CS370 Operating Systems
Colorado State University
Yashwant K Malaiya
Fall 2017 Lecture 21

Main Memory

Slides based on
• Text by Silberschatz, Galvin, Gagne
• Various sources
FAQ

• Why not increase page size as address space increases? Allowed in some systems to a limited extent but not dynamically.

• Page Numbers: Few MSBs of addresses

• Hashed page table entries: pointers to a linked list

• Inverted page tables:
  – An entry/frame, frame number is address
  – Search using PID & page number

• Why multi-level paging?

• TLB 1-5% TIME & multi-level paging: page tables consulted on TLB miss. Multi level TLB possible

• Why 64 bit CPU/OS use less than 64 bit addresses? 40-52 lines

• How to increase hit ratio?
In Systems (HW/OS/Net) algorithms are needed for:

- Process scheduling
- Contiguous allocation policies
- Cache/TLB policies
- Page replacement policies
- Disk scheduling
- Network protocols
- etc.
How to formulate an algorithm

• Examine existing algorithms
• Formulate a new one that may address some of the limitations
  – Ensure that algorithm is valid
  – Compare performance of the new algorithm with existing ones using suitable figures of merit:
    • time,
    • memory requirements,
    • power consumptions etc.
  – If promising, tweak possible options by evaluating them to optimize implementation

Will the best algorithm be best for all individual cases?
How to evaluate an algorithm

**Deterministic evaluation**: use one or a few specific examples
- Pro: Good for understanding the algorithm
- Con: Examples will not be representative of all possible cases

**Probabilistic evaluation**: take randomness into account
- Determine probabilistic distribution of inputs
- Analyze the system performance
  - Mathematical analysis
  - Simulation
Probabilistic evaluation: take randomness into account

- Determine probabilistic distribution of inputs
- Analyze the system performance
  - Mathematical analysis
    - Examples: Queueing theory approaches
  - Simulation:
    - Generate inputs that satisfy probability distributions
      - Ex: process arrival time, CPU/IO burst distribution
    - Simulate system action, measure figure of merit
    - Enough inputs so that results are statistically significant
  - Construct and evaluate an actual system
Examples:

• Performance Evaluation of Cache Replacement Policies for the SPEC CPU2000 Benchmark Suite
• Fairness and interactive performance of O(1) and CFS Linux kernel schedulers
• On the Impact of Scheduler Settings on the Performance of Multi-Threaded SIP Servers
• Page Placement Strategies for GPUs within Heterogeneous Memory Systems
• Linux Scheduler simulation
Segmentation Approach

Memory-management scheme that supports user view of memory

• A program is a collection of segments
  – A segment is a logical unit such as:
    main program
    procedure, function, method
    object
    local variables, global variables
    common block
    stack, arrays, symbol table

• Segment table
  – Segment-table base register (STBR)
  – Segment-table length register (STLR)

• segments vary in length, can very dynamically
• Segments may be paged
• Used for x86-32 bit
• Origin of term “segmentation fault”
Examples

- Intel IA-32 (x386-Pentium)
- x86-64 (AMD, Intel)
- ARM
Logical to Physical Address Translation in IA-32

Diagram:
- Logical address
  - selector
  - offset
  - descriptor table
  - segment descriptor
  - 32-bit linear address

Flow:
- CPU
  - logical address
  - segmentation unit
  - linear address
  - paging unit
  - physical address
  - physical memory

Table:
<table>
<thead>
<tr>
<th>Page number</th>
<th>Page offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>$p_1$</td>
<td>$p_2$</td>
</tr>
<tr>
<td>10</td>
<td>10</td>
</tr>
</tbody>
</table>
Intel IA-32 Paging Architecture

(logical address)

31 22 21 12 11 0

page directory page table offset

page directory

CR3 register

4-KB page

4-MB page

Colorado State University
32-bit address limits led Intel to create page address extension (PAE), allowing 32-bit apps access to more than 4GB of memory space.

- Paging went to a 3-level scheme
- Top two bits refer to a page directory pointer table
- Page-directory and page-table entries moved to 64-bits in size
- Net effect is increasing address space to 36 bits – 64GB of physical memory
Intel x86-64

- Intel x86 architecture based on AMD 64 bit architecture
- 64 bits is ginormous (> 16 exabytes)
- In practice only implement 48 bit addressing or perhaps 52
  - Page sizes of 4 KB, 2 MB, 1 GB
  - Four levels of paging hierarchy
- Can also use PAE so virtual addresses are 48 bits and physical addresses are 52 bits

<table>
<thead>
<tr>
<th>unused</th>
<th>page map level 4</th>
<th>page directory pointer table</th>
<th>page directory</th>
<th>page table</th>
<th>offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>63</td>
<td>48 47</td>
<td>39 38</td>
<td>30 29</td>
<td>21 20</td>
<td>12 11</td>
</tr>
</tbody>
</table>
Example: ARM Architecture

- Dominant mobile platform chip (Apple iOS and Google Android devices for example)
- Modern, energy efficient, 32-bit CPU
- 4 KB and 16 KB pages
- 1 MB and 16 MB pages (termed sections)
- One-level paging for sections, two-level for smaller pages
- Two levels of TLBs
  - Outer level has two micro TLBs (one data, one instruction)
  - Inner is single main TLB
  - First inner is checked, on miss outers are checked, and on miss page table walk performed by CPU
CS370 Operating Systems

Colorado State University
Yashwant K Malaiya
Fall 2017  Lecture 21

VirtualMemory

Slides based on
- Text by Silberschatz, Galvin, Gagne
- Various sources
Virtual Memory: Objectives

- A virtual memory system
- Demand paging, page-replacement algorithms, allocation of page frames to processes
- Threshing, the working-set model
- Memory-mapped files and shared memory and
- Kernel memory allocation

First used in Atlas, Manchester, 1962

PCs: Windows 95
Background

- Code needs to be in memory to execute, but entire program rarely used
  - Error code, unusual routines, large data structures
- Entire program code not needed at the same time
- Consider ability to execute partially-loaded program
  - Program no longer constrained by limits of physical memory
  - Each program uses less memory while running -> more programs run at the same time
    - Increased CPU utilization and throughput with no increase in response time or turnaround time
  - Less I/O needed to load or swap programs into memory
    -> each user program runs faster
• **Virtual memory** – separation of user logical memory from physical memory

• **Virtual address space** – logical view of how process views memory
  – Usually start at address 0, contiguous addresses until end of space
  – Meanwhile, physical memory organized in page frames
  – MMU must map logical to physical

• **Virtual memory can be implemented via:**
  – Demand paging
  – Demand segmentation
Virtual Memory That is Larger Than Physical Memory
Virtual-address Space: advantages

- Usually design logical address space for stack to start at Max logical address and grow “down” while heap grows “up”
  - Maximizes address space use
  - Unused address space between the two is hole
    - No physical memory needed until heap or stack grows to a given new page
- Enables \textit{sparse} address spaces with holes left for growth, dynamically linked libraries, etc.
- System libraries shared via mapping into virtual address space
- Shared memory by mapping pages read-write into virtual address space
- Pages can be shared during \texttt{fork()}, speeding process creation
Shared Library Using Virtual Memory

- stack
- shared library
- heap
- data
- code

shared pages

- stack
- shared library
- heap
- data
- code
Demand Paging

• Could bring entire process into memory at load time
• Or bring a page into memory only when it is needed: **Demand paging**
  – Less I/O needed, no unnecessary I/O
  – Less memory needed
  – Faster response
  – More users
• Similar to paging system with swapping (diagram on right)
• Page is needed ⇒ reference to it
  – invalid reference ⇒ abort
  – not-in-memory ⇒ bring to memory
• **“Lazy swapper”** – never swaps a page into memory unless page will be needed
  – Swapper that deals with pages is a **pager**
Demand paging: Basic Concepts

- Demand paging: pager brings in only those pages into memory what are needed
- How to determine that set of pages?
  - Need new MMU functionality to implement demand paging
- If pages needed are already memory resident
  - No difference from non-demand-paging
- If page needed and not memory resident
  - Need to detect and load the page into memory from storage
    - Without changing program behavior
    - Without programmer needing to change code
Valid-Invalid Bit

- With each page table entry a valid–invalid bit is associated (v ⇒ in-memory – **memory resident**, i ⇒ not-in-memory)
- Initially valid–invalid bit is set to i on all entries
- Example of a page table snapshot:

![Page Table Diagram]

- During MMU address translation, if valid–invalid bit in page table entry is i ⇒ **page fault**
Page Table When Some Pages Are Not in Main Memory

Page 0 in Frame 4 (and disk)
Page 1 in Disk
Page Fault

• If there is a reference to a page, first reference to that page will trap to operating system: Page fault

Page fault

1. Operating system looks at a table to decide:
   – Invalid reference ⇒ abort
   – Just not in memory, but in backing storage, ->2

2. Find free frame

3. Get page into frame via scheduled disk operation

4. Reset tables to indicate page now in memory
   Set validation bit = \( v \)

5. Restart the instruction that caused the page fault

Page fault: context switch because disk access is needed
Questions for you

- What is disk space is full, physical memory is full, and the user launches a process?
- If physical memory (RAM) gets to be very big, do accesses to disk reduce?
- Is there ever a case where adding more memory does not help?
Technical Perspective: Multiprogramming

Solving a problem gives rise to a new class of problem:

• Contiguous allocation. Problem: external fragmentation
• Non-contiguous, but entire process in memory: Problem: Memory occupied by stuff needed only occasionally. Low degree of Multiprogramming.
• Demand Paging: Problem: page faults
• How to minimize page faults?
Steps in Handling a Page Fault

1. Trap
2. Page is on backing store
3. Bring in missing page
4. Reset page table
5. Free frame
6. Restart instruction

Load M

Operating system

Reference

Page table

Physical memory

Colorado State University
Performance of Demand Paging

Stages in Demand Paging (worse case)

1. Trap to the operating system
2. Save the user registers and process state
3. Determine that the interrupt was a page fault
4. Check that the page reference was legal and determine the location of the page on the disk
5. Issue a read from the disk to a free frame:
   1. Wait in a queue for this device until the read request is serviced
   2. Wait for the device seek and/or latency time
   3. Begin the transfer of the page to a free frame
6. While waiting, allocate the CPU to some other user
7. Receive an interrupt from the disk I/O subsystem (I/O completed)
8. Save the registers and process state for the other user
9. Determine that the interrupt was from the disk
10. Correct the page table and other tables to show page is now in memory
11. Wait for the CPU to be allocated to this process again
12. Restore the user registers, process state, and new page table, and then resume the interrupted instruction
Performance of Demand Paging (Cont.)

• Three major activities
  – Service the interrupt – careful coding means just several hundred instructions needed
  – Read the page – lots of time
  – Restart the process – again just a small amount of time

• Page Fault Rate $0 \leq p \leq 1$
  – if $p = 0$ no page faults
  – if $p = 1$, every reference is a fault

• Effective Access Time (EAT)
  \[
  EAT = (1 - p) \times \text{memory access time} + p \times (\text{page fault overhead} + \text{swap page out} + \text{swap page in})
  \]

  Hopefully $p << 1$

Page swap time = seek time + latency time
Demand Paging Example

- Memory access time = 200 nanoseconds
- Average page-fault service time = 8 milliseconds
- EAT = \((1 - p) \times 200 + p\) (8 milliseconds)
  \[= (1 - p) \times 200 + p \times 8,000,000 \text{ nanosec.}\]
  \[= 200 + p \times 7,999,800 \text{ ns}\]

- If one access out of 1,000 causes a page fault, then EAT = 8.2 microseconds.
  This is a slowdown by a factor of 40!!

- If want performance degradation < 10 percent, \(p = \) ?
  \[220 > 200 + 7,999,800 \times p\]
  \[20 > 7,999,800 \times p\]
  \[p < .0000025\]
  \(< \text{one page fault in every 400,000 memory accesses}\)
Issues: Allocation of physical memory to I/O and programs

• Memory used for holding program pages
• I/O buffers also consume a big chunk of memory
• Solutions:
  – Fixed percentage set aside for I/O buffers
  – Processes and the I/O subsystem compete
Demand paging and the limits of logical memory

• Without demand paging
  – All pages of process **must be** in physical memory
  – Logical memory **limited** to size of physical memory

• With demand paging
  – All pages of process **need not be** in physical memory
  – Size of logical address space is **no longer constrained** by physical memory

• Example
  – 40 pages of physical memory
  – 6 processes each of which is 10 pages in size
    • Each process only needs 5 pages as of now
  – Run 6 processes with 10 pages to spare

Higher degree of multiprogramming
Coping with over-allocation of memory

Example

• Physical memory = 40 pages
• 6 processes each of which is of size 10 pages
  – But are using 5 pages each as of now
• What happens if each process needs all 10 pages?
  – 60 physical frames needed

• **Terminate** a user process
  – But paging should be transparent to the user

• **Swap out** a process
  – Reduces the degree of multiprogramming

• **Page replacement**: selected pages. Policy?
What Happens if there is no Free Frame?

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

Continued to Page replacement etc...
Page Replacement

• Prevent **over-allocation** of memory by modifying page-fault service routine to include page replacement

• Use **modify (dirty) bit** to reduce 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
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
     - Write victim frame to disk if dirty

3. Bring the desired page into the (newly) free frame; update the page and frame tables

4. Continue the process by restarting the instruction that caused the trap

Note now potentially 2 page transfers for page fault – increasing EAT
Page Replacement

1. Swap out victim page
2. Change to invalid
3. Swap desired page in
4. Reset page table for new page
FAQ

• How does the virtual memory respond when the stack or heap grows? Or is the hole between them always there? Hole is always there in virtual memory. Frames in memory allocated as needed.

• How is the TLB affected when a page is moved from Memory to Disk, and is replaced by a page brought in from Disk? TLB is cache, has mechanism for removing and adding info to it.

• When does a TLB need to be flushed completely? Context switch

• Can more than one page loaded into memory when a process starts? Prefetching

• Why are disk addresses of non-resident pages not stored in the page table? Generally contains only information used on page hits.
• **Page-replacement algorithm**
  – Which frames to replace
  – Want 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
  – String is just page numbers, not full addresses
  – Repeated access to the same page does not cause a page fault
  – Results depend on number of frames available

• In all our examples, we use 3 frames and the reference string of referenced page numbers is
  \[ 7,0,1,2,0,3,0,4,2,3,0,3,0,3,2,1,2,0,1,7,0,1 \]
Graph of Page Faults Versus The Number of Frames
FIFO page replacement algorithm:
Out with the old; in with the new

- When a page must be replaced
  - Replace the oldest one

- OS maintains list of all pages currently in memory
  - Page at head of the list: Oldest one
  - Page at the tail: Recent arrival

- During a page fault
  - Page at the head is removed
  - New page added to the tail
First-In-First-Out (FIFO) Algorithm

- Reference string: 7,0,1,2,0,3,0,4,2,3,0,3,0,3,2,1,2,0,1,7,0,1
- 3 frames (3 pages can be in memory at a time per process)
- 15 page faults
- Sometimes a page is needed soon after replacement 7,0,1,2,0,3,0,..
Belady’s Anomaly

- Consider Page reference string 1,2,3,4,1,2,5,1,2,3,4,5
  - 3 frames, 9 faults, 4 frames 10 faults!
  - Adding more frames can cause more page faults!

- Belady’s Anomaly

3 frames: 9 page faults
4 frames: 10 page faults
FAQ

- Does Belady’s Anomaly affect all page replacement algorithms?
- When Belady’s Anomaly is present in an algorithm, will adding an additional frame always cause more page faults?
- Paging algorithms seem to assume both temporal and spatial locality... how true is this?