Operating Systems CS-450

Memory Management

Fall 2015
Professor: Ali A. Kooshesh
Department of Computer Science
Sonoma State University

Memory Hierarchy

<table>
<thead>
<tr>
<th>Speed(ns)</th>
<th>1ns</th>
<th>1ns</th>
<th>10-100s</th>
<th>100s</th>
<th>10^7s</th>
<th>10sec</th>
</tr>
</thead>
<tbody>
<tr>
<td>Size:</td>
<td>100B</td>
<td>100B</td>
<td>KB-MB</td>
<td>MBs</td>
<td>GBs</td>
<td>TBs</td>
</tr>
</tbody>
</table>

- CPU Registers
- On Chip Cache
- Level 2 Cache
- Main Memory
- Hard Disk
- Back up tape
Address Space

- Process execution requires access to system resources.
- Resource are limited and as a result, need to be shared.
- Multiplexing (sharing)
  - CPU multiplexing (scheduling: previous chapter)
  - Multiplexing the use of Memory (this chapter)
  - Multiplexing disks and other devices (future chapters)
- A thread is an abstraction: it provides a framework for concurrency.
- Address-space is an abstraction to support memory multiplexing. It gives rise to protection.
- Address-space provides methods that protects programs from programs and operating system from program (and vice-versa!)

Logical Addresses vs Physical Addresses

- We talk about address-space because it is an abstraction that provides protection. We are in the process of learning what types of abstractions there are and how do they work.
- The Compiler generates code using virtual addresses. This is because in modern systems, the compiler has no way of knowing where in the main-memory the code gets loaded.
- As a program gets prepared for execution, the Operating System, depending on its memory management strategy, finds space and stores the program in the main memory.
- Now, we have to map each virtual address to its corresponding physical address.
From Compilation to Running

- Addresses can be bound to physical location in either of the following three stages.
  - Compile-time
  - Linker/Load time
  - Execution-time
- Dynamic Libraries
  - Linking postponed until execution time
  - Stubs (small pieces of code) are used to locate the appropriate memory
  - Stub gets replaced with the address of the routine during execution

Virtual and Physical addresses

```
data1:dw 32
      0x300  00000020
... ...
start: lw r1,0(data1)  0x900  8C2000C0
      jal checkit  0x904  0C000300
loop:  addi r1, r1, -1  0x908  201FFFF
      bnez r1, r0, loop  0x90C  1420FFFF
checkit: ...
```

- Address 0x300 is relative to some base-address chosen by the compiler
- When it gets loaded into the main-memory, the instruction at 0x300 will end up at a totally different physical address. As a consequence, the instruction at 0x904 will point to the wrong physical address.
Single Application No Memory Abstraction

- Uni-Programming requires no translation or protection
  - Only one application runs at a given time and as such, the entire machine is dedicated to its execution.
  - The application gets loaded at and runs from the same memory location. That is, logical addresses map directly to their corresponding physical addresses. The compiler then generates physical addresses.

Multi-Programming Without Memory Abstraction

- The previous scheme does not support multi-programming.
- Can we load multiple programs and avoid address overlap?
- Use Linker/Loader to adjust addresses while the object-code gets loaded into memory.
  - Translation is done by linker/loader
  - Was very common in early operating systems
  - There is no memory protection so bugs could crash the system!
Multi-Programming Using Relocation and Protection

- Address translation during link/load process is expensive
- In addition, object-code can not be moved around in MM
- Can we avoid address translation and provide protection?
- Yes: use two registers, base- and limit-register to translate on the fly and to validate memory address references.
  - If virtual-address > limit-register, error

![Diagram of address translation](image1)

Dynamic Address Translation

- On the fly relocation and protection: Segmentation and the infamous segmentation fault!

![Diagram of dynamic address translation](image2)
Issues Related to Segmentation

- Processes require contiguous memory allocated to them
- Over time, it gives rise to memory fragmentation

- Sharing is not easy
- Need enough physical memory for each process
- Can swap out processes to free up memory

The Idea of Translation

Virtual Addresses

- Program
- Stack
- Globals
- Heap

Translation Map 1

Physical Address Space

Translation Map 2

Virtual Addresses for Program 2

- Heap
- Stack
- Globals
- Program

Operating Systems CS-450
Ali A. Kooshesh
Address Translation Schemes

- **Paging**
  - Divide up main-memory into equal & fixed size segments (*frames*)
  - Divide up virtual addresses into *pages* (frame-size = page-size)
  - Provide a method to translate addresses in pages to their corresponding locations in the frames

- **Multi-level Paging**
  - Same as paging, except for large memory

- **Inverted Page-tables**
  - For large address-spaces; one entry for each frame instead of each virtual page.

- **Pure Segmentation**
  - We just saw an example in the previous slide

- **Segmentation with Paging**
  - Combination of segmentation and paging. Intel Pentium

---

**Paging**

![Diagram of Paging](image-url)
Paging (cont.)

- Each process has its own page-table
  - It resides in the main memory
- 4 KB is a common page-size (12 bits)
  - Number of bytes per page: \(2^{12} = 4096\)
- On a 32 bit machine, that leaves 20 bits to be used for page-table entries
  - Number of possible pages: \(2^{20} = 1,048,576\)
- Pros
  - Simple memory allocation
  - Easy to share pages. Multiple page-tables can point to the same page
- Cons
  - Can waste memory if the address-space is sparse.
  - The address-space becomes large because of the hole between the stack and the heap
  - Where do we store the page-table if it requires more than 4 KB?

Structure of a Page Table Entry

- Present/Absent bit indicates whether
  - The page-table entry is valid or not
  - If valid, is it in the memory or not. Referencing a page that is not in the memory triggers a page-fault
- Modified and Referenced bits are used to help with page-replacement policy algorithms
- Protection bit controls appropriate operations on the page: read, write, and execute
- Page Frame-number contains the physical address of the page in the main-memory
- Caching at times needs to be disabled for very low-level operations to work properly
Translation Look-aside Buffer (TLB)

- Translation by means of page-table adds at least an additional “table-lookup” to each memory access.
- Using page-table increases the running-time of the applications by at least a factor of 2.
- Some caching mechanism is needed to make the translation more efficient.
- One solution is to add a small hardware device, TLB, to the MMU.
TLB (Continue)

• Each entry of TLB contains information for one page-table entry.

• TLB is specifically helpful when locality of reference exists.
  – Instruction accesses remain on the same page or a small number of pages for a while
    • The access is sequential
    • Loops behave in this manner
    • Sequential array accesses
    • Stacks

• Hardware managed TLB
  – On TLB miss, hardware in MMU populates the TLB using the current page-table
    • On translation, if PTE is valid, it gets used
    • If PTE is marked invalid, page-fault occurs

TLB (Continue)

• Software managed TLB
  – On TLB miss, CPU receives TLB fault
    • If PTE valid, fills TLB and returns from fault
    • If PTE invalid, calls page-fault handler and then populates TLB
Page-Faults

- PTE can be used to implement demand paging
  - PTE not valid means page not in memory (page-fault)
  - Use info in PTE to locate and bring the page in the memory

- What happens on a page-fault
  - Choose a page to evict from the memory
  - If the page has been modified, write it to swap-space
  - Mark its PTE and TLB invalid
  - Load new page into memory from disk
  - Update page-table entry and TLB for this page
  - Execute the instruction that caused the page-fault

- The thread that causes page-fault gets suspended; is put in the wait-queue

- All of these actions is transparent to the user-thread
  - Does not know or care about page-faults

Page-Faults (Continued)
Page-Replacement Algorithms

- Replacement is an issue that applies to many cache systems
- The cost of page-fault is very high
- Optimal Page-Replacement algorithm
  - Swap the page that will not be references in the near future
  - Just as SJF scheduling algorithm, requires the knowledge of future
  - Used as the basis of comparison
- The Not Recently Used Page Replacement Algorithm
- The FIFO Algorithm
- The Second-Chance Algorithm
- The Clock Page-Replacement Algorithm
- The LRU Algorithm
- The Working-Set Algorithm

The Not Recently Used Algorithm

- Use the two bits, modified (M) and referenced (R) of the page-table entry (PTE).
- Initially, both bits are clear (set to zero)
- Periodically (20 msec), clear the R bit of all pages in an attempt to distinguish the more recently referenced pages
- If a page is modified, set its M bit to 1
- When a page-fault occurs, each page falls into one of the following categories
  - Category 1: Not referenced, not Modified
  - Category 2: Not referenced, modified*
  - Category 3: Referenced, not modified
  - Category 4: Referenced, modified
- Randomly choose a page from the lowest numbered category
FIFO

- Suppose we have 3 page frames, 4 virtual pages, and following reference stream:
  - A B C A B D A D B C B
- Consider FIFO Page replacement:

<table>
<thead>
<tr>
<th>Ref:</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>A</th>
<th>B</th>
<th>D</th>
<th>A</th>
<th>D</th>
<th>B</th>
<th>C</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>A</td>
<td></td>
<td>D</td>
<td></td>
<td></td>
<td>C</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>B</td>
<td></td>
<td></td>
<td>A</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>C</td>
<td></td>
<td></td>
<td></td>
<td>B</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- FIFO: 7 faults (initial references are counted)
- When referencing D, replacing A is bad choice, since we need A again right away

Optimal Page-Replacement Algorithm

- Suppose we have the same reference stream:
  - A B C A B D A D B C B
- Consider optimal page-replacement:

<table>
<thead>
<tr>
<th>Ref:</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>A</th>
<th>B</th>
<th>D</th>
<th>A</th>
<th>D</th>
<th>B</th>
<th>C</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>A</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>C</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>B</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>C</td>
<td></td>
<td>D</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Optimal: 5 faults
- Where will D be brought in? Look for page not referenced farthest in future.
- What will LRU do?
  - Same decisions as optimal here, but won’t always be true!
Least Recently Used (LRU)

- Replace page that hasn’t been used for the longest time
- The idea is based on the locality of reference. If a page has not been referenced lately, it may not be referenced anytime soon
- It is based on the past behavior as opposed to the optimal algorithms that is based on the future references
- How to implement LRU? Use a list!

- On each use, remove page from list and place at head
- LRU page is at tail
- It is very expensive to implement this way
- In practice, you approximate LRU algorithm

---

LRU (Continued)

- Another implementation for LRU (very expensive in practice)
- Suppose page-references are: 0 1 2 3 2 1 0 3 2 3

```
Access page 0
0 1 2 3
0 0 1 1
1 0 0 0
2 0 0 0
3 0 0 0

Access page 1
0 1 2 3
0 0 1 1
1 0 1 1
1 0 0 1
0 0 0 1

Access page 2
0 1 2 3
0 0 0 0
1 0 1 1
1 0 0 1
0 0 0 1

Access page 3
0 1 2 3
0 0 0 0
0 0 0 1
0 0 0 0
1 1 1 0
```

---
Worst-case Scenario for LRU

- Consider the following: A B C D A B C D A B C D
- LRU Performs as follows (same as FIFO here):

<table>
<thead>
<tr>
<th>Ref:</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td>A</td>
<td>D</td>
<td></td>
<td></td>
<td>C</td>
<td></td>
<td>B</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>B</td>
<td></td>
<td>A</td>
<td>D</td>
<td></td>
<td>C</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>C</td>
<td></td>
<td>B</td>
<td>A</td>
<td>D</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Every reference is a page fault!
- Optimal does much better:

<table>
<thead>
<tr>
<th>Ref:</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td>A</td>
<td></td>
<td></td>
<td></td>
<td>B</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>B</td>
<td></td>
<td>C</td>
<td></td>
<td></td>
<td>C</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>C</td>
<td>D</td>
<td></td>
<td></td>
<td></td>
<td>D</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Second-Change Page-Replacement Algorithm

- Keep a counter for each page and on page-fault, give the page 2 chances (N chances in a more general case)
- On page-fault, check
  - If R is zero, increment counter; if counter == N, replace this page
  - If R is 1, clear both, R and the counter (used since the last sweep)
- How do we pick N?
  - Large N would provide real good approximation. But have to inspect many pages
  - Small N would be more efficient.
- Pages with M bit set to 1 require swapping
- Probably make sense to give dirty pages (M == 1) more chances
Clock Page-Replacement Algorithm

- When page-fault occurs
- Starting with current position of the hand
- **While** not selected **do**
  - If \( R = 0 \)
    - Evict the page
    - Break
  - Set \( R = 0 \)
  - Advance hand

Done

- What if hand
  - Moves slowly
  - Moves quickly

Demand Paging

- In its purest form of paging, a new process starts with none of its pages in the memory
- Of course, running a process in this state will cause an immediate page-fault
- More page-faults follow as the process accesses data or uses the run-time stack or the heap
- The present/absent (valid/invalid) bit in the PTE is used to support demand paging
Working Set

• The set of pages that a process uses at each given time forms its working set
• Page-faults occur until the process’ working set is in memory
• As process progress, its working set changes. This is because its locality of references changes
• If there is not enough memory to hold the process’ working set, the process is likely to start thrashing; cause a page-fault after executing a few instructions
  – The OS spends most of its time swapping pages in and out

• Working set model
  – When a process starts, its working set is of course not known
  – However, once its working set is known, the process might lose the CPU
  – When such process becomes active again, to avoid repeated page-faults, all of its working set pages should be brought back into the memory before it runs

Working Set Model (Continued)

• $\Delta$  Working set window; a fixed number of page references
  – Example: 10,000 instructions
• WS of process $P$ is the total set of pages referenced in the most recent $\Delta$, which varies in time
  – If $\Delta$ too small will not encompass entire locality
  – If $\Delta$ too large will encompass several localities
  – If $\Delta \rightarrow \infty$ will encompass entire program
• In practice, it is easier to compute the working set if $\Delta$ was to represent N units of execution-time
Working-set Page-Replacement Algorithm

• Preparation and Assumptions
  – Every so often (at each clock tick), reset all reference-bits to zero
  – For each process, maintain its current virtual-time
  – Allow room to record the “time of last used” for each frame (page)
  – Choose the working set time (call it $\tau$, where $\tau >$ clock-tick)

• At the time of a page-fault
  – Inspect the value of the reference-bit
  – If it is equal to one
    • The page is in the working set of the process as it must have been
      referenced during this clock tic. Update its “time of last use”
  – If it is equal to zero
    • Is current-virtual-time – time-of-last-use $> \tau$
      – Yes: evict the page as it is not in the working set of the process
      – No: remember its “time of last use” if this is the first page or if its “time of
        last use” is less than all the pages that we have seen so far

Working-set Page-Replacement Algorithm (Cont.)

• If all the pages that we inspect are in the working set, then
  choose the page whose “time of last use” is minimum (oldest) to evict.

• What if the reference-bit of all the pages were
  – 1?
  – 0?
  – In either case, what does it say about $\tau$?
**WSClock Page-Replacement Algorithm**

- **Preparation and Assumptions**
  - We will maintain a circular list of pages (similar to Cock-Algorithm)
  - Allow room to record the “time of last used” for each frame (page)
  - We make use of the reference-bit. It gets cleared at each clock-tick
  - Choose $\tau$ as in the Working-Set Algorithm

- **At a page-fault, inspect the page that the “hand” points to**
  - Is its reference bit a 1?
    - Yes: The page is in the working set. Set its reference bit to zero
    - No: Is its dirty-bit set?
      - Yes: schedule a write for it
      - No: evict the page and use its frame (no need to write it to the disk)
  - If not victim-page has been found yet
    - Advance the hand and repeat the previous step

- **We may set a limit on the number of pages that get written**

---

**WSClock Page-Replacement Algorithm (Cont.)**

- **What if we inspect all the pages and do not find a suitable frame?**
  - Have we scheduled at least one write?
    - Yes: on the next cycle through the pages, we will find and use that frame
    - No: Did we see a clean page during the last scan?
      - Yes: evict it
      - No: evict the page that the hand points to
Process Transitions and Paging

- OS has to manage address-space of a process as it transitions through the system. The major activities include:
  - Process creation
    - Determine the initial size of the address-space
    - Allocate and initialize the page-table
    - Possibly allocate swap-space to the process
  - Process execution
    - Make the page-table active
    - Prepare the TBL for use by this process
    - Decide which pages to make memory resident
  - Page-fault
    - Use the virtual-address that caused the page-fault to identify the referenced page on the disk
    - Find a vacant frame for the referenced page
      - May require using the page-replacement algorithm to evict a page
    - Back up the PC as the instruction that caused the page-fault could not be completed

Process Transitions and Paging (Continued)

- Process Termination
  - Release the page-table, the pages, and swap-space
  - For each shared-page, adjust its reference count

- What do we do with an instruction that causes a page-fault?
  - No side-effects
    - Load RS, (date)  # load the value at address “data” into R5
    - Just back up the PC
  - Instruction with side-effects
    - Copy a data-segment from one page to another page: page-fault occurs after having copied some of the bytes
      - Is the copy overlapping?
    - Auto increment/decrement instructions
      - Mov R2+, -(R3)  # increment R2 (after use); decrement R3 (before use)
  - Hardware can help (in the form of hidden register)
    - Copy PC in a hidden register before it changes
    - Record the registers subject to auto increment/decrement
Handling Page-Faults

- The hardware traps to the kernel
- Trap handler saves the state of the process
  - This will ultimately either leads to a context-switch or the process dies
- OS determines which virtual-page is referenced using the logical address that triggered the page-fault
- If the referenced page is invalid, terminate the process
- Find a free frame for the referenced page
  - Might have to evict a page. In that case:
    - The page-replacement algorithm is applied to locate a victim page
    - If the victim page is dirty, schedule a back up (disk write)
    - Mark the just identified frame as pinned
- Locate the virtual-page referenced and schedule a read for it
- Context-switch to a different process

Handling Page-Faults (Continued)

- When the disk interrupt indicates that the page has arrived
  - Update the page-table entry
  - Unpin the new page
- Back up the PC to point to the instruction that caused the page-fault
- Move the process to the “ready” queue
- Once the process gets scheduled picking up where it left off as though the page-fault never happened
Separation of Policy and Mechanism

• The memory management system can be divided into
  – A low-level MMU handler (machine dependent)
  – A page-fault handler that runs in the kernel space (mechanism, machine independent)
  – An external pager that runs in the user space (enforces policy, machine independent)