Chapter 9: Virtual Memory

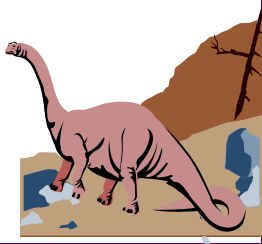
肖 卿 俊
办公室：九龙湖校区计算机楼212室
电邮：csqjxiao@seu.edu.cn
主页： https://csqjxiao.github.io/PersonalPage
电话：025-52091022

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
- To discuss the principle of working-set model
- To explain the IPC model based on memory sharing; To examine the differences between shared memory and memory-mapped files
- To explore how kernel memory is managed

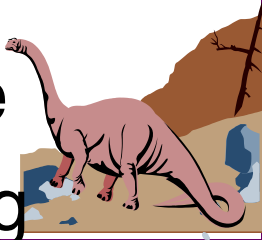
Chapter 9: Virtual Memory

- Background
- Demand Paging
- Copy-on-Write
- Page Replacement within a Process
- Allocation of Frames among Processes
- Thrashing and Working Set Model
- Memory-Mapped Files
- Allocating Kernel Memory
- Other Considerations
- Operating-System Examples



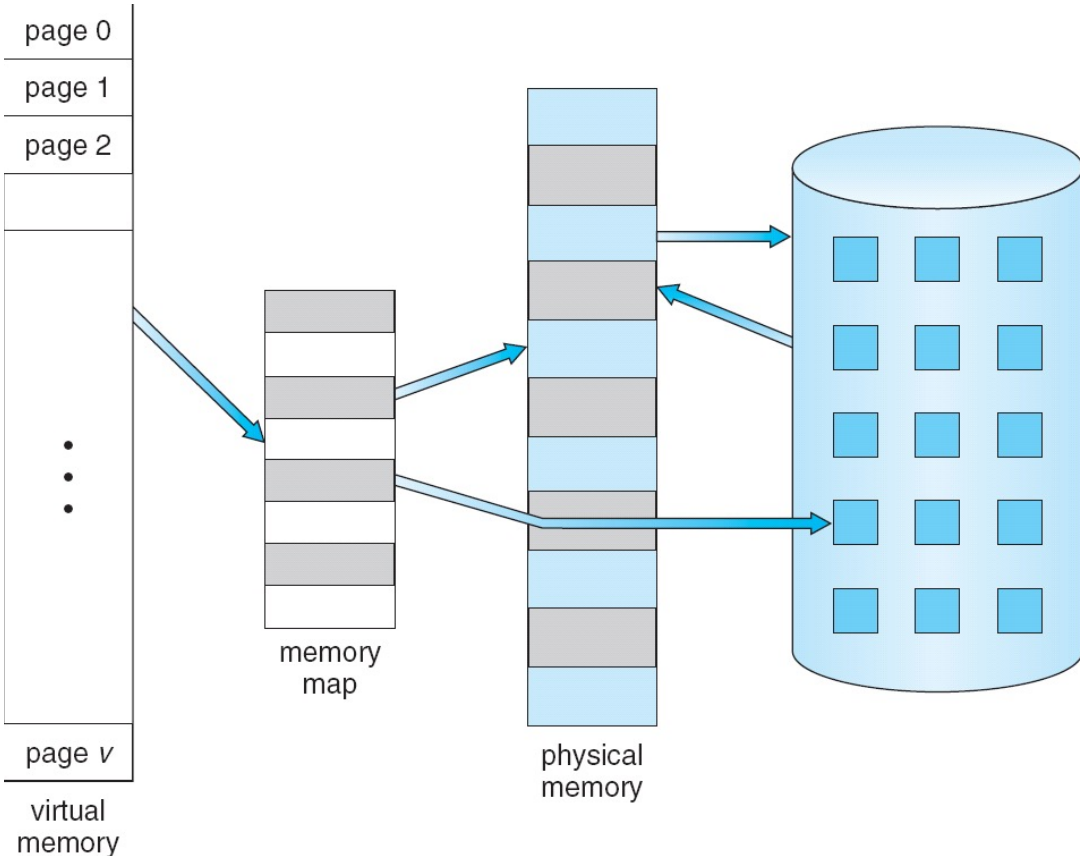
Background

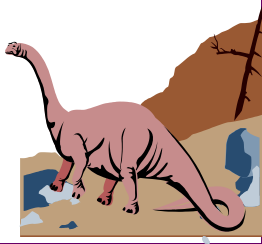
- We previously talked about an entire process swapping into or out of main memory
- **Idea of memory virtualization** – separation of memory address spaces of all user processes
- **Idea of virtual memory** – separation of user logical memory from physical memory.
  - Only part of the program needs to be kept in main memory for execution.
  - Logical address space can therefore be much larger than physical address space.
  - More programs can be run at the same time
  - Less I/O is needed than loading or swapping



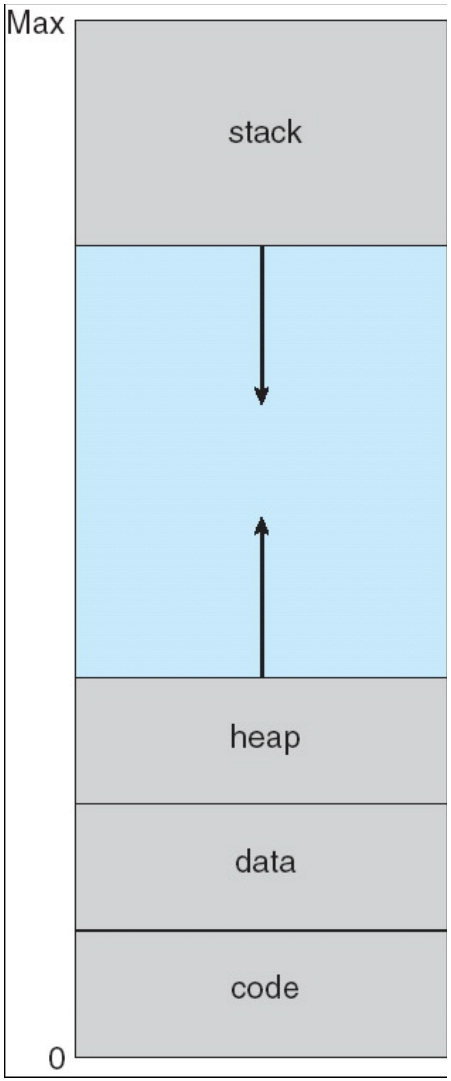
Two Kinds of Implementation for Virtual Memory

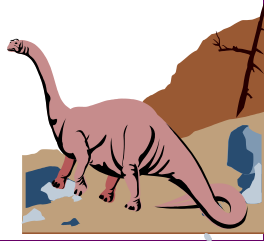
- Virtual memory can be implemented via:
  - Demand paging（按需调页）
  - Demand segmentation（按需调段）



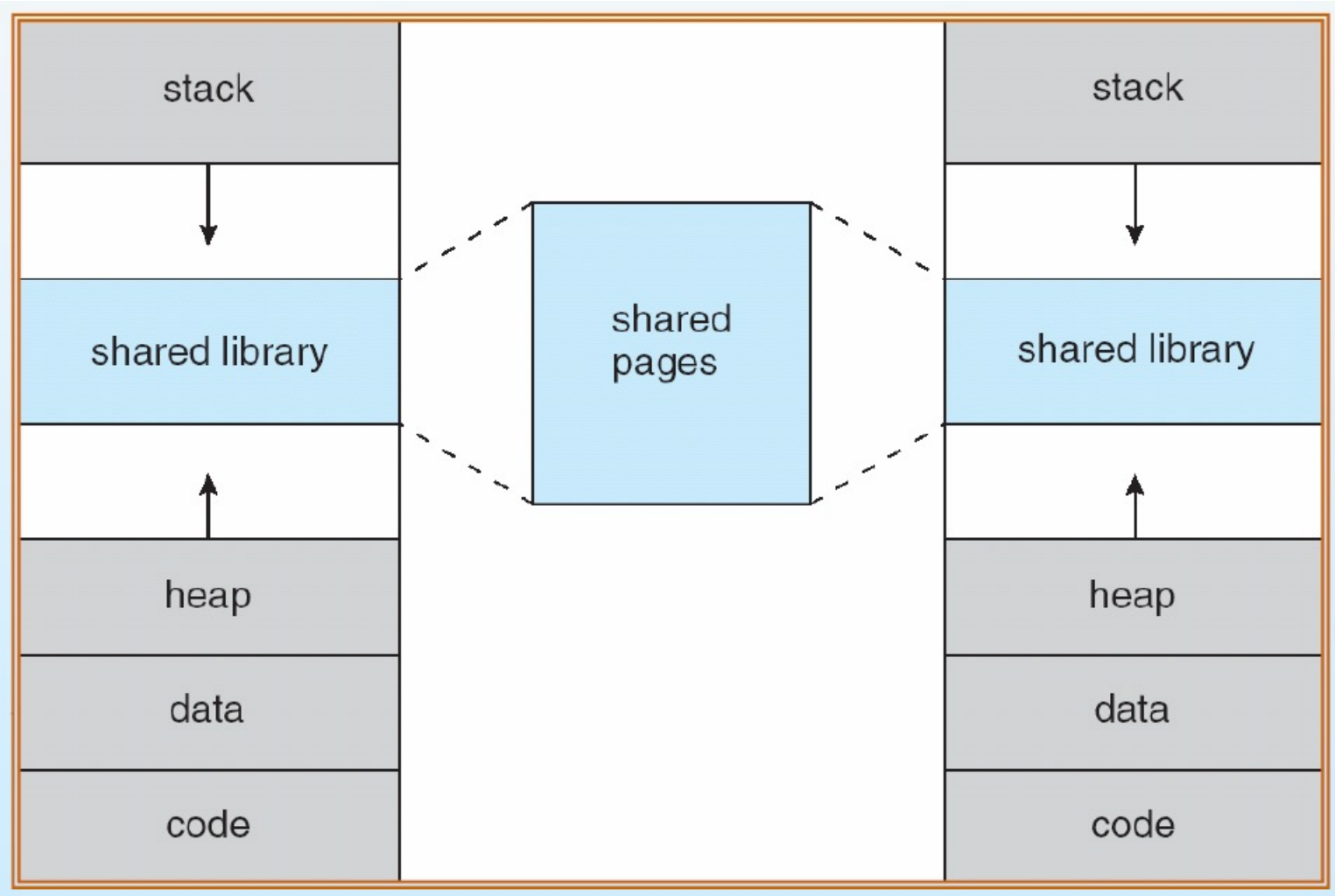


Virtual Address Space with Segmentation



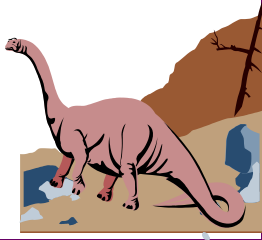


Shared Library Using Virtual Memory with a Shared Segment



Chapter 9: Virtual Memory

- Background
- Demand Paging
- Copy-on-Write
- Page Replacement within a Process
- Allocation of Frames among Processes
- Thrashing and Working Set Model
- Memory-Mapped Files
- Allocating Kernel Memory
- Other Considerations
- Operating-System Examples



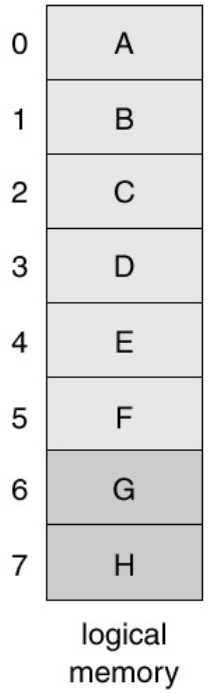
Demand Paging

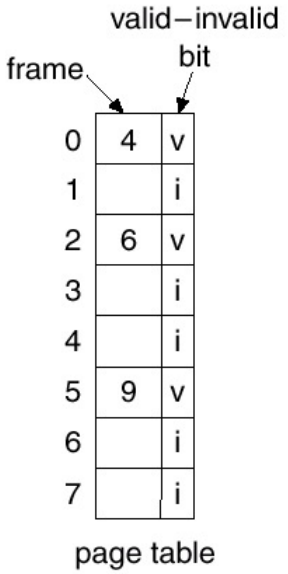
- Bring a page into memory only when it is needed.
  - Less I/O needed
  - Less memory needed
  - Faster response
  - More users
- Page is needed ⇒ reference to it
  - invalid reference ⇒ abort
  - not-in-memory ⇒ bring to memory
- **Pure demand paging**– never bring a page into memory unless page will be needed

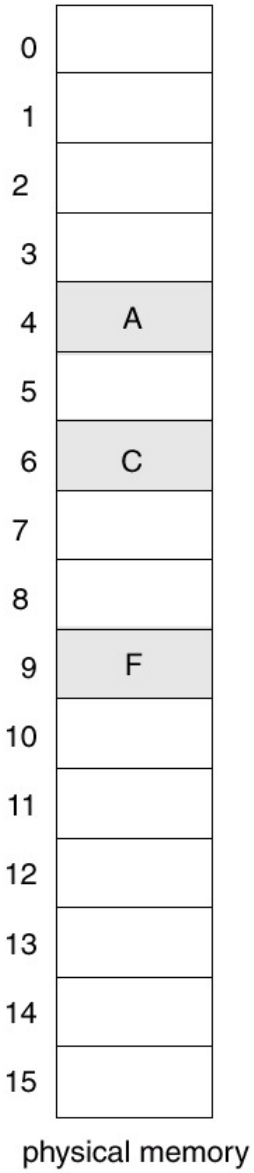
Valid-Invalid Bit

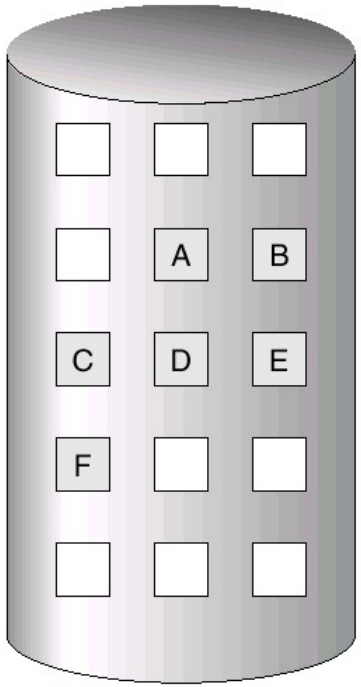
- With each page table entry, a valid-invalid bit is associated
  - (1 ⇒ in-memory, 0 ⇒ not-in-memory)
- Initially, valid-invalid bit is set to 0 on all entries.
- During address translation, if valid-invalid bit in page table entry is 0 ⇒ page fault (**缺页中断**)

Page Table When Some Pages Are Not in Main Memory



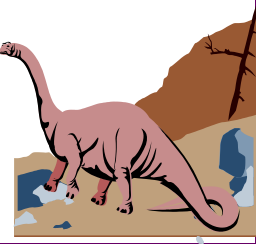




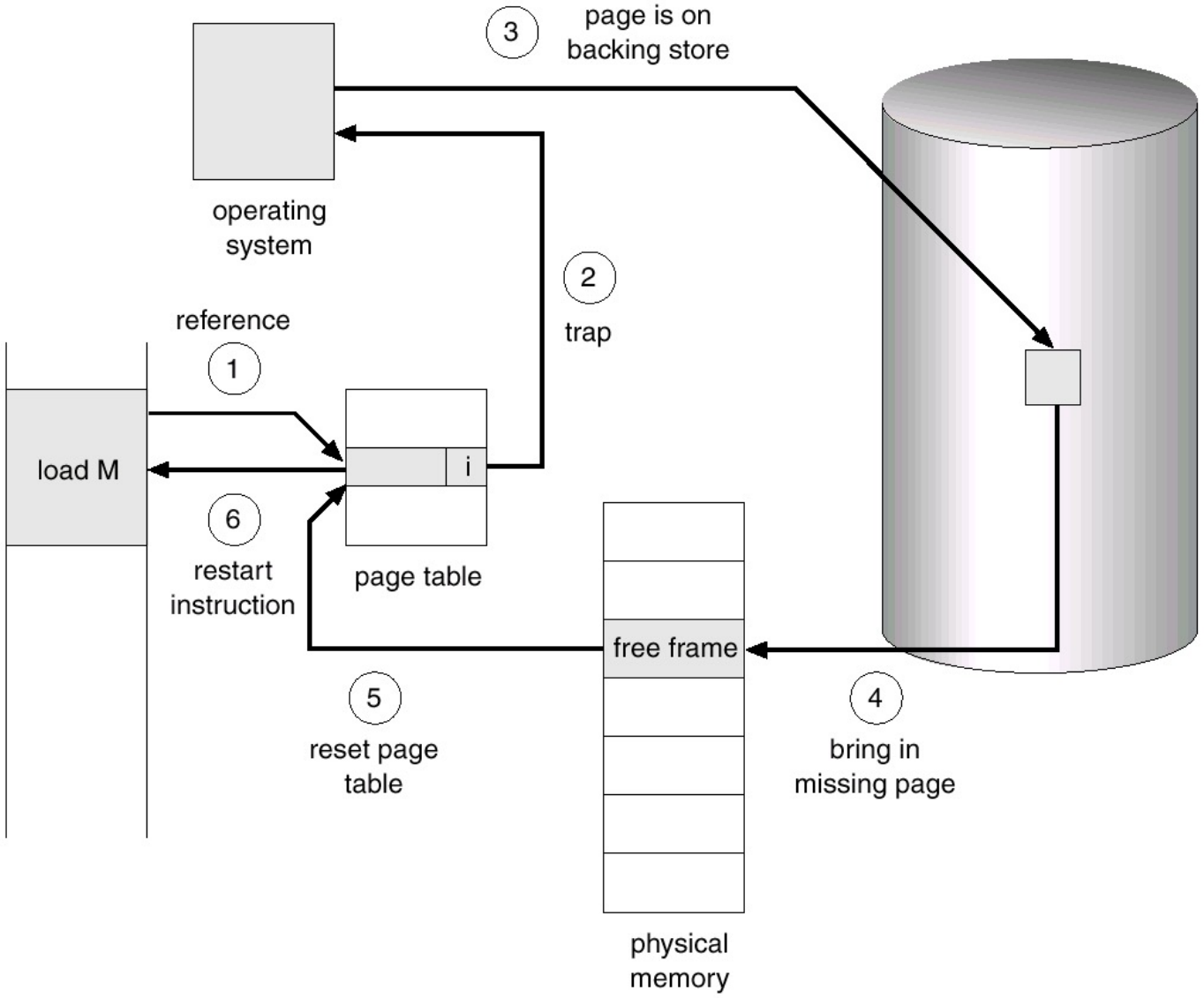


Steps in Handling a Page Fault

- If there is ever a reference to a page, first reference will trap to OS kernel ⇒ page fault
- OS looks at another table to decide:
  - Invalid reference ⇒ abort.
  - Just not in memory.
- Get empty frame.
- Swap page into frame.
- Reset tables, validation bit = 1.
- Restart instruction



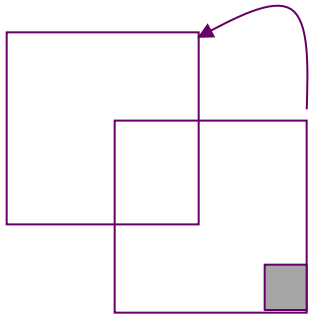
Steps in Handling a Page Fault



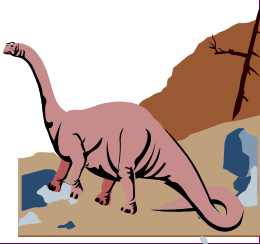
More Details about Restarting an Instruction

- The restart will require fetching instruction again, decoding it again, fetching the two operands again, and applying it again

- Difficulty arises when an instruction may modify multiple virtual pages
  - For example, block move operation



  - Auto increment/decrement location
  - Restart the whole operation?
    - ✓What if source and destination overlap?
    - ✓The source may have been modified

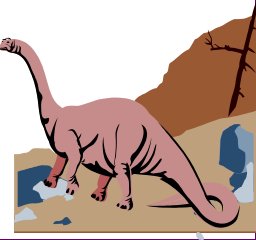


Performance of Demand Paging

- Page Fault Rate $0 \le p \le 1.0$
  - if $p = 0$, no page faults
  - if $p = 1$, every reference is a fault

- Effective Access Time (EAT)

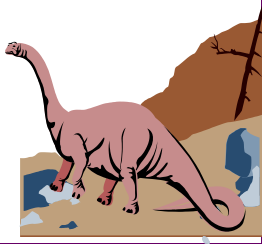
  EAT = (1 – p) x memory access
  + p x (page fault overhead
  [+ swap page out]
  + swap page in
  + restart overhead)



Demand Paging Example

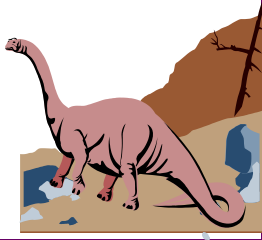
- Memory access time = 1 microsecond
- Swap Page Time = 10 millisec = 10000 microsec
- Assume 50% of the time the page that is being replaced has been modified and therefore needs to be swapped out.
- Ignore the cost of restarting an instruction.

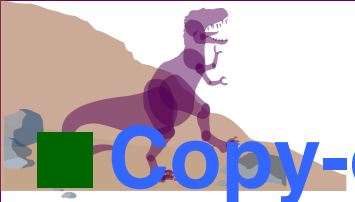
- EAT = (1 - p) x 1 + p x (10000*50%+20000*50%)
  = (1 - p) x 1 + p x (15000)
  = 1 + 14999 x p (in microsecond)



Chapter 9: Virtual Memory

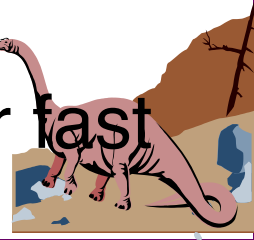
- Background
- Demand Paging
- Copy-on-Write
- Page Replacement within a Process
- Allocation of Frames among Processes
- Thrashing and Working Set Model
- Memory-Mapped Files
- Allocating Kernel Memory
- Other Considerations
- Operating-System Examples



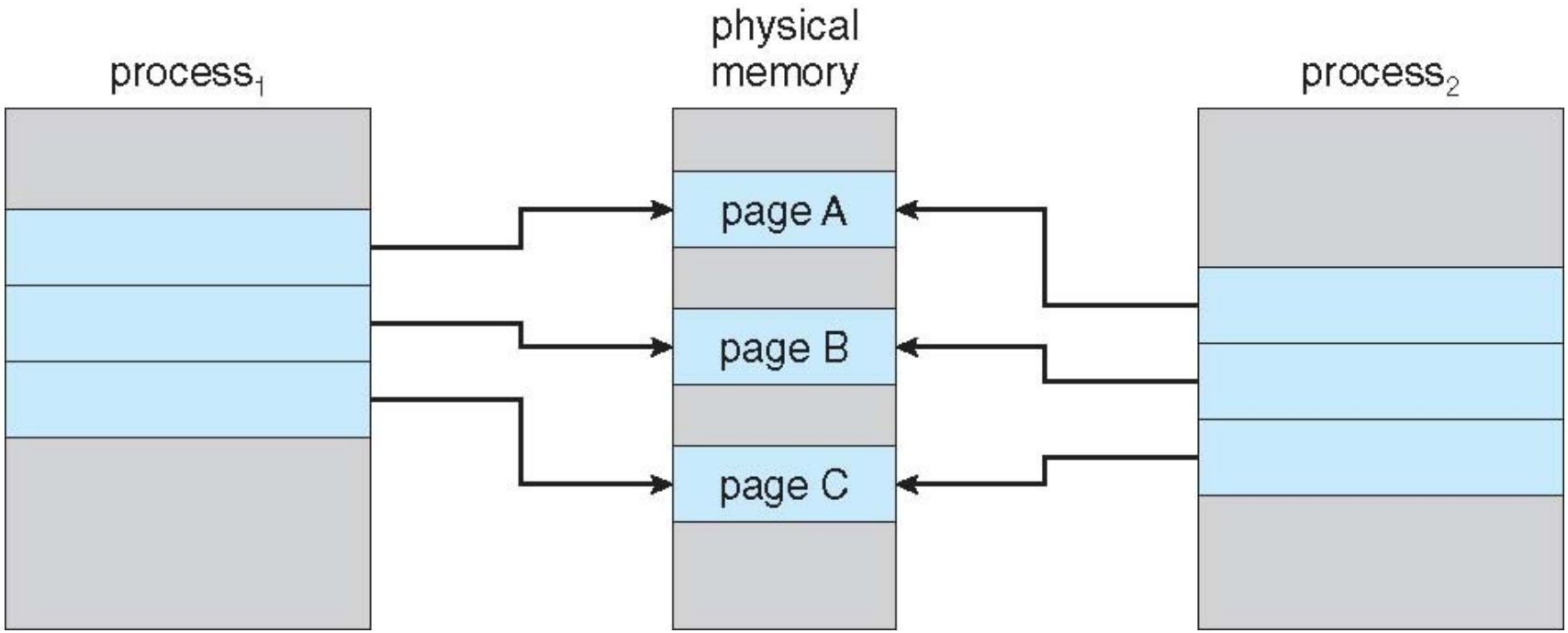


Copy-on-Write

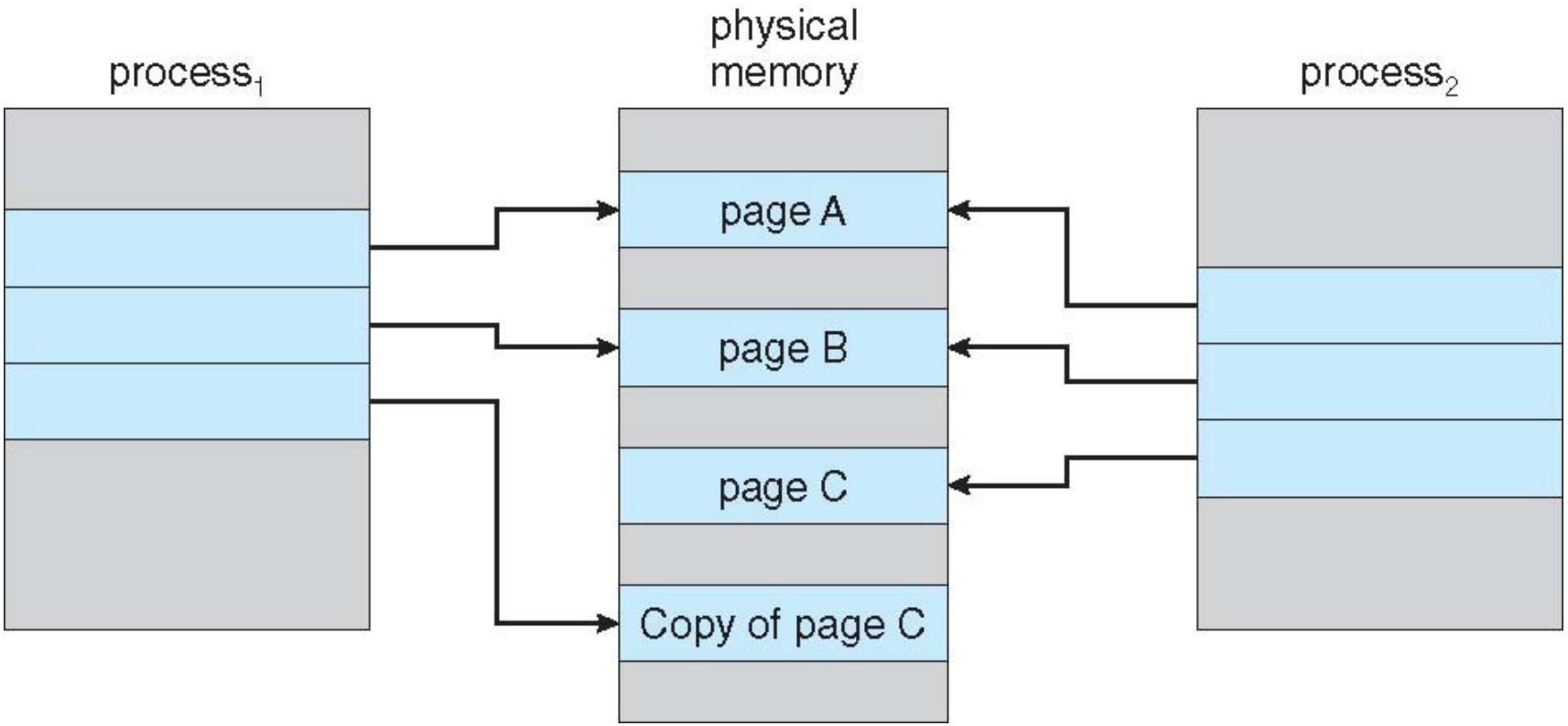
- **Copy-on-Write** (COW) allows both parent and child processes to initially ***share*** the same pages in memory
  - If either process modifies a shared page, only then is the page copied
- COW allows more efficient process creation as only modified pages are copied
- Free pages are allocated from a **pool** of **zero-fill-on-demand** pages
  - Why do we need to zero-out a page before allocating it to a process?
  - The pool should always have free frames for fast demand page execution



Before Process 1 Modifies Page C



After Process 1 Modifies Page C

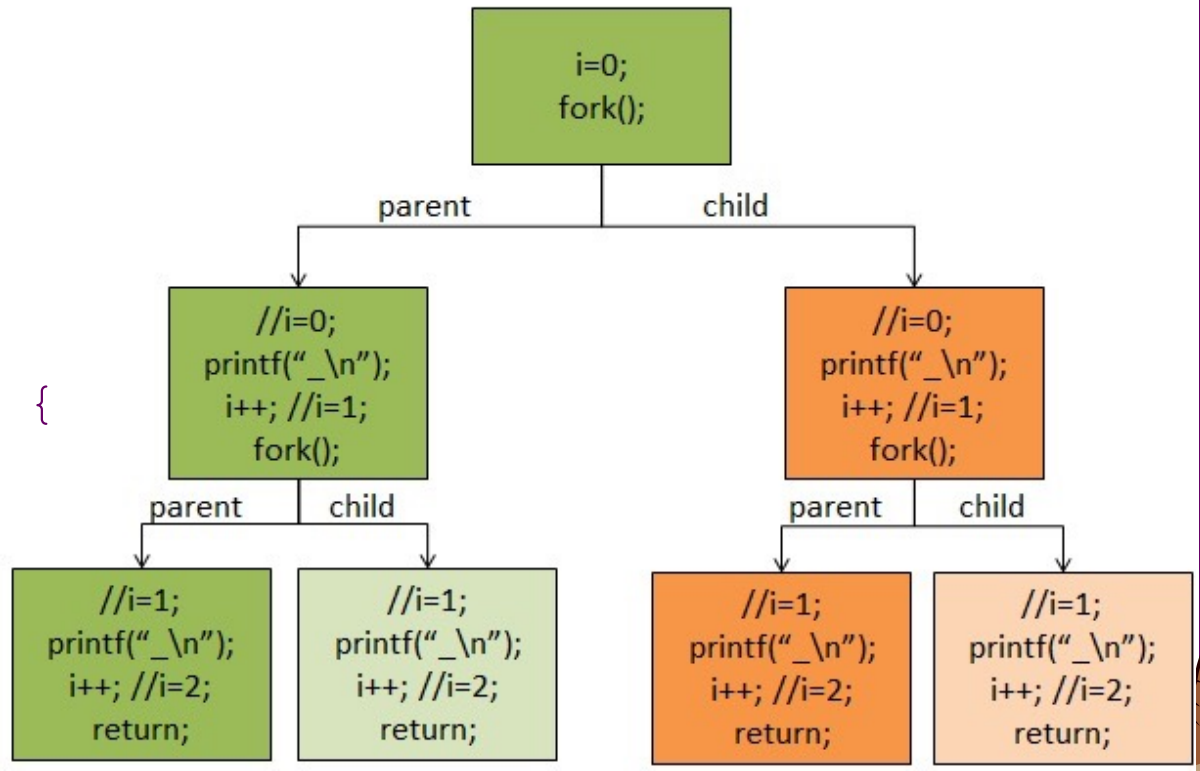


复习Linux fork系统调用

- 问题：Linux fork()系统调用实现了什么功能，返回值含义是什么？请问下面的代码一共输出多少个“_”？请解释原因。

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int main(void) {
    int i;
    for(i=0; i<2; i++) {
        fork();
        printf("_\n");
    }
    wait(NULL);
    wait(NULL);
}
```



fork() and vfork()

- `vfork()`, a variation of `fork()` system call, has the parent suspend and the child without copying the page table of the parent
  - Useful in performance-sensitive applications where a child is created which then immediately issues an `execve()`.
- **以前的fork很低效，它创建一个子进程时，将会创建一个新的地址空间，并且拷贝父进程的资源，而往往在子进程中会执行exec调用，这样，前面的拷贝工作就是白费了。于是，设计者就想出了vfork，它产生的子进程刚开始暂时与父进程共享地址空间（其实就是线程的概念了）。因为这时候子进程在父进程的地址空间中运行，所以子进程不能进行写操作，并且在儿子“霸占”着老子的房子时候，要委屈父亲一下了，让他在外面歇着（阻塞），一旦儿子执行了execve 或者exit 后，相于儿子买了自己的房子了，这时候就相于分家了。**

An Example of fork() and vfork()

```
int main() {
    pid_t pid;
    int cnt = 3;
    pid = fork();
    if(pid<0)
        printf("error in fork!\n");
    else if(pid == 0) {
        cnt++;
        printf("Child process %d, ",getpid());
        printf("cnt=%d\n",cnt);
    } else {
        cnt++;
        printf("Parent process %d,",getpid());
        printf("cnt=%d\n",cnt);
    }
    return 0;
}
```

Execution Result:

Child process 5077, cnt=4

Parent process 5076, cnt=4

If we replace line 4 by pid = vfork(), then

Execution Result:

Child process 5077, cnt=4

Parent process 5076, cnt=1

Segmentation fault: 11

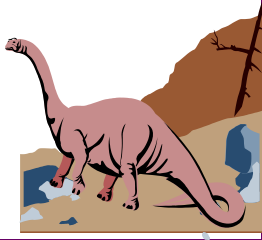
Question: If the cnt variable on stack is shared between parent and child processes, why do we still see cnt =1?

Answer: vfork() differs from fork() in that the calling thread is suspended until the child terminates (either normally by exit() or abnormally after a fatal signal), or it makes a call to execve(). Until that point, the child shares all memory with its parent, including the stack.

Question: What if we insert a command exit(0) before the line "} else {" ?

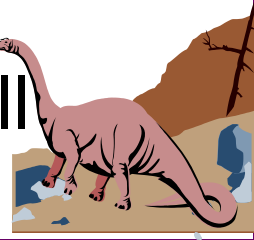
Chapter 9: Virtual Memory

- Background
- Demand Paging
- Copy-on-Write
- Page Replacement within a Process
- Allocation of Frames among Processes
- Thrashing and Working Set Model
- Memory-Mapped Files
- Allocating Kernel Memory
- Other Considerations
- Operating-System Examples



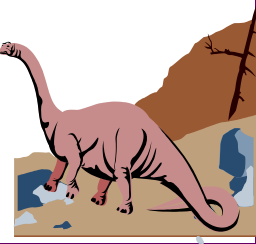
What Happens if There are no Free Frames?

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

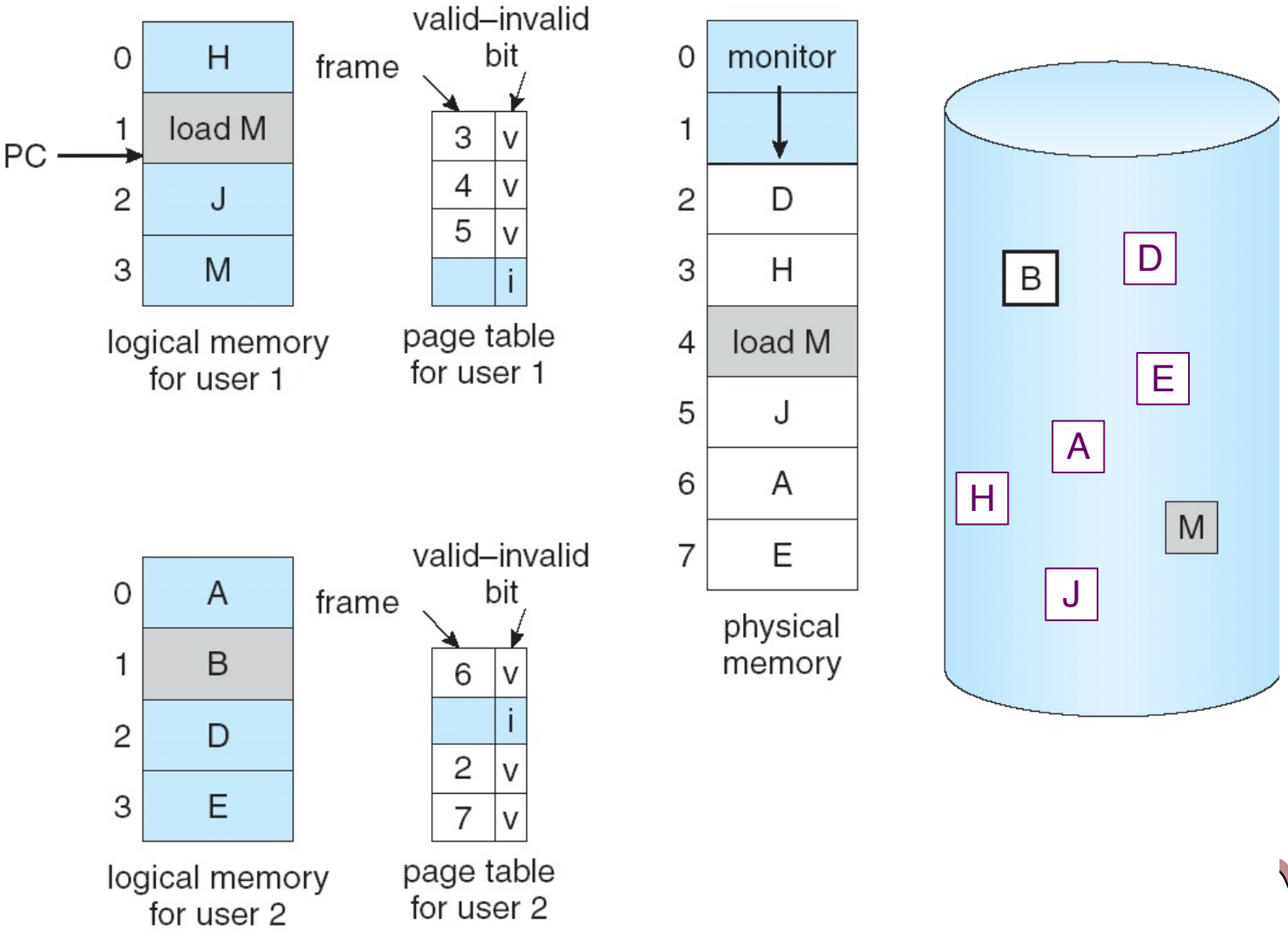


Page Replacement

- Prevent over-allocation of memory by modifying page-fault service routine to include page replacement.
- Use *modify* (*dirty*) *bit* to reduce the 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.



Need For Page Replacement



Basic Page Replacement

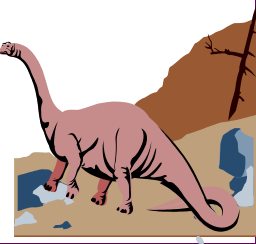
1. Find the location of the desired page on disk.

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 and swap it out

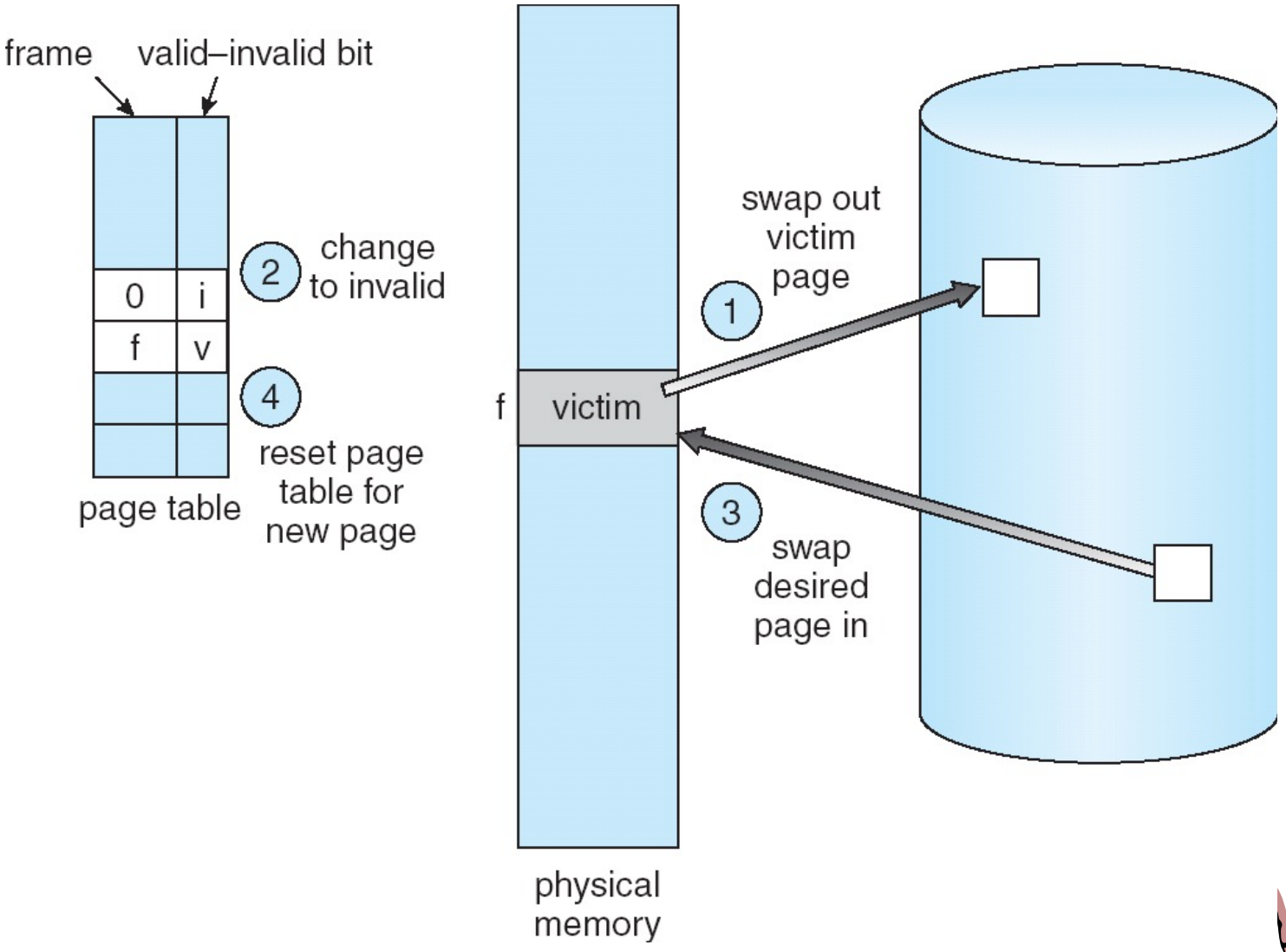
3. Read the desired page into the free frame.

4. Update the page and frame tables.

5. Restart the instruction.



Page Replacement



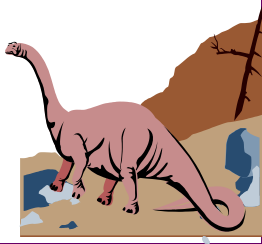
Page Replacement Algorithms

- Key objective: Want the lowest page-fault rate

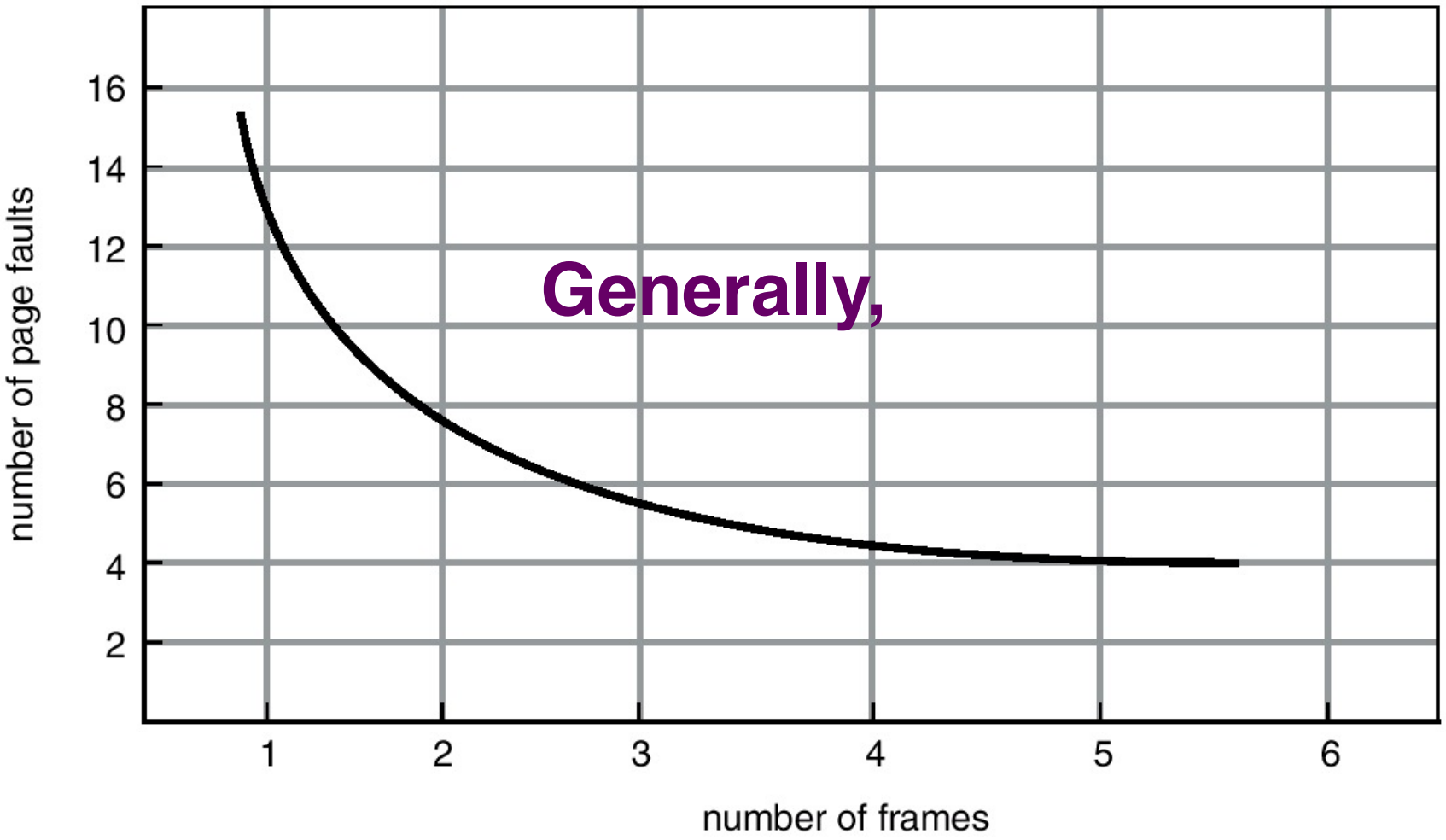
- Evaluate algorithm by running it on a particular string of memory references (reference string) and computing the number of page faults on that string.

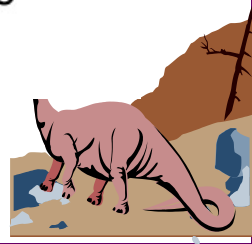
- In all our examples, the reference string is

  1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5.



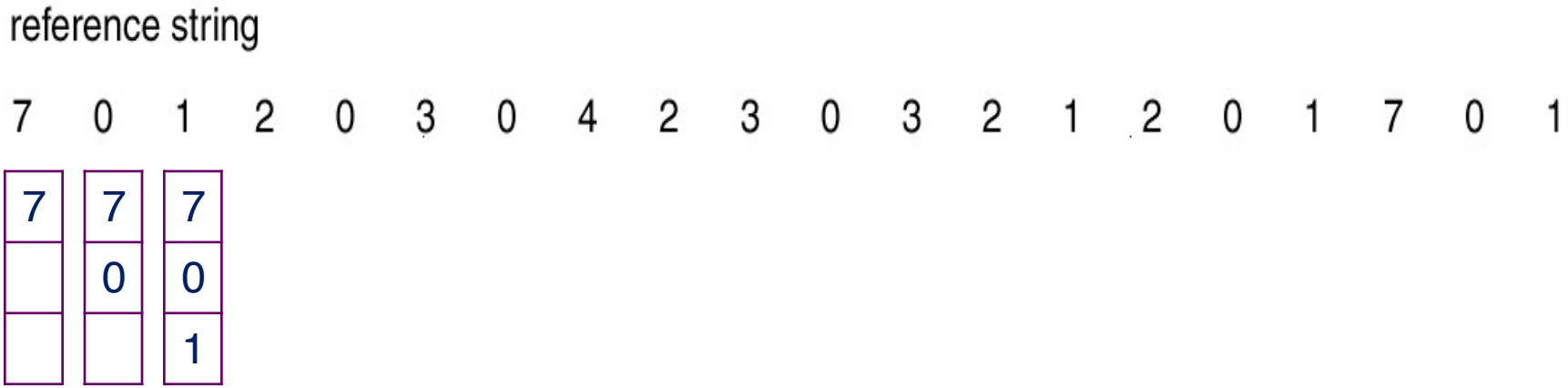
The Number of Page Faults vs. The Number of Frames





First-In-First-Out (FIFO) Page Replacement

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

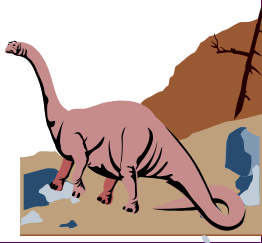


FIFO Page Replacement

reference string

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

3 page frames



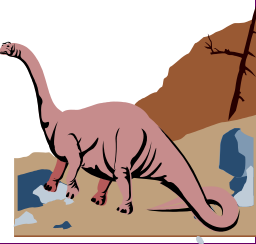
Belady's Anomaly for FIFO Algorithm

- Reference string: 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5
- In all our examples, the reference string is

  1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5.

- Where there are 3 frames (3 pages can be in memory at a time per process)

1	1	4	5	
2	2	1	3	9 page faults
3	3	2	4	

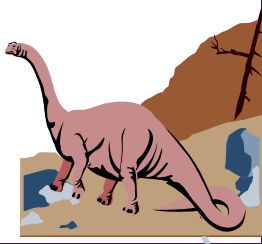


Belady's Anomaly for FIFO Algorithm

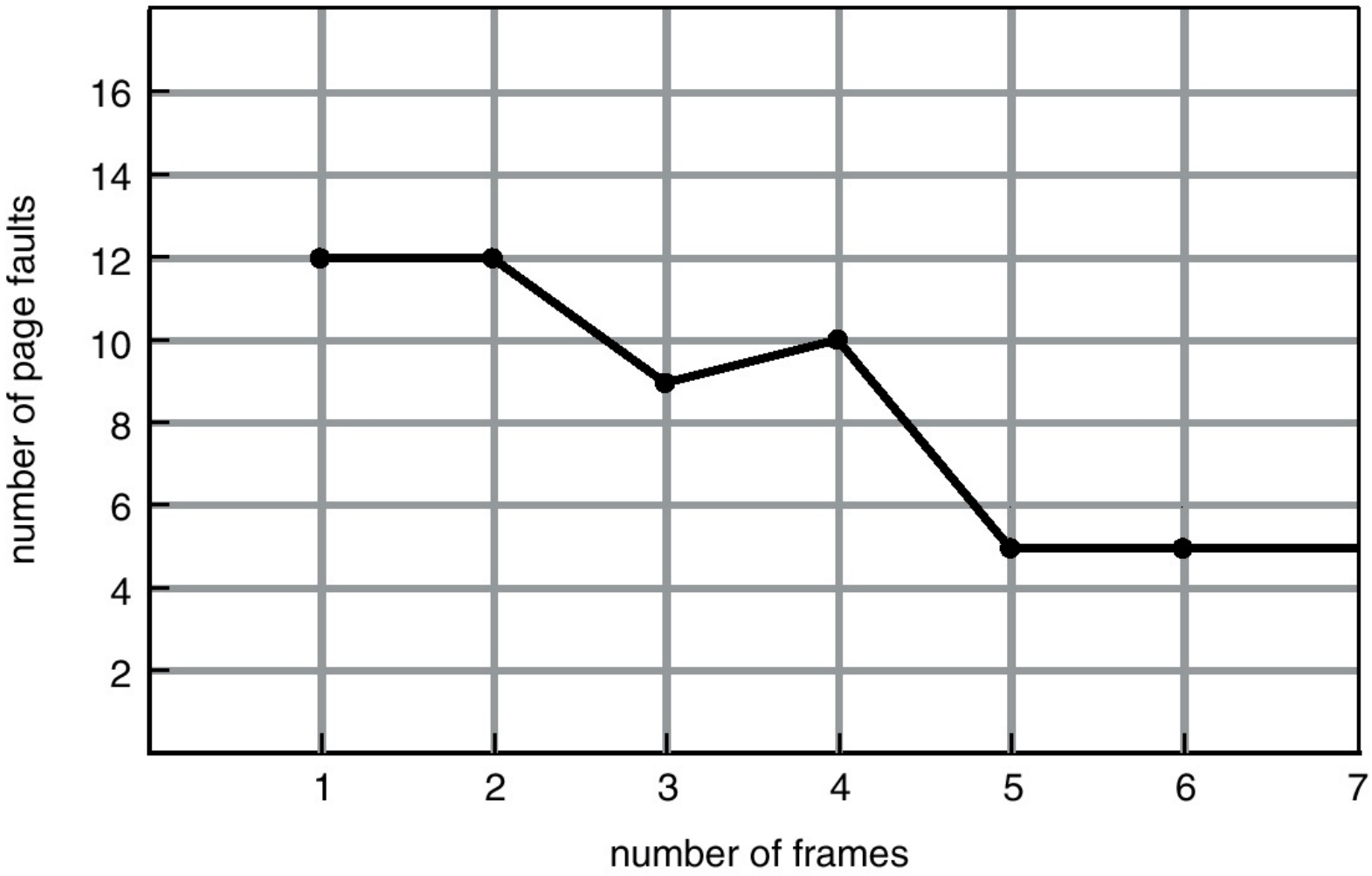
- When there are 4 frames

1	1	5	4	
2	2	1	5	10 page faults
3	3	2		
4	4	3		

- FIFO Replacement – Belady's Anomaly
  - Supposedly, more frames $\Rightarrow$ less page faults
  - However, see the next page



FIFO Illustrating Belady's Anamoly



Optimal Algorithm

- Replace page that will not be used for the longest period of time.

- 4 frames example

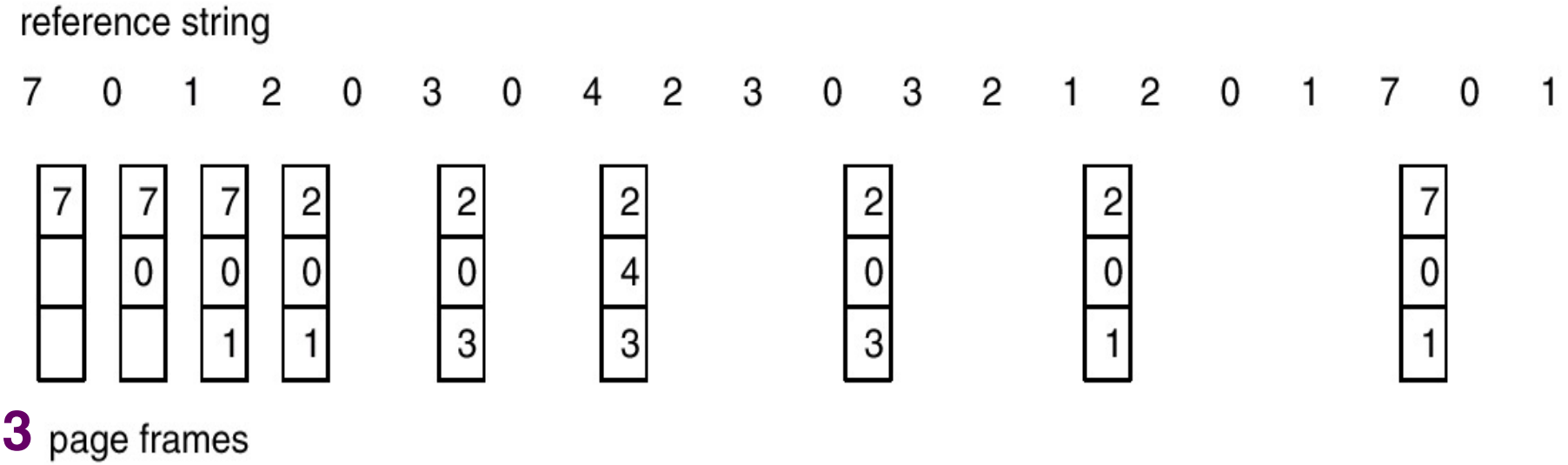
1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5

1	4
2	
3	
4	5

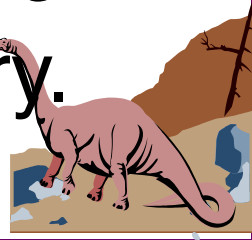
6 page faults

- Need to know the pattern of future memory accesses. So used only for measuring how well your page replacement algorithm performs.

Optimal Page Replacement

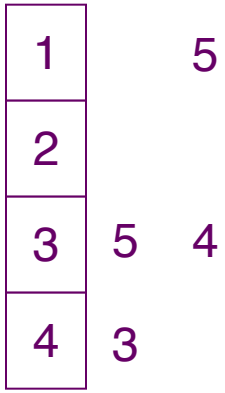


- Replace page that will not be used for the longest period of time.
- But how do you know the future information? What we can know is only the past history.



Least Recently Used (LRU) Algorithm

- Discards the least recently used items first.
- Reference string: 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5

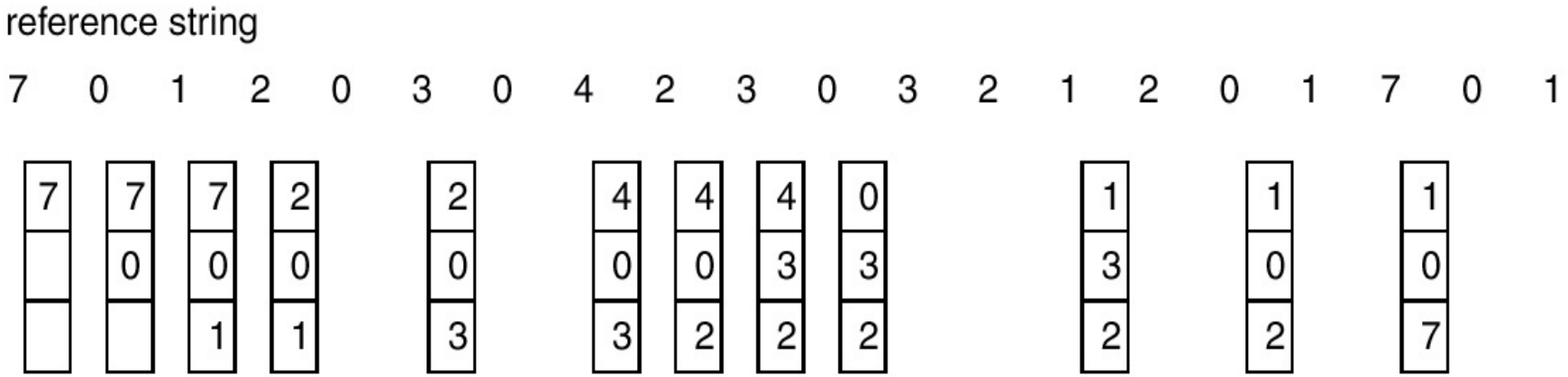


8 page faults,

better than FIFO (10 faults) and

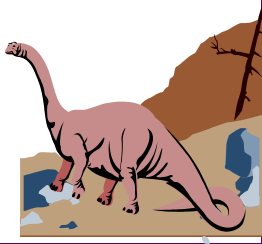
worse than the optimal (6 faults)

- Another Example of LRU



LRU Algorithm Implementations

- 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 determine which are to change.
- Stack implementation – keep a stack of page numbers in a doubly linked list:
  - When a page is referenced:
    - ✓ move it to the top
    - ✓ requires 6 pointers to be changed
  - No search for page replacement

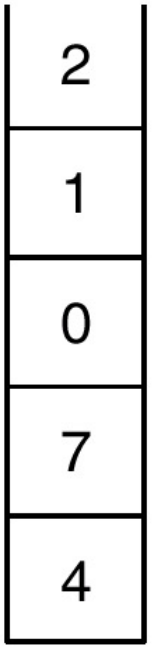


Use A Stack to Record The Most Recent Page References

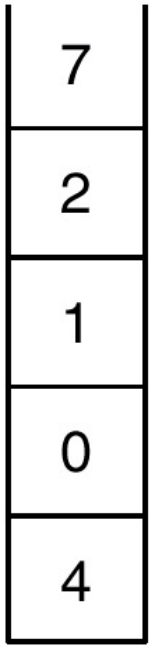
reference string

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

a b



stack before a

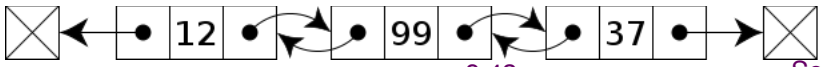


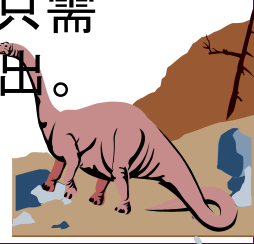
stack after b

请问栈里面访问特定页面号的计算复杂度是多少？ 是O(1)还是O(n)？假设n是栈里的元素个数。

设计和实现O(1)计算复杂度的LRU 缓存机制的数据结构

- 问题：设计和实现一个 LRU (最近最少使用）缓存数据结构，支持 get 和 put 操作，要求O(1)复杂度
  - get(key) - 如果 key 存在于缓存中，则获取 key 的 value(总是正数），否则返回 -1。
  - put(key, value) - 如果 key 不存在，请设置或插入 value。当缓存达到容量时，它应该在插入新项目之前使最近最少使用的项目作废
- 分析：
  - 需要使用的数据结构是哈希表（Hash Table）。基础的哈希表虽具备读写 key-value 数据的功能，但是 key 的存储是无序的。
  - 而本题中当 LRU 存满时，再次存储时需要删除掉最久未使用的数据。
  - 所以，用哈希表和链表两个数据结构。当进行 set & get 操作时，只需把当前节点调整到链表尾，而需要 pop 操作的时候，将链表首弹出。





关于LRU算法的动手小实验

- 该O(1)复杂度的LRU问题来自于leecode：https://leetcode.com/problems/lru-cache

- 一个可能解法来自https://github.com/lamerman/cpp-lru-cache。请从Github上clone该项目。并把该代码仓库编译运行起来。

- 提示：该代码的编译需要cmake工具，并引用单元测试框架googletest，需要命令行下的梯子代理才能解决googletest的下载问题。

- 请问这段代码中这个iterator是什么作用？
- 分析get和put方法的算法复杂度。

```
19 class lru_cache {
20 public:
21     typedef typename std::pair<key_t, value_t> key_value_pair_t;
22     typedef typename std::list<key_value_pair_t>::iterator list_iterator_t;
23
24     lru_cache(size_t max_size) :
25         _max_size(max_size) {
26     }
27
28     void put(const key_t& key, const value_t& value) {
29         auto it = _cache_items_map.find(key);
30         _cache_items_list.push_front(key_value_pair_t(key, value));
31         if (it != _cache_items_map.end()) {
32             _cache_items_list.erase(it->second);
33             _cache_items_map.erase(it);
34         }
35         _cache_items_map[key] = _cache_items_list.begin();
36
37         if (_cache_items_map.size() > _max_size) {
38             auto last = _cache_items_list.end();
39             last--;
40             _cache_items_map.erase(last->first);
41             _cache_items_list.pop_back();
42         }
43     }
44
45     const value_t& get(const key_t& key) {
46         auto it = _cache_items_map.find(key);
47         if (it == _cache_items_map.end()) {
48             throw std::range_error("There is no such key in cache");
49         } else {
50             _cache_items_list.splice(_cache_items_list.begin(), _cache_items_list, it->second);
51             return it->second->second;
52         }
53     }
54
55     bool exists(const key_t& key) const {
56         return _cache_items_map.find(key) != _cache_items_map.end();
57     }
58
59     size_t size() const {
60         return _cache_items_map.size();
61     }
62
63 private:
64     std::list<key_value_pair_t> _cache_items_list;
65     std::unordered_map<key_t, list_iterator_t> _cache_items_map;
66     size_t _max_size;
```

这段代码中的put方法任何情况都首先push_front。如果key在map里面找到了，再用erase接口把这个key删除掉。请问在key存在的这种情况下，能否避免erase。可否直接修改list里面的key的对应节点的value，避免erase调用

```
19 class lru_cache {
20 public:
21     typedef typename std::pair<key_t, value_t> key_value_pair_t;
22     typedef typename std::list<key_value_pair_t>::iterator list_iterator_t;
23
24     lru_cache(size_t max_size) :
25         _max_size(max_size) {
26     }
27
28     void put(const key_t& key, const value_t& value) {
29         auto it = _cache_items_map.find(key);
30         _cache_items_list.push_front(key_value_pair_t(key, value));
31         if (it != _cache_items_map.end()) {
32             _cache_items_list.erase(it->second);
33             _cache_items_map.erase(it);
34         }
35         _cache_items_map[key] = _cache_items_list.begin();
36
37         if (_cache_items_map.size() > _max_size) {
38             auto last = _cache_items_list.end();
39             last--;
40             _cache_items_map.erase(last->first);
41             _cache_items_list.pop_back();
42         }
43     }
44
45     const value_t& get(const key_t& key) {
46         auto it = _cache_items_map.find(key);
47         if (it == _cache_items_map.end()) {
48             throw std::range_error("There is no such key in cache");
49         } else {
50             _cache_items_list.splice(_cache_items_list.begin(), _cache_items_list, it->second);
51             return it->second->second;
52         }
53     }
54
55     bool exists(const key_t& key) const {
56         return _cache_items_map.find(key) != _cache_items_map.end();
57     }
58
59     size_t size() const {
60         return _cache_items_map.size();
61     }
62
63 private:
64     std::list<key_value_pair_t> _cache_items_list;
65     std::unordered_map<key_t, list_iterator_t> _cache_items_map;
66     size_t _max_size;
```

LRU代码的改进方法

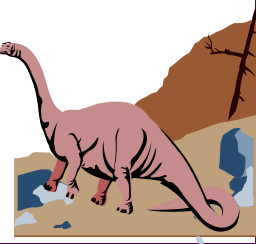
- 如果map中找不到的key，代码不用改变；如果map中找到该key，首先将结点的value赋予新值，然后使用splice函数将结点移动到链表头。

```
void put(const key_t& key, const value_t& value) {
    auto it = _cache_items_map.find(key);
    if (it != _cache_items_map.end()) {
        it->second->second = value;
        _cache_items_list.splice(_cache_items_list.begin(), _cache_items_list, it->second);
    }
    else{
        _cache_items_list.push_front(key_value_pair_t(key, value));
    }
    _cache_items_map[key] = _cache_items_list.begin();

    if (_cache_items_map.size() > _max_size) {
        auto last = _cache_items_list.end();
        last--;
        _cache_items_map.erase(last->first);
        _cache_items_list.pop_back();
    }
}
```

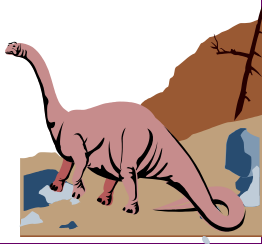
Problems of Previous LRU Implementations

- As to the previous two LRU implementations,
  - Clock: Every page entry has a counter; every time page is referenced through this entry, copy the clock into the counter.
  - Stack: Whenever a page is referenced, it is removed from the stack and put on the top.
- The updating of the clock fields or stack must be done for every memory reference
- Would slow every memory access by a factor of at least ten



LRU Approximation Algorithms

- Reference bit per page (Hardware maintained)
  - Each page is associated with a bit in the page table
  - Initially 0; When page is referenced, set the bit to 1.
  - Replace the one which is 0 (if one exists)
- However, we do not know the order of use.
- This information is the basis for many page-replacement algorithms that approximate LRU replacement



LRU Approximation Algorithms

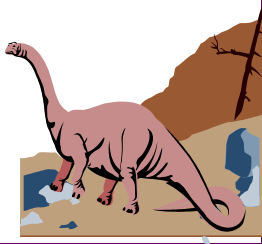
- Rational: Gain additional ordering information by recording the reference bits at regular intervals

- Additional-Reference-Bits Algorithm
  - Keep an 8-bit bytes for each page in main memory
  - At regular intervals, shifts the bits right 1 bit, shift the reference bit into the lower-order bit
  - Interpret these 8-bit bytes as unsigned integers, the page with lowest number is the LRU page
  - E.g., the page with ref bits 11000100 is more recently used than the page with ref bits 01110111

An Example of Additional-Reference-Bits Algorithm (1)

- Assume the following page reference string, where T marks the end of each time interval: 3, 2, 3, T, 8, 0, 3, T, 3, 0, 2, T, 6, 3, 4, 7
- Assume there are 5 frames in memory, and each frame has a Page field (P) and 4 used bits (U3, U2, U1, and U0).
- Initial State

P	U3	U2	U1	U0
-	0	0	0	0
-	0	0	0	0
-	0	0	0	0
-	0	0	0	0
-	0	0	0	0



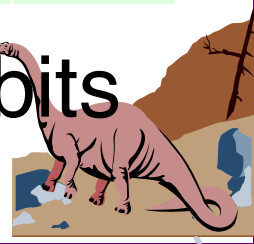
An Example of Additional-Reference-Bits Algorithm (2)

- Assume the following page reference string: 3, 2, 3, T, 8, 0, 3, T, 3, 0, 2, T, 6, 3, 4, 7
- During the first time interval, pages 3, 2, and 3 are referenced.

P	U3	U2	U1	U0
3	1	0	0	0
2	1	0	0	0
–	0	0	0	0
–	0	0	0	0
–	0	0	0	0

P	U3	U2	U1	U0
3	0	1	0	0
2	0	1	0	0
–	0	0	0	0
–	0	0	0	0
–	0	0	0	0

- At the end of the first time interval, all U bits are shifted right one position.



An Example of Additional-Reference-Bits Algorithm (3)

- Assume the following page reference string: 3, 2, 3, T, 8, 0, 3, T, 3, 0, 2, T, 6, 3, 4, 7
- During the second time interval, pages 8, 0, and 3 are referenced.

P	U3	U2	U1	U0
3	1	1	0	0
2	0	1	0	0
8	1	0	0	0
0	1	0	0	0
-	0	0	0	0

P	U3	U2	U1	U0
3	0	1	1	0
2	0	0	1	0
8	0	1	0	0
0	0	1	0	0
-	0	0	0	0

- At the end of the second time interval, all U bits are shifted right one position.

An Example of Additional-Reference-Bits Algorithm (4)

- Assume the following page reference string: 3, 2, 3, T, 8, 0, 3, T, 3, 0, 2, T, 6, 3, 4, 7
- During the third time interval, pages 3, 0, and 2 are referenced.

P	U3	U2	U1	U0
3	1	1	1	0
2	1	0	1	0
8	0	1	0	0
0	1	1	0	0
–	0	0	0	0

P	U3	U2	U1	U0
3	0	1	1	1
2	0	1	0	1
8	0	0	1	0
0	0	1	1	0
–	0	0	0	0

- At the end of the third time interval, all U bits are shifted right one position.

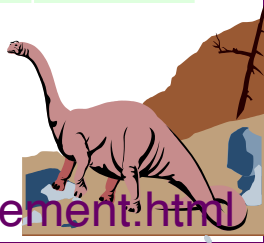
An Example of Additional-Reference-Bits Algorithm (5)

- Assume the following page reference string: 3, 2, 3, T, 8, 0, 3, T, 3, 0, 2, T, 6, 3, 4, 7
- During the fourth time interval, pages 6, 3, 4, and 7 are referenced.

P	U3	U2	U1	U0
3	0	1	1	1
2	0	1	0	1
8	0	0	1	0
0	0	1	1	0
-	0	0	0	0

P	U3	U2	U1	U0
3	1	1	1	1
2	0	1	0	1
4	**1**	**0**	**0**	**0**
0	0	1	1	0
6	1	0	0	0

After pages 6, 3, 4 are referenced, page 8 has been replaced by 4



http://www2.cs.uregina.ca/~hamilton/courses/330/notes/memory/page_replacement.html

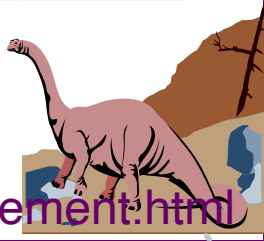
An Example of Additional-Reference-Bits Algorithm (6)

- Assume the following page reference string: 3, 2, 3, T, 8, 0, 3, T, 3, 0, 2, T, 6, 3, 4, 7
- During the fourth time interval, pages 6, 3, 4, and 7 are referenced.

P	U3	U2	U1	U0
3	1	1	1	1
2	0	1	0	1
4	1	0	0	0
0	0	1	1	0
6	1	0	0	0

P	U3	U2	U1	U0
3	1	1	1	1
7	**1**	**0**	**0**	**0**
4	1	0	0	0
0	0	1	1	0
6	1	0	0	0

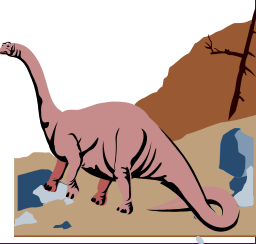
After page 7 is referenced, page 2 has been replaced by 7



http://www2.cs.uregina.ca/~hamilton/courses/330/notes/memory/page_replacement.html

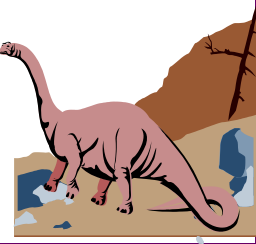
LRU Approximation Algorithms

- Second-Chance Algorithm (FIFO+reference bit)
  - When a page has been selected for replacement, we inspect its reference bit.
  - If the value is 0, we proceed to replace this page;
  - but if the reference bit is set to 1, we give the page a second chance and move on to select the next FIFO page.
  - When a page gets a second chance, its reference bit is cleared, and its arrival time is reset to the current time.

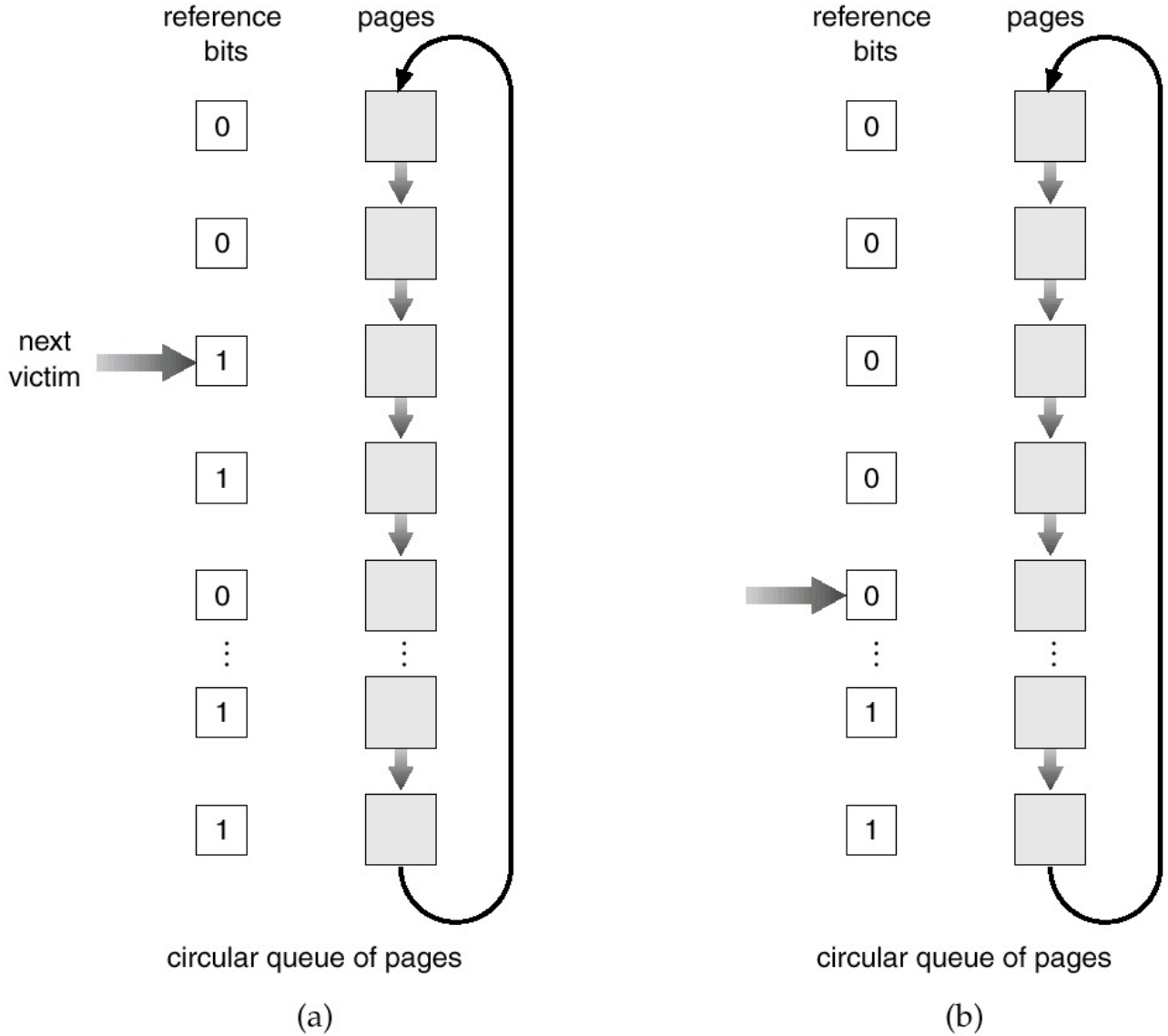


LRU Approximation Algorithms

- Second-Chance Algorithm (clock+reference bit)
  - Given a circular queue, called clock
  - If page to be replaced (in clock order) has reference bit = 1. then:
    - ✓set reference bit 0.
    - ✓leave page in memory.
    - ✓replace next page (in clock order), subject to same rules.

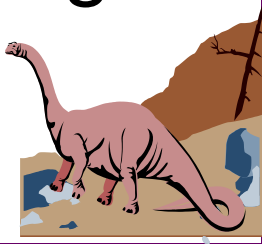


Second-Chance (clock) Page-Replacement Algorithm



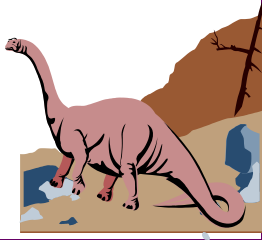
Counting Algorithms

- Keep a counter of the number of references that have been made to each page.
- **Least Frequently Used (LFU) Algorithm**: replaces page with the smallest count.
- **Most Frequently Used (MFU) Algorithm**: based on the argument that the page with the smallest count was probably just brought in and has yet to be used.



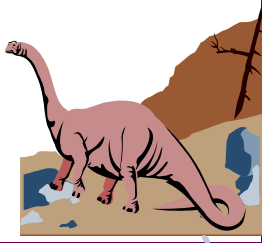
Chapter 9: Virtual Memory

- Background
- Demand Paging
- Copy-on-Write
- Page Replacement within a Process
- Allocation of Frames among Processes
- Thrashing and Working Set Model
- Memory-Mapped Files
- Allocating Kernel Memory
- Other Considerations
- Operating-System Examples



Allocation of Frames among Processes

- Each process needs a **minimum** number of pages.
- Example: IBM 370 – 6 pages to handle SS MOVE instruction:
  - instruction is 6 bytes, might span 2 pages.
  - 2 pages to handle **from**.
  - 2 pages to handle **to**.
- Two major allocation schemes.
  - fixed allocation
  - priority allocation



Fixed Allocation

- Equal allocation – e.g., if 100 frames and 5 processes, give each process 20 pages.

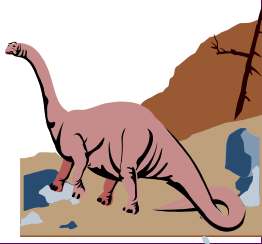
- Proportional allocation – Allocate pages to a process according to the size of the process.

$$s_i = \text{size of process } p_i$$

$$S = \sum s_i$$

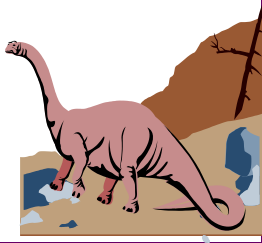
$$m = \text{total number of frames}$$

$$a_i = \text{allocation for } p_i = \frac{s_i}{S} \times m$$



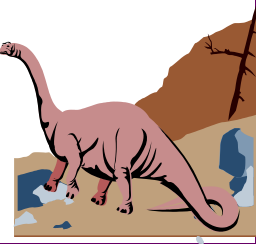
Priority Allocation

- Use a proportional allocation scheme using priorities rather than size.
- If process P_i generates a page fault,
  - select for replacement one of its frames.
  - select for replacement a frame from a process with lower priority number.



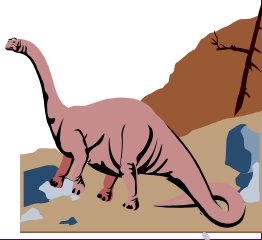
Global vs. Local Allocation

- **Global** replacement – process selects a replacement frame from the set of all frames; one process can take a frame from another.
- **Local** replacement – each process selects from only its own set of allocated frames.



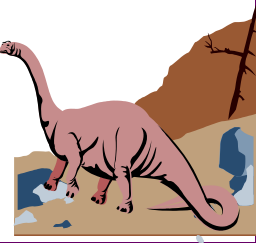
Chapter 9: Virtual Memory

- Background
- Demand Paging
- Copy-on-Write
- Page Replacement
- Allocation of Frames among Processes
- Thrashing and Working Set Model
- Memory-Mapped Files
- Allocating Kernel Memory
- Other Considerations
- Operating-System Examples



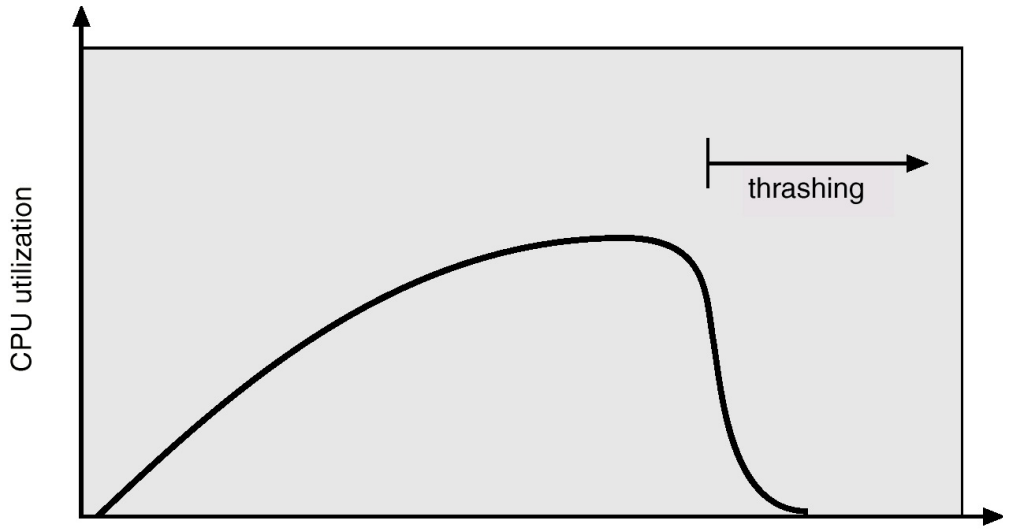
Thrashing

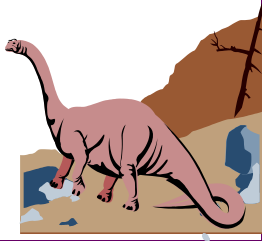
- If a process does not have “enough” frames, the page-fault rate is very high. This leads to:
  - low CPU utilization.
  - operating system thinks that it needs to increase the degree of multiprogramming.
  - another process added to the system.
- **Thrashing** ≡ a process is busy swapping pages in and out.



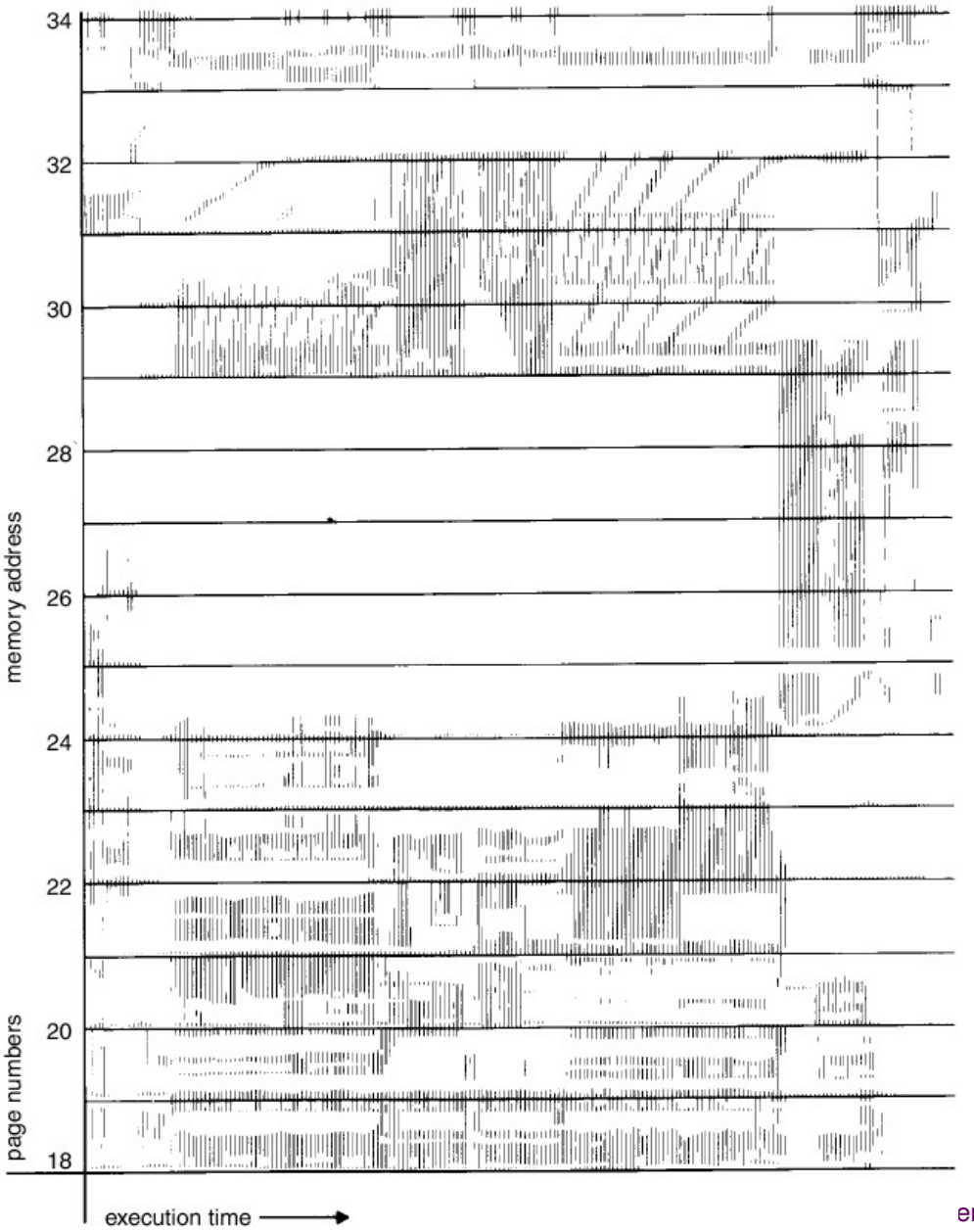
Thrashing

- Why does paging work?
  Locality model
  - Process migrates from one locality to another.
  - Localities may overlap.
- Why does thrashing occur?
  Σ size of locality $>$ total physical memory size





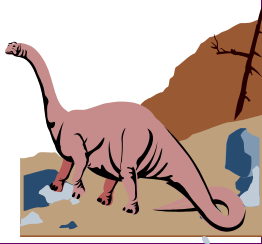
Locality In Memory-Reference Pattern



Working-Set Model

- $\Delta \equiv$ working-set window $\equiv$ a fixed number of page references
  Example: 10,000 instruction

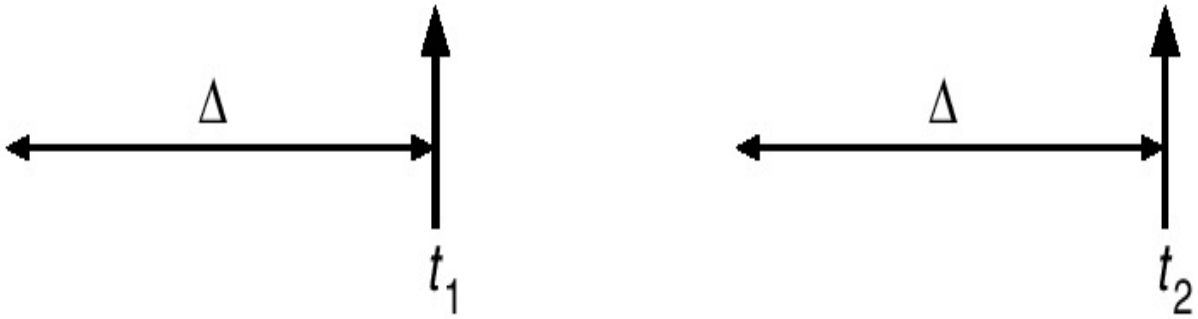
- WSS_i (working set of process P_i) = total number of pages referenced in the most recent Δ (varies in time)
  - if Δ too small will not encompass entire locality.
  - if Δ too large will encompass several localities.
  - if $\Delta = \infty \Rightarrow$ will encompass entire program.



Working-Set Model (cont.)

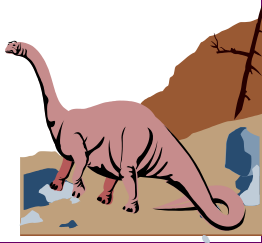
page reference table

. . . 2 6 1 5 7 7 7 7 5 1 6 2 3 4 1 2 3 4 4 4 3 4 3 4 4 4 1 3 2 3 4 4 4 3 4 4 4 . . .



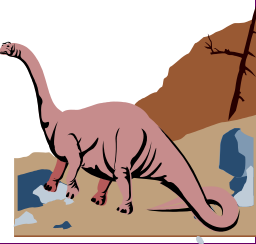
$WS(t_1) = \{1,2,5,6,7\}$

$WS(t_2) = \{3,4\}$



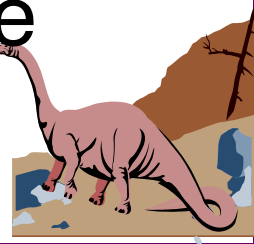
Working-Set Model (cont.)

- $D = \Sigma\ WSS_i \equiv$ total demand frames
- m ≡ total physical memory size
- if $D > m \Rightarrow$ Thrashing
- Policy: if $D >$ m, then suspend one of the processes.

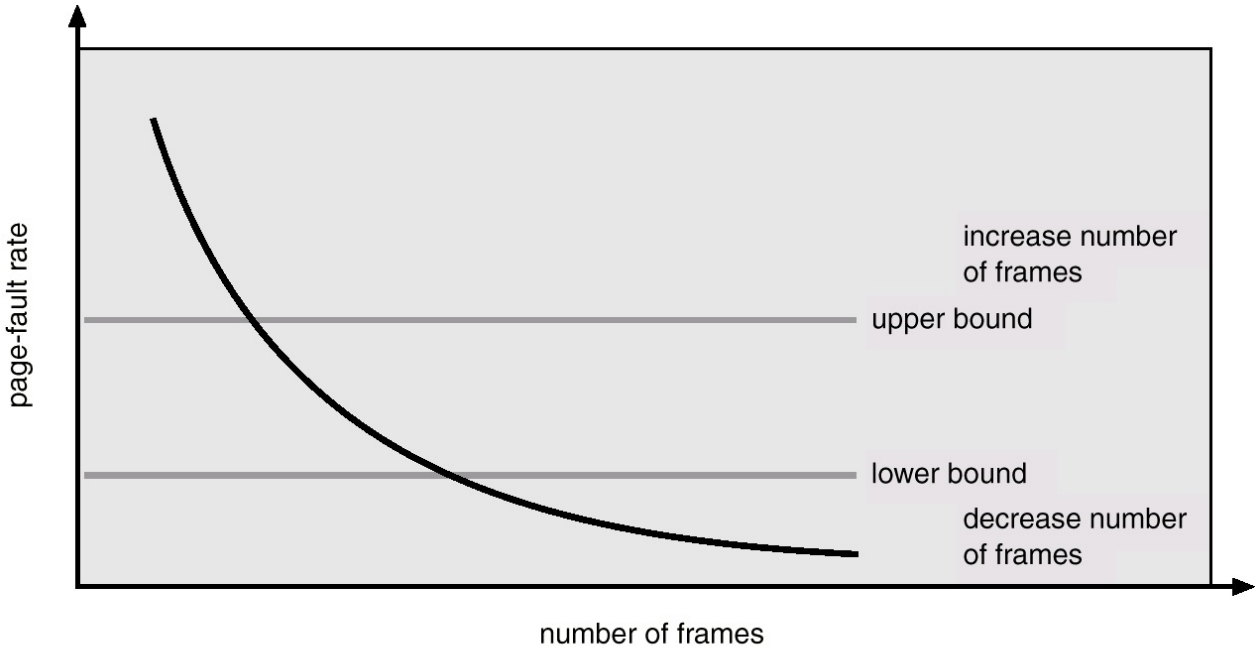


Keeping Track of the Working Set

- Approximate with interval timer + a reference bit
- Example: Δ = 10,000
  - Timer interrupts after every T=5,000 time units.
  - Keep in memory Δ/T=2 bits for each page.
  - Whenever a timer interrupts, copy and set the values of reference bits of all pages to 0.
  - If any one of the bits in memory = 1 $\Rightarrow$ page in working set.
- Why is this not completely accurate?
- Improvement: interrupt every T=1,000 time units, and keep Δ/T=10 bits for each page



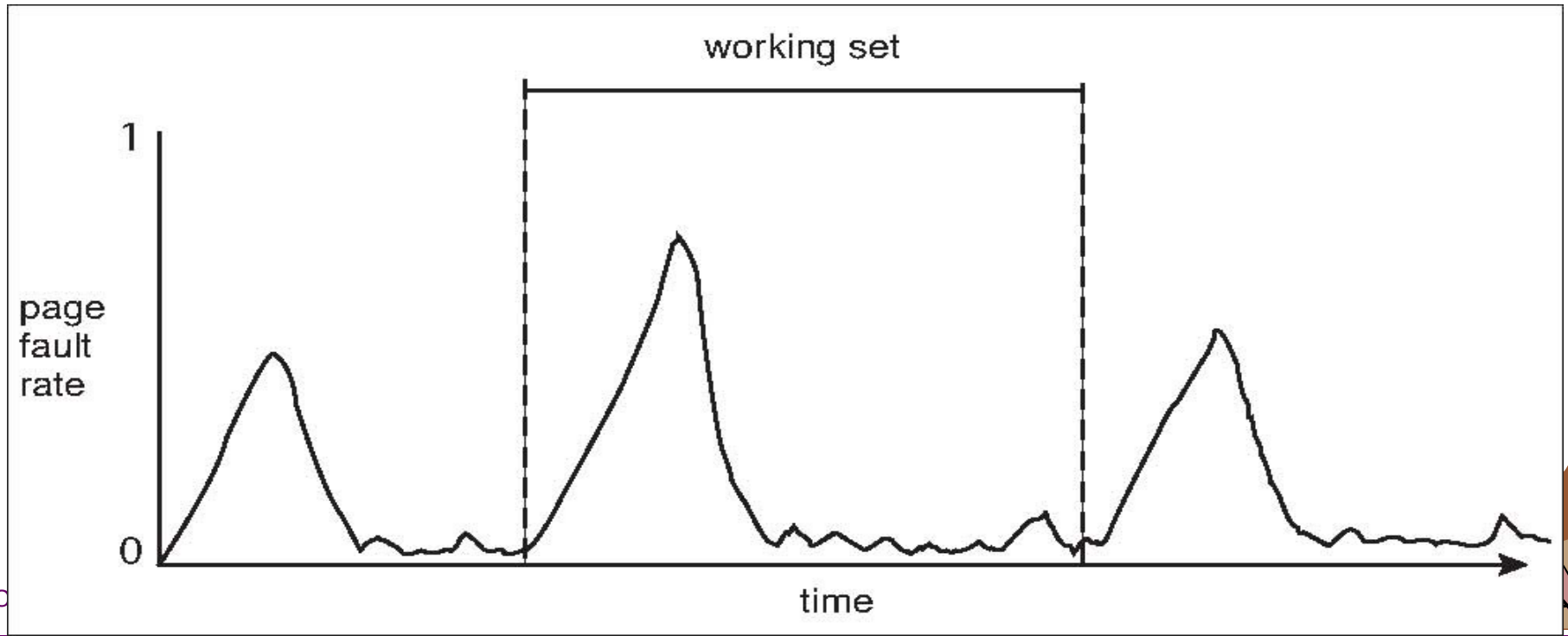
Page-Fault Frequency Scheme



- Establish “acceptable” page-fault rate.
  - If actual rate is too low, process loses frame.
  - If actual rate is too high, process gains frame.

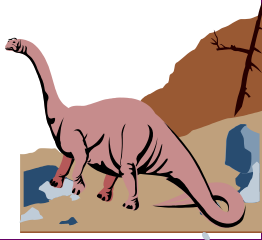
Working Sets and Page Fault Rates

- Direct relationship between working set of a process and its page-fault rate
- Working set changes over time
- Peaks and valleys over time



Chapter 9: Virtual Memory

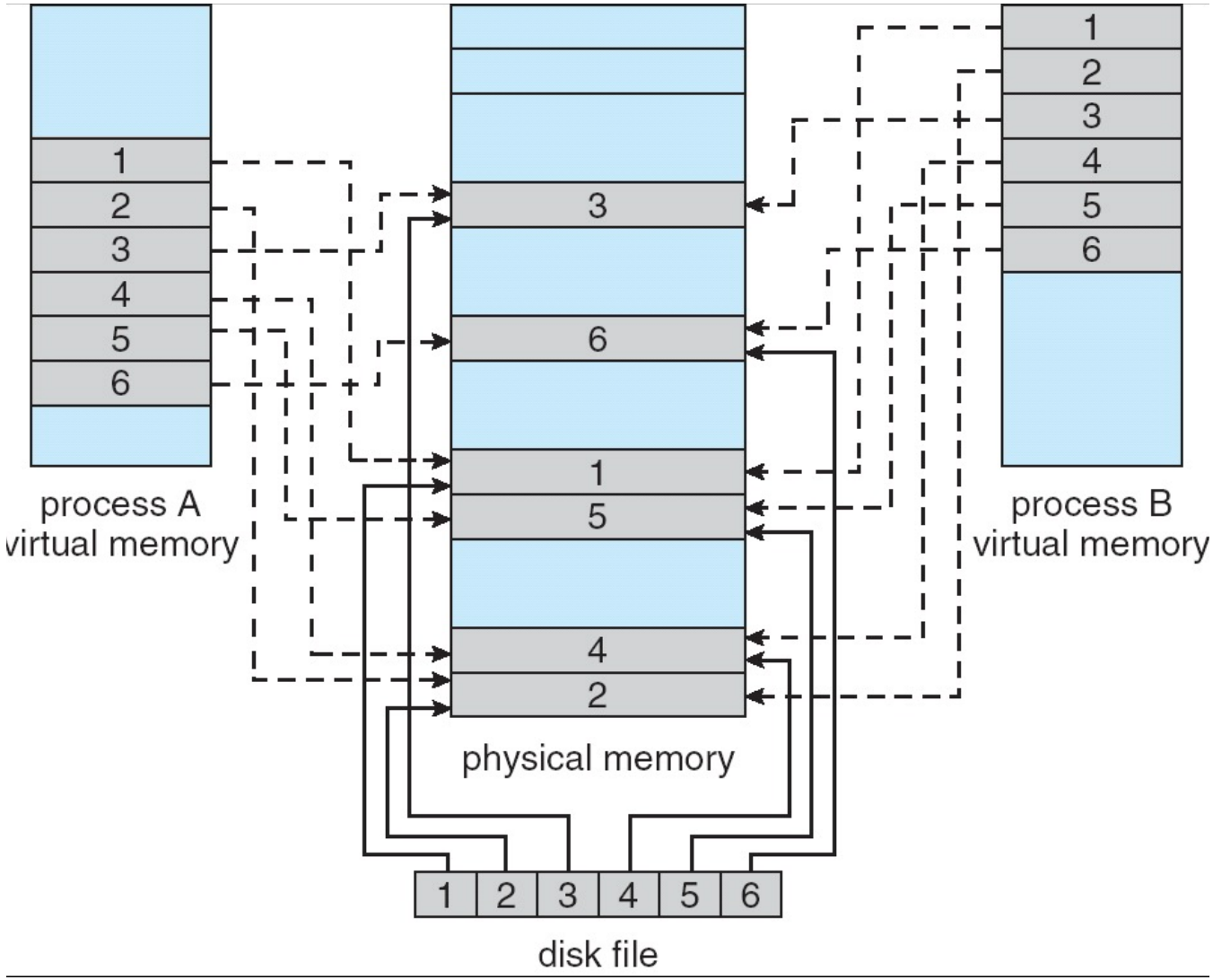
- Background
- Demand Paging
- Copy-on-Write
- Page Replacement
- Allocation of Frames
- Thrashing and Working Set Model
- Memory-Mapped Files
- Allocating Kernel Memory
- Other Considerations
- Operating-System Examples



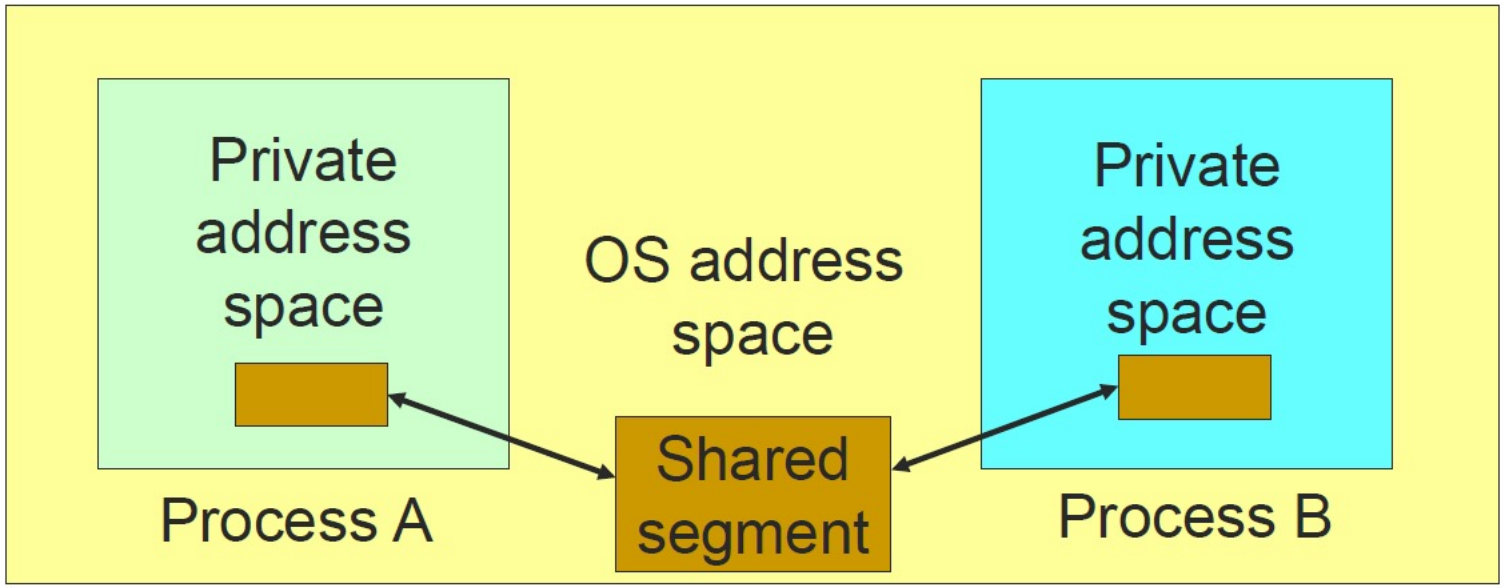
Memory-Mapped Files

- Memory-mapped file I/O allows file I/O to be treated as routine memory access by *mapping* a disk block to a page in memory.
- A file is initially read using demand paging. A page-sized portion of the file is read from the file system into a physical page. Subsequent reads/writes to/from the file are treated as ordinary memory accesses.
- Simplifies file access by treating file I/O through memory rather than **read() write()** system calls.
- Also allows several processes to map the same file allowing the pages in memory to be shared.

Memory-Mapped Files



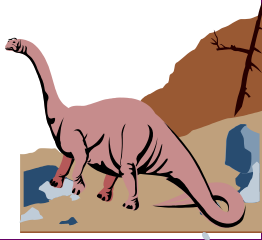
Memory-Mapped Shared Memory



- Processes request the shared segment
- OS maintains the shared segment
- Processes can attach/detach the segment
- Can mark segment for deletion on last detach

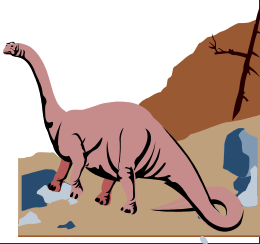
Chapter 9: Virtual Memory

- Background
- Demand Paging
- Copy-on-Write
- Page Replacement
- Allocation of Frames
- Thrashing and Working Set Model
- Memory-Mapped Files
- Allocating Kernel Memory
- Other Considerations
- Operating-System Examples



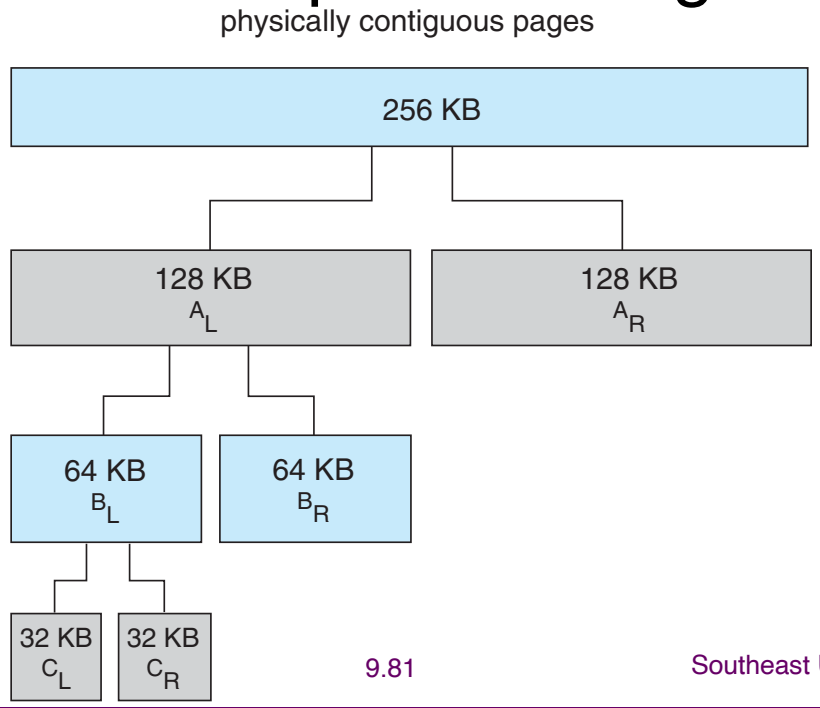
Allocating Kernel Memory

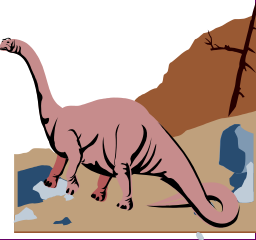
- Treated differently from user memory
- Often allocated from a free-memory pool
  - Kernel requests memory for structures of varying sizes
  - Some kernel memory needs to be contiguous
- Question: Can kernel memory management adopt contiguous memory allocation methods, similar to user-space memory management?



Buddy System

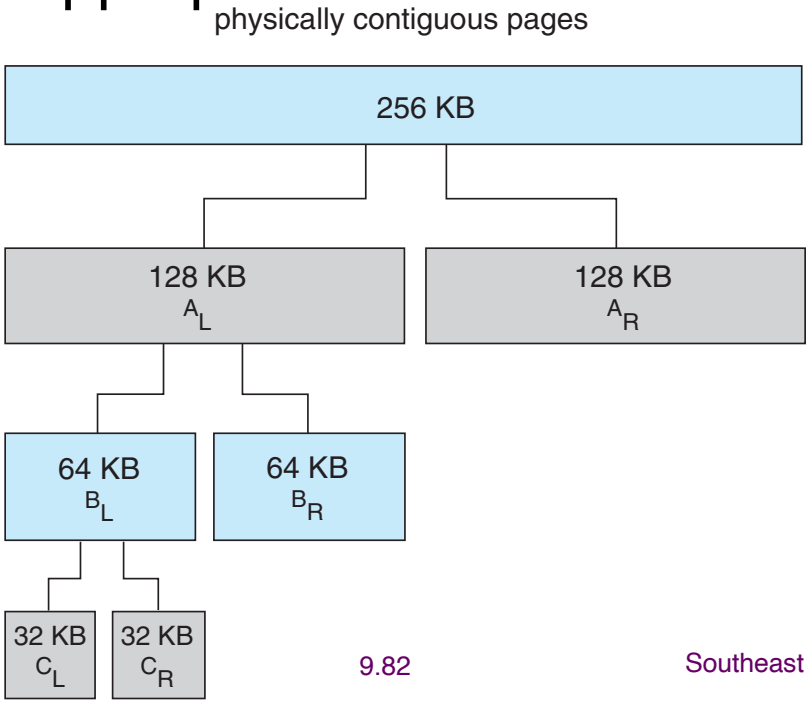
- Allocates memory from fixed-size segment consisting of physically-contiguous pages
- Memory allocated using **power-of-2 allocator**
  - Satisfies requests in units sized as power of 2
  - Request rounded up to next highest power of 2

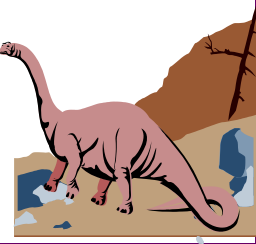




Buddy System

- This algorithm is used to give best fit
  - ✓ *Example* – If the request of 25Kb is made then block of size 32Kb is allocated.
- When smaller allocation needed than available, current chunk split into two buddies of next-lower power of 2
  - ✓ Continue until appropriate sized chunk available

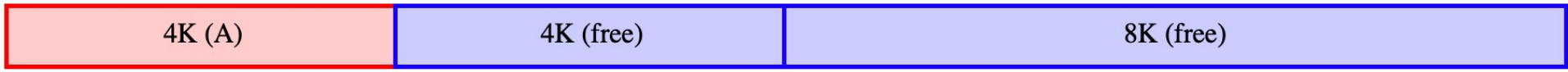




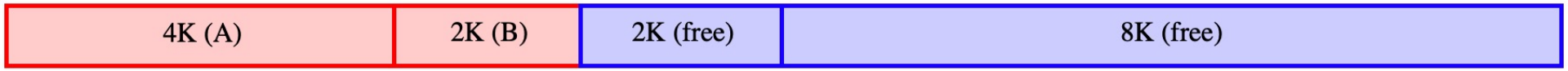
An example of how the buddy system works

- Suppose we have 16K to manage. It starts as one large block:

- Now, we have a request A for a block of 3.6k. We round up to 4K and perform two splits to create such a block.



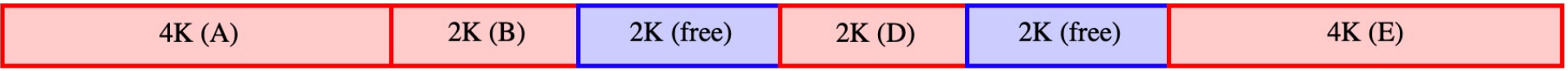
- Next, handle request B for 1.5K. This rounds up to 2, requiring another split:



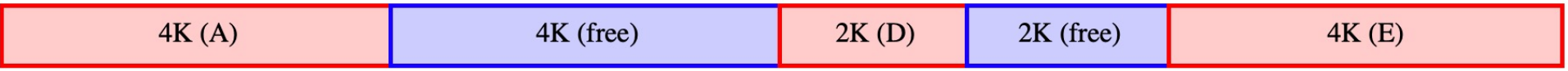
An example of how the buddy system works (cont.)

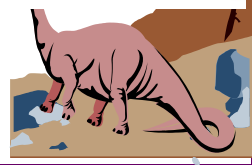
- Now service allocations of 1.2K (C), 1.9K (D) and 2.7K (E):

- Of course, nodes are merged when possible. Suppose the C allocation is freed:



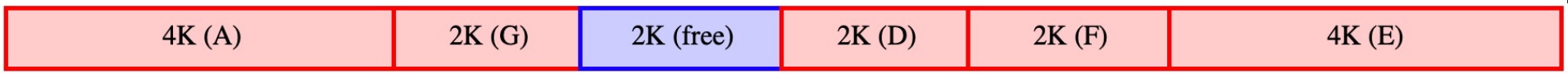
- Then, if B is also freed, the buddies are merged back into a larger node.



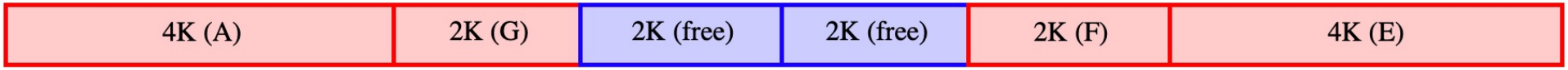


An example of how the buddy system works (cont.)

- However, it is only possible to join nodes which were previously split. For instance, suppose two more allocations, for 1.5K (F) and 1.6K (G), will cause another split:



- Then perhaps that D is freed:

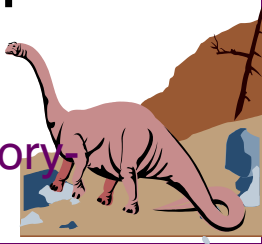


- Now, even though there are two adjacent free blocks of the same size, they cannot be merged because they were not the result of splitting a larger block.

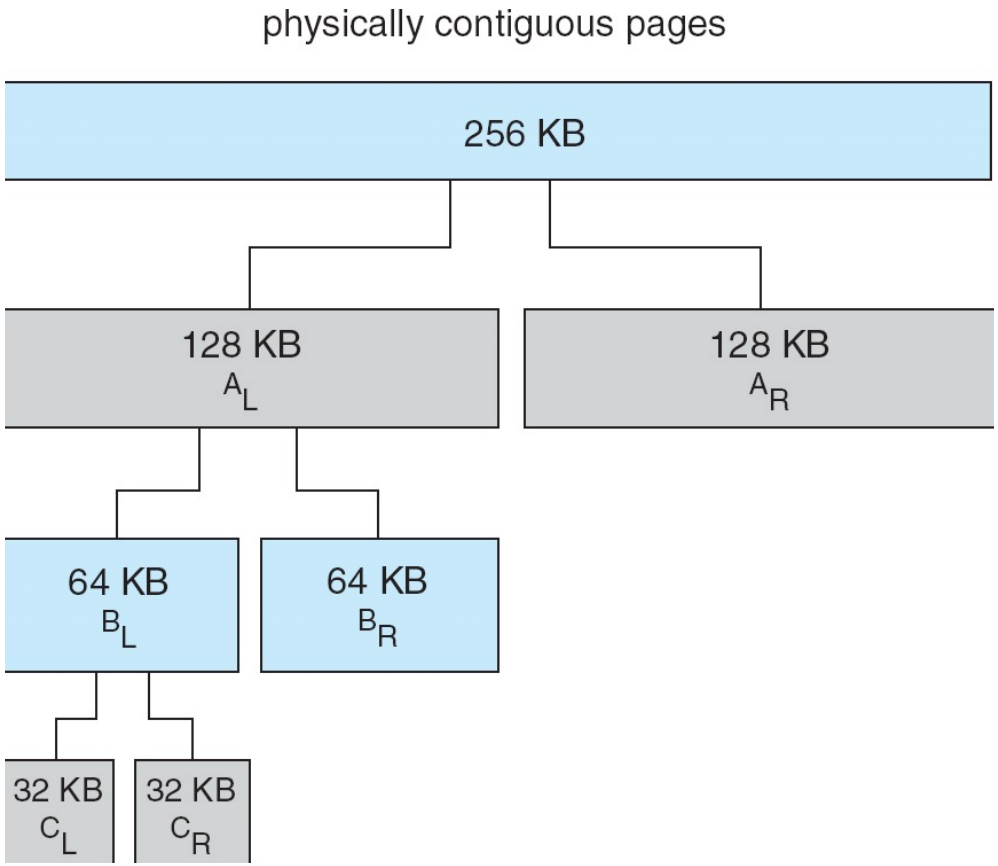
Advantages of Buddy System

- In comparison to other simpler techniques such as dynamic allocation, the buddy memory system has little external fragmentation.
- The buddy memory allocation system is implemented with the use of a binary tree to represent used or unused split memory blocks.
- The buddy system is very fast to allocate or deallocate memory.
- In buddy systems, the cost to allocate and free a block of memory is low compared to that of best-fit or first-fit algorithms.

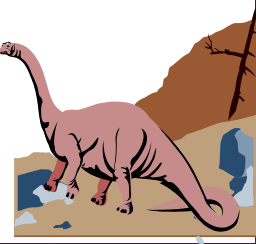
https://www.geeksforgeeks.org/operating-system-allocating-kernel-memory-buddy-system-slab-system/



Buddy System Allocator

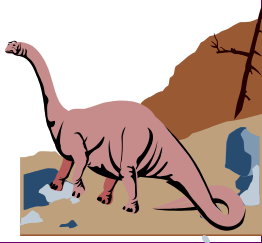


- Question: What is the key inadequacy of this power-of-2 allocator?
- Internal fragmentation
- When the size of an allocated block is x, the expected size of wasted memory is about $\frac{\sqrt{2}}{2}x$?



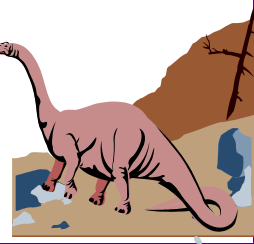
Slab Allocator

- Alternate strategy
- **Slab** is one or more physically contiguous pages
- **Cache** consists of one or more slabs
- Single cache for each unique kernel data structure
  - Each cache filled with **objects** – instantiations of the data structure

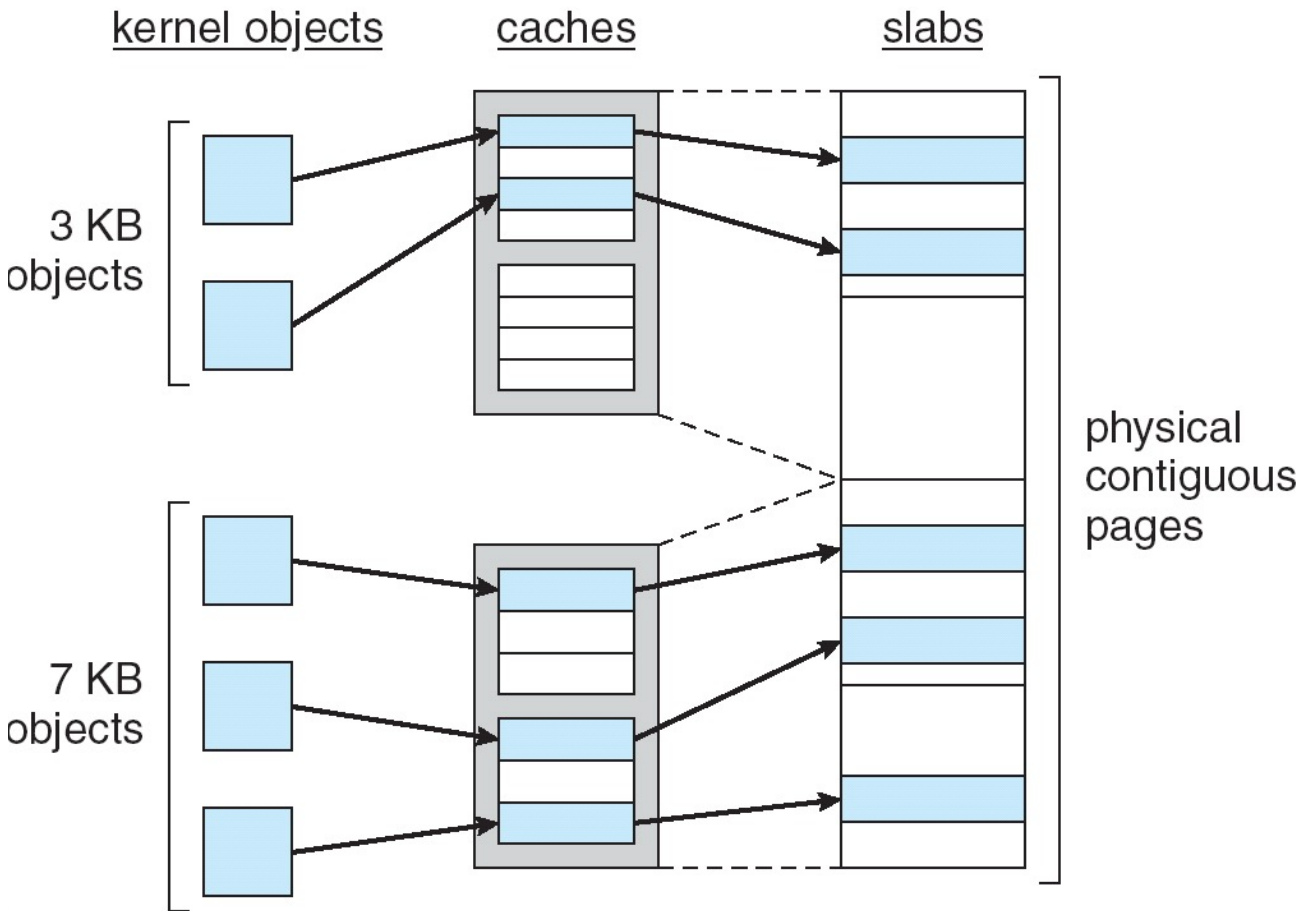


Slab Allocator

- When cache created, filled with objects marked as **free**
- When structures stored, objects marked as **used**
- If slab is full of used objects, next object allocated from empty slab
  - If no empty slabs, new slab allocated
- Benefits include no fragmentation, fast memory request satisfaction

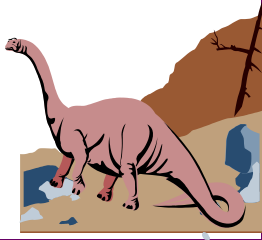


Illustrate the Slab Allocation



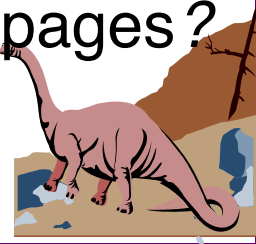
Chapter 9: Virtual Memory

- Background
- Demand Paging
- Copy-on-Write
- Page Replacement
- Allocation of Frames
- Thrashing and Working Set Model
- Memory-Mapped Files
- Allocating Kernel Memory
- Other Considerations
- Operating-System Examples



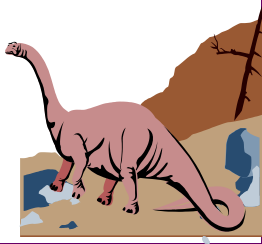
Other Issues -- Prepaging

- Prepaging (**预调页**)
  - To reduce the large number of page faults that occur at process startup
  - Prepage all or some of the pages a process will need, before they are referenced
  - But if prepaged pages are unused, I/O and memory was wasted
  - Assume *s* pages are prepaged and *α* of the pages is used
    - ✓ Is the benefit of *s * α* save pages faults larger or smaller than the cost of prepaging *s * (1- α)* unnecessary pages?
    - ✓ *α* near zero ⇒ prepaging loses



Other Issues – Page Size

- Page size selection must take into consideration:
  - fragmentation
  - table size
  - I/O overhead
  - locality



Other Issues – TLB Reach

- TLB Reach - The amount of memory accessible from the TLB
- TLB Reach = (TLB Size) X (Page Size)
- Ideally, the working set of each process is stored in main memory and its corresponding page table items is in TLB
  - Otherwise, a high degree of two-memory accesses
- Increase the Page Size
- Provide Multiple Page Sizes
  - This allows applications that require larger page sizes the opportunity to use them without a significant increase in fragmentation

Other Issues – Program Structure

- Program structure：**内外存数据交换以页为单位**
  - **int A[][] = new int[2048][1024];**
  - Each row is stored in one page
  - Program 1

```
for (j = 0; j < A.length; j++)
    for (i = 0; i < A.length; i++)
        sum += A[i,j];
```

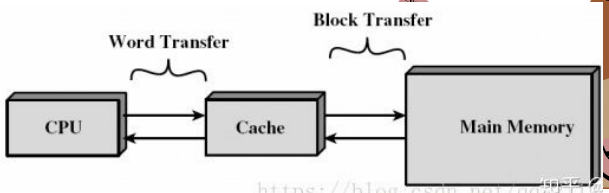
    2048 x 1024 page faults
  - Program 2

```
for (i = 0; i < A.length; i++)
    for (j = 0; j < A.length; j++)
        sum += A[i,j];
```

  - 2048 page faults

https://zhuanlan.zhihu.com/p/37749443

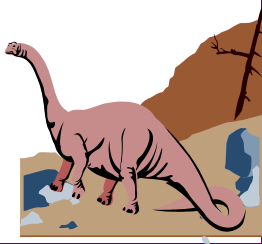
- cache line：每次内存和CPU缓存之间交换数据都是固定大小，cache line就表示这个固定的长度，比如64或128字节。



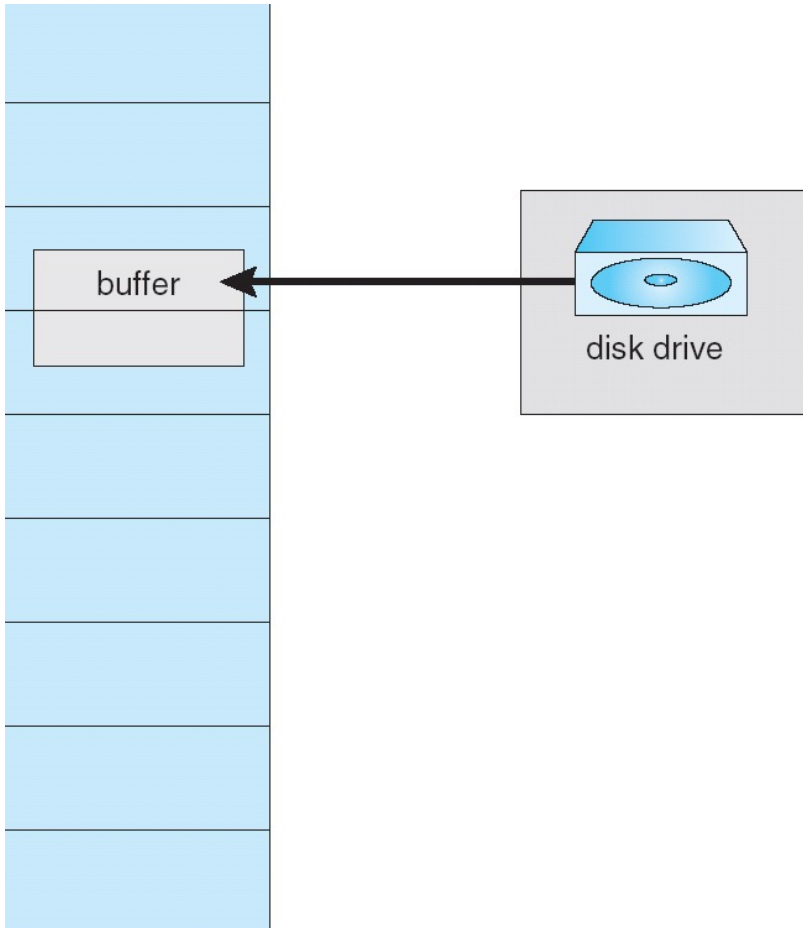
Other Issues – I/O interlock

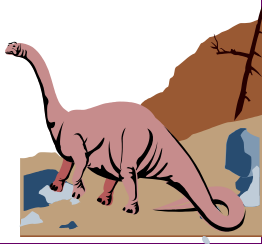
- **I/O Interlock** – Pages must sometimes be locked into memory

- Consider I/O - Pages that are used for copying a file from a device must be locked from being selected for eviction by a page replacement algorithm



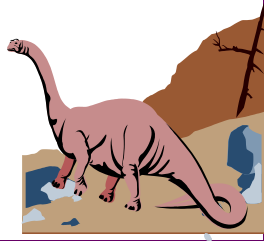
Reason Why Frames Used For I/O Must Be Kept in Memory





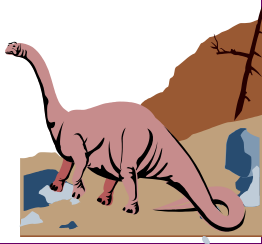
Chapter 9: Virtual Memory

- Background
- Demand Paging
- Copy-on-Write
- Page Replacement
- Allocation of Frames
- Thrashing and Working Set Model
- Memory-Mapped Files
- Allocating Kernel Memory
- Other Considerations
- Operating-System Examples



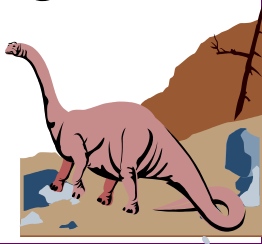
Operating System Examples

- Windows XP
- Solaris



Windows XP

- Uses demand paging with **clustering**. Clustering brings in pages surrounding the faulting page.
- Processes are assigned **working set minimum** and **working set maximum**
- Working set minimum is the minimum number of pages the process is guaranteed to have in memory

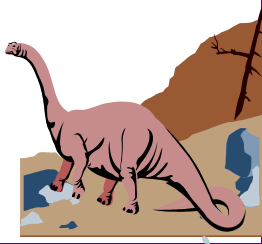


Windows XP (Cont.)

- A process may be assigned as many pages up to its working set maximum
- When the amount of free memory in the system falls below a threshold, **automatic working set trimming** is performed to restore the amount of free memory
- Working set trimming removes pages from processes that have pages in excess of their working set minimum

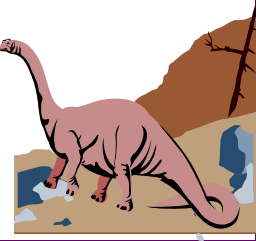
Solaris

- Maintains a list of free pages to assign faulting processes
- *Lotsfree* – threshold parameter (amount of free memory) to begin paging
- *Desfree* – threshold parameter to increasing paging
- *Minfree* – threshold parameter to being swapping



Solaris (Cont.)

- Paging is performed by *pageout* process
- Pageout scans pages using modified clock algorithm
- *Scanrate* is the rate at which pages are scanned. This ranges from *slowscan* to *fastscan*
- Pageout is called more frequently depending upon the amount of free memory available



Solaris 2 Page Scanner