

# 0117401: Operating System 计算机原理与设计

## Chapter 9: Virtual Memory(虚存)

陈香兰

xlanchen@ustc.edu.cn

<http://staff.ustc.edu.cn/~xlanchen>

Computer Application Laboratory, CS, USTC @ Hefei  
Embedded System Laboratory, CS, USTC @ Suzhou

April 28, 2017

## 温馨提示：



为了您和他人的工作学习，  
请在课堂上**关机或静音**。

**不要**在课堂上接打电话。

# 提纲

- 1 Background
- 2 Demand Paging (按需调页)
- 3 Copy-on-Write (写时复制)
- 4 Page Replacement (页面置换)
- 5 Allocation of Frames
- 6 Thrashing (抖动)
- 7 Memory-Mapped Files
- 8 Allocating Kernel Memory
- 9 Other Issues
- 10 Operating System Examples
- 11 小结和作业

## 1 Background

- Instructions must be loaded into memory before execution.
- Solutions in chapter 8:

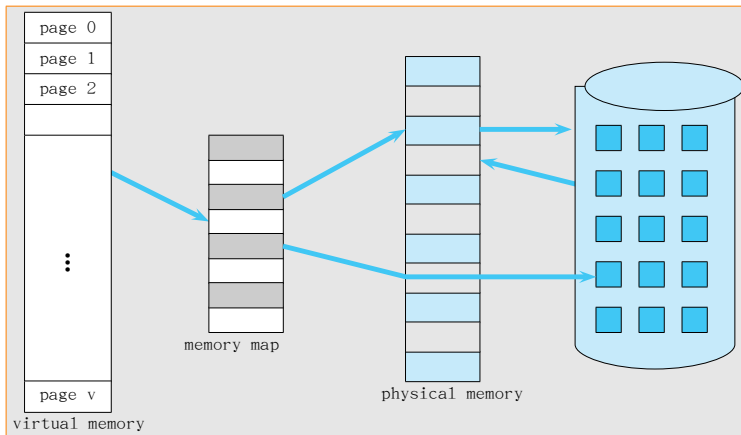
Program entire → Physical memory

- Sometimes, **jobs** may be **too big** or **too many**.  
How to expand the main memory?
  - Physically? **COST TOO HIGH!**
  - Logically? ✓

- Virtual memory: Why and How?
  - Some code may get no, or only little, opportunity of execution,  
for example, code for error handlers
  - Some data may get no opportunity of access
  - **Locality of reference** (程序的局部性原理), 1968, Denning
    - Temporal locality (时间局部性)
    - Spatial locality (空间局部性)
  - **Idea: partly loading** (部分装入)、**demand loading** (按需装入)、**replacement** (置换)

- Virtual Memory (虚拟存储器)  
是指具有请求调页功能和置换功能，能从逻辑上对内存容量加以扩充的一种存储器系统
  - Logical size:  
从系统角度看：内存容量+外存容量  
从进程角度看：地址总线宽度范围内；内存容量+外存容量
  - Speed: close to main memory
  - Cost per bit: close to secondary storage (disks)
- **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

# Background



Example: virtual memory that is larger than physical memory



- Virtual memory can be implemented via:

## ① Demand paging

- Paging technology + pager (请求调页) and page replacement
- Pager VS. swapper  
the unit of swapping in/out is not the entire process but page.

## ② Demand segmentation

# 虚拟存储器的特征

## ① 多次性: 最重要的特征

- 一个作业被分成多次装入内存运行

## ② 对换性

- 允许在进程运行的过程中, (部分) 换入换出

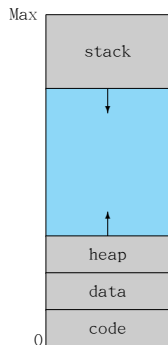
## ③ 虚拟性

- 逻辑上的扩充

- 虚拟性是以多次性和对换性为基础的。
- 多次性和对换性是建立在离散分配的基础上的

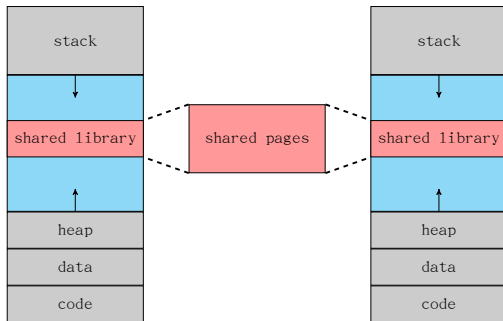
# Virtual-address Space (虚拟地址空间)

- The **virtual address space** of a process refers to **the logical (or virtual) view of how a process is stored in memory**.
  - Typically:  $0 \sim xxx$  & exists in contiguous memory
- **In fact**, the physical memory are organized (partitioned) in **page frames** & the page frames assigned to a process **may not be contiguous**  $\Rightarrow$  MMU



# Some benefits

## 1 Shared library using virtual memory



## 2 Shared memory

## 3 Speeding up process creation

- 2 Demand Paging (按需调页)
  - Basic Concepts (Hardware support)
  - Performance of Demand Paging

# Demand Paging (按需调页)

- Do **not** load the entire program in physical memory at program execution time.  
NO NEED!
- Bring a page into memory **only when it is needed**
  - ① Less I/O needed
  - ② Less memory needed
  - ③ Faster response
  - ④ More users
- A page is **needed**  $\Leftarrow$  Reference to it
  - Invalid reference  $\Rightarrow$  Abort
  - Not-in-memory  $\Rightarrow$  Bring to memory

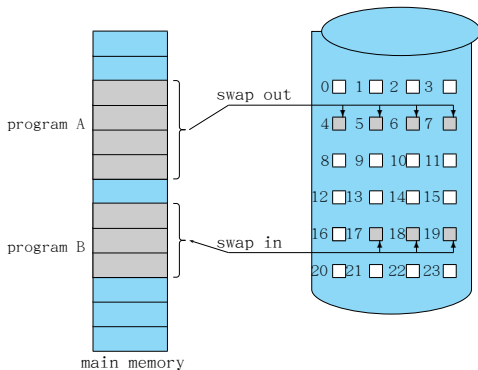
# Demand Paging (按需调页)

- Swapper VS. Pager

- A swapper manipulates the entire processes
- **Lazy swapper**

Never swaps a page into memory unless the page will be needed

- Swapper that deals with individual pages is a **pager**



- 2 Demand Paging (按需调页)
  - Basic Concepts (Hardware support)
  - Performance of Demand Paging



- ① The modified page table mechanism
- ② Page fault
- ③ Address translation
- ④ Secondary memory (as swap space)

# 1) The modified page table mechanism

## ① Valid-Invalid Bit (**PRESENT** bit)

- With each page table entry a valid-invalid bit is associated
  - $v \Rightarrow$  in-memory,  $i \Rightarrow$  not-in-memory
- **Initially** valid-invalid bit is set to **i** on all entries
- During address translation, if valid-invalid bit in page table entry is **i**  $\Rightarrow$  **page fault**

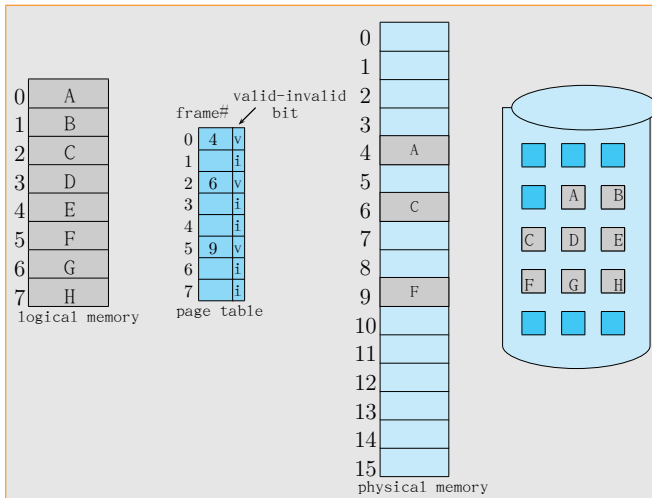
Frame#	valid-invalid bit
	v
	v
	v
	v
	i
...	
	i
	i

page table

- ② Reference bits (for pager out)
- ③ Modify bit (or dirty bit)
- ④ Secondary storage info (for pager in)

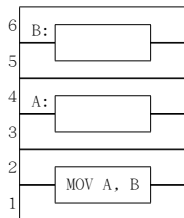
# 1) The modified page table mechanism

- Page table when some pages are not in main memory



## 2) Page Fault (缺页故障)

- **First reference** to a page will trap to OS:  
page fault(缺页故障/异常/中断)
- **Page fault trap (缺页异常)**
  - Exact exception (trap), 精确异常  
Restart the process in exactly the same place and state.  
Re-execute the instruction which triggered the trap
- Execution of one instruction may cause multiply page faults



Example: One instruction and 6 page faults

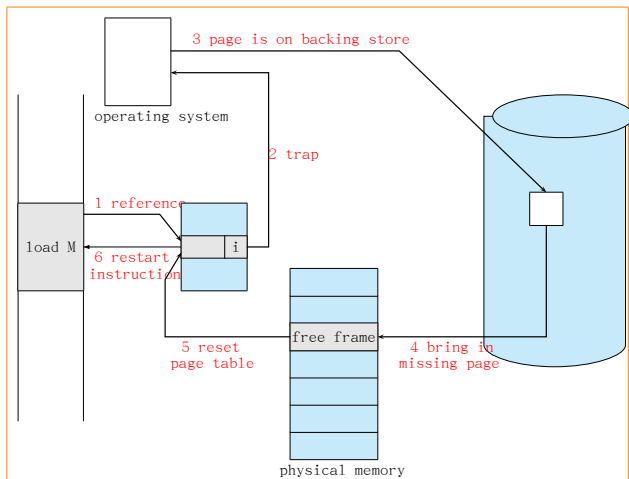
- Page fault may occur at every memory reference
- One instruction may cause multiply page faults while fetching instruction or r/w operators

## 2) Page Fault (缺页故障)

### • Page Fault Handling:

- 1 OS **looks** at **an internal table** to decide:
  - Invalid reference  $\Rightarrow$  abort
  - Just not in memory  $\Rightarrow$
- 2 **Get** empty frame
- 3 **Swap** page into frame
  - Pager out & pager in
- 4 **Modify** the internal tables & Set validation bit = v
- 5 **Restart** the instruction that caused the page fault

## 2) Page Fault (缺页故障)



Steps in handling a page fault

### 3) address translation

- Address translation hardware + page fault handling

- **Context save (保存现场)**

Before OS handling the page fault, the state of the process must be saved

- Example: record its register values, PC

- **Context restore (恢复现场)**

The saved state allows the process to be resumed from the line where it was interrupted.

- **NOTE:** distinguish the following 2 situation

- Illegal reference  $\Rightarrow$  The process is terminated
- Page fault  $\Rightarrow$  Load in or pager in



- 2 Demand Paging (按需调页)
  - Basic Concepts (Hardware support)
  - Performance of Demand Paging

# Performance of Demand Paging

- Let  $p =$  Page Fault Rate ( $0 \leq p \leq 1.0$ )
  - If  $p = 0$ , no page faults
  - If  $p = 1.0$ , every reference is a fault
- **Effective Access Time (EAT)**

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

$$\text{page fault time} = \text{page fault overhead} \\ + \text{swap page out} \\ + \text{swap page in} \\ + \text{restart overhead}$$

# Performance of Demand Paging

## ● Example

- Memory access time = 200ns
- Average page-fault service time = 8ms

$$\begin{aligned} \text{EAT} &= (1 - p) \times 200 + p \times 8\text{ms} \\ &= (1 - p) \times 200 + p \times 8,000,000 \\ &= 200 + p \times 7,999,800 \end{aligned}$$

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

$$\begin{aligned} p &= 0.001 \\ \text{EAT} &= 8,199.8\text{ns} = 8.2\mu\text{s} \end{aligned}$$

This is a slowdown by a factor of  $\frac{8.2\mu\text{s}}{200\text{ns}} = 40!!$

## ● Example

- Memory access time = 200ns
- Average page-fault service time = 8ms

$$\begin{aligned} \text{EAT} &= (1 - p) \times 200 + p \times 8\text{ms} \\ &= (1 - p) \times 200 + p \times 8,000,000 \\ &= 200 + p \times 7,999,800 \end{aligned}$$

- If we want performance degradation < 10%, then

$$\begin{aligned} \text{EAT} = 200 + p \times 7,999,800 &< 200(1 + 10\%) = 220 \\ p \times 7,999,800 &< 20 \\ p &< 20/7,999,800 \approx 0.0000025 \end{aligned}$$

- To keep the fault time low

- ① Swap space, faster than file system
- ② Only dirty page is swapped out, or
- ③ Demand paging only from the swap space, or
- ④ Initially demand paging from the file system, swap out to swap space, and all subsequent paging from swap space

- Keep the fault rate extremely low

- Localization of program executing
  - Time, space

## 3 Copy-on-Write (写时复制)

- Virtual memory allows other benefits during process creation:
  - ① **Copy-on-Write** (写时复制)
  - ② Memory-Mapped Files (later)

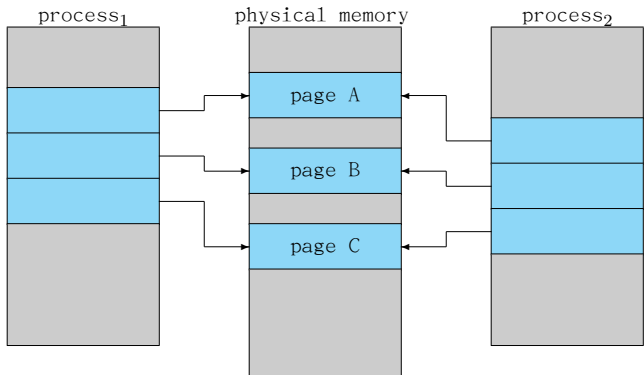
# Copy-on-Write (写时复制)

- Copy-on-Write (COW, 写时复制)
  - 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 zeroed-out pages



# Copy-on-Write (写时复制)

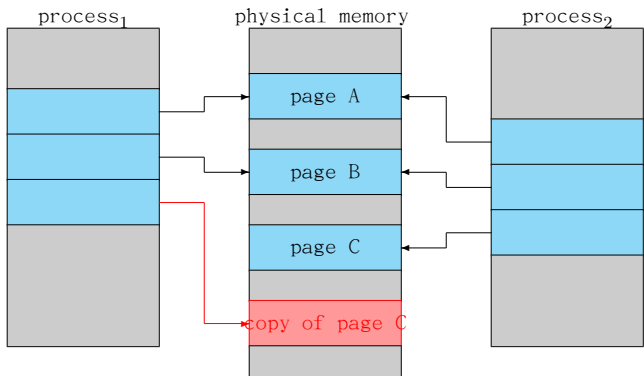
- Example:



Before Process 1 Modifies Page C

# Copy-on-Write (写时复制)

- Example:



## 4 Page Replacement (页面置换)

- Basic Page Replacement
- First-In-First-Out (FIFO) Algorithm
- Optimal Algorithm
- Least Recently Used (LRU) Algorithm
- LRU Approximation Algorithms
- Counting Algorithms
- Page-Buffering Algorithms

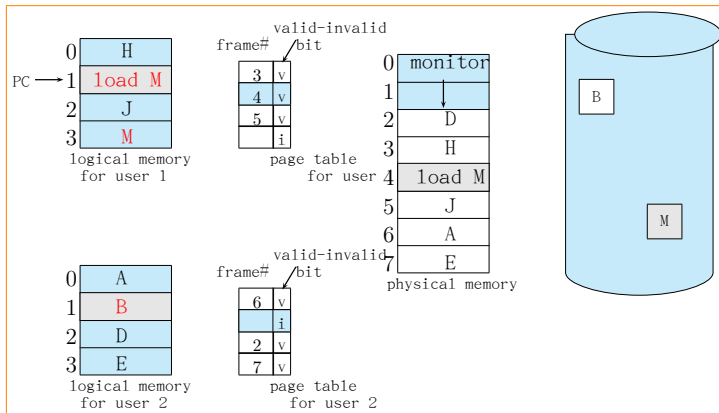
- Page replacement (页面置换)

Find some page in memory, but not really in use, swap it out

- Algorithm?
- Performance?
  - want an algorithm which will result in minimum number of page faults
- Same page may be brought into memory several times

# Need of Page Replacement (页面置换) I

- Free page frame is managed by OS using free-frame-list
- Over-allocation:** No free frames; All memory is in use.



Example of over-allocation

- What happens if there is no free frame?
- **Solution:**  
Page replacement (页面置换)  
Prevent over-allocation of memory by modifying page-fault service routine to include page replacement

## 4 Page Replacement (页面置换)

- Basic Page Replacement
  - First-In-First-Out (FIFO) Algorithm
  - Optimal Algorithm
  - Least Recently Used (LRU) Algorithm
  - LRU Approximation Algorithms
  - Counting Algorithms
  - Page-Buffeing Algorithms

# Basic Page Replacement

## ● Basic Page Replacement

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

② **Find** a free frame:

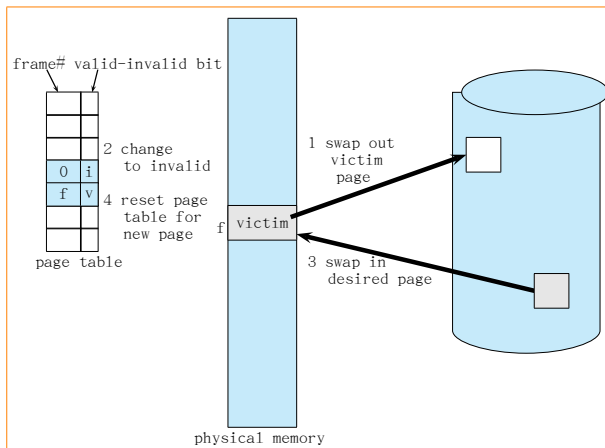
- If there is a free frame, use it
- If there is no free frame, use a page replacement algorithm to select a victim frame

③ **Bring** the desired page into the (newly) free frame;  
**Update** the page and frame tables

④ **Restart** the process



# Basic Page Replacement



# Basic Page Replacement

- **NO MODIFY, NO WRITTEN** (to disk/swap space)
  - Use **modify (dirty) bit** to reduce overhead of page transfers
    - Only modified pages are written to disk
  - This technique also applies to read-only pages
    - For example, pages of binary code
- Page replacement **completes separation between logical memory and physical memory**
  - Large virtual memory can be provided on a smaller physical memory
- Demand paging, to lowest page-fault rate, two major problems
  - 1 Frame-allocation algorithms
  - 2 Page-replacement algorithms

- **GOAL**: to lowest page-fault rate
- Different algorithms are evaluated by running it on a particular string of memory references (**reference string**) and **computing** the number of **page faults** on that string
- A **reference string** is a sequence of addresses referenced by a program  
Example:

- An address reference string:

0100 0432 0101 0612 0102 0103 0104 0101 0611 0103 0104  
0101 0610 0102 0103 0104 0101 0609 0102 0105

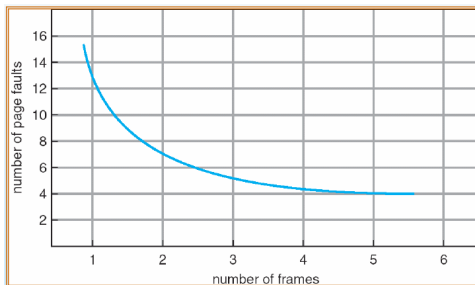
- Assuming page size = 100 B, then its corresponding page reference string is:

1 4 1 6 1 6 1 6 1 6 1

- ② How many page faults?
  - Determined by the number of page frames assigned to the process
  - For the upper example: 1 4 1 6 1 6 1 6 1 6 1
    - If  $\geq 3$ , then only 3 page faults
    - If = 1, 11 pages faults

## How many page faults?

- Determined by the number of page frames assigned to the process
- For the upper example: 1 4 1 6 1 6 1 6 1 6 1
  - If  $\geq 3$ , then only 3 page faults
  - If = 1, 11 pages faults



Graph of Page Faults Versus The Number of Frames

- In all our examples, the reference strings are

① 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5

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

## 4 Page Replacement (页面置换)

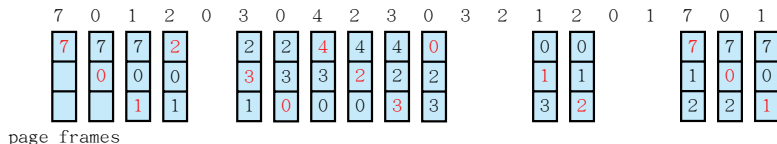
- Basic Page Replacement
- First-In-First-Out (FIFO) Algorithm
- Optimal Algorithm
- Least Recently Used (LRU) Algorithm
- LRU Approximation Algorithms
- Counting Algorithms
- Page-Buffeing Algorithms

# First-In-First-Out (FIFO) Algorithm

- The **simplest** page-replacement algorithm: **FIFO**
  - For each page: a time when it was brought into memory
  - For replacement: **the oldest page is chosen**
  - Data structure: a FIFO queue
    - Replace the page at the head of the queue
    - Insert a new page at the end of the queue

## 1 Example 1: 15 page faults, 12 page replacements

Reference string

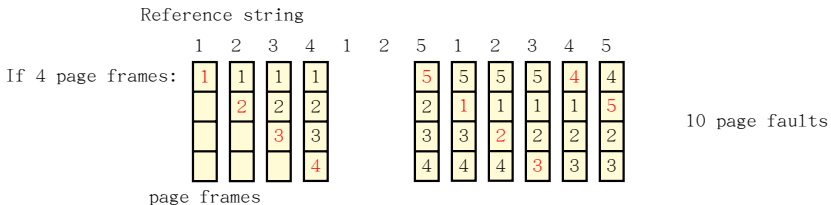
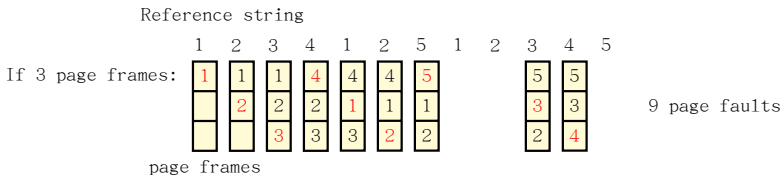




# First-In-First-Out (FIFO) Algorithm

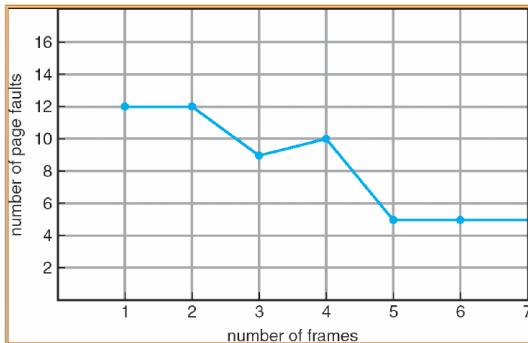
② Example 2: Reference string:

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



# First-In-First-Out (FIFO) Algorithm

- **More memory, better performance?** MAY BE NOT!!
  - **Belady's anomaly** (贝莱迪异常现象):  
more frames  $\Rightarrow$  more page faults



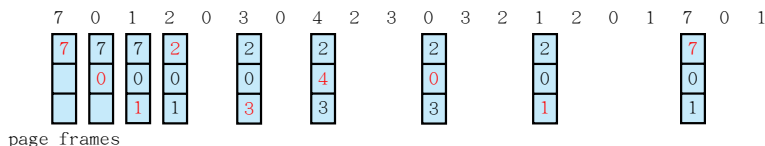
FIFO illustrating Belady's Anomaly

## 4 Page Replacement (页面置换)

- Basic Page Replacement
- First-In-First-Out (FIFO) Algorithm
- Optimal Algorithm
- Least Recently Used (LRU) Algorithm
- LRU Approximation Algorithms
- Counting Algorithms
- Page-Buffeing Algorithms

- **Optimal page-replacement algorithm:**  
Replace page that will not be used for longest period of time
  - It has the **lowest page-fault rate**
  - It will **never suffer from Belady's anomaly**
- Example1: 9 page faults, 6 page replacements

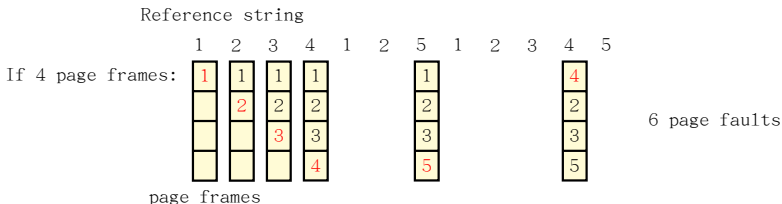
Reference string



# Optimal Algorithm

- 4 frames example

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



- OPT: Difficult to implement

- How to know the future knowledge of the reference string?

- So, it is **only used for measuring** how well other algorithm performs

## 4 Page Replacement (页面置换)

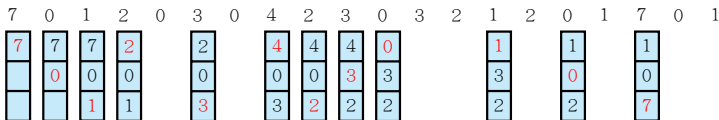
- Basic Page Replacement
- First-In-First-Out (FIFO) Algorithm
- Optimal Algorithm
- Least Recently Used (LRU) Algorithm
- LRU Approximation Algorithms
- Counting Algorithms
- Page-Buffeing Algorithms

# Least Recently Used (LRU) Algorithm

- **LRU**: an approximation of the OPT algorithm  
Use the recent past as an approximation of the near future
  - To replace the page that **has not been used for the longest period of time**
  - For each page: a time of its last use
  - For replace: the oldest time value

## ① Example 1: 12 page faults; 9 page replacements

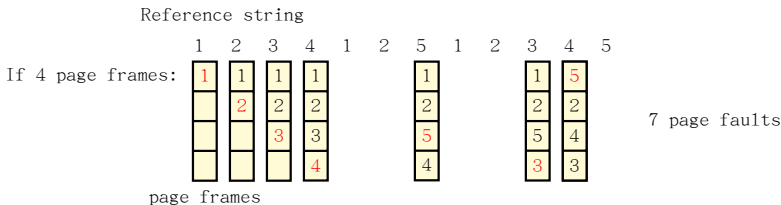
Reference string



page frames

# Least Recently Used (LRU) Algorithm

- **LRU**: an approximation of the OPT algorithm  
Use the recent past as an approximation of the near future
  - To replace the page that **has not been used for the longest period of time**
  - For each page: a time of its last use
  - For replace: the oldest time value
- Reference string: 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5





## HOW to implement LRU replacement?

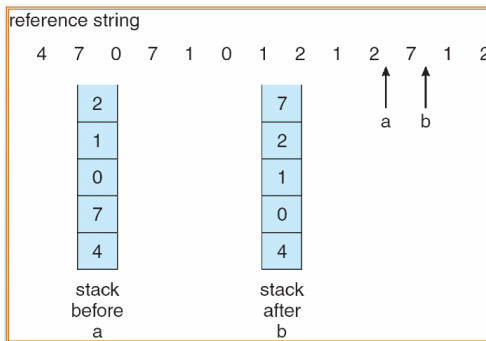
### ① Counter implementation

- 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

# Least Recently Used (LRU) Algorithm

## HOW to implement LRU replacement?

- ② **Stack** implementation — keep a stack of page numbers in a double link form:
  - When page referenced: Move it to the top
    - Requires 6 pointers to be changed
  - No search for replacement



## 4 Page Replacement (页面置换)

- Basic Page Replacement
- First-In-First-Out (FIFO) Algorithm
- Optimal Algorithm
- Least Recently Used (LRU) Algorithm
- **LRU Approximation Algorithms**
- Counting Algorithms
- Page-Buffeing Algorithms

## ● Reference bit

- With each page associate a bit, initially = 0
- When page is referenced bit set to 1
- Replace the one which is 0 (if one exists)
  - We do not know the order, however

## ④ Additinal-Reference-Bits Algorithm:

Reference bits + time ordering, for example: 8 bits

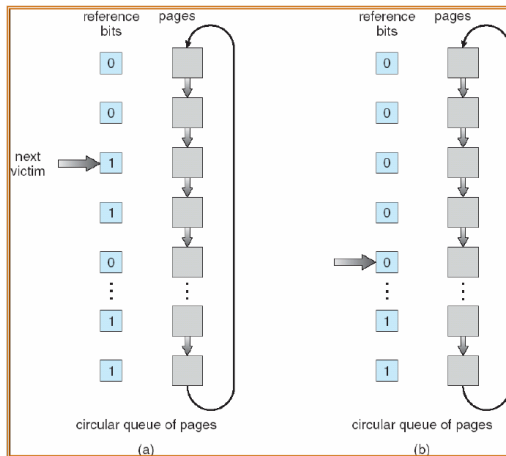
- HW modifies the highest bit, only
- Periodically, right shift the 8 bits for each page
- 00000000, ..., 01110111, ..., 11000100, ..., 11111111

## ② Second chance (clock) Algorithm

- **Need only 1 reference bit**, modified FIFO algorithm
  - First, a page is selected by FIFO
  - Then, the reference bit of the page is checked:
    - 0  $\Rightarrow$  replace it
    - 1  $\Rightarrow$  not replace it, get a second chance with reference bit:  
1  $\rightarrow$  0, and time  $\rightarrow$  current

# LRU Approximation Algorithms

- ② **Second chance (clock) Algorithm**
  - Implementation: **Clock replacement**
    - Clock order



## 8 Enhanced Second-Chance Algorithm

- Reference bit + modify bit
- 4 page classes (访问位, 修改位)
  - (0, 0) — best page to replace
  - (0, 1) — not quite as good
  - (1, 0) — probably be used again soon
  - (1, 1) — probably be used again soon, and be dirty
- Replace the first page encountered in the lowest nonempty class.
  - 1 Scan for (0, 0)
  - 2 Scan for (0, 1), & set reference bits to 0
  - 3 Loop back to step (a)

## 4 Page Replacement (页面置换)

- Basic Page Replacement
- First-In-First-Out (FIFO) Algorithm
- Optimal Algorithm
- Least Recently Used (LRU) Algorithm
- LRU Approximation Algorithms
- Counting Algorithms
- Page-Buffeing Algorithms



- Counting algorithms:

Keep a counter of the number of references that have been made to each page

- ① **LFU Algorithm**: replaces page with smallest count
- ② **MFU Algorithm**: based on the argument that the page with the smallest count was probably just brought in and has yet to be used

## 4 Page Replacement (页面置换)

- Basic Page Replacement
- First-In-First-Out (FIFO) Algorithm
- Optimal Algorithm
- Least Recently Used (LRU) Algorithm
- LRU Approximation Algorithms
- Counting Algorithms
- Page-Buffeing Algorithms

- System commonly keep **a pool of free frames**
  - When replacement occurs, **two frames** are involved
    - 1 **A free frame** from the pool is allocated to the process
      - The desired page is read into the frame
    - 2 **A viction frame** is chosen
      - Written out later and the frame is added to the free pool
  - **NO NEED to write out before read in**
- 
- 1 An expansion
    - Maintain **a list of modified pages**
    - When a paging device is idle, select a modified page, write it out, modify bit $\rightarrow$ 0

## ② Another modification

- Free frame with old page
- The old page can be reused
  - Less write out and less read in
- VAX/VMS
- Some UNIX: + second chance
- ...

## 5 Allocation of Frames

## 1 Minimum number of pages

- Each process needs minimum number of pages
- Determined by ISA (Instruction-Set Architecture )
  - We must have enough frames to hold all the different pages that any single instruction can reference
- 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

## 2 Two major allocation schemes

- Fixed allocation; priority allocation

## 3 Two replacement policy

- Global vs. local

## ① Equal allocation

For example, if there are 100 frames and 5 processes, give each process 20 frames.

$$\text{frame number for any process} = \frac{m}{n}$$

$m$  = total memory frames

$n$  = number of processes

## 2 Proportional allocation

Allocate according to the size of process

• example:

$s_i$  = size of process  $p_i$

$S$  =  $\sum s_i$

$m$  = total number of frames

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

$$m = 64$$

$$S_1 = 10$$

$$S_2 = 127$$

$$a_1 = \frac{10}{137} \times 64 \approx 5$$

$$a_2 = \frac{127}{137} \times 64 \approx 59$$



# Allocation scheme 1: Priority Allocation

- Use a proportional allocation scheme **using priorities** rather than size
- If process  $P_i$  generates a page fault,
  - ① Select for replacement one of its frames
  - ② Select for replacement a frame from a process with lower priority number

- **Global** replacement

process selects a replacement frame from the set of all frames; one process can take a frame from another

- **Problem:** a process cannot control its own page-fault rate

- **Local** replacement

each process selects from only its own set of allocated frames

- Problem?

- 6 Thrashing (抖动)
  - Cause of trashing
  - Working-Set Model (工作集模型)
  - Page-Fault Frequency (缺页频率)

## 6 Thrashing (抖动)

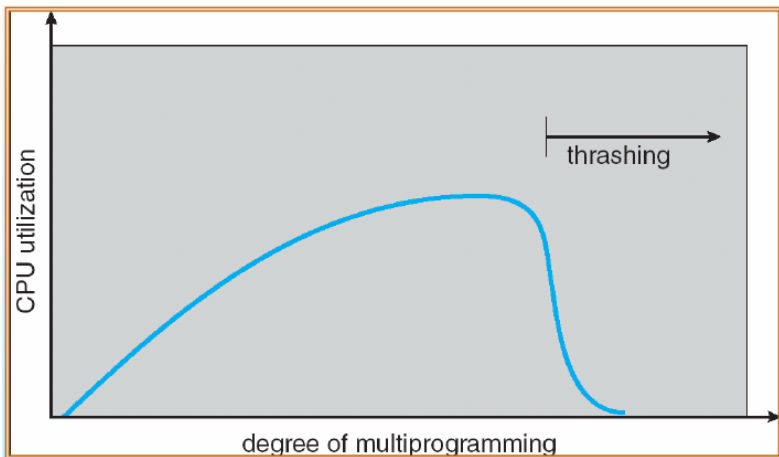
- Cause of thrashing
- Working-Set Model (工作集模型)
- Page-Fault Frequency (缺页频率)

# Thrashing (抖动)

- If a process does not have “enough” pages, the **page-fault rate** is very high. This leads to:
  - Low CPU utilization
  - OS thinks that it needs to increase the degree of multiprogramming
  - Another process added to the system, getting worse!
- **Thrashing**  $\equiv$  a process is busy swapping pages in and out

# Thrashing (抖动)

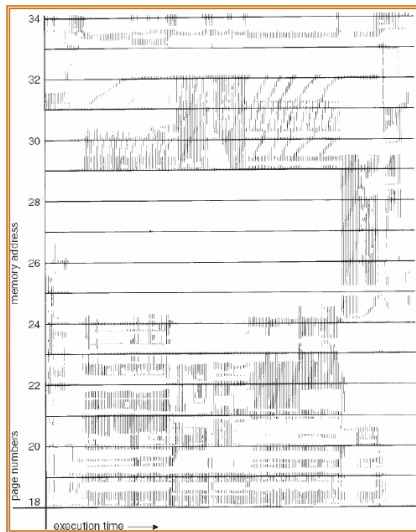
- **Cause of thrashing:** unreasonable degree of multiprogramming (不合理的多道程序度)



# Thrashing (抖动)

- How to limit the effects of thrashing
  - Local replacement algorithm? not entirely solved.
  - **We must provide a process with as many frames as it needs**—locality
  - How do we know how many frames is needed?
    - **working-set strategy**  $\Leftarrow$  **Locality model**
- **Locality model**: This is the reason why demand paging works
  - ① Process migrates from one locality to another
  - ② Localities may overlap
- Why does thrashing occur?  
 $\Sigma$ size of locality  $>$  total memory size

# Thrashing (抖动)





- 6 Thrashing (抖动)
  - Cause of thrashing
  - Working-Set Model (工作集模型)
  - Page-Fault Frequency (缺页频率)

# Working-Set Model (工作集模型)

- The **working-set model** is based on the assumption of **locality**.

- let

$\Delta \equiv$  working – set window

$\equiv$  a fixed number of page references

For example: 10,000 instructions

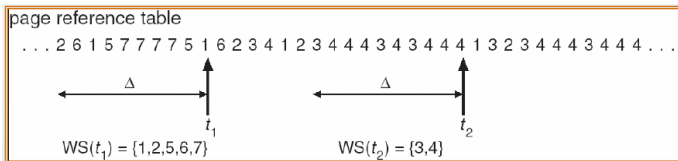
- **Working set (工作集):**

The set of pages in the most recent  $\Delta$  page references.

- An approximation of the program' s locality.

# Working-Set Model (工作集模型)

- Example:  $\Delta = 10$



- Working set size:

$WSS_i$  (working set of Process  $P_i$ )  
= total number of pages referenced in the most recent  $\Delta$  references

- **Varies** in time, depend on the selection of  $\Delta$ 
  - 1 if  $\Delta$  too small will not encompass entire locality
  - 2 if  $\Delta$  too large will encompass several localities
  - 3 if  $\Delta = \infty \Rightarrow$  will encompass entire program

- For all processes in the system, currently

$$D = \sum WSS_i \equiv \text{total demand frames}$$

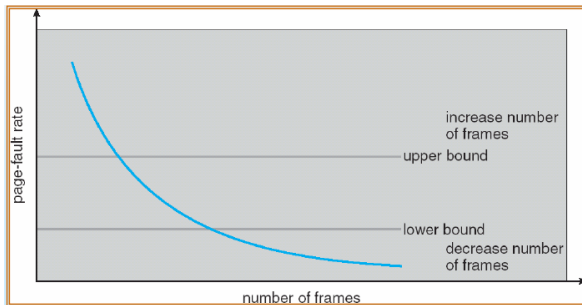
- $D > m \Rightarrow$  Thrashing
- **Policy:**  
if  $D > m$ , then suspend one of the processes

- Approximate with: **interval timer + reference bits**
- Example:  $\Delta = 10,000$ 
  - Timer interrupts after every 5000 time units
  - Keep in memory 2 bits for each page
  - Whenever a timer interrupts, copy and sets the values of all reference bits to 0
  - If one of the bits in memory = 1  $\Rightarrow$  page in working set
- Why is this not completely **accurate**?
  - IN!! But where?
- **Improvement**:
  - 10 bits and interrupt every 1000 time units

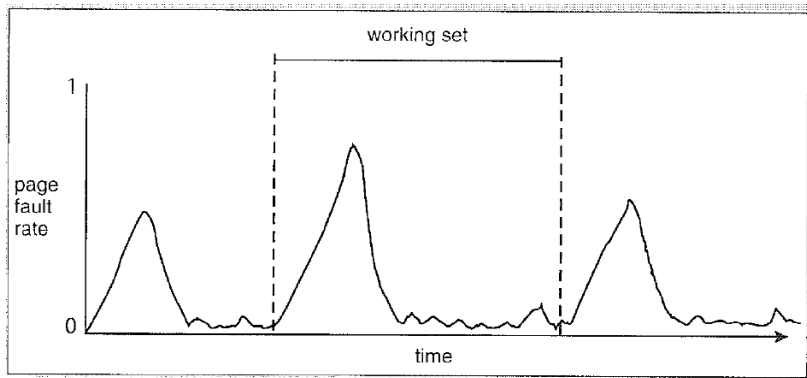
- 6 Thrashing (抖动)
  - Cause of thrashing
  - Working-Set Model (工作集模型)
  - Page-Fault Frequency (缺页频率)

# Page-Fault Frequency Scheme

- **Page-Fault Frequency:** helpful for controlling trashing
  - Trashing has a high page-fault rate.
  - Establish “acceptable” page-fault rate
    - If actual rate too low, process loses frame
    - If actual rate too high, process gains frame



# Working sets and page fault rates





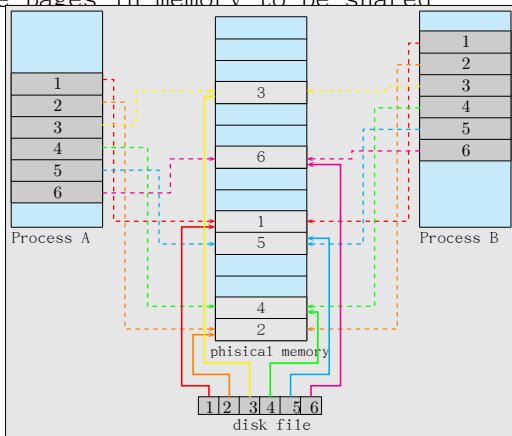
## 7 Memory-Mapped Files

# Memory-Mapped Files

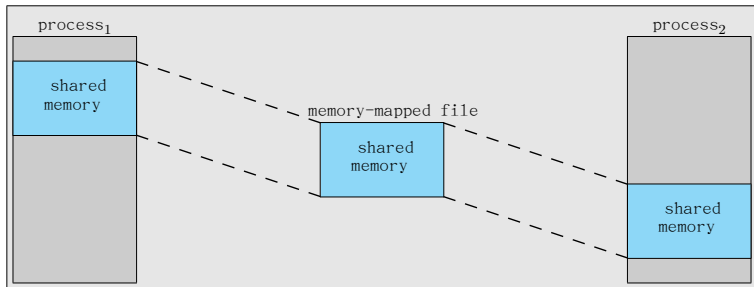
- Memory-mapped file I/O  
allows file I/O to be treated as routine memory access by mapping a disk block to a page in memory
- A file is initially read using demand paging. A page-sized portion of the file is read from the file system into a physical page.  
Subsequent reads/writes to/from the file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

# Memory-Mapped Files

- Also allows several processes to map the same file allowing the pages in memory to be shared



# Shared Memory in Windows using Memory-Mapped I/O



- Many computer architectures provide **memory-mapped I/O**
  - Ranges of memory addresses are set aside and are **mapped to the device registers.**
  - Directly read/write the mapped range of memory address for transfer data from/to device registers
  - Fast response times
  - For example: video controller
    - Displaying text on the screen is almost as easy as writing the text into the appropriate memory-mapped locations.

## 8 Allocating Kernel Memory

# Allocating Kernel Memory

- Kernel memory

Treated differently from user memory

- Process's logical (virtual) address space VS. kernel address space

- different privilege
- allow page fault or not?

- Often allocated from a free-memory pool

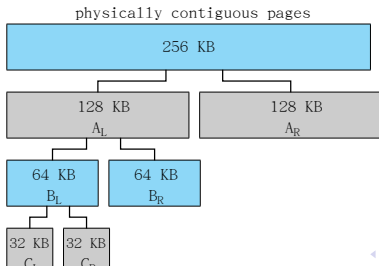
- Kernel requests memory for structures of varying sizes
- Some kernel memory needs to be contiguous

- ① Buddy system (伙伴系统)

- ② Slab allocator (slab分配器)

# 1. Buddy System (伙伴系统)

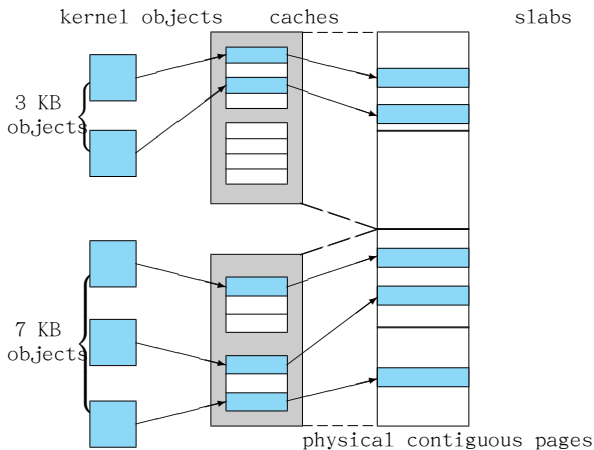
- Allocates memory from **fixed-size segment consisting of physically-contiguous pages**
- Memory allocated using **power-of-2 allocator**
  - Satisfies requests in **units** sized as power of 2
  - Request **rounded up** to next highest power of 2
  - When smaller allocation needed than current size is available, current chunk **split** into two buddies of next-lower power of 2, continue until appropriate sized chunk available





## 2. Slab Allocator (slab分配器) I

- Slab allocator: Alternate strategy



## 2. Slab Allocator (slab分配器) II

- 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
  - 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:** no fragmentation, fast memory request satisfaction

## 9 Other Issues

## ① Prepaging

- To reduce the large number of page faults that occurs 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  $\alpha$  of the pages is used
  - Is cost of  $s * \alpha$  save pages faults  $>$  or  $<$  than the cost of prepagging  $s * (1 - \alpha)$  unnecessary pages?
  - $\alpha$  near zero  $\Rightarrow$  prepagging loses

## ② Page Size

- Page size selection must take into consideration:
  - ① Fragmentation
  - ② Table size
  - ③ I/O overhead
  - ④ Locality

## 8 TLB Reach - The amount of memory accessible from the TLB

- TLB Reach = (TLB Size)  $\times$  (Page Size)
  - Ideally, the working set of each process is stored in the TLB, Otherwise there is a high degree of page faults
  - **Increase the Page Size.**  
This may lead to an increase in fragmentation as not all applications require a large page size
  - **Provide Multiple Page Sizes.**  
This allows applications that require larger page sizes the opportunity to use them without an increase in fragmentation

## 9 Inverted page tables

- This can reduce the memory used to store page tables.
- Need an external page table (one per process) for the information of the logical address space

## 6 Program structure

```
int[128,128] data; // Each row is stored in one page
```

### Program 1

```
for (j = 0; j < 128; j++)  
    for (i = 0; i < 128; i++)  
        data[i,j] = 0;
```

- 128 x 128 = 16,384 page faults

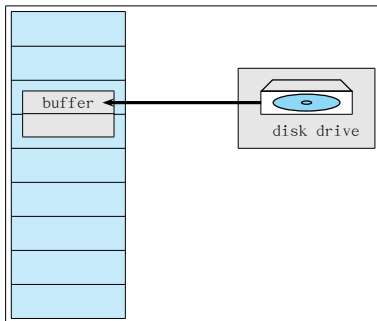
### Program 2

```
for (i = 0; i < 128; i++)  
    for (j = 0; j < 128; j++)  
        data[i,j] = 0;
```

- 128 page faults

## ⑥ 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 in memory

## 10 Operating System Examples

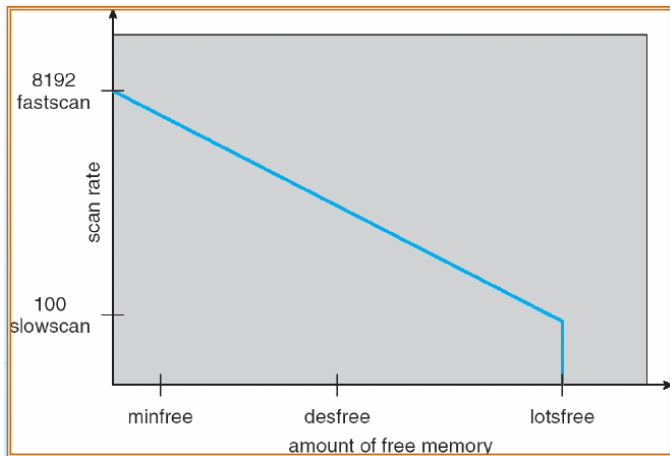


# Operating System Examples

- Windows XP
- Solaris

- Uses **demand paging** with clustering. Clustering brings in pages surrounding the faulting page.
- Processes are assigned **working set minimum** and **working set maximum**
  - 50~345 pages
  - Working set minimum is the minimum number of pages the process is guaranteed to have in memory,
  - A process may be assigned as many pages up to its working set maximum
  - When page fault:
    - if  $<$  working set maximum, allocates a new page
    - if  $=$ max, uses local page-replacement policy
- 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

- Maintains a list of free pages to assign faulting processes
  - Parameter `lotsfree`— threshold (amount of free memory) to begin paging,  $1/64$  the size of physical memory
  - check the amount of free pages 4 times per second
- Paging is performed by pageout process using modified second-chance algorithm (with two hands)
  - `Desfree`— threshold parameter to increasing paging
  - `Minfree`— threshold parameter to being swapping
  - `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

## 11 小结和作业

# 小结

- 1 Background
- 2 Demand Paging (按需调页)
- 3 Copy-on-Write (写时复制)
- 4 Page Replacement (页面置换)
- 5 Allocation of Frames
- 6 Thrashing (抖动)
- 7 Memory-Mapped Files
- 8 Allocating Kernel Memory
- 9 Other Issues
- 10 Operating System Examples
- 11 小结和作业

- 10.3 某个计算机给它的用户提供了 $2^{32}$ B的虚拟地址空间。计算机有 $2^{18}$ B的物理内存。虚拟内存使用页面大小为4KB的分页机制实现。一个用户进程产生虚拟地址11123456，现在说明一下系统怎样建立相应的物理地址。区分一下软件操作和硬件操作。
- 10.4 对于请求调页，下面变成技巧和结构哪些“好”？哪些“坏”？为什么？
  - a, 堆栈
  - b, 哈希表
  - c, 顺序检索
  - d, 二分法检索
  - e, 纯代码
  - f, 向量操作
  - g, 间接寻址

- 10.10 假设有二维数组A:

```
int A[][] = new int[100][100];
```

在一个页面大小为200的分页内存系统中，A[0][0]存放在地址200中。一个操作数组A的小进程驻存在页面0（地址0到199）；这样，每条指令都将从页面0中获取。

对于3个页帧，下面2中不同的数组初始化循环将分别产生多少个页错误？假设使用LRU置换算法，页帧1中存放进程，另外两个初始时空。

- a,

```
for (int j=0;j<100;j++)  
    for (int i=0;i<100;i++)  
        A[i][j]=0;
```

- b,



```
for (int i=0;i<100;i++)  
    for (int j=0;j<100;j++)  
        A[i][j]=0;
```

- 10.11 假设有下面引用序列:

1, 2, 3, 4, 2, 1, 5, 6, 2, 1, 2, 3, 7, 6, 3, 2, 1, 2, 3, 6

下面的页面置换算法会产生多少次缺页异常？分别假设帧有1、2、3、4、5、6、7个。所有的帧初始时空。第一个页调入时都会引发一次页错误。

- LRU置换算法
  - FIRO置换算法
  - 最优置换算法
- 10.20 抖动的原因是什么？系统怎样检测抖动？一旦系统检测到抖动，系统怎样消除这个问题？

“纸上得来终觉浅，绝知此事要躬行”——宋·陆游《冬夜读书示子聿·选一》

谢谢！