Virtual Memory - Part 1 (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 Required**: A process may be broken up into pieces that need not be contiguously located in main memory.
  - **Protection**: Memory references only to own physical memory

Demand Paging

- Bring a page into main 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.
Matrix Multiply (1)

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?

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];

Instruction Page Reference String: I, I, ..., I
A[][] Page Reference String: A0, A0, ..., A0, A1, A1, ...
B[][] Page Reference String: B0, B1, B2, ..., B0, B1, B2, ...
C[][] Page Reference String: C0, C0, ..., C0, C1, C1

Matrix Multiply (2)

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

R1 ← 0; // accumulator
for (int k=0; k<N; k++) {
  // test and jump
  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
R1 → C[i][j]; // Store value of C[i][j]

Page Fault Rate

What if inner loop is:

R1 ← 0; // accumulator
for (int k=0; k<N; k++) {
  // test and jump
  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
R1 → C[i][j]; // Store value of C[i][j]

Page Fault Rate

What if inner loop is:

R1 ← 0; // accumulator
for (int k=0; k<N; k++) {
  // test and jump
  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
R1 → C[i][j]; // Store value of C[i][j]

9N+1 memory references per row
N^2 rows

What if physical memory allocation of 4 frames

Initial Placement:
1. page 0, A[0][*] → 1, B[0][*] → 2, C[0][*] → 3

Page reference string (F = Page Fault):
0, 0, 1, 0, 2, 0, 0, 0, 0, 0, 1, 0, F, 0, 0, 0, 0, 0, 0, 1, 0, F, 0, 0, 0, 0, 0; ...

Composite memory reference string (n = N-1):
II, A0, I, B0, I, I, I, I, I, A0, I, B1, I, I, I, I, I, I, I, I, I, A0, I, Bn, I, I, I, I, I, C0 // C[0][1]
II, A0, I, B0, I, I, I, I, I, A0, I, B1, I, I, I, I, I, I, I, I, I, A0, I, Bn, I, I, I, I, I, C0 // C[0][n]

9N+1 memory references per row
N^2 rows

Page Fault Rate

The number of page faults per memory reference
About 1/9 for large N
More precisely
(9N+1) x N^2 page references
N^3 + N page faults Æ Page fault rate of (N^3 + N)/[(9N+1) x N^2]]
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 = \(<\text{Frame } #, \text{Offset}>\)

- **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 main memory?**
  - Almost anywhere in main memory \(\Rightarrow\) Need associative hardware for address translation

- **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

Basic Page Table Structure

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

- Physical Address
  - frame\# offset

Program Paging Hardware Memory

Huge Virtual Address Spaces

- **If the page size is 1 KB in a virtual memory that can be as large as $$2^{32}$$ bytes (4 gigabytes), how many entries will the 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!
  - \(\Rightarrow\) Page tables can be huge and are subject to paging
Two-Level Page Table (1)

Example: Pentium

Indexes into page table broken into 1-page subtables

Logical Address

Physical Address

Main Memory

Page Frame F

Directory

Page Subtable

Two-Level Page Table (2)

Objective: Reduce amount of RAM for page tables

- 4 KB pages, 32-bit addresses, 4-byte PT entries
- 4 MB single-level page table $\Rightarrow$ 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 1024 x 4 KB = 4 MB
- 32-bit address $\Rightarrow$ 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 (real) memory

Virtual Address

Physical Address

Virtual Memory (VM)

PID

Paging Hardware

Memory

matches page# P and PID

Inverted Page Table With Hashing

Virtual Address

Physical Address

Virtual Memory (VM)

PID

Paging Hardware

Memory

matches page# P and PID
**Translation Lookaside Buffer (TLB)**

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

**Typical TLB Parameters**

- **Block Size**: 4 to 8 bytes (1 page entry)
- **Hit Time**: 2.5 to 5 nsec (1 clock cycle)
- **Miss Penalty**: 50 to 150 nsec
- **TLB Size**: 32 bytes to 8 KB
- **Desired Hit Rate**: 98% to 99.9%

**Page Fault Handling (1)**

1. **Trap**
   - If not in TLB, use page table (part of PCB) to update TLB (page must be in memory (valid bit ON))
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**