -> work = work + allocation = [7 4 5]
- finish[2] = true, needed[2] < work -> work = work + allocation = [10 4 7]
- finish[0] = true, needed[0] < work -> work = work + allocation = [10 5 7]

Banker's Algorithm: Example

- P1 requires and gets allocated 1 0 2 more

	allocation	max	need
	A B C	A B C	A B C
P_0	0 1 0	7 5 3	7 4 3
P_1	3 0 2	4 2 4	1 2 2
P_2	3 0 2	9 0 2	6 0 0
P_3	2 1 1	2 2 2	0 1 1
P_4	0 0 2	4 3 3	4 3 1

available
A B C
2 3 0

- Check whether it is in safe state?
  - We cannot find a process that the need[i] < work[i]

Deadlock Detection

- Allow system to enter deadlock state, but detect and recover from it
- Detection algorithm and recovery scheme

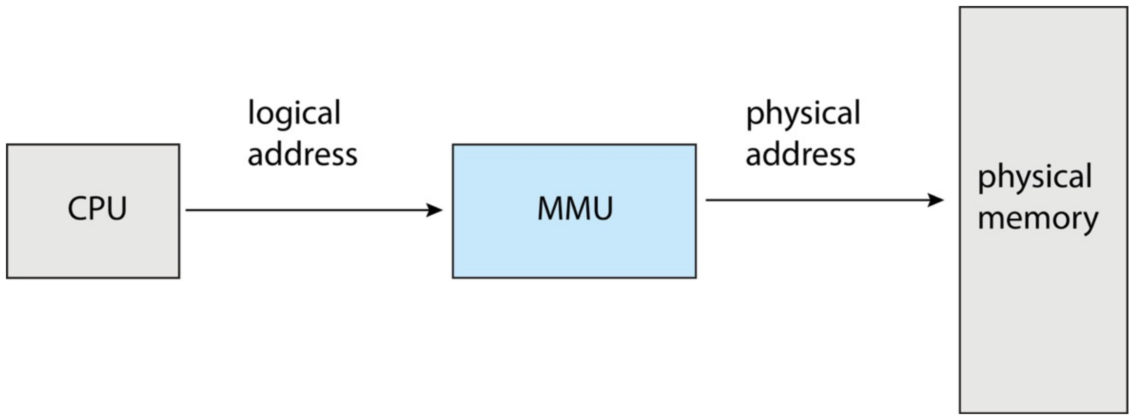
09: Main Memory

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 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

Memory-Management Unit (MMU)

- Hardware device that at run time maps virtual to physical address



Contiguous Allocation

- Main memory must support both OS and user processes
- Limited resource, must allocate efficiently
- Contiguous allocation is one early method
- Main memory usually into two **partitions**:
  - Resident operating system, usually held in low memory with interrupt vector
  - User processes then held in high memory
    - Each process contained in single contiguous section of memory

Memory Allocation Strategies

- Fixed partitions
- Variable partitions

Partitioning Strategies – Fixed

- Fixed Partitions – divide memory into equal sized pieces (except for OS)
  - Degree of multiprogramming = number of partitions
  - Simple policy to implement
    - All processes must fit into partition space
    - Find any free partition and load process
- Problem – Internal Fragmentation
  - Unused memory in partition not available to other processes

Question:
What is the “right” partition size?

Partitioning Strategies – Variable

- Memory is dynamically divided into partitions based on process needs
  - More complex management problem
    - Need data structures to track free and used memory
    - New process allocated memory from hole large enough to fit it
- Problem – External Fragmentation
  - Unused memory between partitions too small to be used by any processes

Memory Allocation

- How to satisfy a request of size *n* from a list of free memory blocks?
  - **first-fit**: allocate from the first block that is big enough
  - **best-fit**: allocate from the smallest block that is big enough
    - must search entire list, unless ordered by size
    - produces the smallest leftover hole
  - **worst-fit**: allocate from the largest hole
    - must also search entire list
    - produces the largest leftover hole
- **Fragmentation** is big problem for all three methods
  - first-fit and best-fit usually perform better than worst-fit

Fragmentation

- **External fragmentation**
  - Unusable memory between allocated memory blocks
    - **Total amount** of free memory space **is larger than a request**
    - The request cannot be fulfilled because the free memory is not **contiguous**
  - External fragmentation can be reduced by **compaction**
    - Shuffle memory contents to place all free memory in one large block
    - Program needs to be **relocatable** at runtime
    - Performance overhead, timing to do this operation
  - Another solution: **paging**

Fragmentation

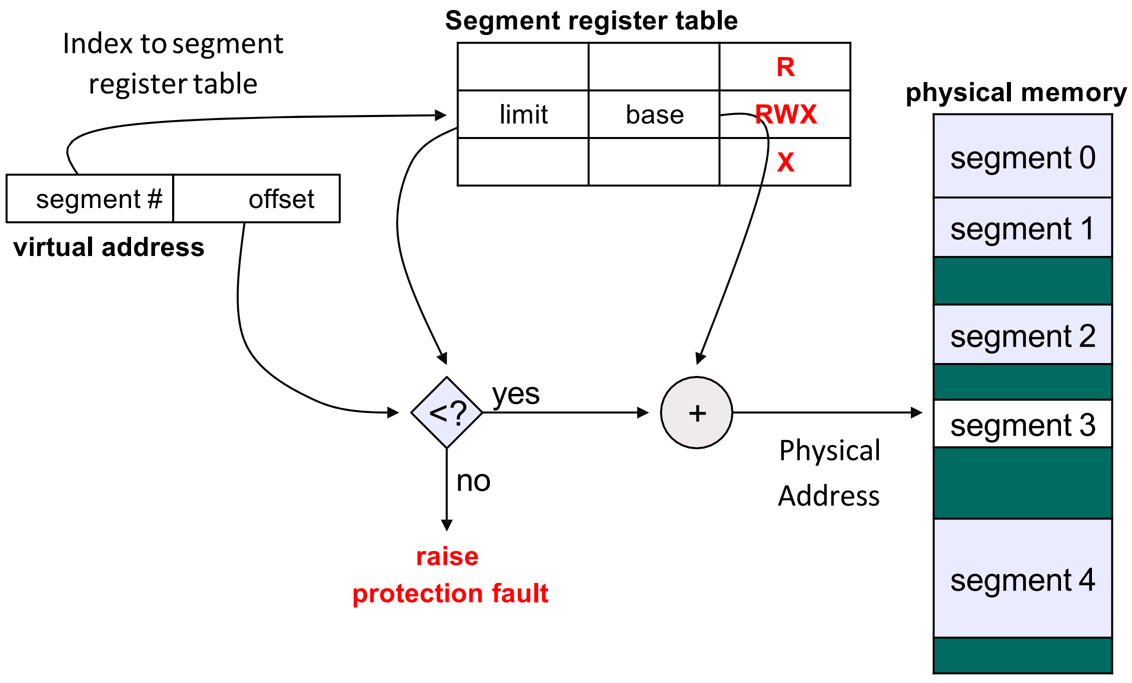
- **Internal fragmentation**
  - memory allocated may be larger than the requested size
  - this size difference is memory *internal to a partition*, but not being used
  - Example: free space 18464 bytes, request 18462 bytes
- Sophisticated algorithms are designed to avoid fragmentation
  - none of the first-/best-/worst-fit can be considered sophisticated

Segmentation

- Logical address consists of a pair:
  - <segment-number, offset>

- Segment table where each entry has:
  - Base: starting physical address
  - Limit: length of segment

Segment Lookup



Address Binding

- Addresses are represented in different ways at different stages of a program's life
  - **source code** addresses are usually **symbolic** (e.g., variable name)
  - **compiler** binds symbols to **relocatable addresses**
    - e.g., "14 bytes from beginning of this module"
  - **linker** (or loader) binds relocatable addresses to **absolute addresses**
    - e.g., 0x0e74014
  - Each binding maps one address space to another

Binding of Instructions and Data to Memory

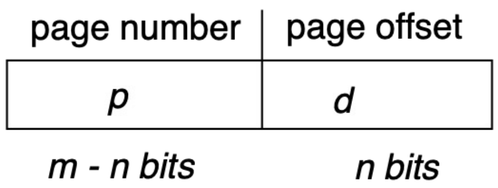
- Address binding of instructions and data to memory addresses can happen at three different stages
  - **Compile time**: If memory location known a priori, **absolute code** can be generated; must recompile code if starting location changes
  - **Load time**: Must generate relocatable code if memory 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
    - Need **hardware support** for address maps (e.g., base and limit registers)

Paging

- Physical address space of a process can be **noncontiguous**; process is allocated physical memory whenever the latter is available
  - Avoids **external fragmentation -> avoid for compacting**
  - Avoids problem of varying sized memory chunks
- Basic methods
  - 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, where?**

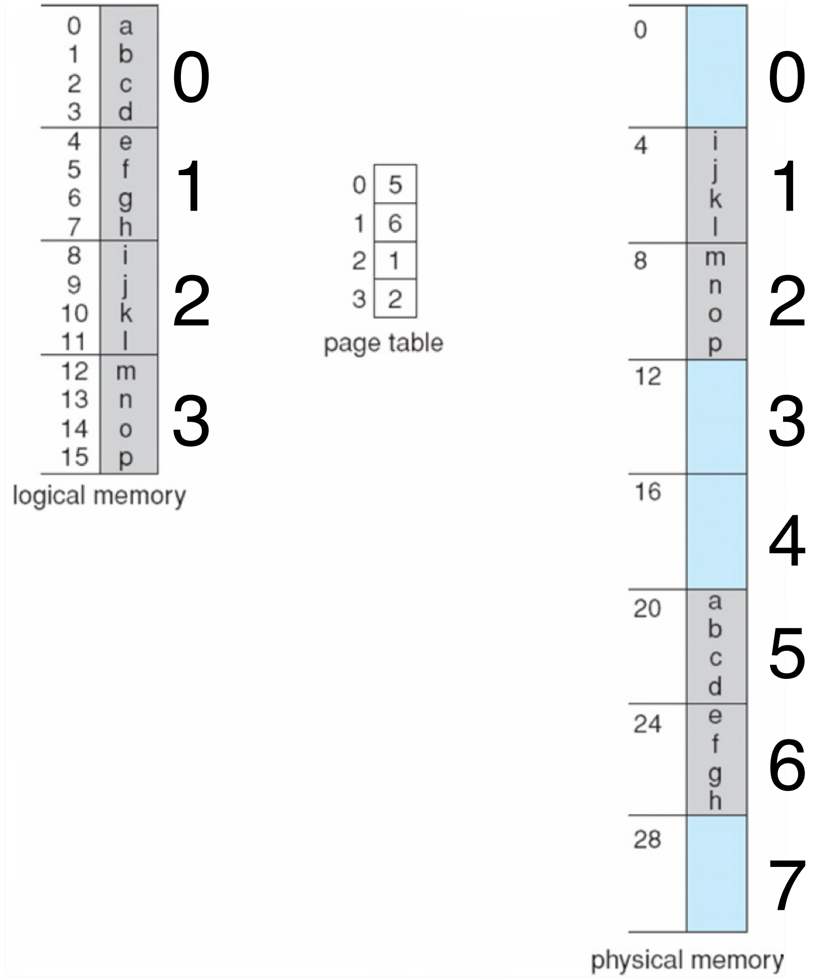
Paging: Address Translation

- A logical address is divided into:
  - **page number** (p)
    - used as an index into a page table
    - page table entry contains the corresponding **physical** frame number
  - **page offset** (d)
    - offset within the page/frame
    - combined with frame number to get the physical address



m bit logical address space, *n* bit page size

Paging Example



$m = 4$ and $n = 2$ 32-byte memory and 4-byte pages

TLB

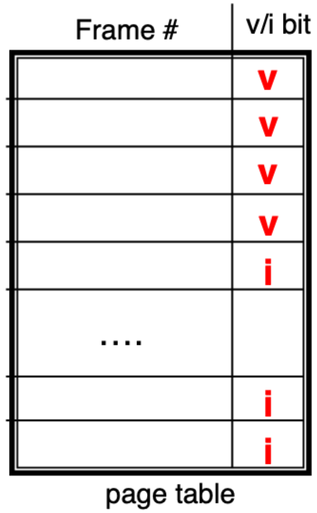
- TLB and context switch
  - Each process has its own page table
    - Switching process needs to switch page table
  - **TLB must be consistent with page table**
    - TLB entries are from page table of current process
    - Option I: flush TLB at every context switch, **or**,
    - Option II: tag TLB entries with **address-space identifier** (ASID) that uniquely identifies a process
  - Some TLB entries can be **shared** by processes, and fixed in the TLB
    - E.G., TLB entries for the kernel
- TLB and operating system
  - MIPS: OS should deal with TLB miss exception
  - X86: TLB miss is handled by hardware

Effective Access Time

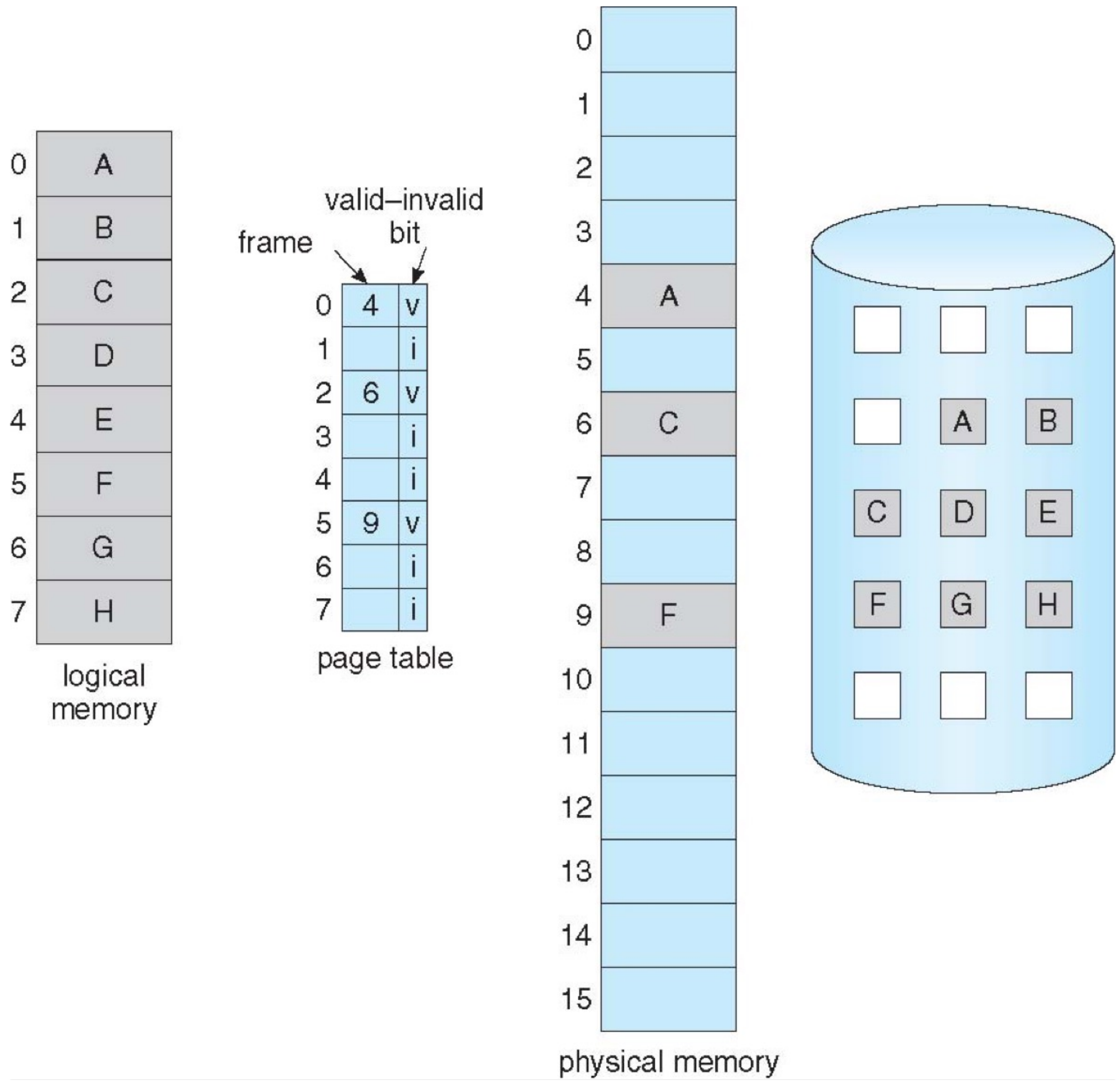
- **Hit ratio** – percentage of times that a page number is found in the TLB
- An 80% hit ratio means that we find the desired page number in the TLB 80% of the time.
- Suppose that 10 nanoseconds to access memory.
  - If we find the desired page in TLB then a mapped-memory access take 10 ns
  - Otherwise we need **two memory access** so it is 20 ns: page table + memory access
- Effective Access Time (EAT)
- EAT = 0.80 x 10 + 0.20 x 20 = 12 nanoseconds
- Implying 20% slowdown in access time
- What if TLB access time is non-zero
  - For example, TLB access time is 2 ns, memory access is 10 ns, hit ratio is p
    - With TLB: total time = p*(2+10)+(1-p)*(2+20)
    - Without TLB: 20

Valid-Invalid Bit

- Each page table entry has a valid–invalid (present) bit
  - *V* ➠ in memory (memory is resident), *I* ➠ not-in-memory
  - initially, valid–invalid bit is set to *i* on all entries
  - during address translation, if the entry is invalid, it will trigger a **page fault**
- Example of a page table snapshot:



Page Table (Some Pages Are Not in Memory)

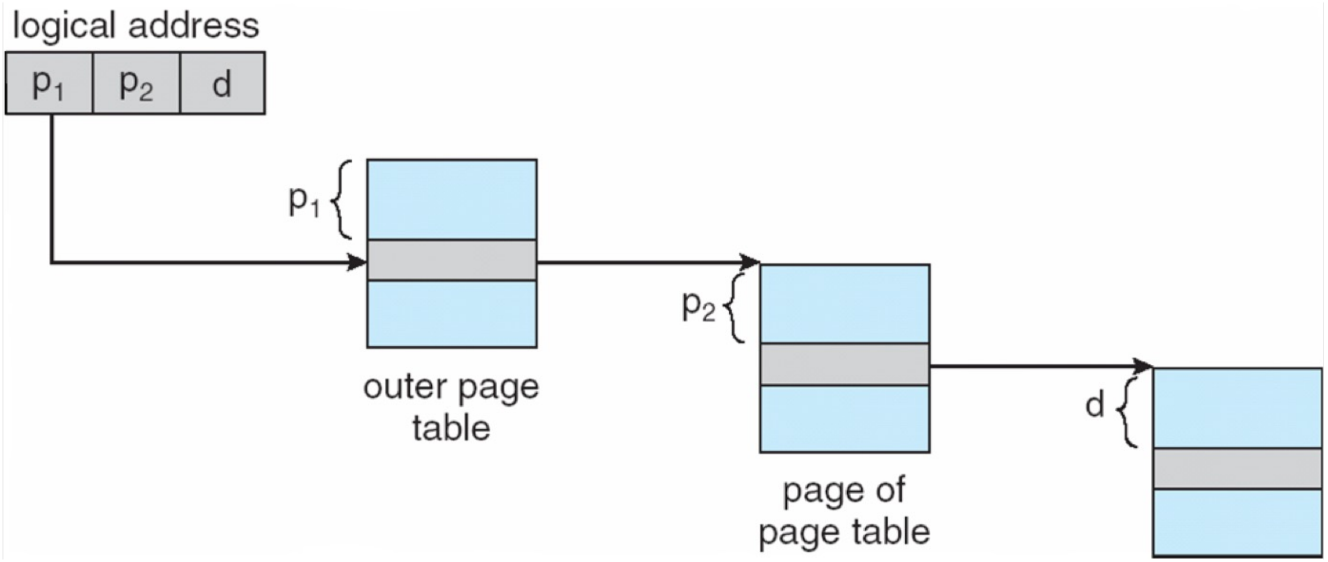


Two-Level Paging

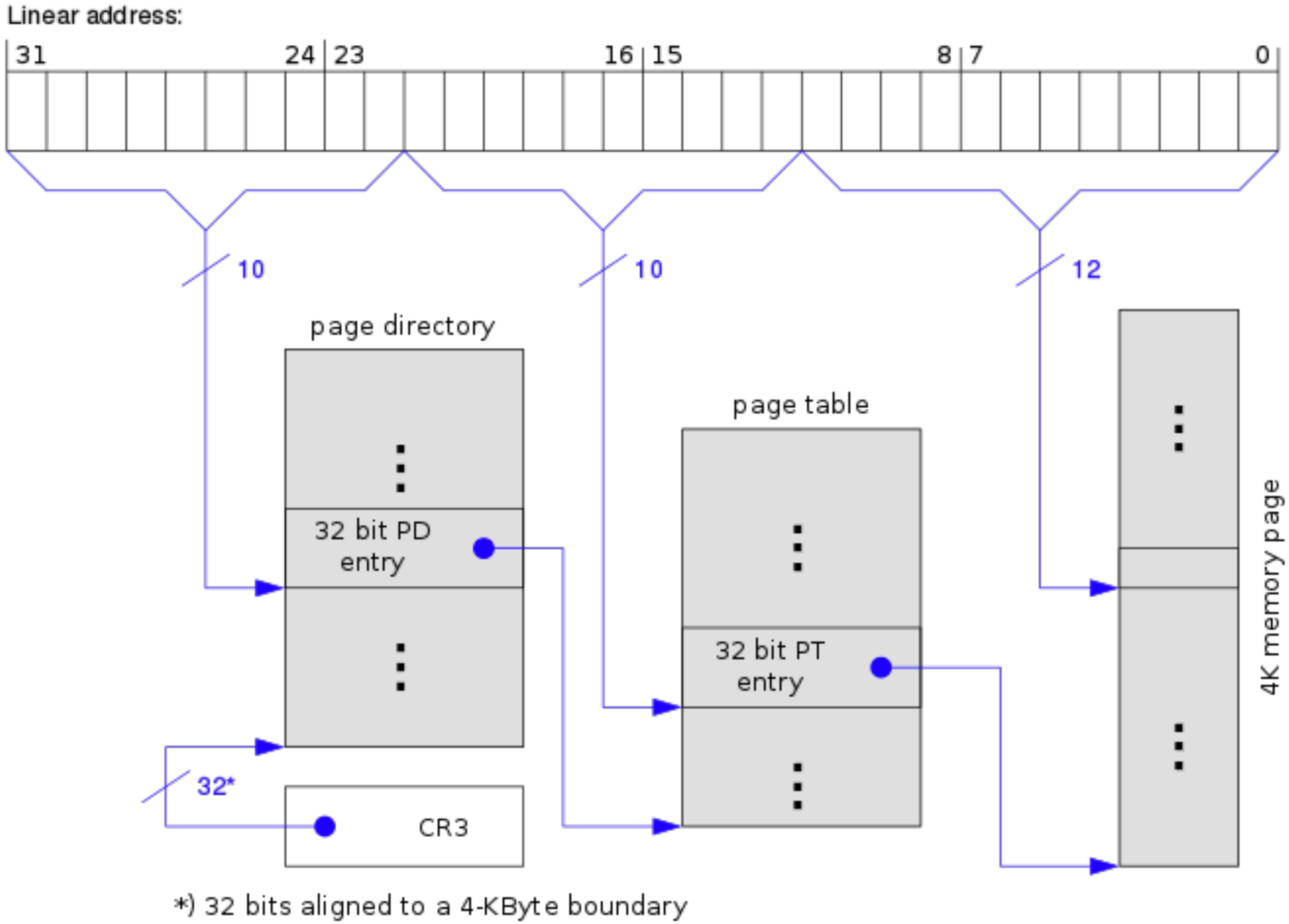
- A logical address is divided into:
  - A **page directory number** (first level page table)
  - A **page table number** (2nd level page table)
  - A **page offset**
- Example: 2-level paging in 32-bit intel cpus
  - 32-bit address space, 4KB page size
  - 10-bit page directory number, 10-bit page table number
  - Each page table entry is 4 bytes, one frame contains 1024 entries (2^{10})

p_1	p_2	d
10	10	12

Address-Translation Scheme



Page Table in Linux

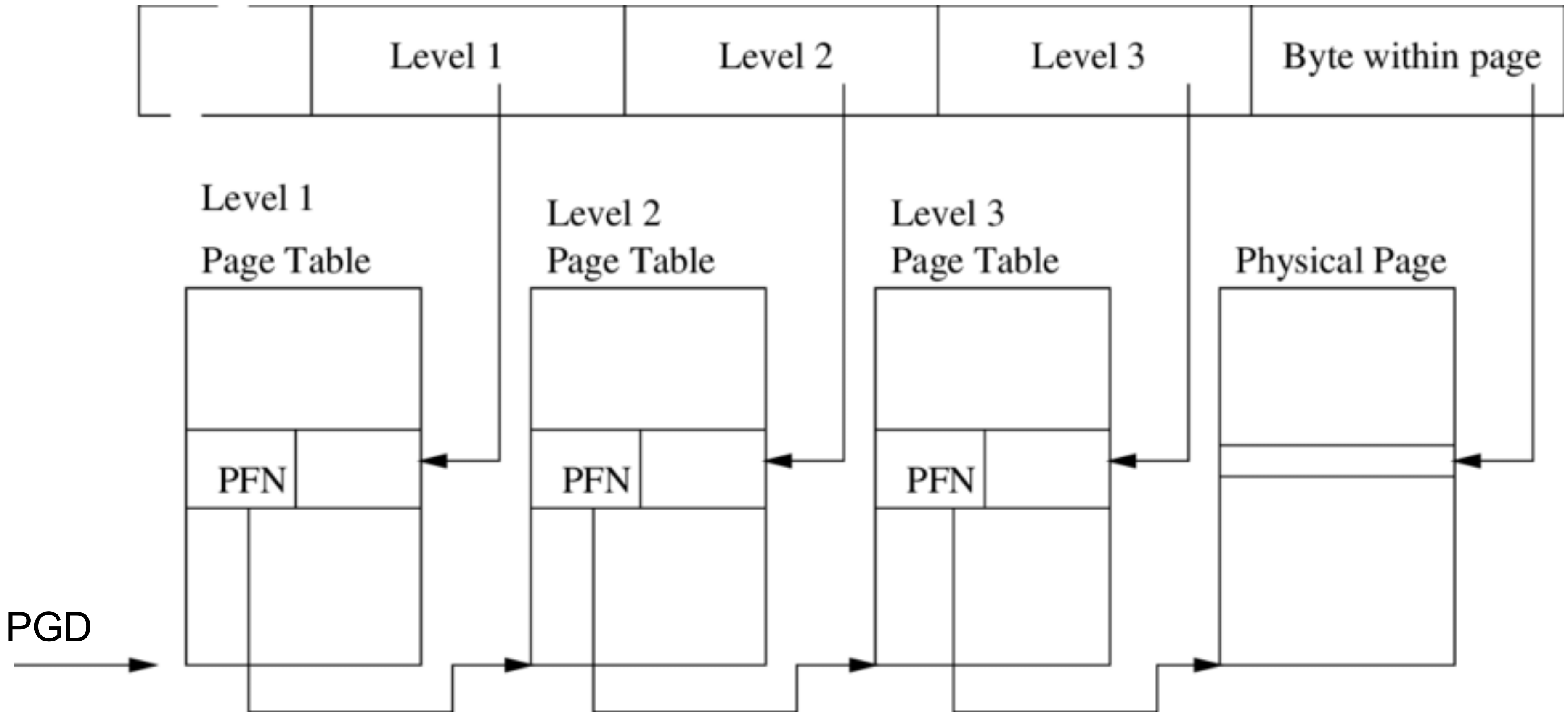


64-bit Logical Address Space

- 64-bit logical address space requires more levels of paging
  - two-level paging is not sufficient for 64-bit logical address space
    - if page size is 4 KB (2^{12}), outer page table has 2^{42} entries, inner page tables have 2^{10} 4-byte entries
  - one solution is to add more levels of page tables
    - e.g., three levels of paging: 1st level page table is 2^{32} bytes in size
    - and possibly 4 memory accesses to get to one physical memory location
  - usually **not support full 64-bit virtual address space**
    - AMD-64 supports 48-bit
    - ARM64 supports 39-bit, 48-bit

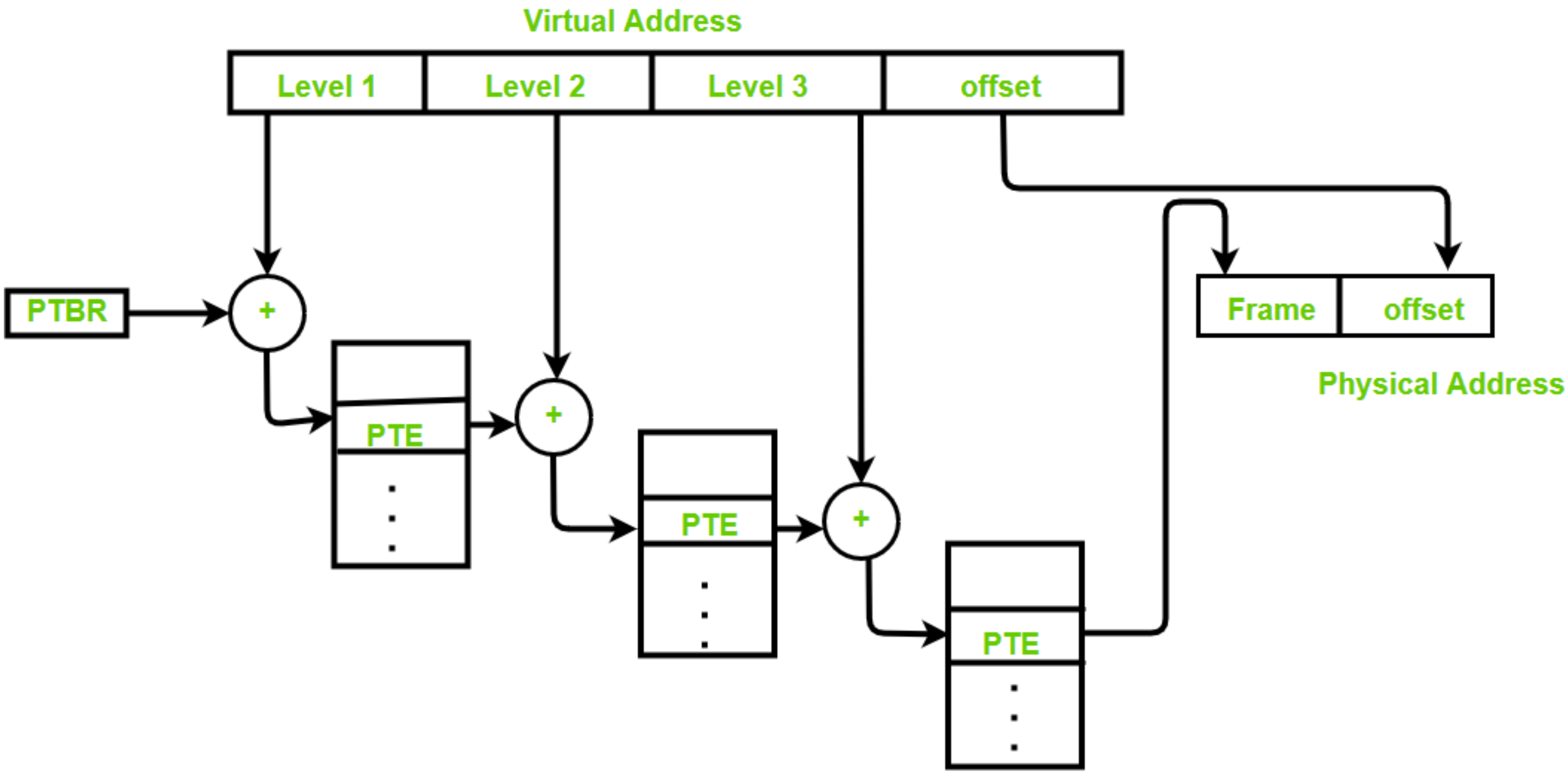
64-bit Logical Address Space

- ARM64: 39 bits = 9+9+9+12



64-bit Logical Address Space

- ARM64: 39 bits = 9+9+9+12

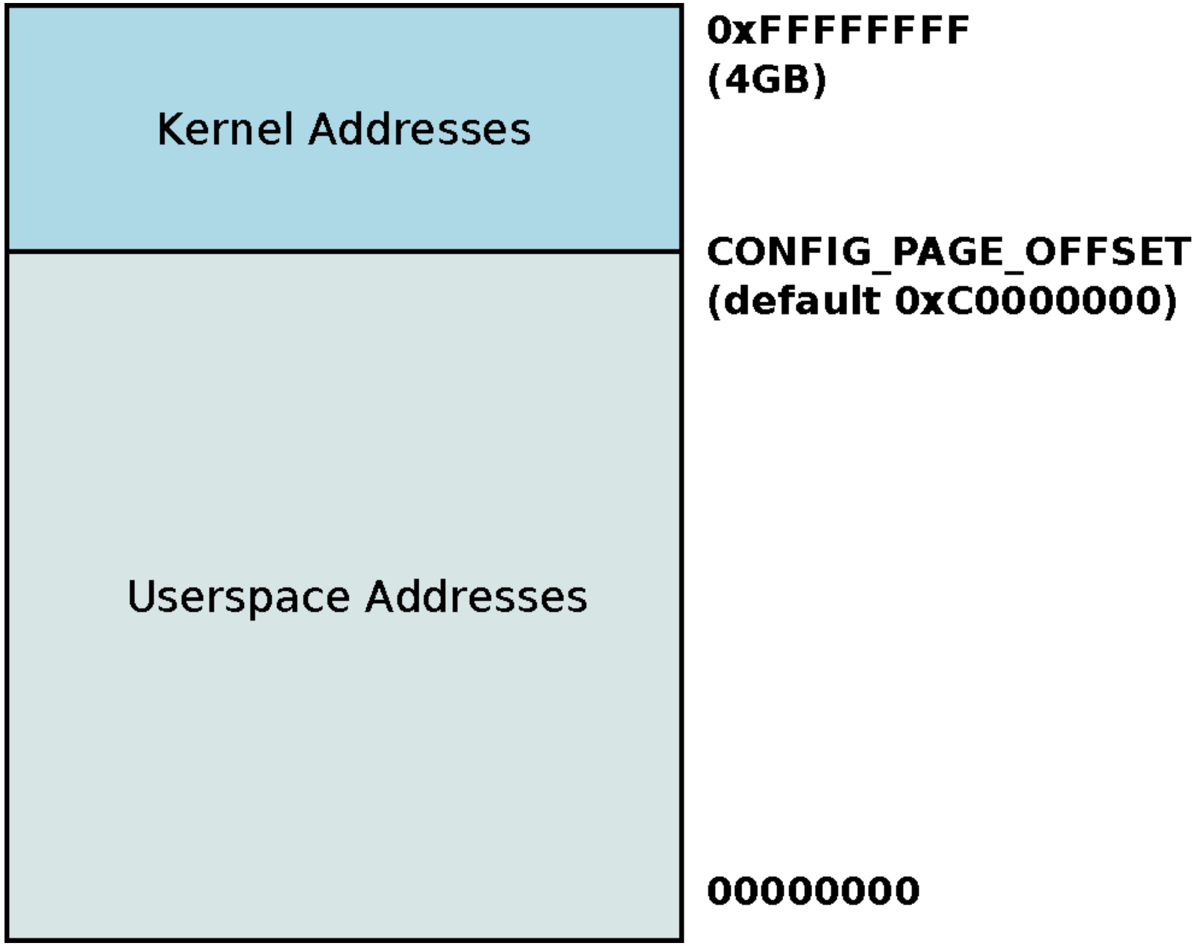


3 Level paging system

Memory Protection

- Accomplished by protection bits with each frame
- Each page table entry has a **present** (aka. valid) bit
  - present: the page has a valid physical frame, thus can be accessed
- Each page table entry contains some protection bits
  - **kernel/user**, **read/write**, **execution**?, **kernel-execution**?
  - why do we need them?
- Any violations of memory protection result in a trap to the kernel

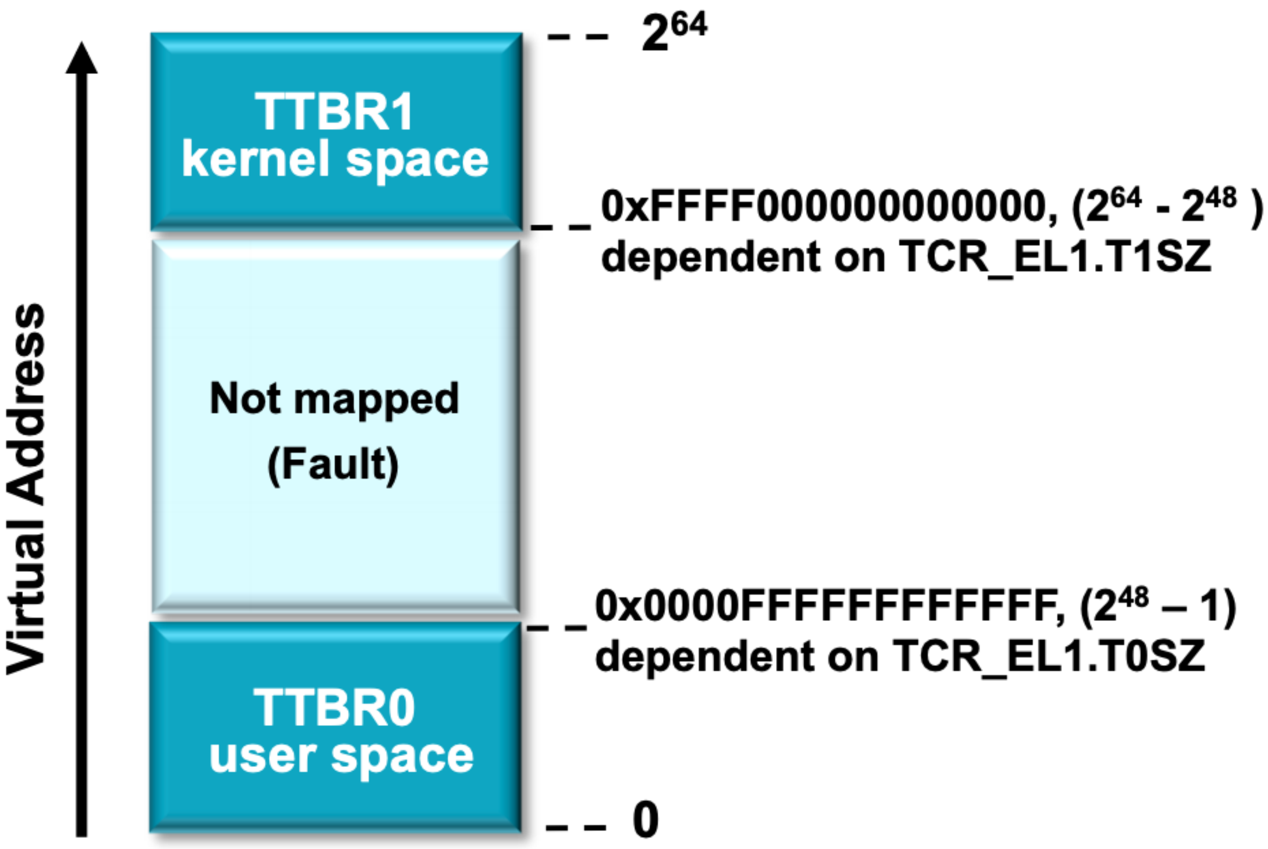
Virtual Addresses 32-bit – Linux



- By default, the kernel uses the top 1GB of virtual address space.
- Each userspace processes get the lower 3GB of virtual address space.

Virtual Addresses 64-bit

- ARM 64
  - 48-bit page table
  - 4-level: 9+9+9+9+12



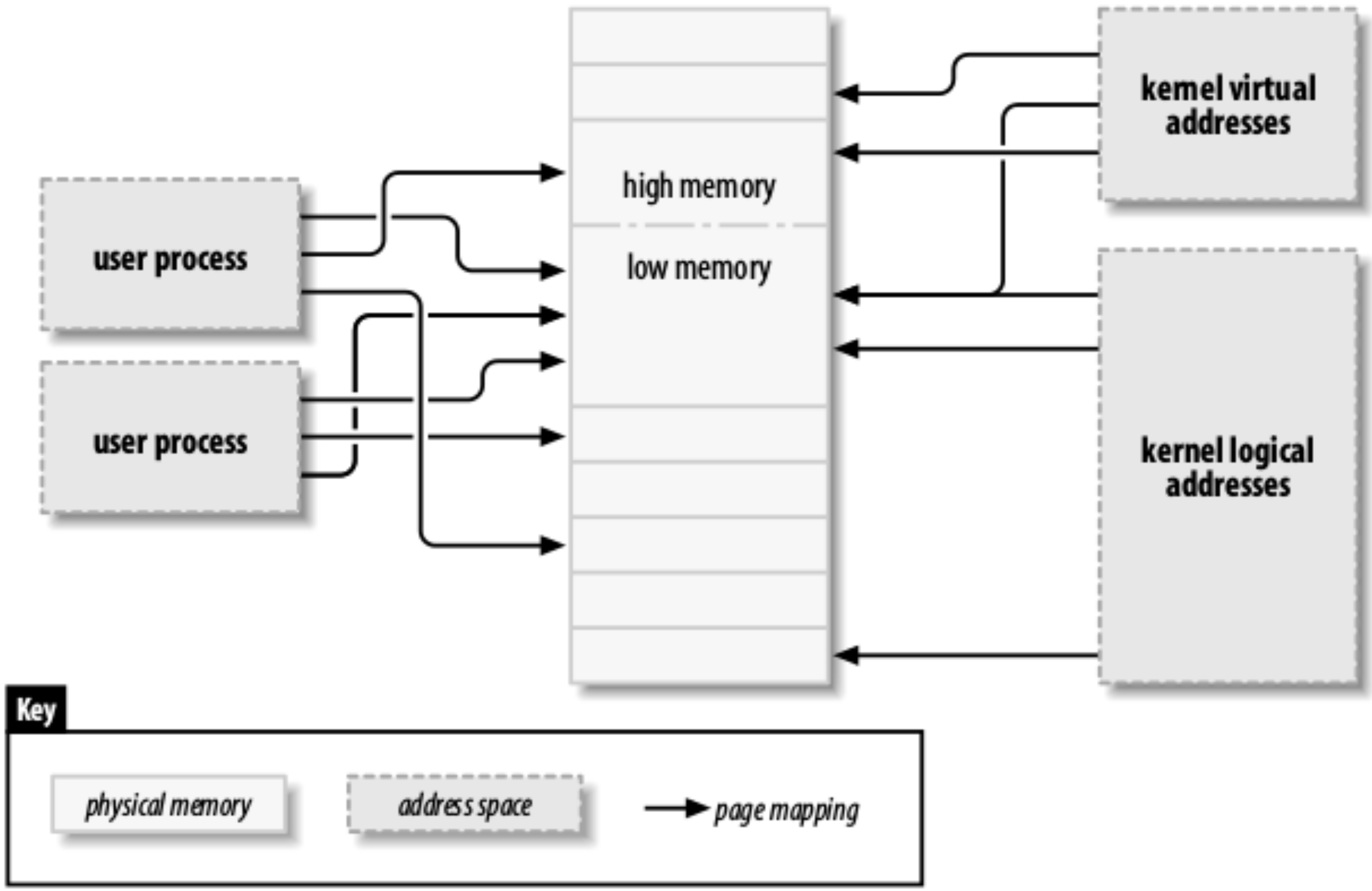
Virtual Addresses 64-bit

- ARM 64
  - 39-bit page table, 3-level: 9+9+9+12

    - 39-bit virtual address for both user and kernel
      - `0000000000000000-0000007fffffffff` (512GB): user
      - [architectural gap]
      - `ffffff8000000000-ffffffbbfffeffff` (~240MB): vmalloc
      - `ffffffbbffff0000-ffffffbcffffffff` (64KB): [guard]
      - `ffffffbc00000000-ffffffbdffffffff` (8GB): vmemmap
      - `ffffffbe00000000-ffffffbffbffffff` (~8GB): [guard]
      - `ffffffbffc000000-ffffffbfffffffff` (64MB): modules
      - `ffffffc000000000-ffffffffffffffff` (256GB): mapped RAM
    - 4KB page configuration
      - 3 levels of page tables (pgtable-nopud.h)
      - Linear mapping using 4KB, 2MB or 1GB blocks
      - AArch32 (compat) supported

Virtual Addresses - Linux

- User virtual address
- Kernel virtual address
- Kernel logical address



10: Virtual Memory

Demand Paging Background

- Code needs to be in memory to execute, but entire program **rarely** needed or used at the same time
  - unused code: error handling code, unusual routines
  - unused data: large data structures
- Consider ability to execute partially-loaded program
  - program no longer constrained by limits of physical memory
  - programs could be larger than physical memory

Demand Paging

- **Demand paging** brings a page into memory only when it is demanded
  - demand means access (read/write)
  - if page is invalid (error) ➠ abort the operation
  - if page is valid but not in memory ➠ bring it to memory
    - Memory here means physical memory
    - This is called page fault
    - via swapping for swapped pages
    - via mapping for new page
    - no unnecessary I/O, less memory needed, slower response, more apps

Page Fault

- First reference to a non-present page will trap to kernel: page fault, the reasons can be
  - **invalid reference** ➠ deliver an exception to the process
  - **valid but not in memory** ➠ swap in
- get an empty physical frame
- swap page into frame via disk operation
- set page table entry to indicate the page is now in memory
- restart the instruction that caused the page fault

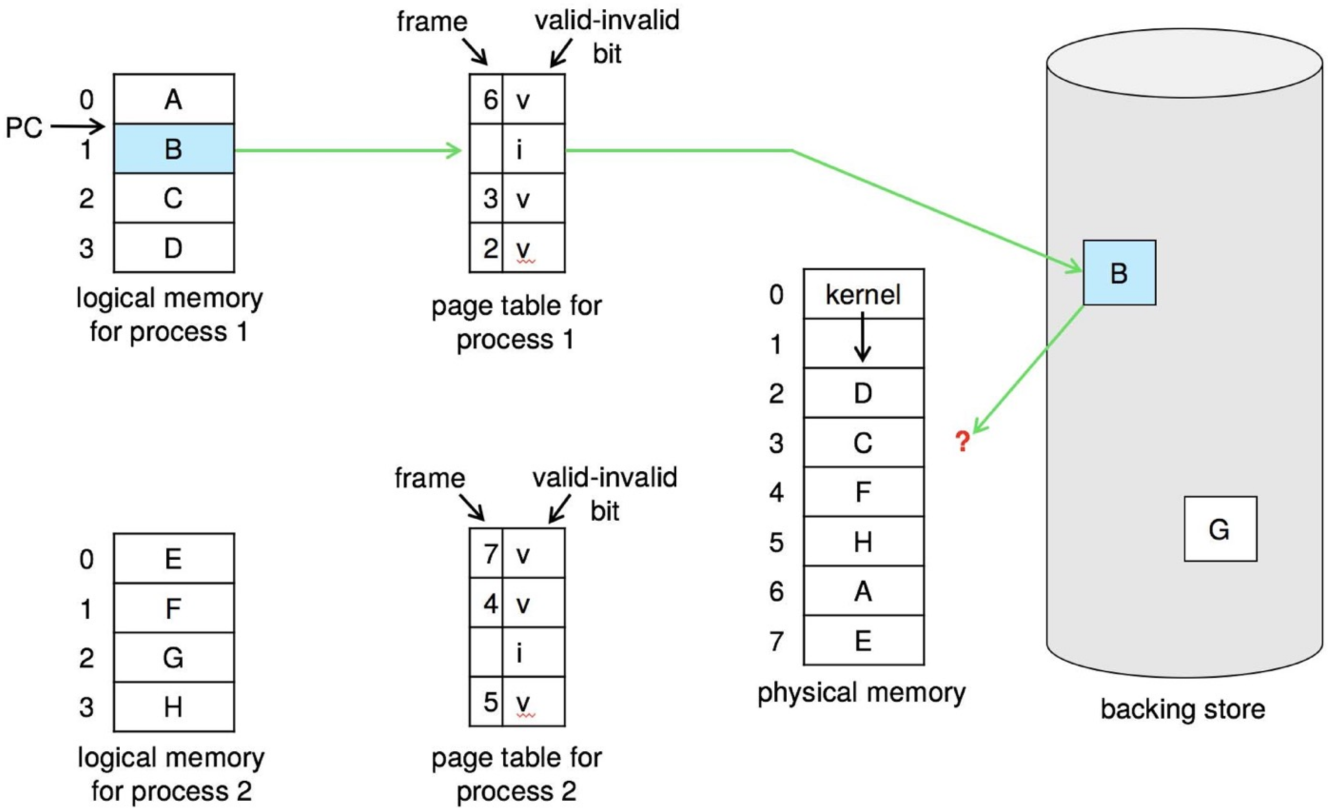
What Happens if There is no Free Frame?

- Used up by process pages
- Also in demand from the kernel, I/O buffers, etc
- How much to allocate to each?
- **Page replacement** – find some page in memory, but not really in use, page it out
  - Algorithm – terminate? swap out? replace the page?
  - Performance – want an algorithm which will result in minimum number of page faults
- Same page may be brought into memory several times

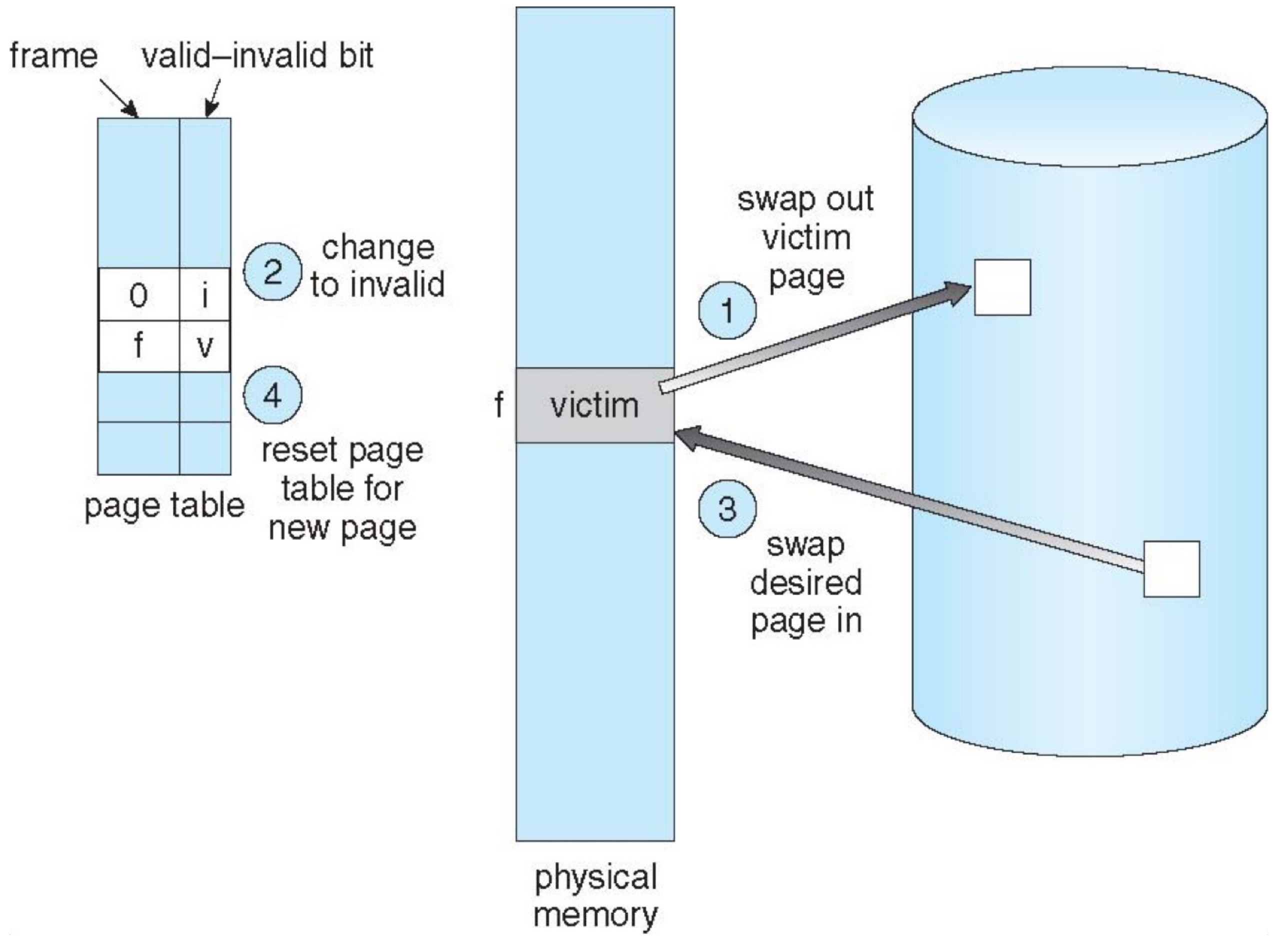
Page Replacement

- Memory is an important resource, system may run out of memory
- To prevent out-of-memory, swap out some pages
  - Page replacement usually is a part of the page fault handler
  - Policies to select victim page require careful design
    - Need to reduce overhead and avoid thrashing
  - Use modified (dirty) bit to reduce number of pages to swap out
    - Only modified pages are written to disk
  - Select some processes to kill (last resort)
- Page replacement completes separation between logical memory and physical memory – large virtual memory can be provided on a smaller physical memory

Need For Page Replacement



Page Replacement



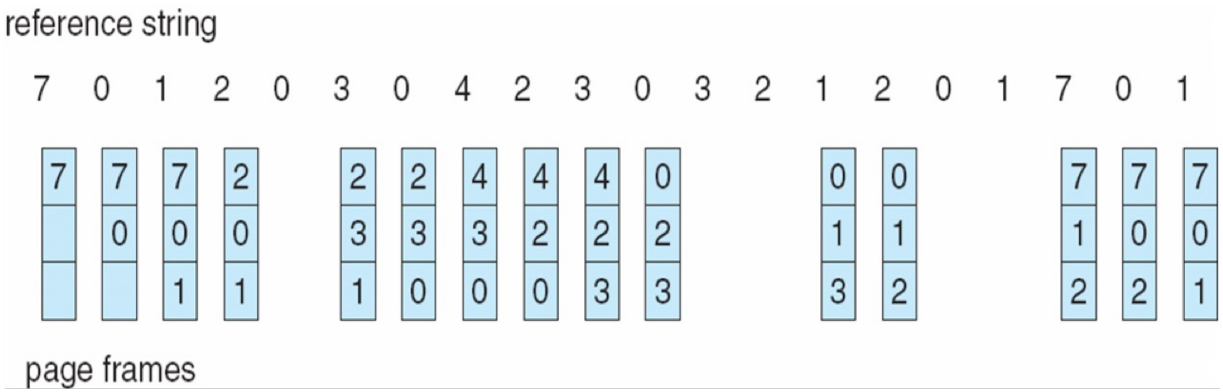
Page Replacement Algorithms

- Page-replacement algorithm should have lowest page-fault rate on both first access and re-access
  - **FIFO**, **optimal**, **LRU**, **LFU**, **MFU**…

- To evaluate a page replacement algorithm:
  - Run it on a particular string of memory references (reference string)
    - String is just page numbers, not full addresses
  - Compute the number of page faults on that string
    - Repeated access to the same page does not cause a page fault
  - In all our examples, the reference string is
    7,0,1,2,0,3,0,4,2,3,0,3,0,3,2,1,2,0,1,7,0,1

First-In-First-Out (FIFO)

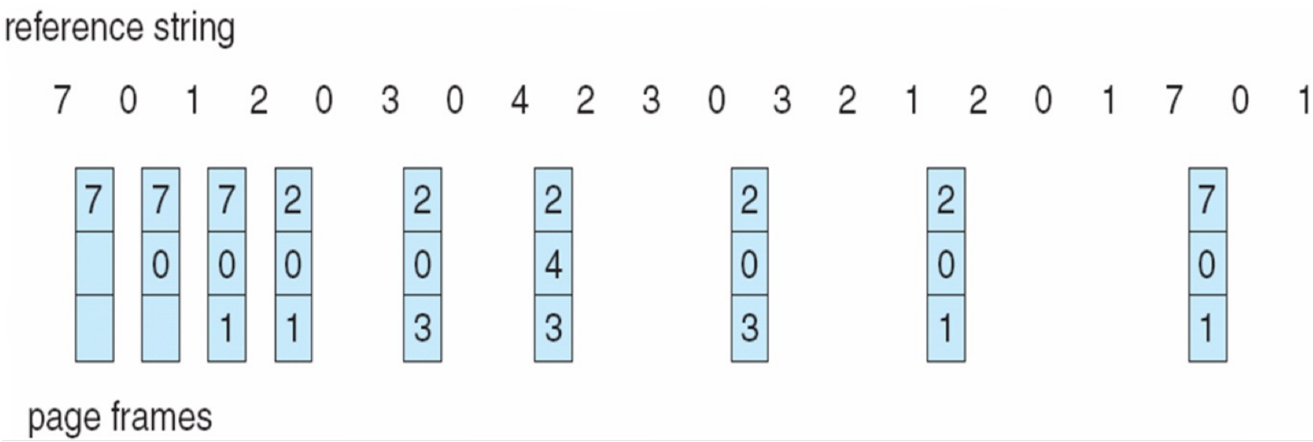
- **FIFO**: replace the first page loaded
  - similar to sliding a window of n in the reference string
  - our reference string will cause 15 page faults with 3 frames
  - how about reference string of 1,2,3,4,1,2,5,1,2,3,4,5 /w 3 or 4 frames?
- For FIFO, adding **more frames** can cause **more page faults**!
  - **Belady's Anomaly**



15 page faults

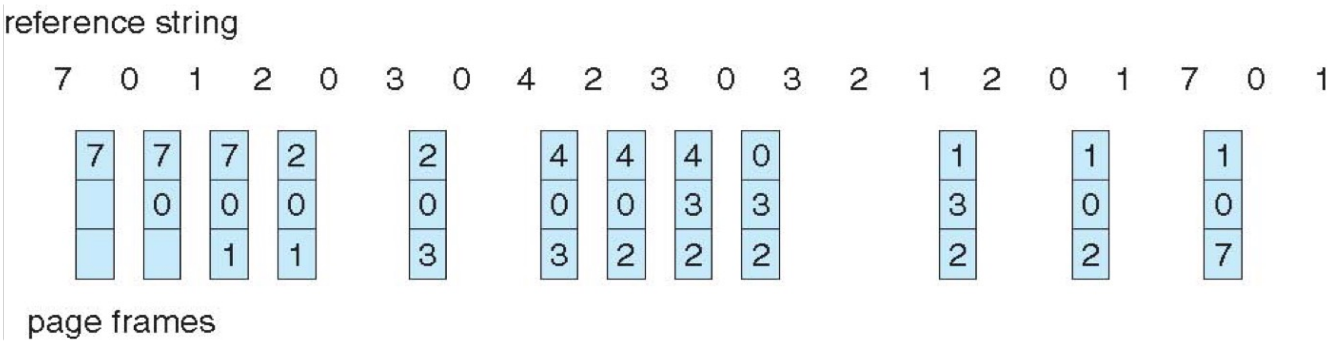
Optimal Algorithm

- **Optimal** : replace page that will not be used for the longest time
  - 9 page fault is optimal for the example on the next slide
- **How do you know which page will not be used for the longest time?**
  - **can’t read the future**
  - used for measuring how well your algorithm performs



Least Recently Used (LRU)

- **LRU** replaces pages that **have not been used for the longest time**
  - associate time of last use with each page, select pages w/ oldest timestamp
  - generally good algorithm and frequently used
  - 12 faults for our example, better than FIFO but worse than OPT
- LRU and OPT do **NOT** have Belady’s Anomaly



Thrashing

- If a process doesn't have "enough" pages, page-fault rate may be high
  - page fault to get page, replace some existing frame
  - but quickly need replaced frame back
  - this leads to:

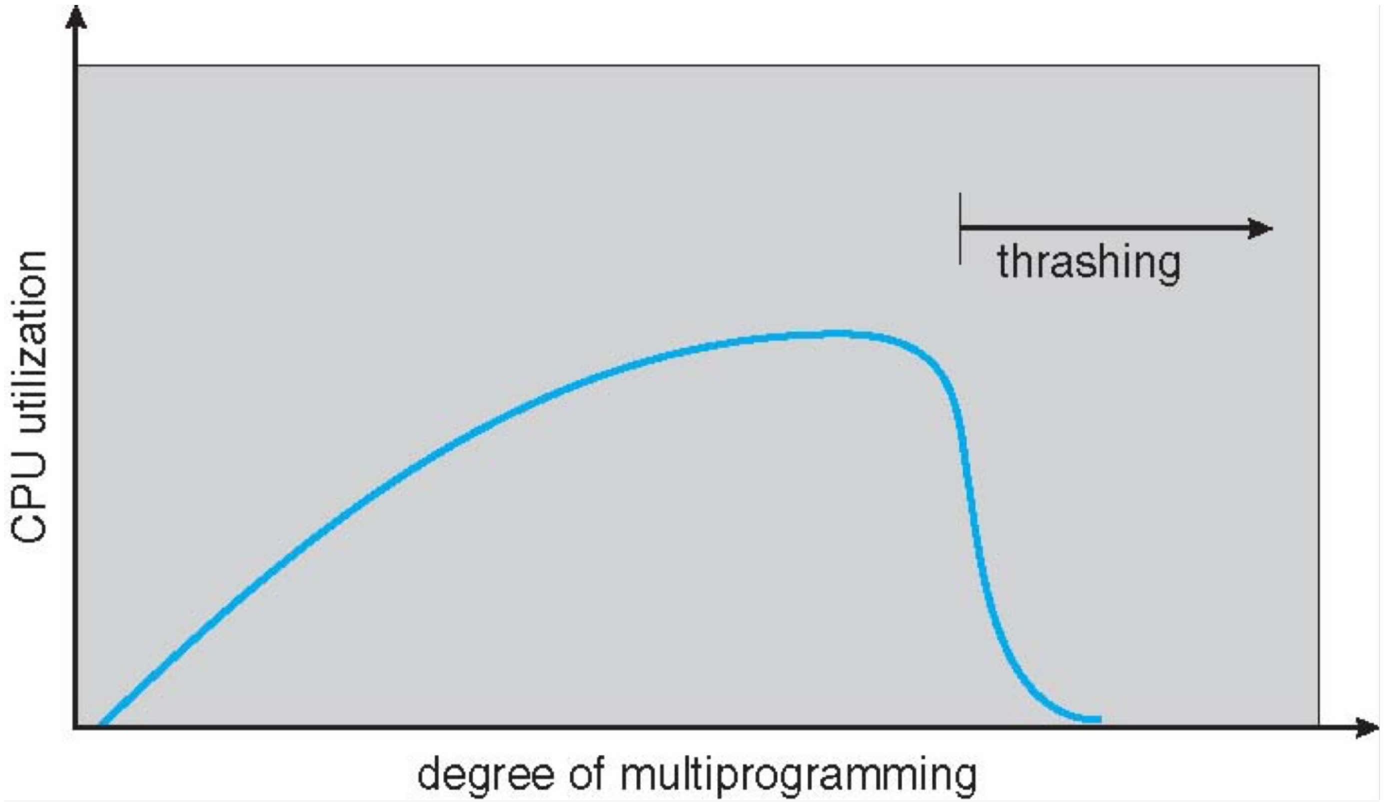
    low CPU utilization ➠

    kernel thinks it needs to increase the degree of

    multiprogramming to maximize CPU utilization ➠

    another process added to the system
- **Thrashing**: a process is busy swapping pages in and out

Thrashing



Demand Paging and Thrashing

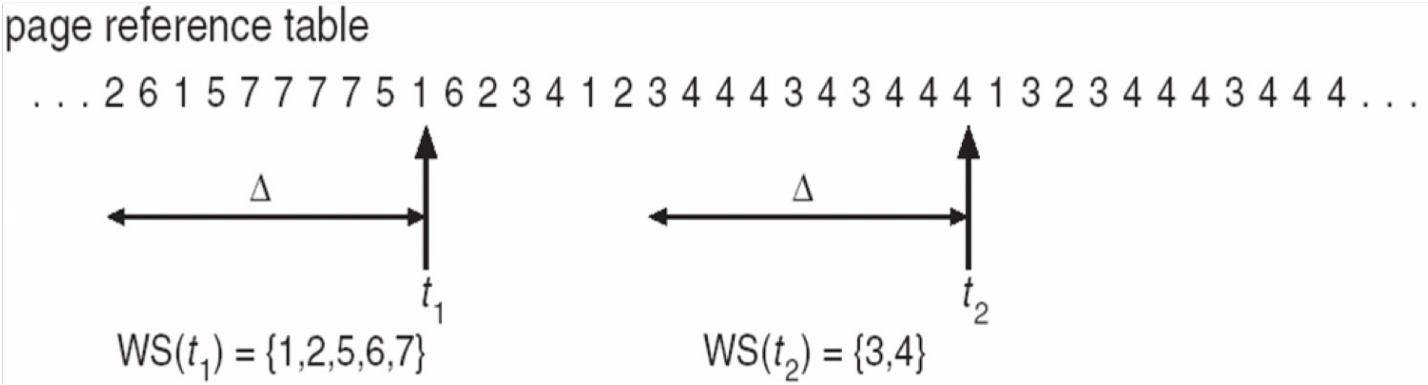
- Why does demand paging work?
    - process memory access has **high locality**
    - process migrates from one locality to another, localities may overlap
- Why does thrashing occur?
    - total size of locality > total memory size

Working-Set Model

- **Working-set window**(Δ): a fixed number of page references
  - if Δ too small ➠ will not include entire locality
  - if Δ too large ➠ will include several localities
  - if Δ = ∞ ➠ will include entire program
- **Working set** of process p_i (WSSi): total number of pages referenced in the most recent Δ (varies in time)
- **Total working sets**: $D = \sum WSS_i$
  - approximation of total locality
  - if D > m ➠ possibility of thrashing
  - to avoid thrashing: if D > m, suspend or swap out some processes

Working-Set Model

- Working-set window Δ = 10



Page table quiz

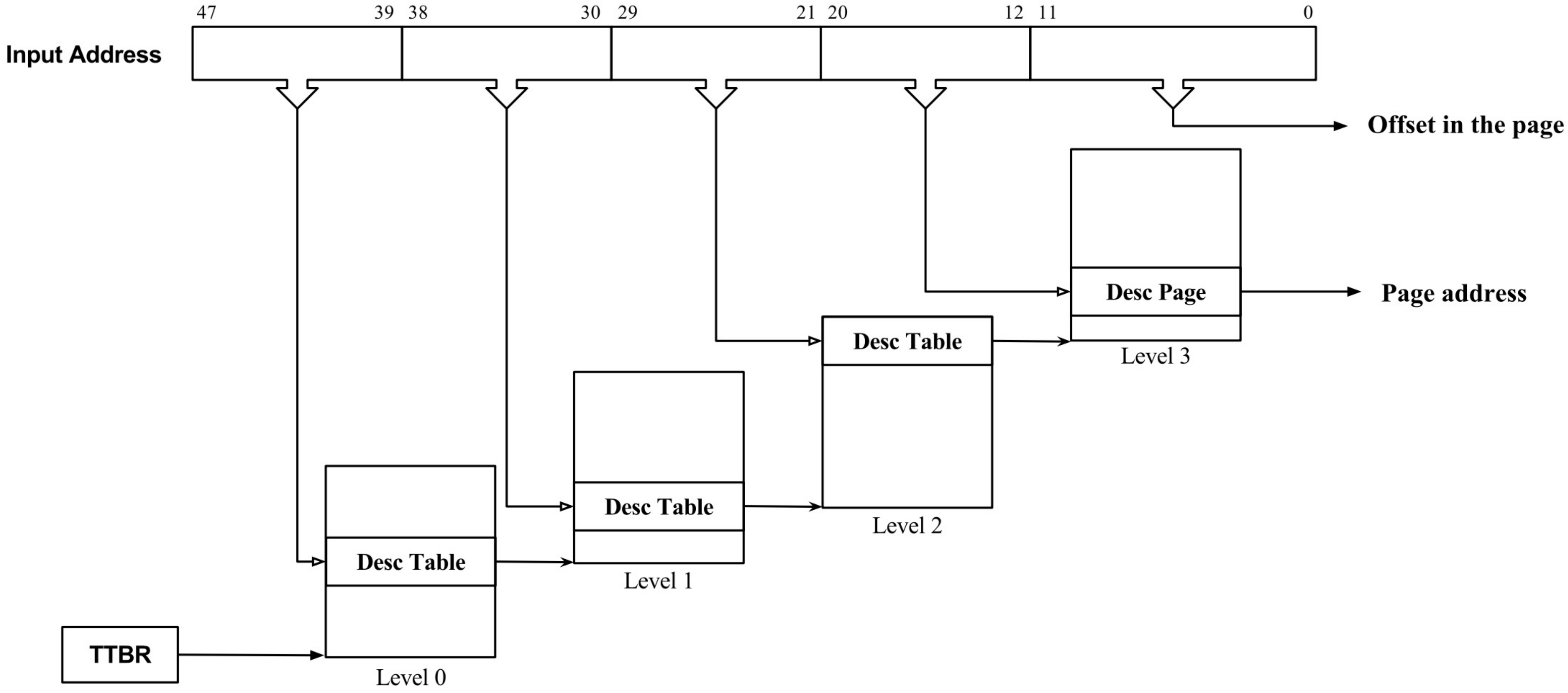
1) In 32-bit architecture, for 1-level page table, how large is the whole page table?
2) In 32-bit architecture, for 2-level page table, how large is the whole page table?
    1) How large for the 1st level PGT?
    2) How large for the 2nd level PGT?
3) Why can 2-level PGT save memory?
4) 2-level page table walk example
    1) Page table base register holds 0x0061 9000
    2) Virtual address is 0xf201 5202
    3) Page table base register holds 0x1051 4000
    4) Virtual address is 0x2190 7010

Page table quiz

- How about page size is 64KB
  - What is the virtual address format for 32-bit?
  - What is the virtual address format for 64-bit?
    - 39-bit VA
    - 48-bit VA

Virtual address format

- 48-bit VA with 4KB page



Takeaway

- The whole slides