Virtual Memory (CSE 422S)

Ken Wong
Washington University

kenw@wustl.edu
www.arl.wustl.edu/~kenw

Virtual Memory

Map memory addresses at run-time
» Physical (actual) address = f(logical address)
» Typically, f( ) is implemented as a page table

Basic Ideas
» Hide details of real physical memory from user
» Each user has n contiguous (linear) address spaces
  • Each begins at address 0
  • Paging (n = 1) versus Segmentation (n ≥ 1)

Virtual Memory Overview

- Real Memory: Main memory
- Virtual Memory: The memory perceived by the user or programmer
  » Implemented through paging and segmentation with page swapping

Properties of Paging and Segmentation
- Dynamic Address Translation: Memory references are logical addresses that are dynamically translated into physical addresses at run time.
- Non-contiguous Main Memory: A process may be broken up into pieces that need not be contiguously located in main memory.

Potential Benefits
- Effective Multiprogramming: More processes in main memory
- Less Memory Constrained: Loosens memory constraints
- Protection: Memory references only to own physical memory

Memory Management Requirements

- Relocation
  » Easily swap process (in part or entirely) in and out of memory

- Protection
  » Process should be protected by unauthorized access

- Sharing
  » Allow processes to share data and programs in memory

- Support for Program Modules
  » Treat program modules as units (e.g., segmentation)

- Effective Memory Usage
  » Keep most active parts of a process in main memory
  » Store the rest on secondary storage
Demand Paging

- Bring page into memory only when it is needed
  - Less I/O needed
  - Less memory needed
  - Faster response
  - More users
- Page is needed only when it is referenced
  - Abort invalid references
  - Swap in pages when referenced but not in main memory
- Effective only when there is a large degree of locality over short time periods.
- **Thrashing**: A condition in which a processor spends most of its time swapping pages rather than executing instructions.

Simple Paging

- Basic Idea
  - Break processes into equal sized *pages*
  - e.g., 1 KB = 2^10 bytes
  - Break real memory into the same sized *page frames*
  - Load process pages into frames
    - Not necessarily contiguous frames
  - Some internal fragmentation (last page), but no external fragmentation
- Like fixed partitioning, but:
  - Partitions are small
  - A process can occupy more than 1 partition
  - Partitions need not be contiguous

Simple Paging Example

<table>
<thead>
<tr>
<th>Frame</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>A.0</td>
<td>A.0</td>
<td>A.0</td>
<td>A.0</td>
<td>A.0</td>
<td>A.0</td>
<td>A.0</td>
<td>A.0</td>
</tr>
<tr>
<td></td>
<td>A.1</td>
<td>A.1</td>
<td>A.1</td>
<td>A.1</td>
<td>A.1</td>
<td>A.1</td>
<td>A.1</td>
<td>A.1</td>
</tr>
<tr>
<td></td>
<td>B.0</td>
<td>B.0</td>
<td>B.0</td>
<td>B.0</td>
<td>B.0</td>
<td>B.0</td>
<td>B.0</td>
<td>B.0</td>
</tr>
<tr>
<td></td>
<td>B.1</td>
<td>B.1</td>
<td>B.1</td>
<td>B.1</td>
<td>B.1</td>
<td>B.1</td>
<td>B.1</td>
<td>B.1</td>
</tr>
<tr>
<td></td>
<td>C.0</td>
<td>C.0</td>
<td>C.0</td>
<td>C.0</td>
<td>C.0</td>
<td>C.0</td>
<td>C.0</td>
<td>C.0</td>
</tr>
<tr>
<td></td>
<td>D.0</td>
<td>D.0</td>
<td>D.0</td>
<td>D.0</td>
<td>D.0</td>
<td>D.0</td>
<td>D.0</td>
<td>D.0</td>
</tr>
<tr>
<td></td>
<td>D.1</td>
<td>D.1</td>
<td>D.1</td>
<td>D.1</td>
<td>D.1</td>
<td>D.1</td>
<td>D.1</td>
<td>D.1</td>
</tr>
<tr>
<td></td>
<td>D.2</td>
<td>D.2</td>
<td>D.2</td>
<td>D.2</td>
<td>D.2</td>
<td>D.2</td>
<td>D.2</td>
<td>D.2</td>
</tr>
<tr>
<td></td>
<td>D.3</td>
<td>D.3</td>
<td>D.3</td>
<td>D.3</td>
<td>D.3</td>
<td>D.3</td>
<td>D.3</td>
<td>D.3</td>
</tr>
</tbody>
</table>

Address Translation (Paging)

```
0000010111011110
```

- Logical Address
- Page #
- Offset
- Process Page Table:
  - 0 1 2
  - 0 0 0 1 0
  - 0 0 0 1 1
  - 0 0 0 1 0 1

- Physical Address
### Typical Virtual Memory Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Page Size</td>
<td>512 to 8KB</td>
</tr>
<tr>
<td>Hit Time</td>
<td>5 to 100 nsec</td>
</tr>
<tr>
<td>Miss Penalty</td>
<td>5 to 30 msec</td>
</tr>
<tr>
<td>Main Memory Size</td>
<td>64 MB to 2 GB</td>
</tr>
<tr>
<td>Desired Hit Rate</td>
<td>$\geq 99.99%$</td>
</tr>
</tbody>
</table>

- **Page fault** occurs when program references a page frame that is not in main memory.

### Paged Virtual Memory Issues

- **How is a page frame found if it is in main memory?**
  - Page Table: Physical address = <Frame #, Offset>

- **How large should a page frame be?**
  - Big page: 1) Small page table; 2) more efficient read/write; 3) greater internal fragmentation; 4) higher process load time
  - Which page frame should be replaced on a virtual memory miss?
  - The one that is the least likely to be referenced in the future

- **Where can a page be placed in memory?**
  - In almost any free page frame

- **How long does it take to translate a virtual address?**
  - Ideal: no more than 1 clock cycle
  - Need associative hardware

- **Which page frame should be replaced on a virtual memory miss?**
  - The one that is the least likely to be referenced in the future

- **When should a page frame be written back to the swap device (disk)?**
  - Only if it has been modified (it is dirty) and as late as possible

### Matrix Multiply (1)

```c
int A[N][N], B[N][N], C[N][N];
for (int i=0; i<N; i++)
  for (int j=0; j<N; j++)
    for (int k=0; k<N; k++)
      C[i][j] += A[i][k] * B[k][j];
```

- **Highest locality:** Instructions and A matrix
- **What if 1 page will hold 1 row?**

### Matrix Multiply (2)

- **Access to instructions and A[][] show high temporal locality**
- **What if inner loop is:**

```c
R1 < 0; // accumulator
for (int k=0; k<N; k++) {
  R2 < A[i][k]; // Load R2 with A[i][k]
  R3 < B[k][j]; // Load R3 with B[k][j]
  R4 < R2 * R3; // A[i][k] + B[k][j]
  R1 < R1 + R4; // C[i][j] += ...
}
```

- // update pointers at end of loop

```c
R1 < C[i][j]; // Store value of C[i][j]
```
Matrix Multiply (3)

- Composite memory reference string \((n = N-1)\):

\[
\begin{align*}
II, A0, B0, I, I, I, I, : II, A0, Bn, I, I, I, : CO // C0[0] \\
II, A0, B0, I, I, I, I, : II, A0, Bn, I, I, I, : CO // C0[1] \\
\vdots \\
\end{align*}
\]

- What if physical memory allocation of 4 frames
  - Initial Placement:
    - \(I \rightarrow \text{page 0, } A[0][\ast] \rightarrow 1, B[0][\ast] \rightarrow 2, C[0][\ast] \rightarrow 3\)
  - Page reference string \((F = \text{Page Fault})\):
    - \(0,0,1,0,0,0,0,0; 0,0,1,0,0,0,0,0,0,0; 0,0,1,0,0,0,0,0,0,0; \ldots\)
  - \(9N+1\) memory references per row
  - \(N^2\) rows

Page Fault Rate

- What if inner loop is:

\[
\begin{align*}
R1 & \leftarrow 0; // accumulator \\
& \text{for (int } k=0; k<N; k++) \{ // test and jump} \\
& R2 \leftarrow A[i][k]; // Load R2 with A[i][k] \\
& R3 \leftarrow B[k][j]; // Load R3 with B[k][j] \\
& R4 \leftarrow R2 * R3; // A[i][k] + B[k][j] \\
& R1 \leftarrow R1 + R4; // C[i][j] += ... \\
& \} // update pointers at end of loop \\
& R1 \rightarrow C[i][j]; // Store value of C[i][j] \\
\end{align*}
\]

- The number of page faults per memory reference
- About \(1/9\) for large \(N\)
- More precisely
  - \((9N+1) \times N^2\) page references
  - \(N^3 + N\) page faults ➔ Page fault rate of \((N^3 + N)/((9N+1) \times N^2)\)

Basic Page Table Structure

- Virtual Address
  - page\# offset
- Physical Address
  - frame\# offset
- Page Table Address
- Page Table
- Page Frame
- Valid Bit (In memory?)
- Modified Bit (Dirty?)

Huge Virtual Address Spaces

- If the page size is 1 KB in a 32-bit virtual address space, how many entries will a single-level page table have?
  - \(2^{32} / 2^{10} = 2^{22}\), about 4 million entries
- How many pages will be occupied by the page table if each row is 32 bits (4 bytes)?
  - \(2^{22} 2^2 / 2^{10} = 2^{14} = 16\) K pages!
    - Page tables can be huge and are subject to paging
- If there are 128 processes, how space is needed to store the worst-case page tables?
  - \(128 \times 16\) K pages = \(128 \times 16\) K x 1 KB = 2 GB
### Two-Level Page Table (1)

- **Example:** Pentium

- Indexes into page table broken into 1-page subtables

```
<table>
<thead>
<tr>
<th>Logical Address</th>
<th>Physical Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1   P2   D</td>
<td>F    D</td>
</tr>
</tbody>
</table>
```

- Directory
- Page Subtable
- Main Memory
- Page Frame

### Two-Level Page Table (2)

- **Goal:** Reduce amount of RAM for page tables
  - 4 KB pages, 32-bit addresses, 4-byte PT entries
  - 4 MB single-level page table
  - 1,000 pages

- **Example (100 KB of text and initial data):**
  - Need only about 3 pages to hold two-level page table
  - One page holds 4096/4 = 1024 entries
  - Each directory entry potentially addresses 1024x4 KB = 4 MB
  - 32-bit address 4 GB which is addressable by 4 GB/4 MB = 1,024 PT entries = 4 KB
  - Directory fits into 1 page

### Inverted Page Table

- **Example:** Power PC
  - One row for each page frame in physical memory

```
<table>
<thead>
<tr>
<th>Virtual Address</th>
<th>Physical Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>P   D</td>
<td>F    D</td>
</tr>
</tbody>
</table>
```

- <PID, P> Search
- Inverted Page Table
- Page Frame
- matches page # P and PID

### Inverted Page Table With Hashing

```
<table>
<thead>
<tr>
<th>Virtual Address</th>
<th>Physical Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>P   D</td>
<td>F    D</td>
</tr>
</tbody>
</table>
```

- <PID, P> Hash
- Hash Table
- Inverted Page Table
- Search
- Page Frame
- Synonym Chain
- matches page # P and PID
Translation Lookaside Buffer (TLB)

- Reduces number of memory accesses from 2 to 1 for each address reference

Typical TLB Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Block Size</td>
<td>4 to 8 bytes (1 page entry)</td>
</tr>
<tr>
<td>Hit Time</td>
<td>2.5 to 5 nsec (1 clock cycle)</td>
</tr>
<tr>
<td>Miss Penalty</td>
<td>50 to 150 nsec</td>
</tr>
<tr>
<td>TLB Size</td>
<td>32 bytes to 8 KB</td>
</tr>
<tr>
<td>Desired Hit Rate</td>
<td>98% to 99.9%</td>
</tr>
</tbody>
</table>

Page Fault Handling (1)

1. Use TLB
   - If not in TLB, use page table (part of PCB) to update TLB (page must be in memory (V=1))
2. If invalid, trap to OS and terminate process; if valid but not in memory, page in from swap area
3. Find location on disk
4. Find a free page frame for incoming page
5. Read page into free frame
   - If memory is full, perform page replacement
6. Update page table
7. Restart interrupted instruction
Protection in Paging Systems

Virtual Address  Physical Address

Page#  offset

Page Table Address

Page Table

Program  Paging Hardware  Memory

Access Privileges (rwx)

Page Frame

Physical Address

frame#  offset

Virtual Address

Access Privileges (rwx)

read-only  Text, Constants

readable  Text

writeable  Heap, Stack

Code Sharing

VM1

printf(...);

Logical Addresses

transfer vector

VM2

printf(...);

Physical Memory and Addresses

May be preloaded

TV Determined at load time

Memory Allocation in the Kernel

- Not as easy as user-space memory allocation
- Factors
  - Kernel can not sleep
  - Kernel can not easily deal with memory errors
  - Need lightweight allocation scheme
- Pages
  - Physical page: basic unit of memory management
    - Memory Management Unit (MMU)
      - Hardware that manages memory
      - Maps virtual addresses to physical addresses
    - MMU deals in pages
  - One page descriptor for each physical page
    - 4 GB addr space and 4 KB pages → 1 M descriptors

Memory Management in Linux

- Linux process (32-bit machine)
  - 3 GB virtual address space
  - +1 GB reserved for page tables and other data
    - only visible when in kernel mode
  - Memory is allocated by kernel as needed
- Hardware limitations ➔ memory zones
  - ZONE_DMA pages for DMA operations
    - some devices can DMA only to certain addresses
    - on x86 architectures, < 16 MB
  - ZONE_NORMAL normally mapped pages
  - ZONE_HIGHMEM high-memory addresses, not permanently mapped
    - on x86 architectures, all physical addresses above 896MB
- Actual layout of zones: architecture dependent
Linux Memory Allocation Mechanisms

- **Page allocator (Buddy algorithm)**
  - Main kernel memory allocation mechanism
  - Allocates physically contiguous pages of memory
  - No external fragmentation
  - But can have internal fragmentation
    - unused space in allocated chunks

- **Slab allocator**
  - Built on top of page allocator
  - Addresses internal fragmentation

- **vm allocator**
  - Used when memory need not be physically contiguous
  - Allocates virtually contiguous pages

Buddy System (Tree Representation)

- Simple, fast
- Good for allocating contiguous memory blocks
- Subject to internal fragmentation

Free Lists:
- 16MB
- 8MB
- 4MB
- 2MB

Buddy System Example

<table>
<thead>
<tr>
<th>16 MB</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
</tr>
<tr>
<td>A</td>
</tr>
<tr>
<td>A</td>
</tr>
<tr>
<td>A</td>
</tr>
<tr>
<td>A</td>
</tr>
<tr>
<td>E</td>
</tr>
<tr>
<td>E</td>
</tr>
<tr>
<td>E</td>
</tr>
<tr>
<td>E</td>
</tr>
<tr>
<td>E</td>
</tr>
<tr>
<td>E</td>
</tr>
</tbody>
</table>

Page Replacement Algorithms (1)

- **Example** Page Reference Stream (String)
  - 2, 3, 2, 1, 5, 2, 4, 5, 3, 2, 5, 2

- Assume
  - **Global Policy**
    - Entire set of page frames is shared by all processes
    - Number of frames allocated to a process can vary over time
  - When free frame needed
    - All resident pages are candidates for eviction
  - Alternative is a **Local Replacement** policy
    - Number of pages allocated to a process is fixed (a constant)

- **Goal of global replacement algorithm**
  - Select a **good** page to be replaced when a new page must be swapped into memory
Page Replacement Algorithms (2)

Basic Algorithms

- **Optimal (Impractical)**
  - Idea: Replace the P for which the time to the next reference is the longest.
  - Impossible to implement, but serves as a reference point

- **First-In, First-Out (FIFO)**
  - Idea: Replace the oldest P
  - One of the simplest algorithms, but performs poorly
  - Treat page frames of a process as a circular buffer
  - Pages are removed in RR order
  - Implement as a pointer that cycles through the page frames of a process

- **Least Recently Used (LRU)**
  - Idea: Replace the page that hasn't been referenced for the longest time in the past
  - Difficult to implement in hardware
  - Time stamp each page and replace the oldest one
  - Use a stack with the most recently referenced page on top

Replacement Example (1)

<table>
<thead>
<tr>
<th>OPT</th>
<th>FIFO</th>
<th>LRU</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>4</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>3</td>
</tr>
</tbody>
</table>

LRU Page Replacement

- **Least Recently Used (LRU)**
  - Idea: Replace the page that hasn't been referenced for the longest time in the past
  - Does almost as well as optimal algorithm on some reference sequences
  - Difficult to implement in hardware
  - Time stamp each page and replace the oldest one
  - Use a stack with the most recently referenced page on top
### Clock Page Replacement

**Clock (Second Chance Algorithm)**
- **Idea**: Faster approximation of LRU
- Select an unmarked frame in RR order
  - Cursor cycles through page frames looking for an unmarked page as the replacement page
- When to **unmark** the page underneath the cursor:
  - If the page is marked, then go to next page
  - If all pages are marked, the cursor will return to the first one it unmarked
- When to **mark** a page:
  - When loaded into main memory
  - When already in main memory and it has been referenced

**Enhanced Clock (Third Chance Algorithm)**
- Scan 1: Search for \((R,M) = (0,0)\) for replacement
  - \((1,1)\) requires disk write
- Scan 2: Search for \((R,M) = (0,0)\) for replacement

### The Working Set Model
- **W(t, \Delta)** is the *working set* at virtual time \(t\) with a window size of \(\Delta\) and is:
  - Defined over the page reference string for each process
  - The set of pages that have been referenced in the time interval \([t - \Delta, t]\).
- **Example**: 
  
  \[
  W(4,4) = \{0, 3, 8, 9\} \\
  W(2,2) = \{0, 9\} \\
  W(15,5) = \{0, 2, 9\}
  \]

- **W(t, \Delta)** varies over time \(t\) even with a fixed window size \(\Delta\)

### The Working Set Strategy
- **The Strategy**
  - Monitor \(W(t, \Delta)\) for each process
  - Periodically remove pages from the resident set of a process that are not in its \(W(t, \Delta)\)
  - Schedule a process only if its working set is in main memory

- **Problems**
  - The past doesn't always predict the future
  - An exact measurement of \(W(t, \Delta)\) is impractical because it requires a time-ordered queue of pages.
  - The optimal value of \(\Delta\) is unknown
Example

- Reference String: \((4,3,0,2,2,3,1,2,4,2,4,0,3)\)
- Working Set \((\Delta=4)\)

<table>
<thead>
<tr>
<th>Working Set</th>
<th>Page 0</th>
<th>Page 1</th>
<th>Page 2</th>
<th>Page 3</th>
<th>Page 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Page 0</td>
<td>(x)</td>
<td>(x)</td>
<td>(x)</td>
<td>(x)</td>
<td>(x)</td>
</tr>
<tr>
<td>Page 1</td>
<td>(x)</td>
<td>(x)</td>
<td>(x)</td>
<td>(x)</td>
<td>(x)</td>
</tr>
<tr>
<td>Page 2</td>
<td>(x)</td>
<td>(x)</td>
<td>(x)</td>
<td>(x)</td>
<td>(x)</td>
</tr>
<tr>
<td>Page 3</td>
<td>(x)</td>
<td>(x)</td>
<td>(x)</td>
<td>(x)</td>
<td>(x)</td>
</tr>
<tr>
<td>Page 4</td>
<td>(x)</td>
<td>(x)</td>
<td>(x)</td>
<td>(x)</td>
<td>(x)</td>
</tr>
</tbody>
</table>

PFF \((F = 3)\): expand RSS if inter-fault time < 3

<table>
<thead>
<tr>
<th>Working Set</th>
<th>Page 0</th>
<th>Page 1</th>
<th>Page 2</th>
<th>Page 3</th>
<th>Page 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Page 0</td>
<td>(\times)</td>
<td>(\times)</td>
<td>(\times)</td>
<td>(\times)</td>
<td>(\times)</td>
</tr>
<tr>
<td>Page 1</td>
<td>(\times)</td>
<td>(\times)</td>
<td>(\times)</td>
<td>(\times)</td>
<td>(\times)</td>
</tr>
<tr>
<td>Page 2</td>
<td>(\times)</td>
<td>(\times)</td>
<td>(\times)</td>
<td>(\times)</td>
<td>(\times)</td>
</tr>
<tr>
<td>Page 3</td>
<td>(\times)</td>
<td>(\times)</td>
<td>(\times)</td>
<td>(\times)</td>
<td>(\times)</td>
</tr>
<tr>
<td>Page 4</td>
<td>(\times)</td>
<td>(\times)</td>
<td>(\times)</td>
<td>(\times)</td>
<td>(\times)</td>
</tr>
</tbody>
</table>

Design Issues (Paging)

- Local versus Global policy
  - Global policies tend to work better ➔
    - Allow WS size to vary over a process’ lifetime to avoid thrashing
- Load Control
  - Sum of all WS sizes > Size of memory ➔ Thrashing likely
  - Approach: Swap out some processes
- Page Size
  - Selectable by OS (to some extent) (Typical: 4 or 8 KB)
  - Issues: internal fragmentation, PT size, paging time
- Sharing Pages
- Cleaning Policy
  - Keep a supply of free frames ➔ Better performance
  - Paging daemon: Two-handed clock algorithm evicts pages

Practical Systems

- Machines typically don’t support LRU or WS
  - Typically, have an R-bit (referenced)
- Portable kernel code
  - ➔ May not use all features of VM hardware
- Aging – a software solution
  - Based on NFU (Not Frequently Used) algorithm
  - OS scans all pages at each clock interrupt (10 msec)
    - Right: shift each R-bit into its 8-bit age counter
    - Then, zero (reset) each R-bit
  - Replacement page is the one with the smallest age
  - NFU
    - Adds R-bit to age counter
    - Problem is that past behavior can incorrectly effect replacement

Aging Example

<table>
<thead>
<tr>
<th>R-bits</th>
<th>Tick 0</th>
<th>Tick 1</th>
<th>Tick 2</th>
<th>Tick 3</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>10000000</td>
<td>11000000</td>
<td>11100000</td>
<td>11110000</td>
</tr>
<tr>
<td></td>
<td>00000000</td>
<td>01000000</td>
<td>01100000</td>
<td>01110000</td>
</tr>
<tr>
<td></td>
<td>10000000</td>
<td>10100000</td>
<td>00100000</td>
<td>00010000</td>
</tr>
<tr>
<td></td>
<td>00000000</td>
<td>00100000</td>
<td>10000000</td>
<td>01000000</td>
</tr>
<tr>
<td></td>
<td>10000000</td>
<td>01000000</td>
<td>10100000</td>
<td>01010000</td>
</tr>
</tbody>
</table>

Source: Tanenbaum, Modern Operating Systems
**WS Clock Page Replacement (1)**

- **Current VT = 45**
- **VT(A) = 45**
  - Active, Running
- **VT(B) = 19**
  - Active
- **WS Threshold**
  - T=4

![Page Frame Table](chart1)

**WS Clock Page Replacement (2)**

- **Current VT = 45**
- **VT(A) = 45**
  - Active, Running
- **VT(B) = 19**
  - Active
- **WS Threshold**
  - T=4
- **Clock interrupt occurs**
  - VT'(A)=VT(A)+1
  - Update TS if R=1
  - Clear R

![Page Frame Table](chart2)

**WS Clock Page Replacement (3)**

- **Current VT = 46**
- **VT(A) = 46**
- **VT(B) = 19**
- **WS Threshold**
  - T=4
- **Process A**
  - refs page 5
  - page fault
  - Scan frames for replacement
    - age ≥ T=4? and
    - clean: (0,0)?
- **None ➔ Deactivate B**
  - overcommitted memory

![Page Frame Table](chart3)

**WS Clock Page Replacement (4)**

- **Bits (flags) in page frame table (FT)**
  - R = 1: page has been referenced (read/write)
  - M = 1: page has been modified (write)
  - V = 1: page entry is valid
- **Every page has a TS (time of last use) field**
- **Every process has a VT (virtual time)**
  - Advances only when process is active
- **When clock interrupt**
  - Set TS(i) = VT(i) if R-bit = 1
  - Evict page (reduce WS) if page is clean (R=0,M=0) and age < threshold
    - where age = VT(i) – TS(i)
  - Clear all R-bits
WS Clock Page Replacement (5)

- Each page falls into one of four (R,M) classes
  - (0,0): Neither recently used nor dirty (CLEAN)
  - (0,1): Not used but dirty (want to convert to (0,0))
  - (1,0): Recently used and clean
    - maybe don’t replace
  - (1,1): Recently used and dirty
    - might be used again soon

- Page replacement (when page fault)
  - Ideal replacement page is (R=0,M=0)
  - Page is (R=0,M=1) ➔ Schedule write to disk
    - Convert to (R=0, M=0)
    - Limit number of writes
    - Avoids context switch
    - Hope to find clean page later in scan

Summary

- Optimal: Not implementable, but useful benchmark
- FIFO: Might throw out important pages
- LRU: Excellent but difficult to implement in HW
- WS: Expensive to implement
- Aging: Efficient approximate LRU algorithm
- WSClock: Good, efficient algorithm

Interesting References

Implementation Issues

- Paging Work Done by OS
  - process creation, execution and termination, page fault
- Handling a Page Fault
- Instruction Backup
  - Need to restart instruction causing a page fault
- Locking Pages in Memory
  - What if process reading file directly into buffer gets swapped out ??
- Backing Store Design
  - Where to put pages on disk ??
- Separating Policy From Mechanism