

Final Review 03



Operating Systems
Wenbo Shen

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 and 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);

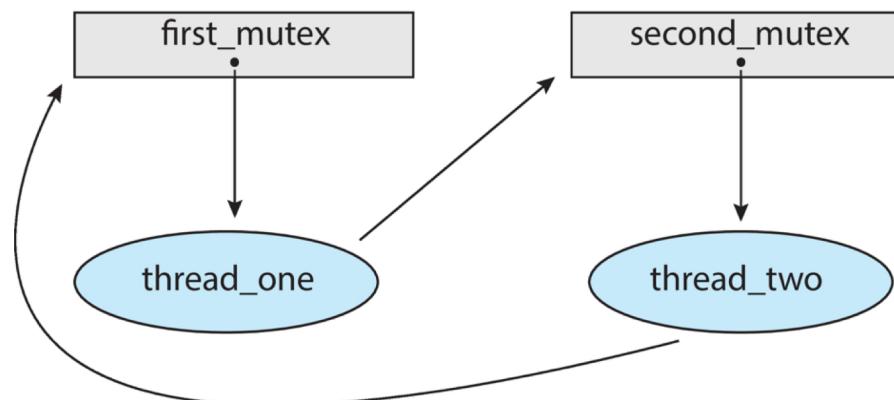
    pthread_exit(0);
}

/* 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 $\{P_0, P_1, \dots, P_n\}$
 - P_0 is waiting for a resource that is held by P_1
 - P_1 is waiting for a resource that is held by $P_2 \dots$
 - 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

- How?
 - **Prevention: that the possibility of deadlock is excluded!!!**
 - **Avoidance**
 - **Deadlock detection and recovery**
 - **Ignore the problem and pretend deadlocks never occur in the system**



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

Deadlock Detection

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

09: Main Memory

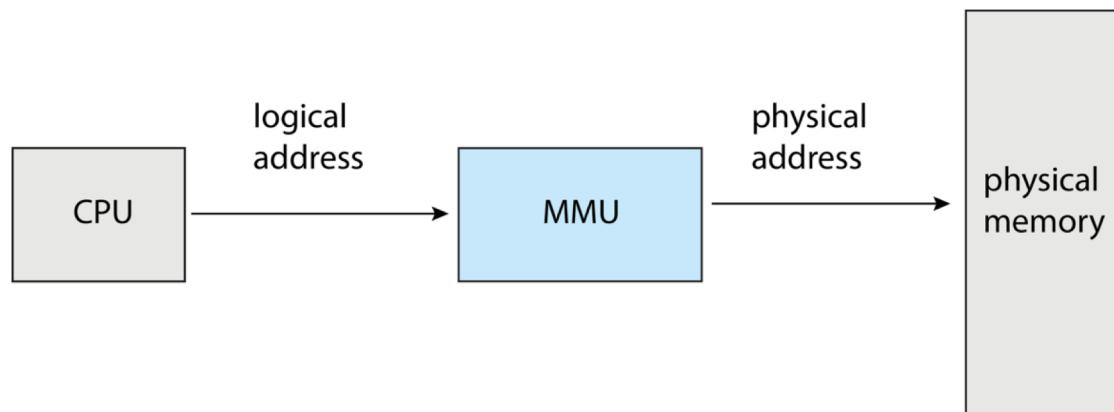


Logical vs. Physical Address Space

- 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



- Many methods possible, covered in the rest of this chapter

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

- 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**
 - 50-percent rule: N allocated blocks, $0.5N$ will be lost due to fragmentation. $1/3$ is unusable!

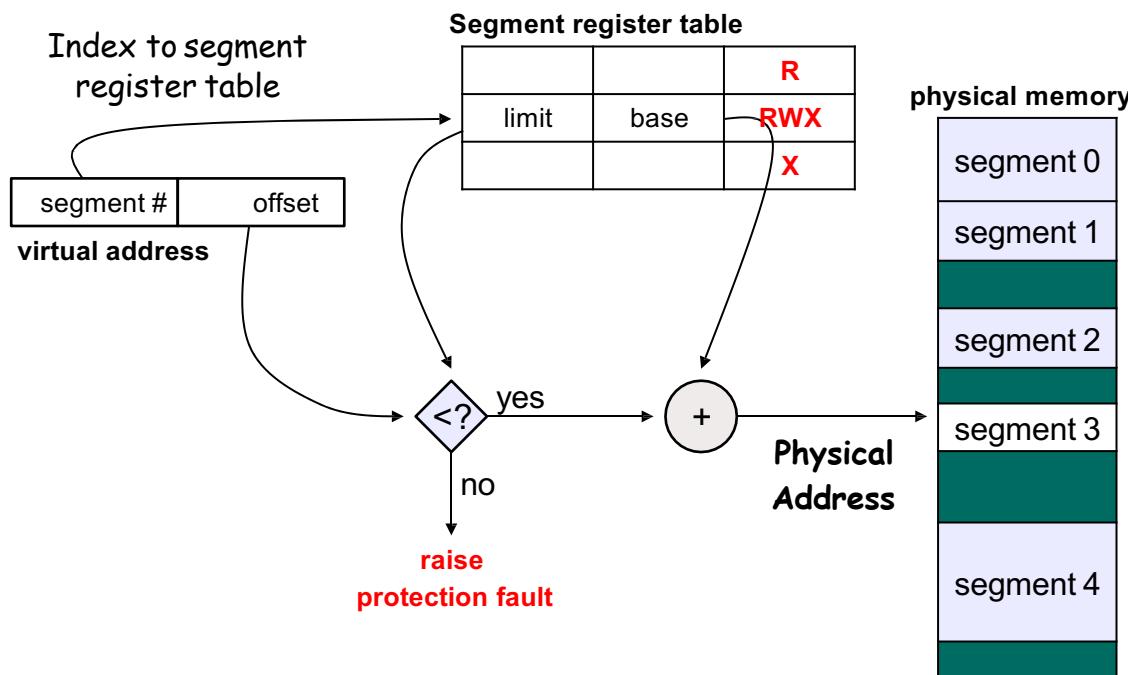
Fragmentation

- **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

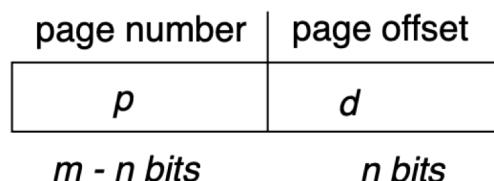


Paging

- 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

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)
 - $EAT = 0.80 \times 10 + 0.20 \times 20 = 12$ 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

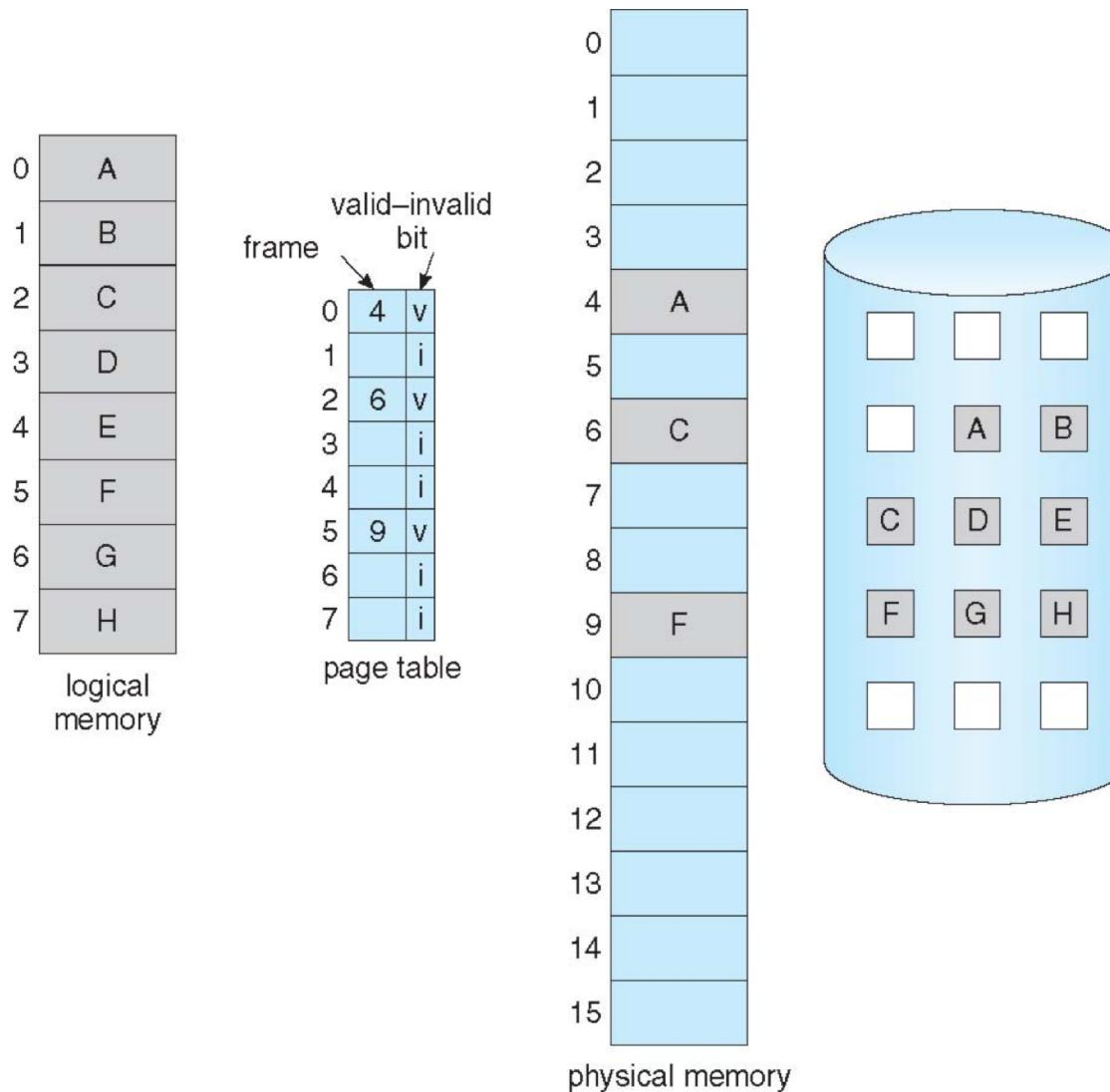
Valid-Invalid Bit

- 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:

Frame #	v/i bit
	v
	v
	v
	v
	i
....	
	i
	i

page table

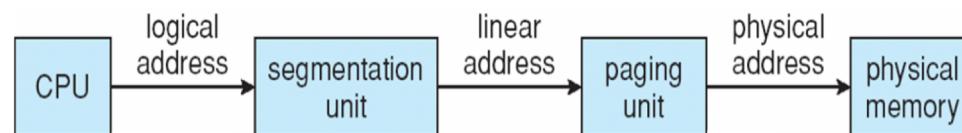
Page Table (Some Pages Are Not in Memory)



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

Logical to Physical Address Translation in IA-32



page number		page offset
p_1	p_2	d
10	10	12

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

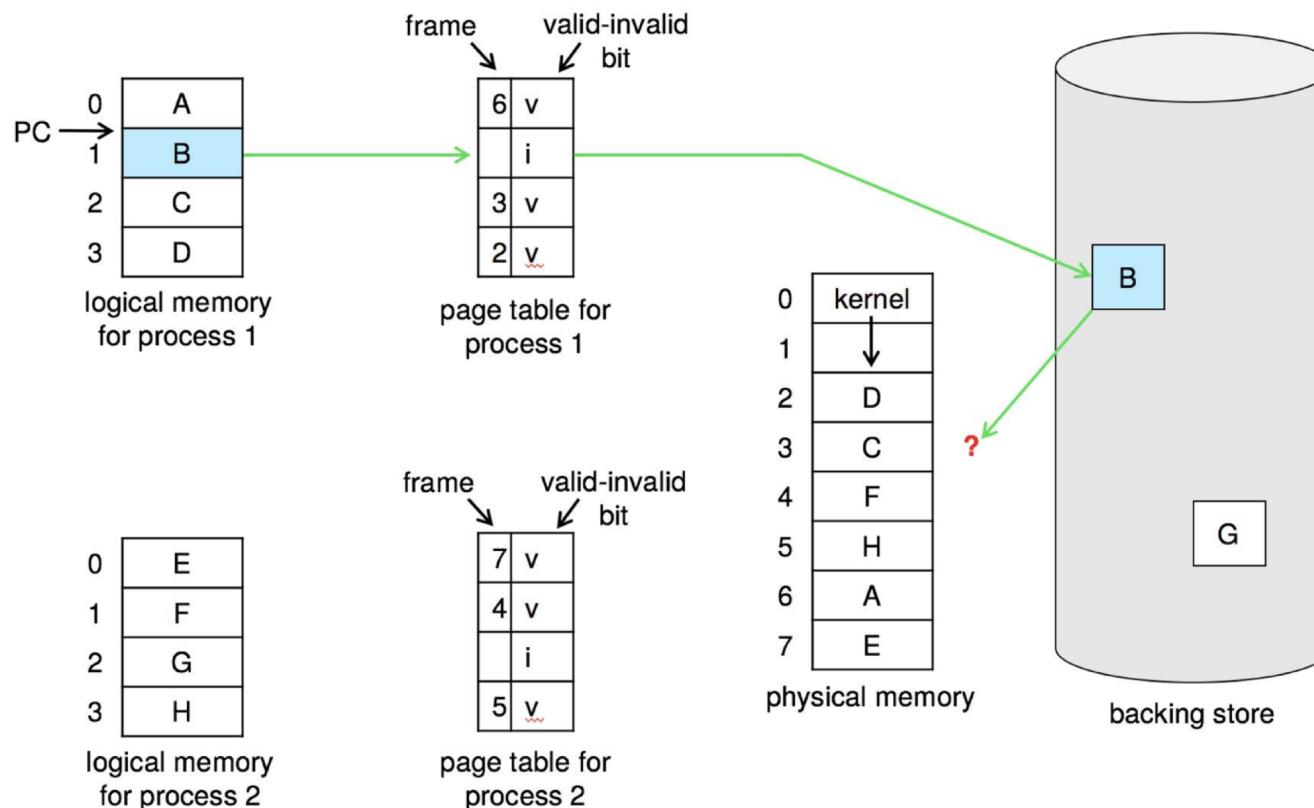
What Happens if There is no Free Frame?

- 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

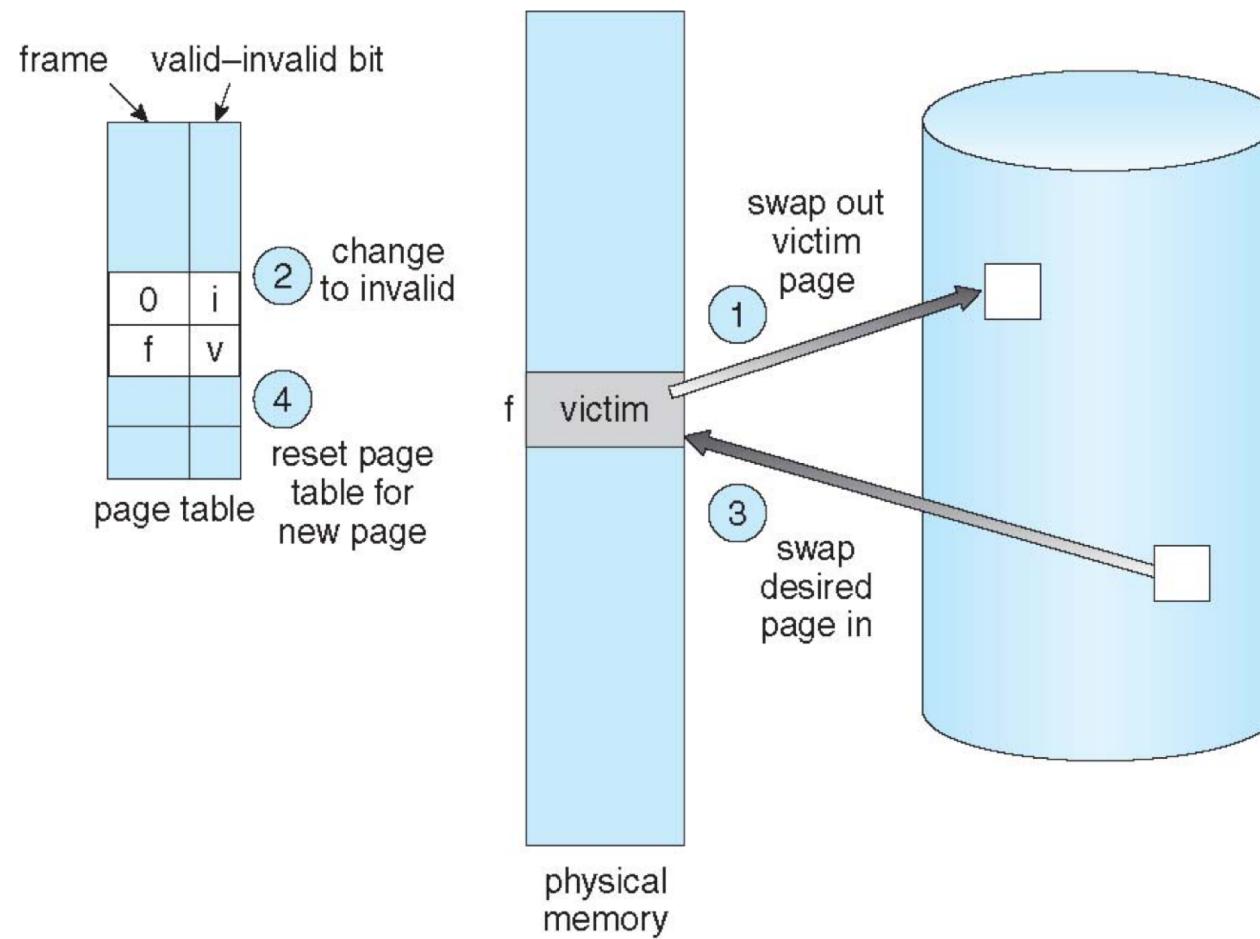
Page Replacement

- 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

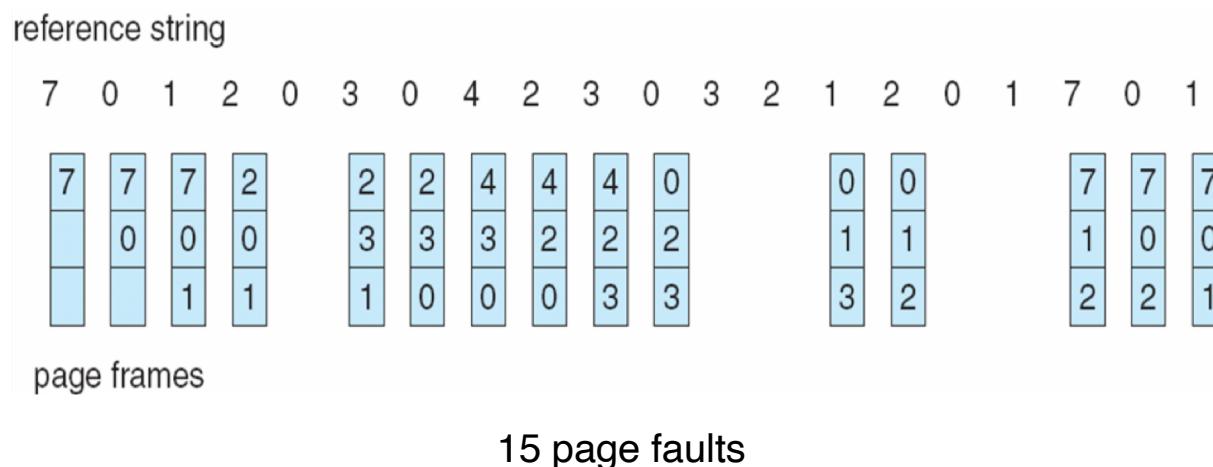


Page Replacement Algorithms

- 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**

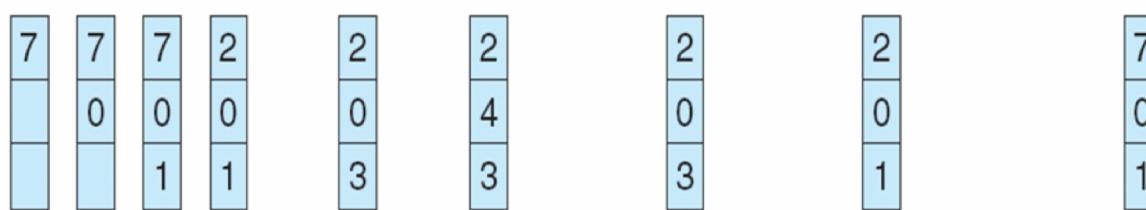


Optimal Algorithm

- **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

reference string

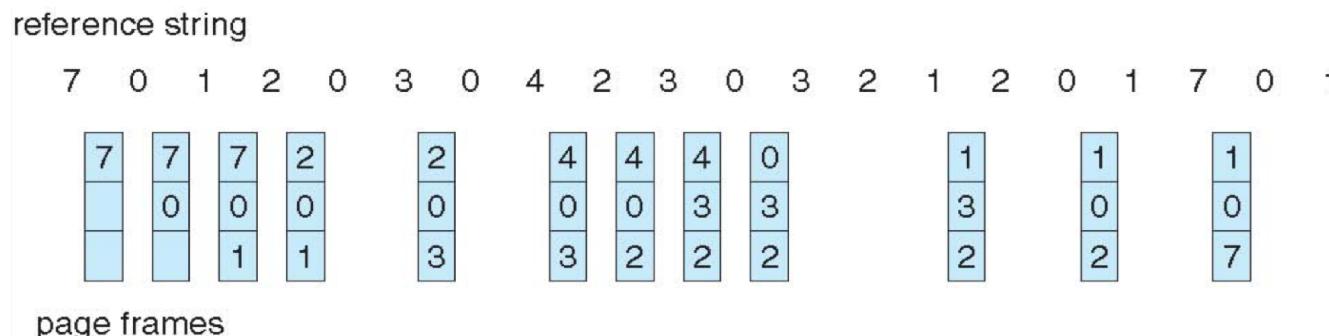
7 0 1 2 0 3 0 4 2 3 0 3 2 1 2 0 1 7 0 1



page frames

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



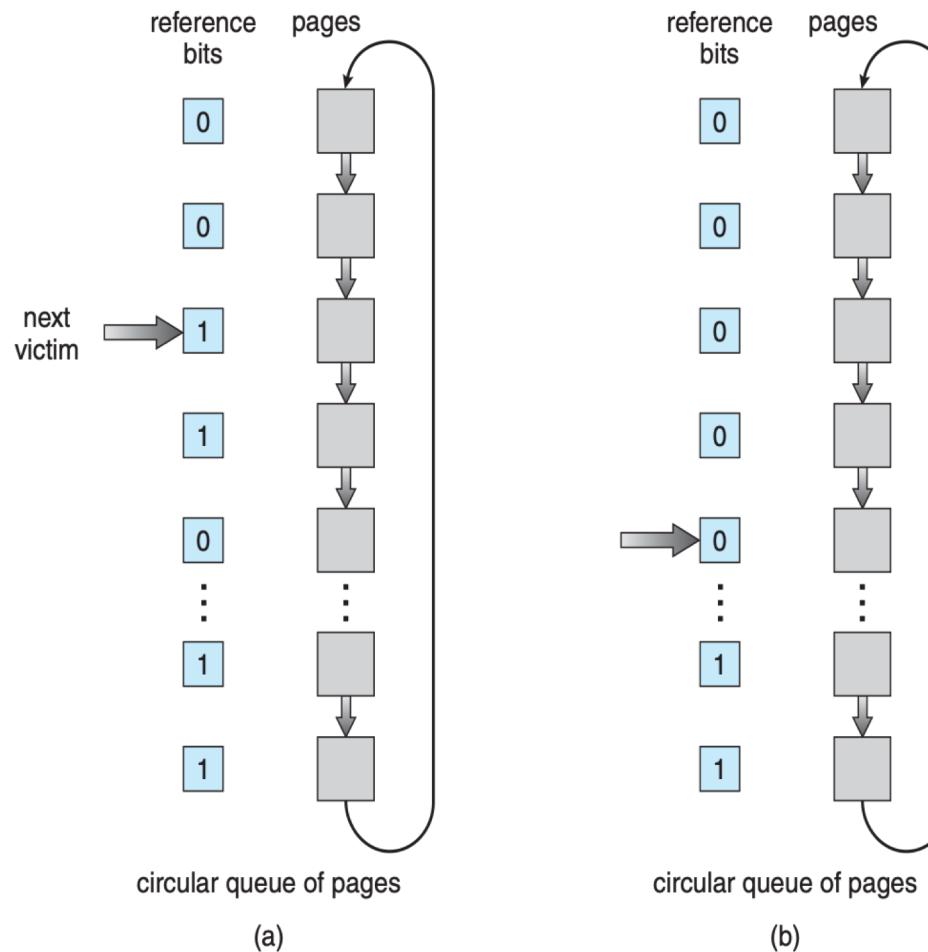
LRU Approximation Implementation

- Counter-based and stack-based LRU have high performance overhead
- Hardware provides a **reference bit**
- LRU approximation with a **reference bit**
 - associate with each page a reference bit, initially set to 0
 - when page is referenced, set the bit to 1 (done by the hardware)
 - replace any page with reference bit = 0 (if one exists)
 - We do not know the order, however

LRU Implementation

- **Second-chance** algorithm
 - Generally FIFO, plus hardware-provided **reference bit**
 - **Clock** replacement
 - If page to be replaced has
 - Reference bit = 0 -> replace it
 - reference bit = 1 then:
 - set reference bit 0, leave page in memory
 - replace next page, subject to same rules

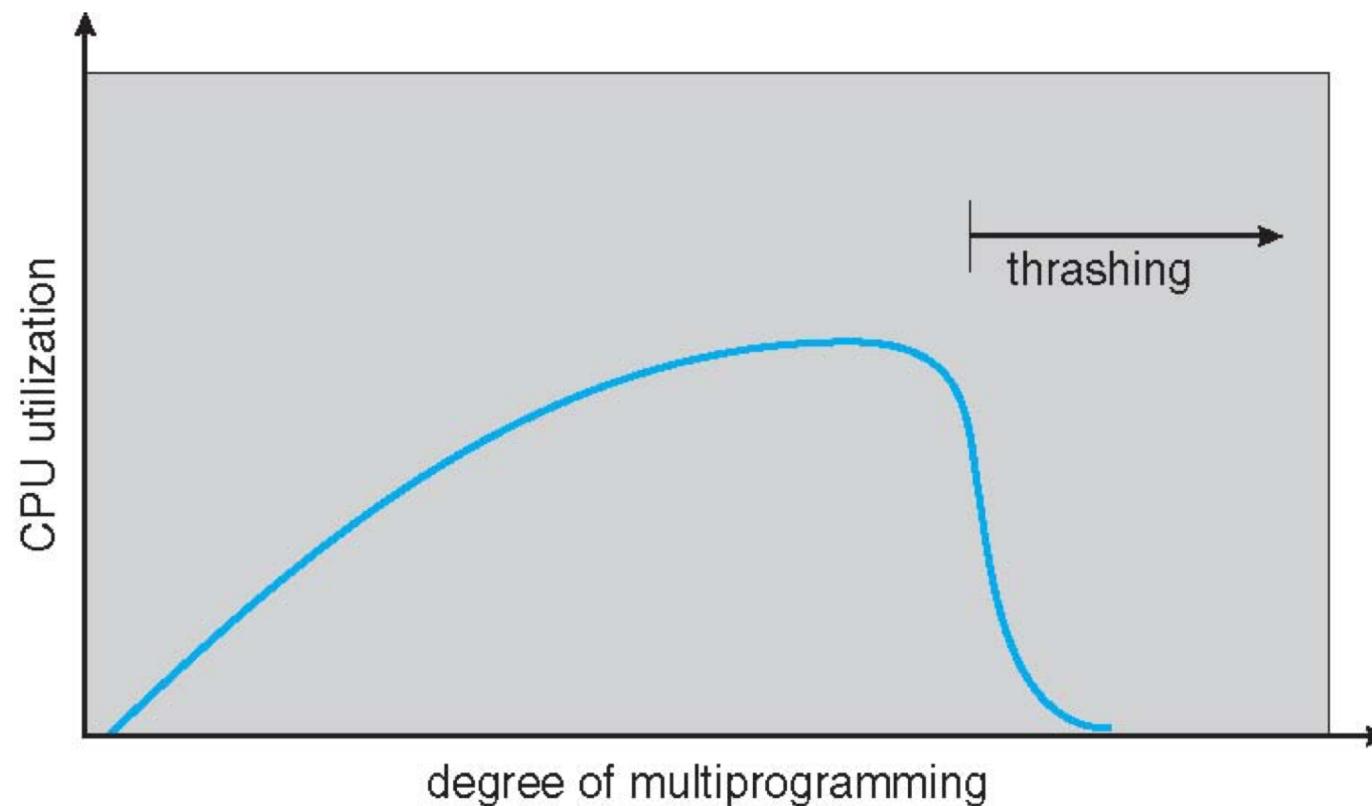
Second-chance (clock) Page-replacement Algorithm



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

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

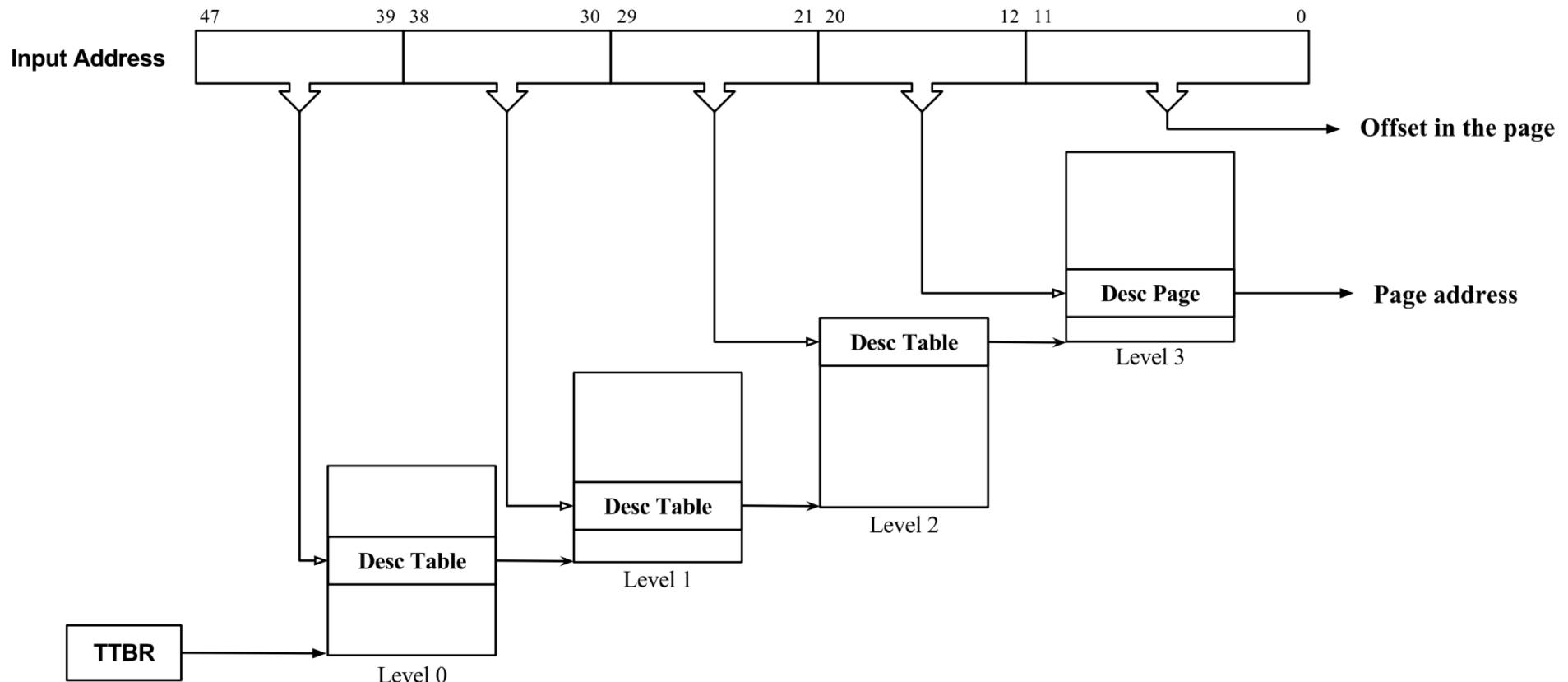
 - 3) Page table base register holds 0x1051 4000
 - 4) Virtual address is 0x2190 7010

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