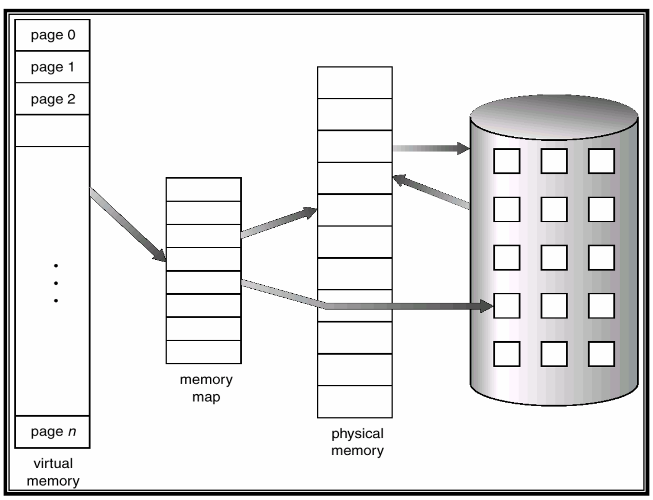
Operating System Software to support Virtual Memory

Chapter 8.2, Livro do William Stallings

SISTEMAS OPERATIVOS, 1º Semestre, 2004-2005

A VM Larger Than Physical Memory



Operating System Policies for Virtual Memory

- **Fetch Policy**
  - Demand
  - Prepaging
- **VM: Files and Processes**
- **Placement Policy**
- **Replacement Policy**

  Basic Algorithms:
  - Optimal
  - LRU
  - FIFO
  - Clock
  - Page Buffering
- **Resident Set Management**
  - Resident Set Size
    - Fixed
    - Variable
  - Replacement Scope
    - Global
    - Local
- **Cleaning Policy**
  - Demand
  - Precleaning
- **Load Control**
  - Degree of multiprogramming

Fetch Policy

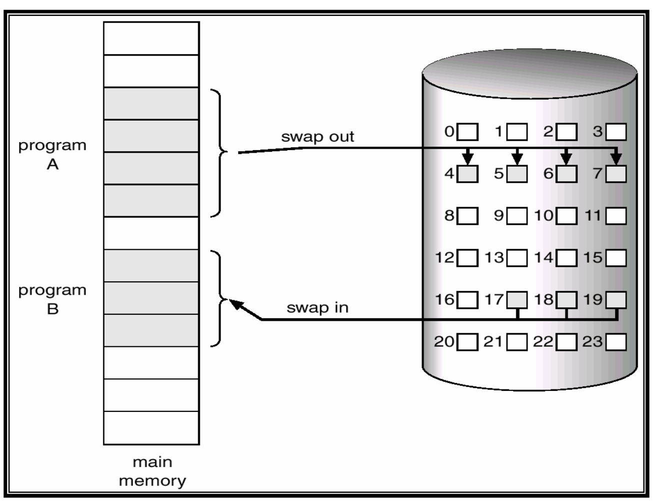
- **Fetch Policy**
  - Determines when a page should be brought into memory.
  - **Demand paging** only brings pages into main memory when a reference is made to a location on the page.
    - Many page faults when process first started.
  - **Prepaging** brings in more pages than needed
    - More efficient to bring in pages that reside contiguously on the disk.

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 $\Rightarrow$ reference to it
  - invalid reference $\Rightarrow$ abort
  - not-in-memory $\Rightarrow$ bring to memory

Demand Paging



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

  $+ p$ [page fault overhead]

[swap out + swap in + OS overhead]

Example

- Average page fault overhead: 25 milliseconds
- Memory access time: 100 nanoseconds
- EAT = (1 – p) x (100) + p (25,000,000)
  = 100 + 24,999,900 x p

EAT: is directly proportional to the page-fault rate.
If one access out of 1000 causes a page-fault,
p = 0.001 → EAT = 25 microseconds.
The computer will be slowed down by a factor of 250!...

Another Example

- Memory access time = 40 ns (4x10-8s)
- 50% of time page being replaced has been modified and so must be written back to disk.
- Each page fault: swap in page, 50% written back.
- Swap Page Time (IN or OUT) = 10 msec = 1x10-2s

EAT = (1 – p) x 4 x 10-8s + *p* (1.5 x 10-2s + $_{\text{OS\_Overhead}}$)
= 4x10-8s + 0.01499996*p*s
= (40 + 14999960*p*)ns

Virtual Memory: Processes and Files

Virtual Memory: Processes and Files

- Virtual memory allows other benefits during process creation and when using files:

  - **Copy-on-Write**

  - **Memory-Mapped Files**

Copy-on-Write

- Copy-on-Write 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.

- Copy-on-Write allows more efficient process creation as only modified pages are copied.

Copy-on-Write vs vfork()

- Copy-on-Write is used in many operating systems including Linux, Windows 2000/XP, and Solaris.

- Some systems use a version of fork() - vfork().

- vfork() is different than copy-on-write.
- vfork() suspends the parent until the child has called exec() or has exited.
- vfork() is intended for use when the child calls exec() immediately after being forked. If the child does anything to the address space of the parent before calling exec(), the changes will be reflected in the parent as well.

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.

- Allows several processes to map the same file allowing the pages in memory to be shared.

Memory-Mapped Files: API

- **Mapping a file**

  void *mmap(void **addr*, size_t *len*, int *prot*, int *flags*, int *fildes*, off_t *off*);

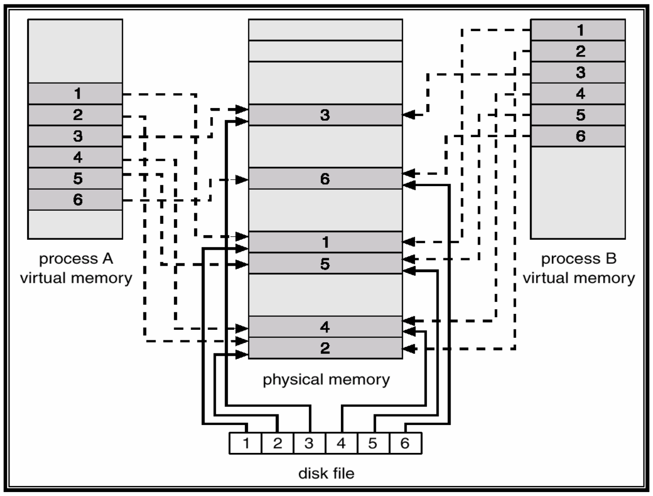
- **Unmapping the file**

  int munmap(caddr_t *addr*, size_t *len*);

Memory-Mapped Files: Example

```
int main(int argc, char *argv[])
  {
     int fd, offset;
     char *data;
     struct stat sbuf;

       if ((fd = open("xpto", O_RDONLY)) == -1) {
          perror("open");
          exit(1);
     }

     if ((data = mmap((caddr_t)0, sbuf.st_size, PROT_READ, MAP_SHARED, fd, 0)) == (caddr_t)(-1)){
               perror("mmap");
               exit(1);
     }

     offset = 10; // vai buscar o 10º byte do ficheiro...
     printf("byte at offset %d is '%c'\n", offset, data[offset]);
     return 0;
  }
```

Memory-Mapped Files

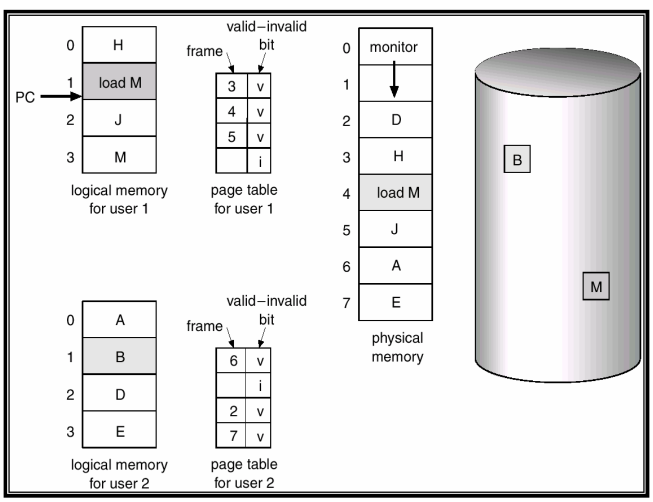


Replacement Policy

Replacement Policy

- **Placement Policy**
  - Which page is replaced?
  - Use *dirty* (*modify*) bit to reduce overhead of page transfers – only modified pages written back to disk.
  - Page removed should be the page least likely to be referenced in the near future.
  - Most policies predict the future behavior on the basis of past behavior.

Need for Page Replacement



Basic Page Replacement

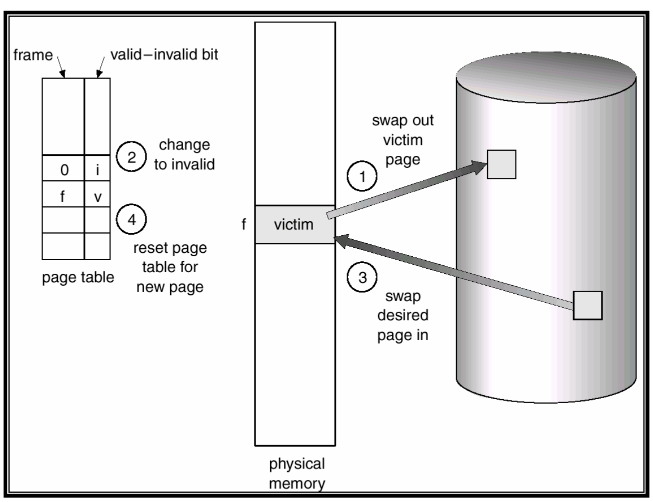
1. Find the location of the desired page on disk.

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.
   - If victim is *dirty* write back to disk

3. Read the desired page into the (newly) free frame. Update the page and frame tables.

4. Restart the process.

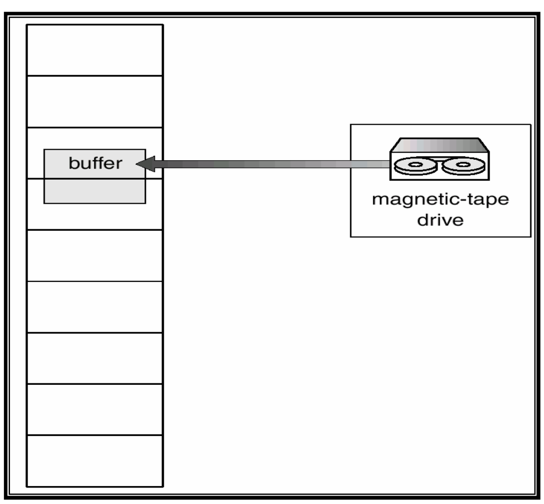
Example



Page Locking and the Replacement Policy

- **Frame Locking**
  - If frame is locked, it may not be replaced
  - Kernel of the operating system
  - Control structures
  - I/O buffers (Pages that are used for copying a file from a device must be locked from being selected for eviction by a page replacement algorithm).
  - Associate a lock-bit with each frame

Frames Used For I/O Must Be In Memory



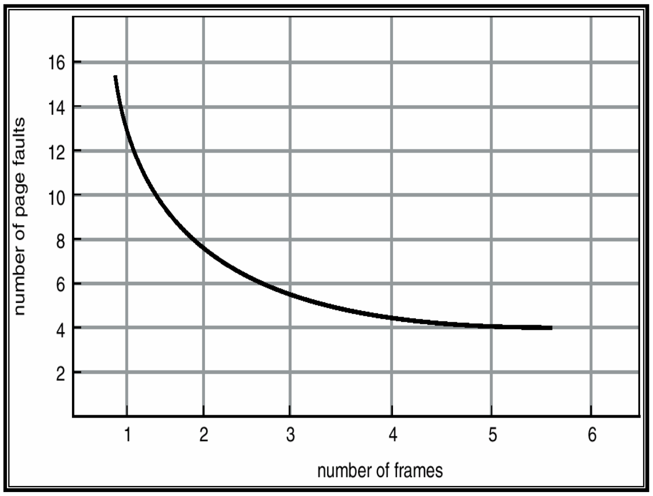
Page-Replacement Algorithms

Page-Replacement Algorithms

- Goal: get the lowest page-fault rate
- Evaluate algorithm by running on given string of memory references and compute the number of page faults
- Example of a reference string:

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

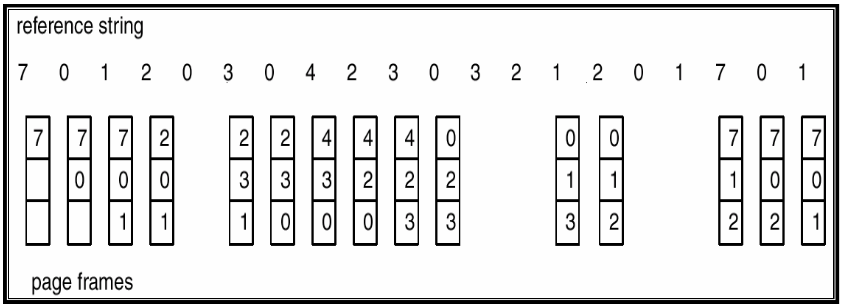
Graph of Page Faults Versus The Number of Frames



Replacement Algorithm: FIFO

- **First-in, first-out (FIFO)**
  - Treats page frames allocated to a process as a circular buffer.
  - Pages are removed in round-robin style.
  - Simplest replacement policy to implement.
  - Page that has been in memory the longest is replaced.
  - These pages may be needed again very soon.

FIFO Page Replacement



FIFO Algorithm

- Reference string: 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5
- 3 frames (for *all* processes: 3 pages in memory at a time)

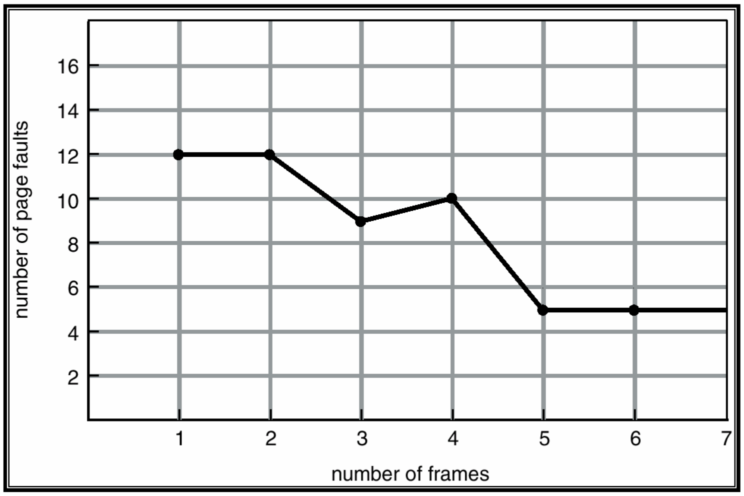
1	1	4	5	
2	2	1	3	9 page faults
3	3	2	4	

- 4 frames

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

- FIFO replacement bad, can lead to – **Belady's Anomaly**
  - more frames $\Rightarrow$ *more* page faults !!

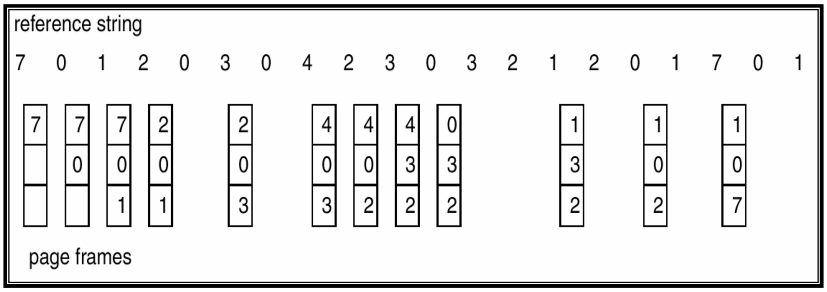
Belady's Anomaly



Replacement Algorithm: LRU

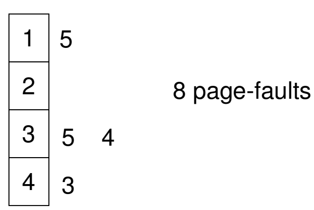
- **Least Recently Used (LRU)**
  - Replaces the page that has not been referenced for the longest time.
  - By the principle of locality, this should be the page least likely to be referenced in the near future.
  - Each page could be tagged with the time of last reference. This would require a great deal of overhead.

LRU Page Replacement



LRU Algorithm

- Reference string: 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5

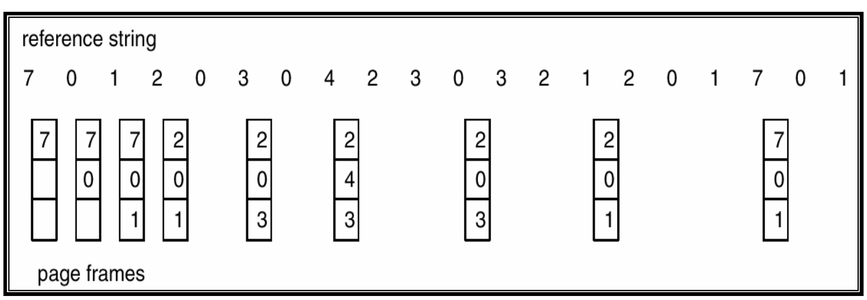


- Counter implementation
  - every page table entry has timestamp; every time page referenced through entry, copy clock (counter) into timestamp
  - when page needs to be replaced, look at timestamp to choose

Optimal Replacement Algorithm

- **Optimal policy**
  - Selects for replacement that page for which the time to the next reference is the longest.
  - Impossible to have perfect knowledge of future events.

Optimal Page Replacement



Optimal Algorithm

- Replace page not to be used for longest time in future
- 4 frames example: 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5

1	4	
2		6 page faults
3		
4	5	

- How do you know which frame to replace?
- Useful as standard for comparing realistic algorithms

Clock Replacement Algorithm

- **Clock Policy**
  - Additional bit called a use bit.
  - When a page is first loaded in memory, the use bit is set to 0.
  - When the page is referenced, the use bit is set to 1.
  - When it is time to replace a page, the first frame encountered with the use bit set to 0 is replaced.
  - During the search for replacement, each use bit set to 1 is changed to 0.

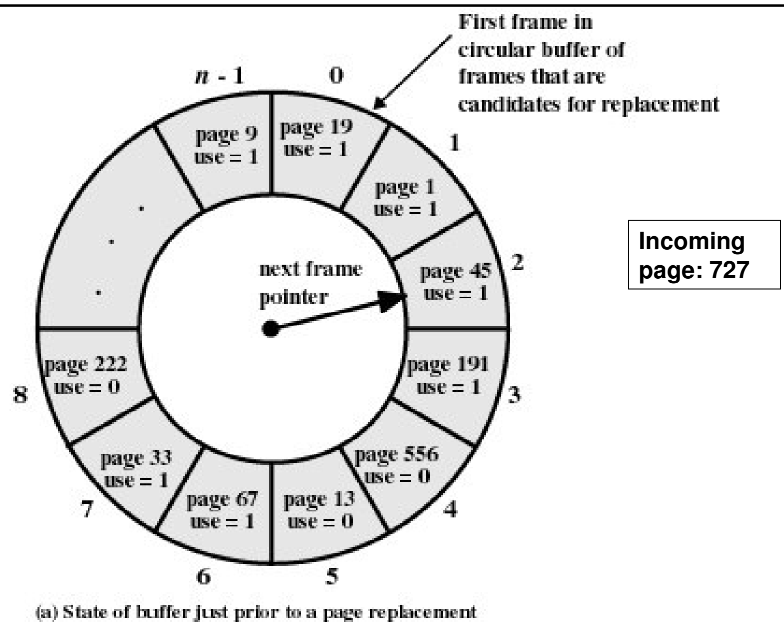


Figure 8.16 Example of Clock Policy Operation

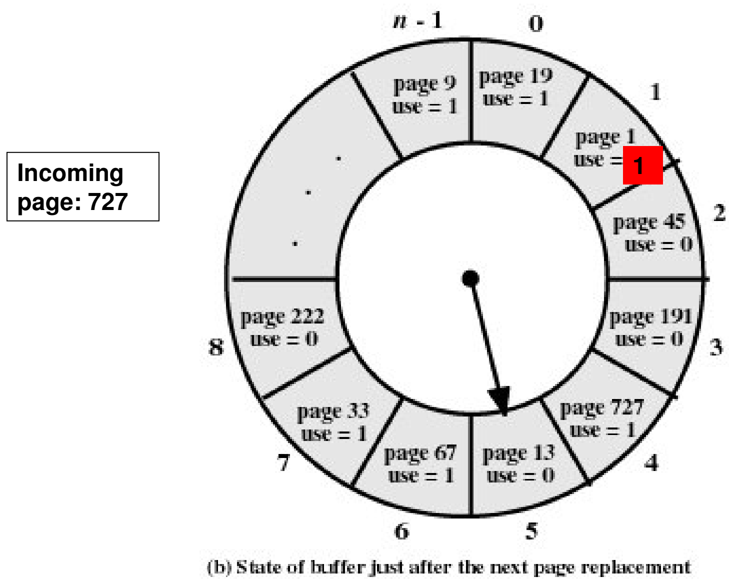
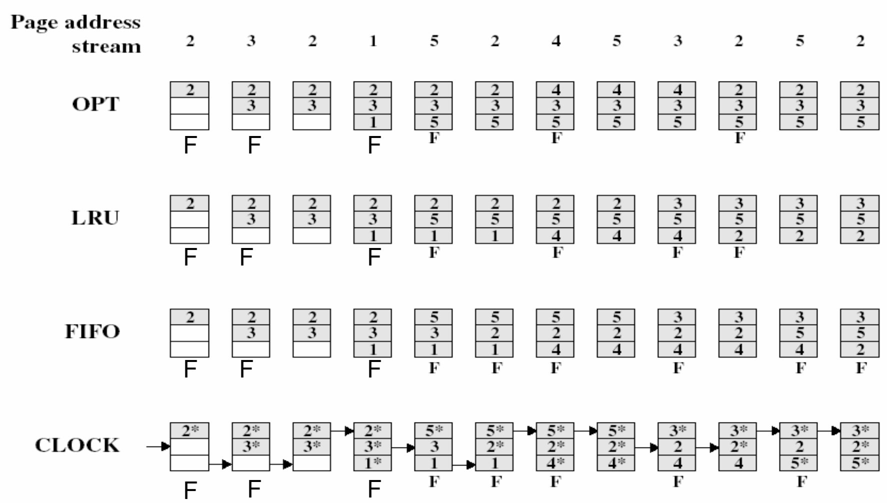
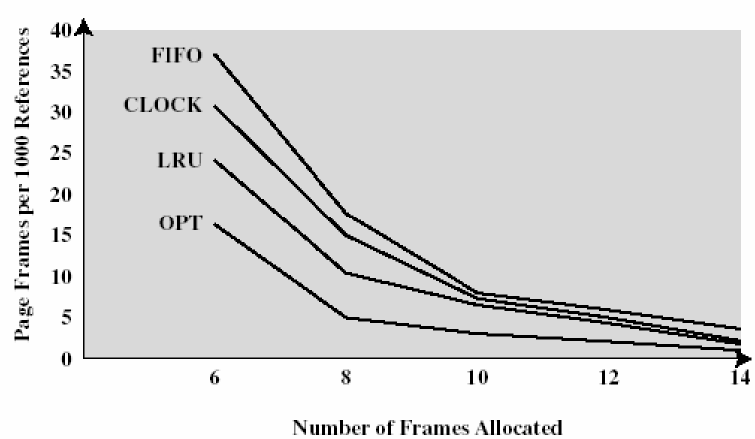


Figure 8.16 Example of Clock Policy Operation

Page Replacement Algorithms



Local Page Replacement Algorithms



goto QUIZ#5...

Resident Set Size

- **Fixed-allocation**
  - gives a process a fixed number of pages within which to execute.
  - when a page fault occurs, one of the pages of that process must be replaced.
- **Variable-allocation**
  - number of pages allocated to a process varies over the lifetime of the process.

Replacement Scope

- **Local replacement policy**:
  - Chooses only among the resident pages of the process that generated the page fault in selecting a page to replace.
- **Global replacement policy**:
  - Considers all unlocked pages in main-memory as candidates for replacement.

Fixed Location, Local Scope

- A process is running in memory with a fixed number of frames.
- When a page fault occurs the OS must choose which page to replace, among the resident pages of this process.
- With a fixed-allocation policy it is necessary to decide ahead of time the amount of allocation to give to a process.
- Trade-off decision...

Variable Allocation, Global Scope

- Easiest to implement; adopted by many operating systems.
- Operating system keeps list of free frames.
- Free frame is added to resident set of process when a page fault occurs.
- If no free frame, replaces one from another process.

Variable Allocation, Local Scope

- When new process added, allocate number of page frames based on application type, program request, or other criteria.
- When page fault occurs, select page from among the resident set of the process that suffers the fault.
- Reevaluate allocation from time to time.

Cleaning Policy

- **Demand cleaning**
  - – a page is written out only when it has been selected for replacement.
- **Pre-cleaning**
  - – pages are written out in batches.

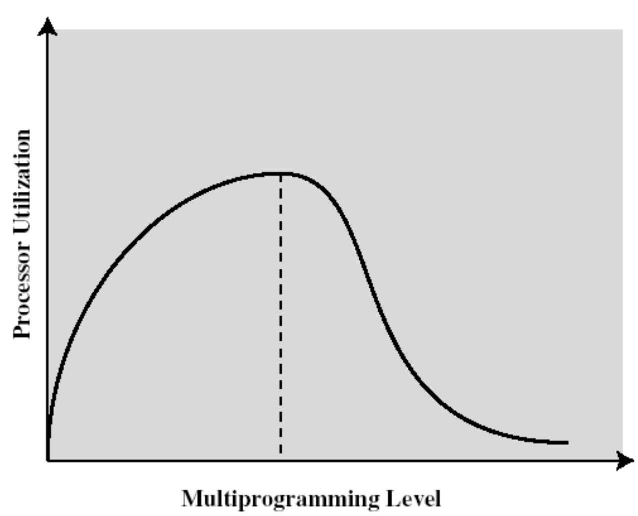
Cleaning Policy

- Best approach uses **page buffering**
  - – Replaced pages are placed in two lists
    - Modified and unmodified.
  - – Pages in the modified list are periodically written out in batches.
  - – Pages in the unmodified list are either reclaimed if referenced again or lost when its frame is assigned to another page.

Load Control

- Determines the number of processes that will be resident in main memory.
- Too few processes, many occasions when all processes will be blocked and much time will be spent in swapping.
- Too many processes will lead to: **thrashing**.

Multiprogramming Effects

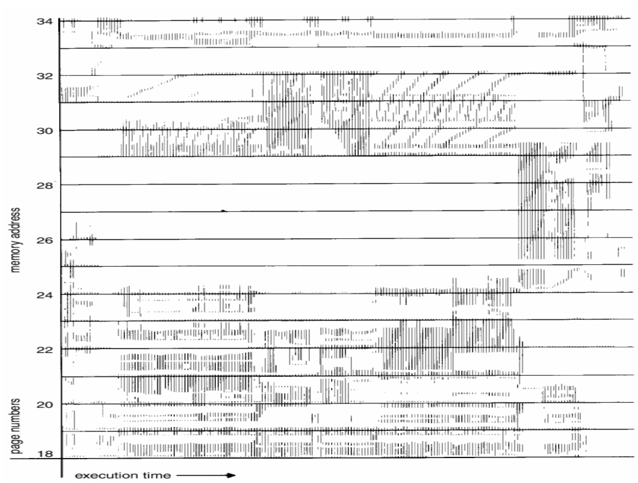


Why Trashing occurs?

- 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 memory size

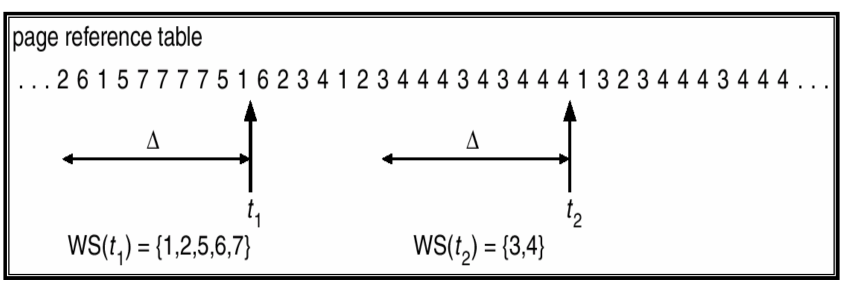
Locality in a Memory-Reference Pattern



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.
- $D = \Sigma\ WSS_i \equiv$ total demand frames
- (m = total number of page frames)
- if $D > m \Rightarrow$ Thrashing
- Policy: if $D >$ m, then suspend one of the processes.

Working Set Model

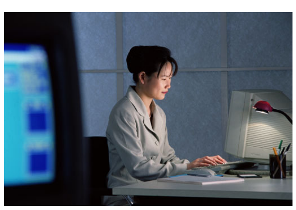


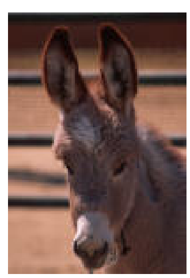
Exemplo: dois programas...

```
int A[][] = new int[128][128];
int i,j;

// page size = 512 bytes
// each row is stored in one
// page


// Programa A
for (j = 0; j < 128; j++)
    for (i = 0; i < 128; i++)
            A[i][j] = 0;
```

```
int A[][] = new int[128][128];
int i,j;

// page size = 512 bytes
// each row is stored in one
// page


// Programa B
for (i = 0; i < 128; i++)
    for (j = 0; j <  128; j++)
            A[i][j] = 0;
```

Quem escreveu o programa A?
E o programa B?



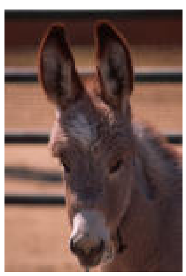


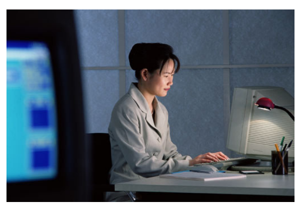
Exemplo: dois programas...

int A[][] = new int[128][128];
 	
// Programa A
for (j = 0; j < 128; j++)
for (i = 0; i < 128; i++)
A[i][j] = 0;
	int A[][] = new int[128][128];
	
	// Programa B
	for (i = 0; i < 128; i++)
	for (j = 0; j < 128; j++)
	A[i][j] = 0;

128 x128 = 16,384 page faults

128 page faults





int A[][] = new int[128][128];
 	
// Programa A
for (j = 0; j < 128; j++)
for (i = 0; i < 128; i++)
A[i][j] = 0;
	int A[][] = new int[128][128];
	
	// Programa B
	for (i = 0; i < 128; i++)
	for (j = 0; j < 128; j++)
	A[i][j] = 0;

128 x128 = 16,384 page faults

128 page faults

Process Suspension

- **Lowest priority process.**
- **Faulting process**
  - this process does not have its working set in main memory so it will be blocked anyway.
- **Last process activated**
  - this process is least likely to have its working set resident.

Process Suspension

- **Process with smallest resident set**
  - this process requires the least future effort to reload
- **Largest process**
  - obtains the most free frames
- **Process with the largest remaining execution window**.

Overview of Virtual Memory Systems:

- Unix and Solaris
- Linux
- Windows 2000

UNIX and Solaris Memory Management

- Two separate memory management schemes in Unix SVR4 and Solaris
  - Paging System – Allocate page frames
  - Kernel Memory Allocator – Allocate memory for the kernel
- Paging System Data Structures
  - Page Table – One entry for each page of virtual memory for that process
    - Page Frame Number – Physical frame #
    - Age – How long in memory w/o reference
    - Copy on Write – Are two processes sharing this page: after fork(), waiting for exec()
    - Modify – Page modified?
    - Reference – Set when page accessed
    - Valid – Page is in main memory
    - Protect – Are we allowed to write to this page?

Paging Data Structures

- Disk Block Descriptor
  - Swap Device Number – Logical device
  - Device Block Number – Block location
  - Type of storage – Swap or executable, also indicate if we should clear first
- Page Frame Data Table
  - Page State – Available, in use (on swap device, in executable, in transfer)
  - Reference Count – # processes using page
  - Logical Device – Device holding copy
  - Block Number – Location on device
  - Pfdata pointer – For linked list of pages
- Swap-use Table
  - Reference Count – # entries pointing to a page on a storage device
  - Page/storage unit number – Page ID

Data Structures

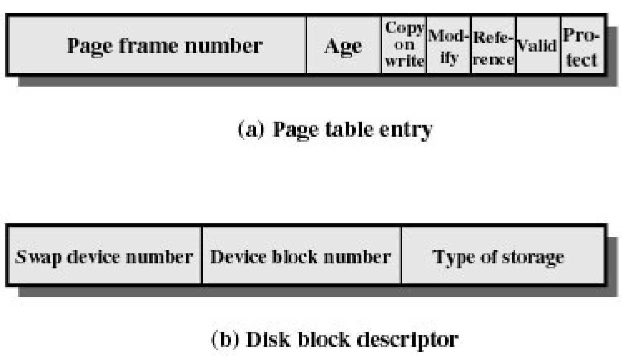


Figure 8.22 UNIX SVR4 Memory Management Formats

Data Structures

Page state	Reference count	Logical device	Block number	Pfdata pointer

(c) Page frame data table entry

Reference count	Page/storage unit number

(d) Swap-use table entry

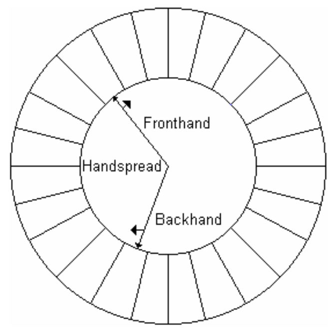
Figure 8.22 UNIX SVR4 Memory Management Formats

UNIX and Solaris Memory Management

- Page Replacement
  - refinement of the clock policy
- Kernel Memory Allocator
  - most blocks are smaller than a typical page size

Unix SVR4 Page Replacement

- Clock algorithm variant
- *Fronthand* – Clear Use bits
- *Backhand* – Check Use bits, if use=0 prepare to swap page out
- *Scanrate* – How fast the hands move
  - Faster rate frees pages faster
- *Handspread* – Gap between hands
  - Smaller gap frees pages faster
- System adjusts values based on free memory



Linux Memory Management

- Virtual Memory Addressing
  - Supports 3-level page tables
    - Page Directory
      - One page in size (must be in memory)
    - Page Middle Directory
      - Can span multiple pages
      - Will have size=1 on Pentium
    - Page Table
      - Points to individual pages
- Page Allocation
  - Uses a buddy system with 1-32 page block sizes
- Page Replacement
  - Based on clock algorithm
  - Uses age variable
    - Incremented when page is accessed
    - Decremented as it scans memory
    - When age=0, page may be replaced
  - Has effect of least frequently used method
- Kernel Memory Allocation
  - Uses scheme called *slab allocation*
  - Blocks of size 32 through 4080 bytes

Windows 2000 Memory Management

- Virtual Address Map
  - 00000000 to 00000FFF – Reserved
    - Help catch NULL pointer uses
  - 00001000 to 7FFFEFFF – User space
  - 7FFFEFFF to 7FFFFFFF – Reserved
    - Help catch wild pointers
  - 80000000 to FFFFFFFF – System
- Page States
  - Available – Not currently used
  - Reserved – Set aside, but not counted against memory quota (not in use)
    - No disk swap space allocated yet
    - Process can declare memory that can be quickly allocated when it is needed
  - Committed: space set aside in paging file (in use by the process)

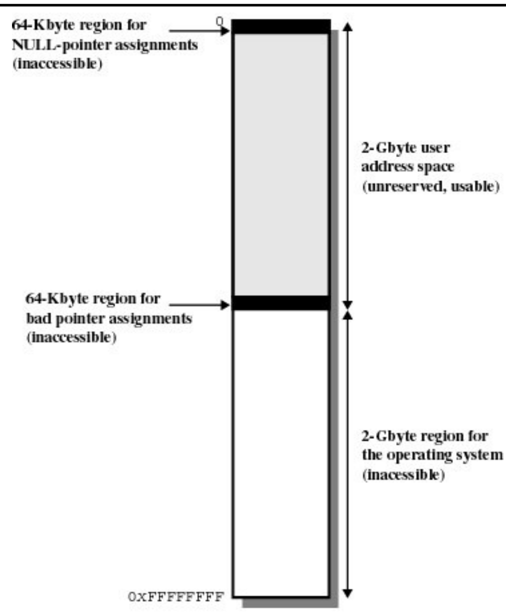


Figure 8.25 Windows 2000 Default Virtual Address Space

Windows 2000 Memory Management

- Uses variable allocation, local scope.
- When a page fault occurs, a page is selected from the local set of pages.
- If main memory is plentiful, allow the resident set to grow as pages are brought into memory.
- If main memory is scarce, remove less recently accessed pages from the resident set.