Review: Summary of Chapter 3

- Deadlocks and its modeling
- Deadlock detection
- Deadlock recovery
- Deadlock avoidance
  - Resource trajectories
  - Safe and unsafe states
  - The banker's algorithm
- Two-phase locking
- More reading: Textbook 3.1 - 3.9
Chapter 4: Memory Management

4.1 Basic memory management
4.2 Swapping
4.3 Virtual memory
4.4 Page replacement algorithms
4.5 Modeling page replacement algorithms
4.6 Design issues for paging systems
4.7 Implementation issues
4.8 Segmentation

Memory Management

- Ideally programmers want memory that is
  - large
  - fast
  - non volatile
  - and cheap

- Memory hierarchy
  - small amount of fast, expensive memory – cache
  - some medium-speed, medium price main memory
  - gigabytes of slow, cheap disk storage

- Memory manager handles the memory hierarchy

Memory is cheap and large in today’s desktop, why memory management still important?
Monoprogramming without Swapping or Paging

- One program at a time, sharing memory with OS

Three simple ways of organizing memory. (a) early mainframes. (b) palmtop and embedded systems. (c) early PC.

Multiprogramming with Fixed Partitions

- Fixed-size memory partitions, without swapping or paging
  - Separate input queues for each partition
  - Single input queue
  - Various job schedulers

What are disadvantages?
Modeling Multiprogramming

° Degree of multiprogramming: how many programs in memory?
  • Independent process model: CPU utilization = 1 – \( p^n \)
    - A process spends a fraction \( p \) of its time waiting for I/O

Assumption on independence is not true!

Example
  • under the independent process model, a computer has 32 MB memory, with OS taking 16 MB and each program taking 4 MB. With an 80% average I/O wait, what is the CPU utilization? How much more CPU utilization if adding another 16 MB memory?

\[
1 - 0.8^4 = 60\% \\
1 - 0.8^8 = 83\% \\
1 - 0.8^{12} = 93\%
\]
Analysis of Multiprogramming System Performance

- Performance analysis in batching systems with R-R scheduling

<table>
<thead>
<tr>
<th>Job</th>
<th>Arrival time</th>
<th>CPU minutes needed</th>
<th># Processes</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>10:00</td>
<td>4</td>
<td>CPU idle</td>
</tr>
<tr>
<td>2</td>
<td>10:10</td>
<td>3</td>
<td>CPU busy</td>
</tr>
<tr>
<td>3</td>
<td>10:15</td>
<td>2</td>
<td>CPU process</td>
</tr>
<tr>
<td>4</td>
<td>10:20</td>
<td>2</td>
<td></td>
</tr>
</tbody>
</table>

- Arrival and work requirements of 4 jobs
- CPU utilization for 1 - 4 jobs with 80% I/O wait
- Sequence of events as jobs arrive and finish
  - note numbers show amount of CPU time jobs get in each interval

![Diagram of job arrival and CPU utilization](image)

Relocation and Protection

- Relocation: what address the program will begin in memory
  - Linker includes a list/bitmap with the binary program during loading
- Protection: must keep a program out of other processes’ partitions
  - Use base and limit values
    - address locations added to base value to map to physical address
    - address locations larger than limit value is an error

![Diagram of relocation and protection](image)

What are disadvantages?
Swapping (1) – Memory Relocation

- Swapping: bring in each process in its *entirety*, M-D-M- ...
- Key issues: allocating and de-allocating memory, keep track of it

Memory allocation changes as
- processes come into memory
- leave memory

Shaded regions are unused memory (memory holes)

Why not memory compaction?

Swapping (2) – Memory Growing

(a) Allocating space for growing *single* (data) segment
(b) Allocating space for growing stack & data segment

Why stack grows downward?
Memory Management with Bit Maps

- Keep track of dynamic memory usage: bit map and free lists

- Part of memory with 5 processes, 3 holes
  - tick marks show allocation units (*what is its desirable size?*)
  - shaded regions are free
  - Corresponding bit map (*searching a bitmap for a run of n 0s?*)
  - Same information as a list (better using a double-linked list)

Memory Management with Linked Lists

- De-allocating memory is to update the list

Four neighbor combinations for the terminating process X
Memory Management with Linked Lists (2)

- How to allocate memory for a newly created process (or swapping)?
  - First fit
  - Best fit; surely slow, but why could be more wasteful than first fit?
  - Worst fit
  - How about separate P and H lists for searching speedup?

- Example: a block of size 2 is needed for memory allocation

Virtual Memory

- Virtual memory: the combined size of the program, data, and stack may exceed the amount of physical memory available.
  - Swapping with overlays; but hard and time-consuming to split a program into overlays by the programmer
  - What to do more efficiently?
Logical program works in its contiguous virtual address space.

Address translation done by MMU

Actual locations of the data in physical memory

Terms
- Pages
- Page frames
- Page hit
- Page fault
- Page replacement

Examples:
- MOV REG, 0
- MOV REG, 8192
- MOV REG, 20500
- MOV REG, 32780

Page table gives the relation between virtual addresses and physical memory addresses.
Finding a Page in Memory or in Disk

Two data structures created by OS on creating a process
- To track which virtual address(es) use each physical page (in PT)
- To record where each virtual page is stored on disk (in PT or not)

In practice, could be in two tables.

CA: Placing a Page

What is page size?
Why no tags?
- indexed with the virtual page #
Page Tables

Two issues:

1. Page table can be large
   * Using registers?

2. Mapping must be fast
   * PT in the main mem.?

Who handles page faults?

Multi-level Page Tables

(a) 32 bit address with 2 page table fields.
(b) Two-level page tables
Structure of a Page Table Entry

- **Virtual Address**
  - Virtual page number
  - Page offset
  - Caching disabled
  - Modified / dirty
  - Present/absent
  - Referenced
  - Protection
  - Page frame number

**Who sets all those bits?**

CA: Translation Look-aside Buffers

Taking advantage of Temporal Locality:

A way to speed up address translation is to use a special cache of recently used page table entries -- this has many names, but the most frequently used is **Translation Lookaside Buffer** or **TLB**

<table>
<thead>
<tr>
<th>Virtual page number (virtual page #)</th>
<th>Cache</th>
<th>Ref/use</th>
<th>Dirty</th>
<th>Protection</th>
<th>Physical Address (physical page #)</th>
</tr>
</thead>
</table>

- TLB access time comparable to cache access time;
  much less than Page Table (usually in main memory) access time

**Who handles TLB management and handling, such as a TLB miss?**
A TLB Example

<table>
<thead>
<tr>
<th>Valid</th>
<th>Virtual page</th>
<th>Modified</th>
<th>Protection</th>
<th>Page frame</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>140</td>
<td>1</td>
<td>RW</td>
<td>31</td>
</tr>
<tr>
<td>1</td>
<td>20</td>
<td>0</td>
<td>R X</td>
<td>38</td>
</tr>
<tr>
<td>1</td>
<td>130</td>
<td>1</td>
<td>RW</td>
<td>29</td>
</tr>
<tr>
<td>1</td>
<td>129</td>
<td>1</td>
<td>RW</td>
<td>62</td>
</tr>
<tr>
<td>1</td>
<td>19</td>
<td>0</td>
<td>R X</td>
<td>50</td>
</tr>
<tr>
<td>1</td>
<td>21</td>
<td>0</td>
<td>R X</td>
<td>45</td>
</tr>
<tr>
<td>1</td>
<td>860</td>
<td>1</td>
<td>RW</td>
<td>14</td>
</tr>
<tr>
<td>1</td>
<td>861</td>
<td>1</td>
<td>RW</td>
<td>75</td>
</tr>
</tbody>
</table>

A TLB to speed up paging (usually inside of MMU traditionally)

Page Table Size

Given a 32-bit virtual address, 4 KB pages, 4 bytes per page table entry (memory addr. or disk addr.)

What is the size of the page table?

The number of page table entries:

\[ 2^{32} / 2^{12} = 2^{20} \]

The total size of page table:

\[ 2^{20} * 2^2 = 2^{22} \text{ (4 MB)} \]

When we calculate Page Table size, the index itself (virtual page number) is often NOT included!

What if the virtual memory address is 64-bit?
Inverted Page Tables

- Inverted page table: one entry per page frame in physical memory, instead of one entry per page of virtual address space.

Given a 64-bit virtual address,
4 KB pages,
256 MB physical memory

How many entries in the Page Table?
Home many page frames instead?
How large is the Page Table if one entry 8B?

Inverted Page Tables (2)

- Inverted page table: how to execute virtual-to-physical translation?
  - TLB helps! But what if a TLB miss?
CA: Integrating TLB, Cache, and VM

Just like any other cache, the TLB can be organized as fully associative, set associative, or direct mapped.

TLBs are usually small, typically not more than 128 - 256 entries even on high end machines. This permits fully associative lookup on these machines. Most mid-range machines use small n-way set associative organizations.

Translation with a TLB

Page Replacement Algorithms

° Like cache miss, a page fault forces choice
  • which page must be removed
  • make room for incoming page

° Modified page must first be saved
  • Modified/Dirt bit
  • unmodified just overwritten

° Better not to choose an often used page
  • will probably need to be brought back in soon
  • Temporal locality
**Optimal Page Replacement Algorithm**

- Replace page needed at the farthest point in future
  - Optimal but unrealizable
  - OS has to know when each of the pages will be referenced next
  - Good as a benchmark for comparison
    - Take two runs, the first run gets the trace, and the second run uses the trace for the replacement
    - Still, it is *only* optimal with respect to that specific program

**Least Recently Used (LRU)**

- Assume pages used recently will be used again soon
  - throw out page that has been unused for longest time
  - Example: 0 5 2 0 1 5

- Must keep a linked list of pages
  - most recently used at front, least at rear
  - update this list every memory reference !!!
    - finding, removing, and moving it to the front

- Special hardware:
  - Equipped with a 64-bit counter
  - keep a counter field in each page table entry
  - choose page with lowest value counter
  - periodically zero the counter (NRU)
  - And more simulation alternatives
Simulating LRU in Software

For a RAM with \( n \) page frames, maintain a matrix of \( n \times n \) bits; set all bits of row \( k \) to 1, and then all bits of column \( k \) to 0. At any instant, the row whose binary value is lowest is the least recently used.

<table>
<thead>
<tr>
<th>Page</th>
<th>Page</th>
<th>Page</th>
<th>Page</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

(a) (b) (c) (d) (e)

LRU using a matrix – pages referenced in order 0,1,2,3,2,1,0,3,2,3

Not Recently Used (NRU)

- Each page has R bit (referenced; r/w) and M bit (modified)
  - bits are set when page is referenced and modified
  - OS clears R bits periodically (by clock interrupts)
- Pages are classified
  1. not referenced, not modified
  2. not referenced, modified
  3. referenced, not modified
  4. referenced, modified
- NRU removes a page at random
  - From the lowest numbered non-empty class
**FIFO Page Replacement Algorithm**

- Maintain a linked list of all pages
  - in order they came into memory
- Page at beginning of list replaced (the oldest one)
- Disadvantage
  - page in memory the longest (oldest) may be often used

---

**Second Chance Page Replacement Algorithm**

- OS clears R bits periodically (by clock interrupts)
- Second chance (FIFO-extended): looks for an oldest and not referenced page in the previous clock interval; if all referenced, FIFO
  - (a) pages sorted in FIFO order
  - (b) Page list if a page fault occurs at time 20, and A has R bit set (numbers above pages are loading times);
    - (c) what if A has R bit cleared?
The Clock Page Replacement Algorithm

- The clock page replacement algorithm differs from Second Chance only in the implementation.
  - No need to move pages around on a list.
  - Instead, organize a circular list as a clock, with a hand pointing to the oldest page.

When a page fault occurs, the page the hand is pointing to is inspected. The action taken depends on the R bit:
  - R = 0: Evict the page
  - R = 1: Clear R and advance hand

Not Frequent Used (NFU)

- NFU (Not Frequently Used): uses a counter per page to track how often each page has been referenced, and choose the least to kick out.
  - OS adds R bit (0 or 1) to the counter at each clock interrupt.
  - Problem: never forgets anything.
Aging - Simulating LRU/NFU in Software

- Aging: the counters are each shifted right 1 bit before the R bit is added in; the R bit is then added to the leftmost
  - The page whose counter is the lowest is removed when a page fault

<p>| R bits for | R bits for | R bits for | R bits for | R bits for |
| pages 0-5, | pages 0-5, | pages 0-5, | pages 0-5, | pages 0-5, |</p>
<table>
<thead>
<tr>
<th>clock tick 0</th>
<th>clock tick 1</th>
<th>clock tick 2</th>
<th>clock tick 3</th>
<th>clock tick 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>01000000</td>
<td>10000000</td>
<td>00100000</td>
<td>00010000</td>
<td>00001000</td>
</tr>
<tr>
<td>10000000</td>
<td>11000000</td>
<td>10100000</td>
<td>01000000</td>
<td>01000000</td>
</tr>
<tr>
<td>00000000</td>
<td>00000000</td>
<td>10000000</td>
<td>01000000</td>
<td>01000000</td>
</tr>
<tr>
<td>01000000</td>
<td>01000000</td>
<td>01100000</td>
<td>10110000</td>
<td>10110000</td>
</tr>
<tr>
<td>00000000</td>
<td>00100000</td>
<td>10100000</td>
<td>01010000</td>
<td>01010000</td>
</tr>
</tbody>
</table>

The aging algorithm simulates LRU in software, 6 pages for 5 clock ticks, (a) – (e)

The Working Set and Pre-Paging

- Demand paging vs. pre-paging
- Working set: the set of pages that a process is currently using
- Thrashing: a program causing page faults every few instructions
- Observation: working set does not change quickly due to locality
  - Pre-paging working set for processes in multiprogramming

![Example: 0, 2, 1, 5, 2, 5, 4](image)

The working set is the set of pages used by the k most recent memory references w(k,t) is the size of the working set at time, t
The Working Set Page Replacement Algorithm

The WSClock Page Replacement Algorithm

Operation of the WSClock Algorithm
### Review of Page Replacement Algorithms

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>Optimal</td>
<td>Not implementable, but useful as a benchmark</td>
</tr>
<tr>
<td>NRU (Not Recently Used)</td>
<td>Very crude</td>
</tr>
<tr>
<td>FIFO (First-In, First-Out)</td>
<td>Might throw out important pages</td>
</tr>
<tr>
<td>Second chance</td>
<td>Big improvement over FIFO</td>
</tr>
<tr>
<td>Clock</td>
<td>Realistic</td>
</tr>
<tr>
<td>LRU (Least Recently Used)</td>
<td>Excellent, but difficult to implement exactly</td>
</tr>
<tr>
<td>NFU (Not Frequently Used)</td>
<td>Fairly crude approximation to LRU</td>
</tr>
<tr>
<td>Aging</td>
<td>Efficient algorithm that approximates LRU well</td>
</tr>
<tr>
<td>Working set</td>
<td>Somewhat expensive to implement</td>
</tr>
<tr>
<td>WSClock</td>
<td>Good efficient algorithm</td>
</tr>
</tbody>
</table>

#### Belady’s Anomaly

- More page frames of memory, fewer page faults, true or not?
  - FIFO of 0 1 2 3 0 1 4 0 1 2 3 4 in 3-page and 4-page memory

- (a) FIFO with 3 page frames.
- (b) FIFO with 4 page frames.