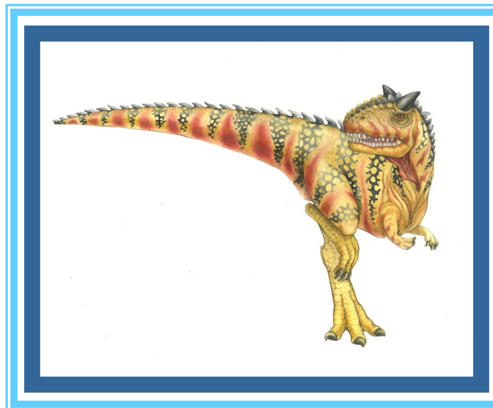
Chapter 9: Virtual Memory



Silberschatz, Galvin and Gagne ©2013

Chapter 9: Virtual Memory

- Background
- Demand Paging
- Page Replacement

Objectives

- To describe the benefits of a virtual memory system
- To explain the concepts of demand paging, page-replacement algorithms, and allocation of page frames

Introduction

- So far in the memory management and process concepts, it was seen that the process instructions being executed must be in physical memory.

- **Virtual Memory** is a storage scheme where secondary memory can be treated as if it is a part of main memory.
- **Virtual Memory** is a technique that allows the execution of processes that are not completely in memory.
- Virtual memory abstracts main memory into an extremely large, uniform array of storage,
- Thus, a computer can address more memory than the amount physically installed on the system.
- This extra memory is actually called **virtual memory** and it is a section of a secondary storage (hard disk) that's set up to emulate the computer's RAM.

Background

- Code needs to be in memory to execute, but entire program rarely used
  - Error code, unusual routines, large data structures
- Entire program code not needed at same time
- Consider ability to execute partially-loaded program
  - Program no longer constrained by limits of physical memory
  - Each program takes less memory while running -> more programs run at the same time
    - Increased CPU utilization and throughput with no increase in response time or turnaround time
  - Less I/O needed to load or swap programs into memory -> each user program runs faster
- Running a program that is not entirely in memory would benefit both the system and the user.

Background (Cont.)

- **Virtual memory** – separation of user logical memory from physical memory
  - Only part of the program needs to be in memory for execution
  - Logical address space can therefore be much larger than physical address space
  - Allows address spaces to be shared by several processes
  - Allows for more efficient process creation
  - More programs running concurrently
  - Less I/O needed to load or swap processes

Virtual Memory That is Larger Than Physical Memory

Background (Cont.)

- **Virtual address space** – logical view (virtual view) of how a process is stored in memory
  - Usually start at address 0, contiguous addresses until end of space
  - Meanwhile, physical memory organized in page frames
  - MMU must map logical to physical
- Virtual memory can be implemented via:
  - Demand paging
  - Demand segmentation

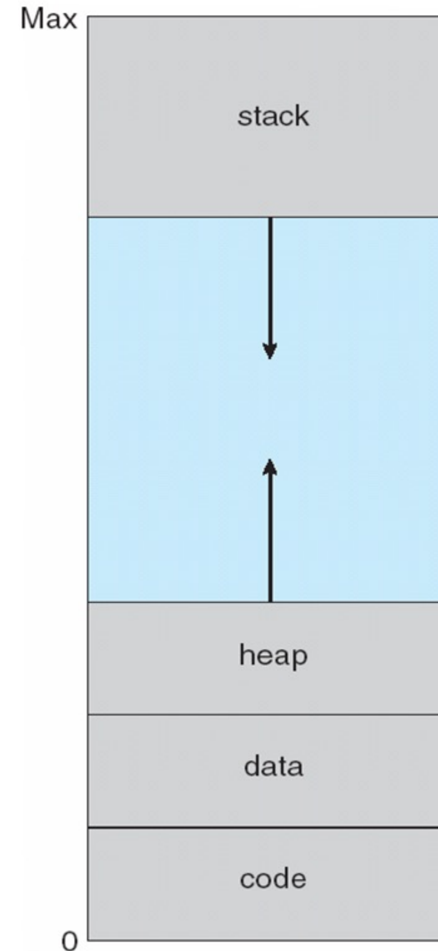
Virtual-address Space

- Usually designing logical address space for stack to start at Max logical address and grow "down" while heap grows "up"
  - Maximizes address space use
  - Unused address space between the two is hole
    - No physical memory needed until heap or stack grows
- Virtual address spaces that include holes are known as **sparse** address spaces.
- Enables **sparse** address spaces with holes left for growth, dynamically linked libraries, etc (benefits)

  System libraries shared via mapping into virtual address space

  Shared memory by mapping pages read-write into virtual address space

  Pages can be shared during `fork()`, speeding process creation



Shared Library Using Virtual Memory

stack

shared library

heap

data

code

shared
pages

stack

shared library

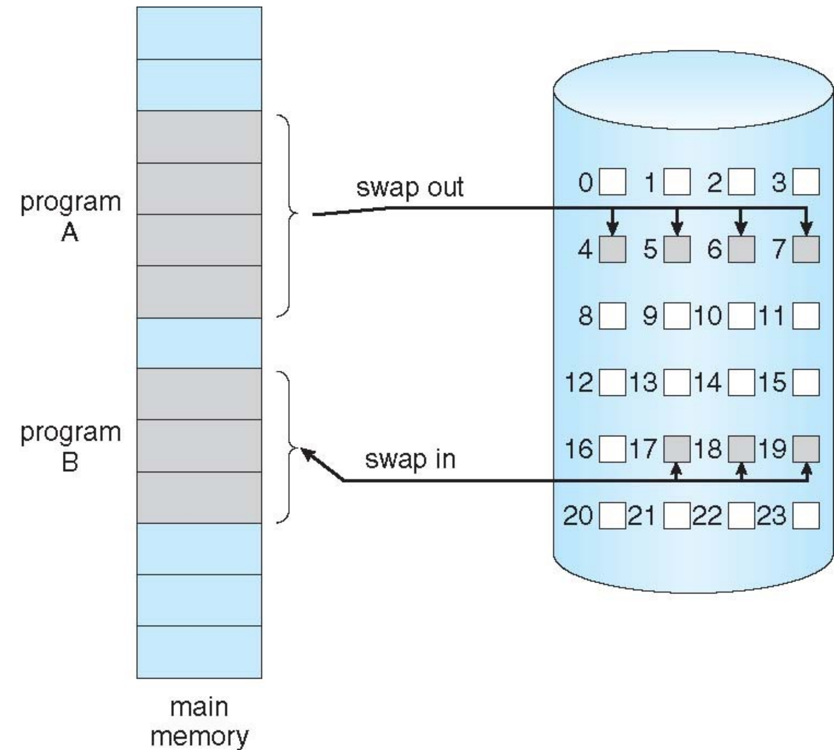
heap

data

code

Demand Paging

- Could bring entire process into memory at load time
- Or bring a page into memory only when it is needed
  - Less I/O needed, no unnecessary I/O
  - Less memory needed
  - Faster response
  - More users
- Similar to paging system with swapping (diagram on right)
- Page is needed ⇒ reference to it
  - invalid reference ⇒ abort
  - not-in-memory ⇒ bring to memory
- **Lazy swapper** – never swaps a page into memory unless page will be needed
  - Swapper that deals with pages is a **pager**



Basic Concepts

- With swapping, pager guesses which pages will be used before swapping out again
- Instead, pager brings in only those pages into memory
- **How to determine that set of pages?**
  - Need new MMU functionality to implement demand paging
- If pages needed are already **memory resident**
  - No difference from non demand-paging
- If page needed and not memory resident
  - Need to detect and load the page into memory from storage
    - Without changing program behavior
    - Without programmer needing to change code

Valid-Invalid Bit

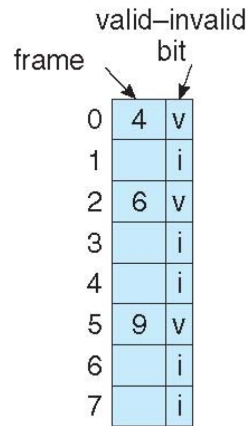
- With each page table entry a **valid–invalid** bit is associated
  (**v** ⇒ in-memory – **memory resident**, **i** ⇒ not-in-memory)
- Initially valid–invalid bit is set to **i** on all entries
- Example of a page table snapshot:

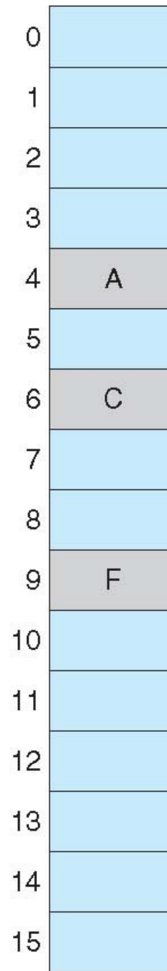
Frame #	valid-invalid bit
	v
	v
	v
	i
. . .	
	i
	i

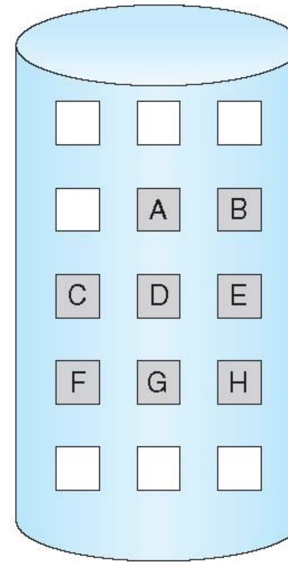
page table

- During MMU address translation, if valid–invalid bit in page table entry is **i** ⇒ page fault

Page Table When Some Pages Are Not in Main Memory







If tried to access a page that does not belong to the process, what will happen?

Page Fault

- If there is a reference to a page, first reference to that page will trap to operating system:

  page fault

1. Operating system looks at another table to decide:
   - Invalid reference $\Rightarrow$ abort
   - Just not in memory
2. Find free frame
3. Swap page into frame via scheduled disk operation
4. Reset tables to indicate page now in memory
   Set validation bit = **v**
5. Restart the instruction that caused the page fault

A crucial requirement for demand paging is the ability to restart any instruction after a page fault.

Steps in Handling a Page Fault

page is on backing store (3)

operating system

reference (1)

trap (2)

load M

i

page table

restart instruction (6)

free frame

reset page table (5)

bring in missing page (4)

physical memory

Page Fault Example

Consider a three-address instruction for ADDING the contents of A to B and placing the result in C:

1. Fetch and decode the instruction (ADD).
2. Fetch A.
3. Fetch B.
4. Add A and B.
5. Store the sum in C.

Assume that C is a page which is not yet present in memory.
Then a page fault will occur when trying to store the sum in C.
So,
In this case, we have to get the desired page, bring it into memory, correct the page table and restart the instruction.

While restarting we have to:

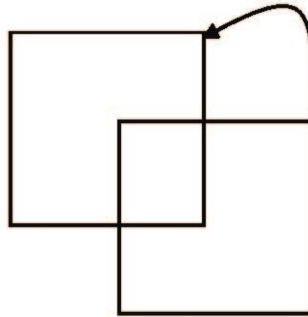
- Fetch the instruction again
- Decode it again.
- Fetch the 2 operands A and B again
- Perform the addition again and
- Store the sum in C

Aspects of Demand Paging

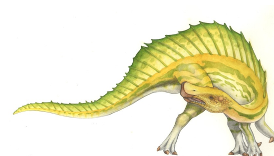
- Extreme case – start process with *no* pages in memory
  - OS sets instruction pointer to first instruction of process, non-memory-resident -> page fault
  - And for every other process pages on first access
  - ➔ This scheme is **Pure demand paging**
- Actually, a given instruction could access multiple pages -> multiple page faults
  - Consider fetch and decode of instruction which adds 2 numbers from memory and stores result back to memory
  - Pain decreased because of **locality of reference**
- Hardware support needed for demand paging
  - Page table with valid / invalid bit
  - Secondary memory (swap device with **swap space**)
  - Instruction restart

Instruction Restart

- Consider an instruction that could access several different locations
  - block move



  - Auto increment/decrement location
  - Restart the whole operation?
    - What if source and destination overlap?



Performance of Demand Paging

- Demand paging can significantly affect the performance of a computer system.
- To see why, let's compute the **effective access time (EAT)** for a demand-paged memory.
- For most computer systems, the memory-access time, denoted *ma,* ranges from 10 to 200 nanoseconds.
- No page faults, the effective access time = the memory access time (**EAT=ma)**.
- What would happen if page faults occur?

Performance of Demand Paging

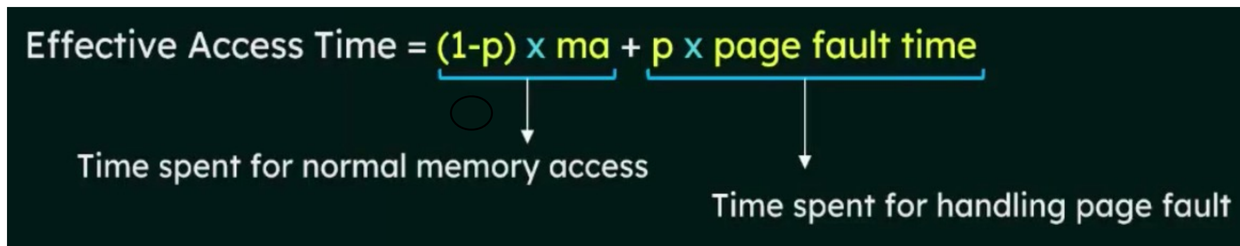
- Stages in Demand Paging (worse case)

1. Trap to the operating system
2. Save the user registers and process state
3. Determine that the interrupt was a page fault
4. Check that the page reference was legal and determine the location of the page on the disk
5. Issue a read from the disk to a free frame:
    1. Wait in a queue for this device until the read request is serviced
    2. Wait for the device seek and/or latency time
    3. Begin the transfer of the page to a free frame
6. While waiting, allocate the CPU to some other user
7. Receive an interrupt from the disk I/O subsystem (I/O completed)
8. Save the registers and process state for the other user
9. Determine that the interrupt was from the disk
10. Correct the page table and other tables to show page is now in memory
11. Wait for the CPU to be allocated to this process again
12. Restore the user registers, process state, and new page table, and then resume the interrupted instruction

Performance of Demand Paging (Cont.)

- Three major activities
  - Service the interrupt – careful coding means just several hundred instructions needed (may take 1 to 100 microseconds)
  - Read the page – lots of time (can take close to 8 milliseconds)
  - Restart the process – (may take 1 to 100 microseconds)
- **Page Fault Service Time =** page fault overhead + swap page out + swap page in
- Page Fault Rate $0 \le p \le 1$
  - if $p = 0$ no page faults
  - if $p = 1$, every reference is a fault
- **Effective Access Time (EAT)**

$$EAT = (1 - p) \times \text{memory access} + p \times \text{page fault time}$$



Demand Paging Example

- Memory access time = 200 nanoseconds
- Average page-fault service time = 8 milliseconds

  EAT = (1 – p) x 200 + p (8 milliseconds)

  = (1 – p) x 200 + p x 8,000,000

  = 200 + p x 7,999,800 (EAT is directly proportional to the **page-fault rate**)

- If one access out of 1,000 causes a page fault, then

  EAT = 8199.8 nanosecond = 8.2 microseconds

- The computer will be slowed down by a factor of 40 because of demand paging! ($\frac{8199.8}{200} \cong 40$)
- If want performance degradation < 10 percent
  - **p = 0 → EAT = 200; degrade by 10%, then EAT = 200+(10%*200) = 220**
  - Calculate p:
    - 220 > 200 + 7,999,800 x p

      20 > 7,999,800 x p

      p < .0000025

      < one page fault in every 400,000 memory accesses

Demand Paging Optimizations

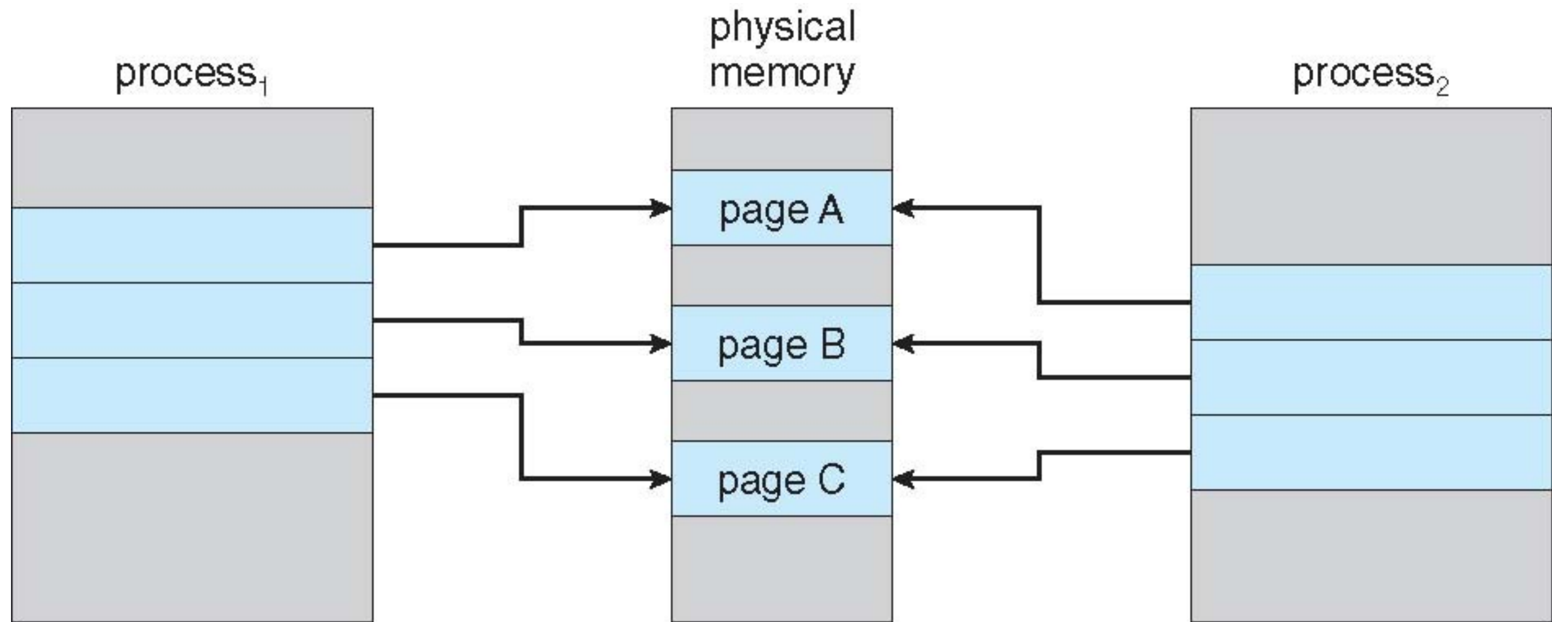
- Swap space I/O faster than file system I/O even if on the same device
  - Swap allocated in larger chunks, less management needed than file system
- Copy entire process image to swap space at process load time
  - Then page in and out of swap space
  - Used in older BSD Unix
- Demand page in from program binary on disk, but discard rather than paging out when freeing frame
  - Used in Solaris and current BSD
  - Still need to write to swap space
    - Pages not associated with a file (like stack and heap) – **anonymous memory**
    - Pages modified in memory but not yet written back to the file system
- Mobile systems
  - Typically don't support swapping
  - Instead, demand page from file system and reclaim read-only pages (such as code)

Copy-on-Write

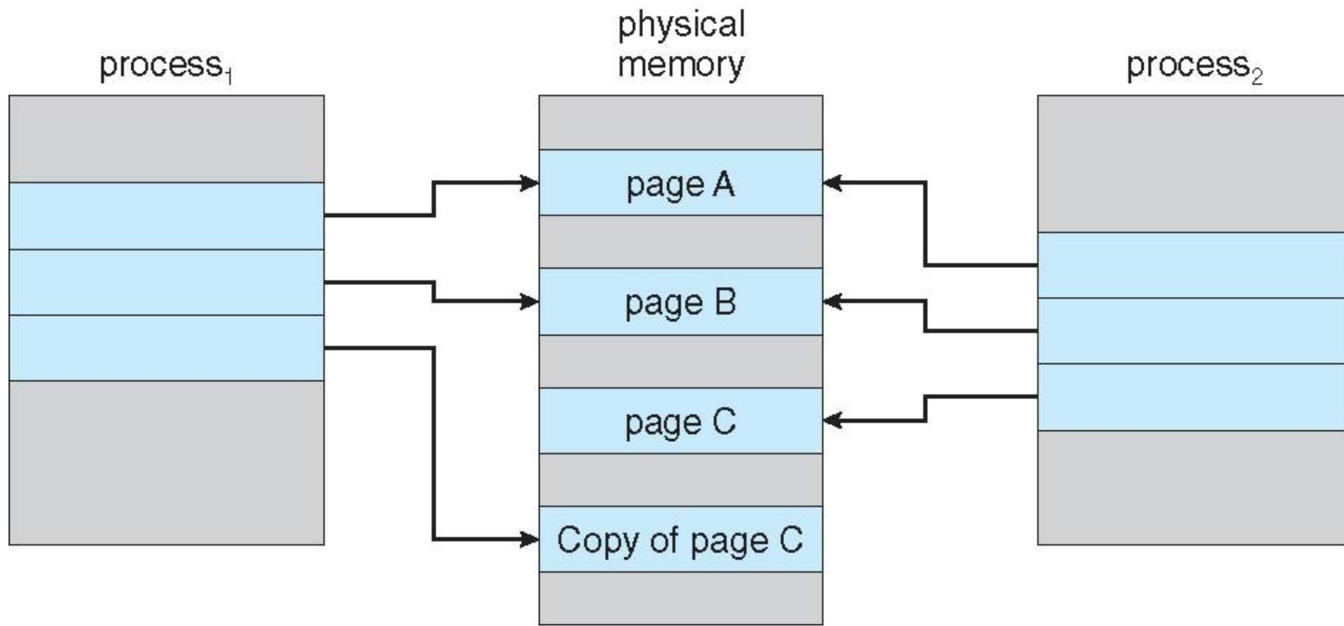
- **Recall that the fork() system call creates a child process that is a duplicate of its parent.**
- **Copy-on-Write** (COW) allows both parent and child processes to initially ***share*** the same pages in memory
  - If either process modifies a shared page, only then is the page copied
- COW allows more efficient process creation as only modified pages are copied
- In general, free pages are allocated from a **pool** of **zero-fill-on-demand** pages
  - Pool should always have free frames for fast demand page execution

- Several versions of UNIX uses `vfork()` (for **virtual memory fork**) a variation on `fork()`, the parent process is suspended, and the child process uses the address space of the parent.
- `vfork()` does not use copy-on-write ➔ if the child process changes any pages of the parent's address space, the altered pages will be visible to the parent once it resumes
  - Designed to have child call `exec()`
  - Very efficient

Before Process 1 Modifies Page C



After Process 1 Modifies Page C



Problems of Demand Paging

What Happens if There is no Free Frame?

- Used up by process pages
- Also in-demand from the kernel, I/O buffers (buffers for I/O also consume a considerable amount of memory)
- How much to allocate to each process?
- If increasing the degree of multiprogramming➔ **over-allocating** memory

Solution:

- 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

Need For Page Replacement

PC → 1

	logical memory for user 1
0	H
1	load M
2	J
3	M

page table for user 1 (frame, valid–invalid bit)

frame	valid–invalid bit
3	v
4	v
5	v
	i

	logical memory for user 2
0	A
1	B
2	D
3	E

page table for user 2 (frame, valid–invalid bit)

frame	valid–invalid bit
6	v
	i
2	v
7	v

	physical memory
0	monitor
1	
2	D
3	H
4	load M
5	J
6	A
7	E

B

M

Page Replacement

- Prevent **over-allocation** of memory by modifying page-fault service routine to include page replacement
- Use **modify** (**dirty**) **bit** to reduce overhead of page transfers – only modified pages are written to disk
- Page replacement completes separation between logical memory and physical memory – large virtual memory can be provided on a smaller physical memory

Basic Page Replacement

1. Find the location of the desired page on disk

2. Find a free frame:
   - If there is a free frame, use it
   - If there is no free frame, use a page replacement algorithm to select a **victim frame**
     - Write victim frame to disk if *dirty*

3. Bring the desired page into the (newly) free frame; update the page and frame tables

4. Continue the process by restarting the instruction that caused the trap (page fault)

Note that if no frames are free, potentially 2 page transfers for page fault (one out and one in) are required – increasing EAT

Page Replacement

frame
valid–invalid bit
0 i
f v
page table
2 change to invalid
4 reset page table for new page
f victim
physical memory
1 swap out victim page
3 swap desired page in

Page and Frame Replacement Algorithms

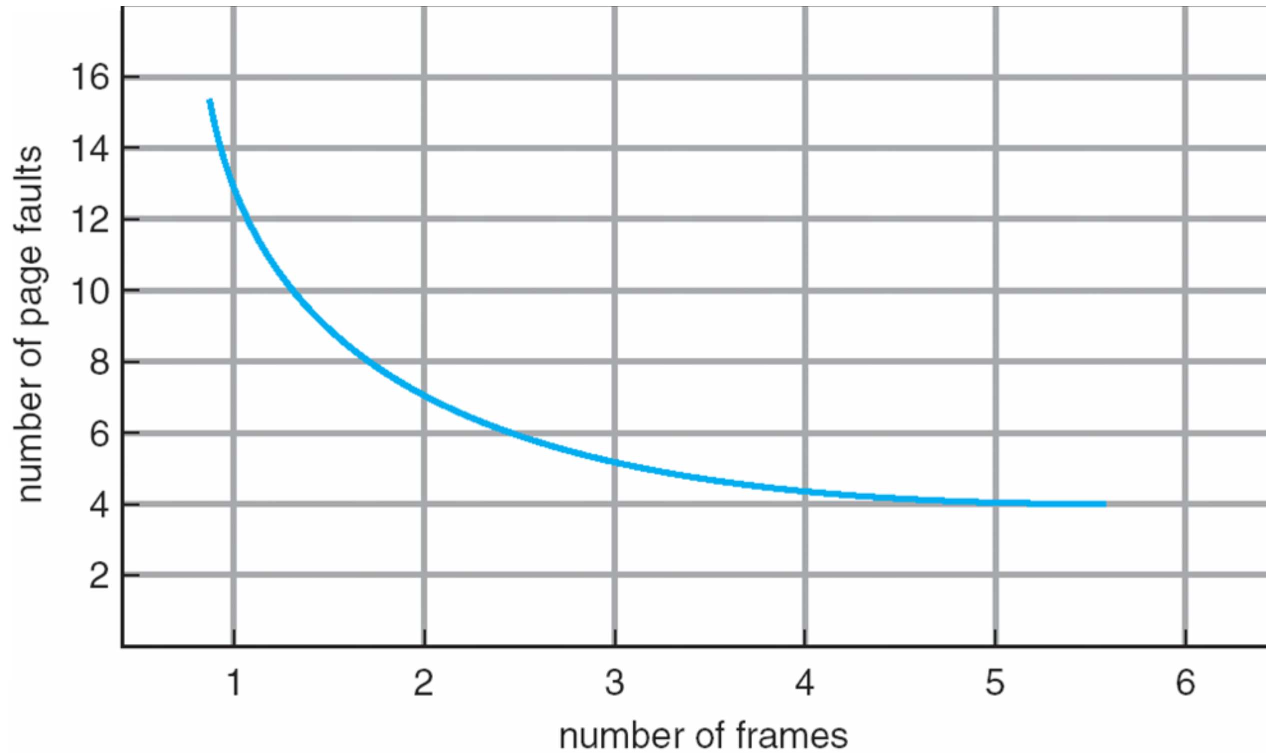
We must solve two major problems to implement demand paging:

1. **Frame-allocation algorithm** determines
   - How many frames to give each process
   - Which frames to replace
2. **Page-replacement algorithm**
   - Want lowest page-fault rate on both first access and re-access

- Evaluate algorithm by running it on a particular string of memory references (reference string) and computing the number of page faults on that string
  - String is just page numbers, not full addresses
  - Repeated access to the same page does not cause a page fault
  - Results depend on number of frames available
- In all our examples, the **reference string** of referenced page numbers is

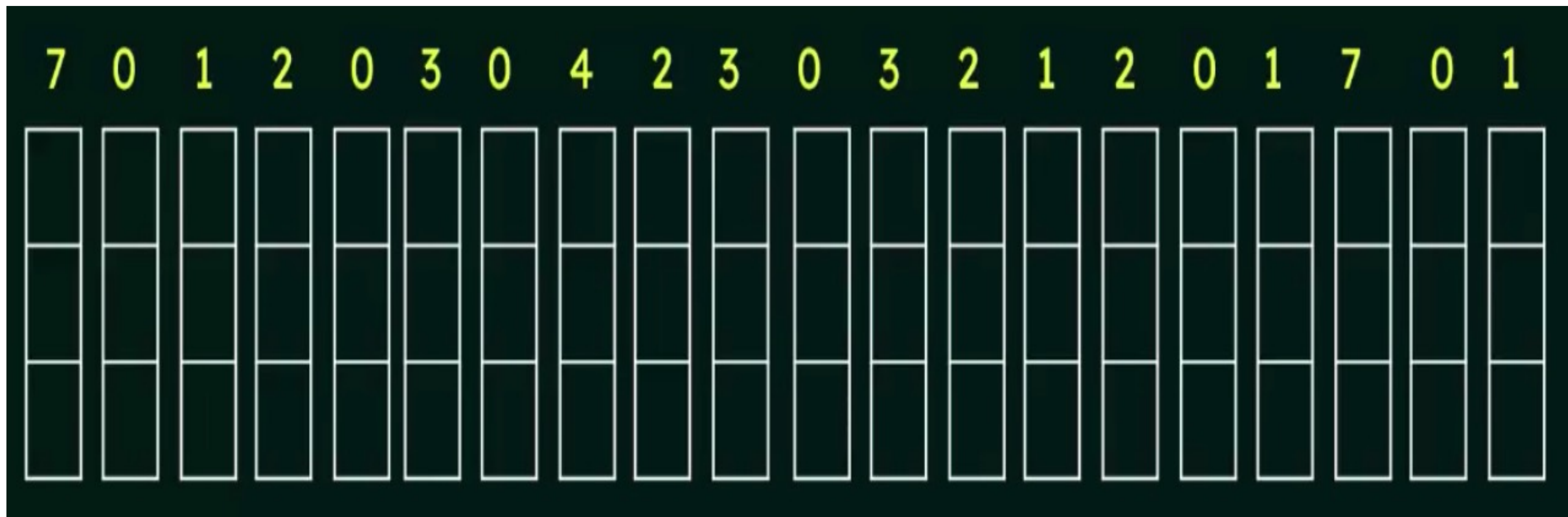
7,0,1,2,0,3,0,4,2,3,0,3,0,3,2,1,2,0,1,7,0,1

Graph of Page Faults Versus The Number of Frames



First-In-First-Out (FIFO) Algorithm

- The simplest page-replacement algorithm is (FIFO) algorithm.
- We can create a FIFO queue to hold all pages in memory.
  - We replace the page at the head of the queue.
  - When a page is brought into memory, we insert it at the tail of the queue.
- Reference string: **7,0,1,2,0,3,0,4,2,3,0,3,0,3,2,1,2,0,1,7,0,1**
- 3 frames (3 pages can be in memory at a time per process)



First-In-First-Out (FIFO) Algorithm

- Reference string: **7,0,1,2,0,3,0,4,2,3,0,3,0,3,2,1,2,0,1,7,0,1**
- 3 frames (3 pages can be in memory at a time per process)

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

15 Page Faults
when
FIFO Page Replacement
was used

FIFO Illustrating Belady's Anomaly

Can vary by reference string: consider 1,2,3,4,1,2,5,1,2,3,4,5

- Adding more frames can cause more page faults!

  Belady's Anomaly (for some page-replacement algorithms, the page-fault rate may ***increase*** as the number of allocated frames increases)

number of page faults

16
14
12
10
8
6
4
2

1 2 3 4 5 6 7

number of frames

Optimal Algorithm

- Replace page that will not be used for longest period of time
  - 9 is optimal for the example

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

Optimal Algorithm

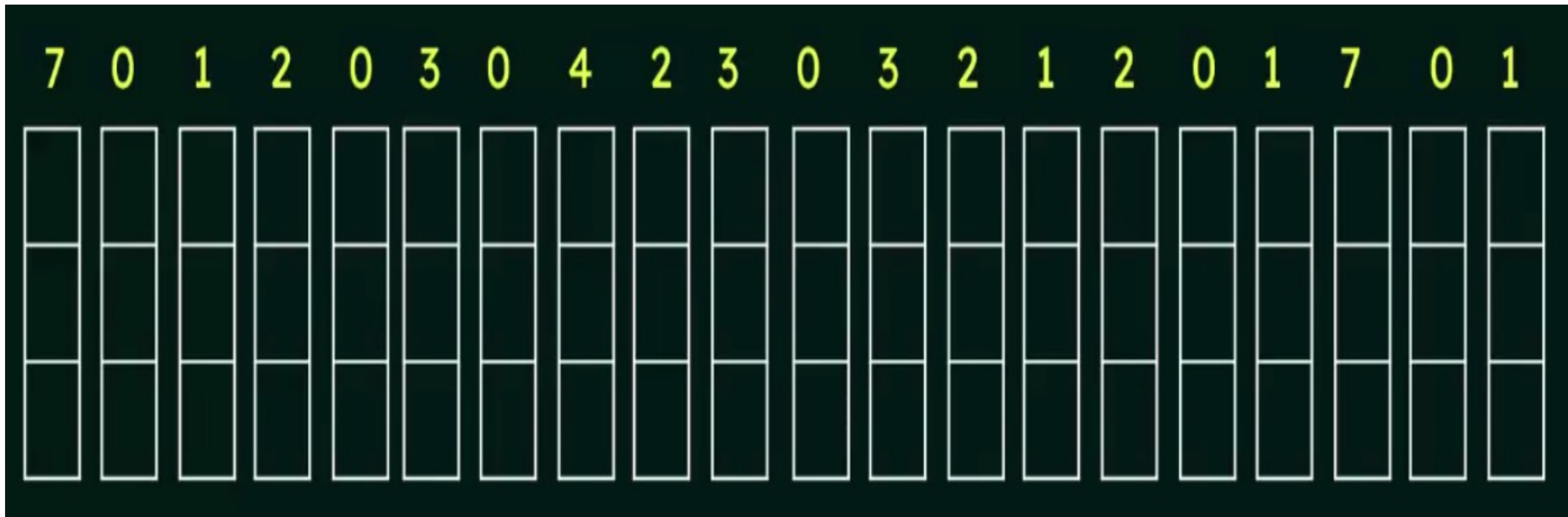
7	0	1	2	0	3	0	4	2	3	0	3	2	1	2	0	1	7	0	1
7	7	7	2	2	2	2	2	2	2	2	2	2	2	2	2	2	7	7	7
	0	0	0	0	0	0	4	4	4	0	0	0	0	0	0	0	0	0	0
		1	1	1	3	3	3	3	3	3	3	3	1	1	1	1	1	1	1
F	F	F	F		F		F			F			F				F		

9 Page Faults

- It has the lowest page-fault rate of all algorithms and will never suffer from Belady's anomaly
- How do you know this?
  - Can't read the future
- Used for measuring how well your algorithm performs

Least Recently Used (LRU) Algorithm

- Use past knowledge rather than future
- Replace page that has not been used in the most amount of time
- Associate time of last use with each page



Least Recently Used (LRU) Algorithm

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

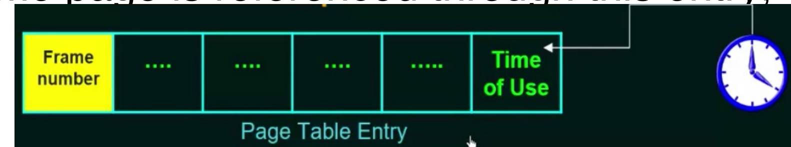
12 Page Faults

- 12 faults – better than FIFO but worse than OPT
- Generally good algorithm and frequently used
- But how to implement?

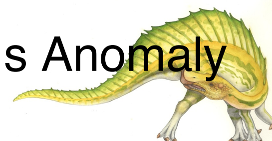
LRU Algorithm (Cont.)

Two ways to implement LRU:

- **Counter implementation**
  - Every page entry has a counter; every time page is referenced through this entry, copy the clock into the counter



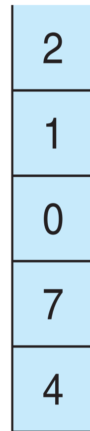
  - When a page needs to be changed, look at the counters to find smallest value
    - Search through table needed
- **Stack implementation**
  - Keep a stack of page numbers in a doubly linked list form:
  - Page referenced:
    - move it to the top
    - requires 6 pointers to be changed
  - But each update more expensive
  - No search for replacement
- LRU and OPT are cases of **stack algorithms** that don't have Belady's Anomaly



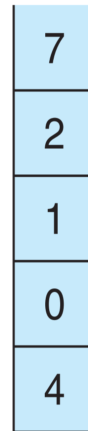
Use Of A Stack to Record Most Recent Page References

reference string

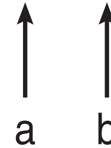
4 7 0 7 1 0 1 2 1 2 7 1 2



stack
before
a



stack
after
b



Thrashing

- **Thrashing** occurs when virtual memory system is in a constant state of paging,
- Rapidly exchanging data in memory for data on disk, to the exclusion of most application-level processing.
- This causes the performance of the computer to degrade or collapse greatly.
- The set of pages that a process is currently using is its **working set .**
- If the entire working set is in memory, the process will run without causing many faults until it moves into another execution phase (e.g., the next pass of the compiler).
- If the available memory is too small to hold the entire working set, the process will cause many page faults and run slowly,

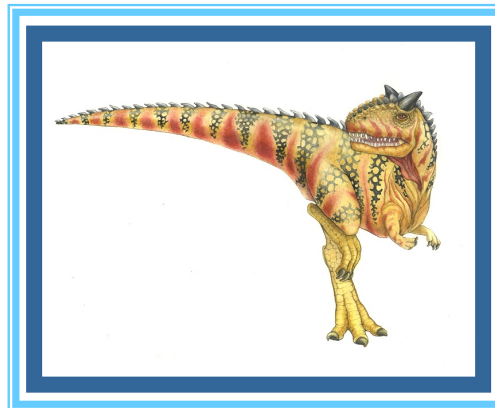
Thrashing

- If a process does not have "enough" pages, the page-fault rate is very high
  - Page fault to get page
  - Replace existing frame
  - But quickly need replaced frame back
  - This leads to:
    - Low CPU utilization
    - Operating system thinking that it needs to increase the degree of multiprogramming
    - Another process added to the system
- **Thrashing** ≡ a process is busy swapping pages in and out

Thrashing

- Since executing an instruction takes a few nanoseconds.
- and reading in a page from the disk typically takes 10 msec.
- At a rate of one or two instructions per 10 msec, it will take ages to finish.
- A program causing page faults every few instructions is said to be **thrashing**

End of Chapter 9



Silberschatz, Galvin and Gagne ©2013