Lecture.10.Virtual.Memory.txt

- Virtual Memory
    - Separation of user logical memory from physical memory
        - Logical address space can therefore be much larger than physical address space
            - The memory management unit (MMU) is used to correlate logical and physical addresses
        - Allows address spaces to be shared by several processes
    - Virtual address space is the logical view of how processes are stored in memory
    - If there is not enough space in memory, paging or segmentation is used
        - Segments are considered as huge pages
    - Virtual memory can be implemented via:
        - Demand paging
        - Demand segmentation

- Virtual Memory That Is Larger Than Physical Memory
    - i.e. Diagram of Virtual Memory

```
|--------|
| page 0 |
|--------|
| page 1 |
|--------|
| page 2 |
|--------|
|        |
|--------|
|        |
|    *   |
|    *   |
|    *   |
|        |
|--------|
| page v |
|--------|
 Virtual
 Memory
```

    - Data moves from virtual memory to physical memory to backing store
        - It can also move from backing store to physical memory
        - This is mostly a concept

- Virtual Address Space
    - Logical address space for stack to start at max and grow "down", while heap grows "up"
        - Heap is dynamically allocated memory
        - The heap and stack grow toward each other
    - Enables sparse address spaces with holes left for growth,

dynamically linked libraries, etc.

- System libraries shared via mapping into virtual address space
  - This virtual address space is used for all processes that require the system library
- i.e. Diagram

```
Max |-------|
    | stack |
    |--||---|
    |  ||   |
    |  \/   |
    |       |
    |   /\  |
    |   ||  |
    |---||--|
    | heap  |
    |-------|
    | data  |
    |-------|
    | code  |
  0 |-------|
```

- Shared Library Using VM
  - i.e. Diagram

```
|---------|                     |---------|
|  stack  |                     |  stack  |
|----||---|                     |----||---|
|    ||   |                     |    ||   |
|    ||   |                     |    ||   |
|    \/   |   |-------------|   |    \/   |
|         |   |             |   |         |
|-------------|             |-------------|
| shared  |        shared       | shared  |
| library |        pages        | library |
|-------------|             |-------------|
|         |   |             |   |         |
|   /\    |   |-------------|   |   /\    |
|   ||    |                     |   ||    |
|   ||    |                     |   ||    |
|---||----|                     |---||----|
|  heap   |                     |  heap   |
|---------|                     |---------|
|  data   |                     |  data   |
|---------|                     |---------|
|  code   |                     |  code   |
|---------|                     |---------|
 Process P1                      Process P2
```

  - In the diagram above, both processes, P1 and P2, share data
    - They share data on the same virtual address space

- Demand Paging

```
    - The entire process is swapping pages in and out
        - Could bring entire process into memory at load time
        - Or bring a page into memory only when it is needed
    - If page is needed, reference to it
        - An invalid reference -> Abort
        - If it is not-in-memory -> Bring it to memory

- Page Table When Some Pages Are Not In Main Memory
    - i.e. Diagram of Logical Memory
          |---|
        0 | A |
          |---|
        1 | B |
          |---|
        2 | C |
          |---|
        3 | D |
          |---|
        4 | E |
          |---|
        5 | F |
          |---|
        6 | G |
          |---|
        7 | H |
          |---|
           Logical
           Memory
    - i.e. Diagram of Page Table
        |---|---|
        | 4 | v |
        |---|---|
        |   | i |
        |---|---|
        | 6 | v |
        |---|---|
        |   | i |
        |---|---|
        |   | i |
        |---|---|
        | 9 | v |
        |---|---|
        |   | i |
        |---|---|
        |   | i |
        |---|---|
         Page Table
    - i.e. Diagram of Physical Memory
           |-----|
         0 |     |
```

```
          |------|
        1 |      |
          |------|
        2 |      |
          |------|
        3 |      |
          |------|
        4 |  A   |
          |------|
        5 |      |
          |------|
        6 |  C   |
          |------|
        7 |      |
          |------|
        8 |      |
          |------|
        9 |  F   |
          |------|
       10 |      |
          |------|
       11 |      |
          |------|
       12 |      |
          |------|
       13 |      |
          |------|
       14 |      |
          |------|
       15 |      |
          |------|
          Physical
          Memory
   - i.e. Diagram of Backing Store
       |---------------------------|
       |                           |
       |   |---|   |---|   |---|   |
       |   |   |   |   |   |   |   |
       |   |---|   |---|   |---|   |
       |                           |
       |   |---|   |---|   |---|   |
       |   |   |   | A |   | B |   |
       |   |---|   |---|   |---|   |
       |                           |
       |   |---|   |---|   |---|   |
       |   | C |   | D |   | E |   |
       |   |---|   |---|   |---|   |
       |                           |
       |   |---|   |---|   |---|   |
       |   | F |   | G |   | H |   |
```

```
|   |---|   |---|   |---|   |
|                           |
|   |---|   |---|   |---|   |
|   |   |   |   |   |   |   |
|   |---|   |---|   |---|   |
|                           |
|---------------------------|
```

- In the diagrams above, Page 0 (that holds 'A') in the page table is mapped to frame 4, and the valid-invalid bit is set to 'v'
  - This process is located at address 4, inside physical memory
- In the diagrams above, Page 1 (that holds 'B') in the page table is not mapped to any frame
  - Thus, 'B' is not inside physical memory
    - Instead, 'B' is inside the backing store
  - If you wanted to run process 'B', it is the job of the operating system to copy 'B' from the backing store into physical memory
- With each page table entry a valid-invalid bit is associated
  - Indicates if the page has a corresponding frame in physical memory
- During MMU address translation, if valid-invalid bit in page table entry is invalid -> page fault exception

- Demand Paging
  - Get the bad virtual address that caused the page fault
  - Allocate a physical page
    - Find the page in the backing store
  - Call the file system to copy the page from the disk to the physical page in memory
    - Copy page from backing store into physical memory
  - Update the page table
  - Re-execute the instruction

- Steps In Handling A Page Fault
  - Assume that process 'M' is being loaded, and it is not already in physical memory
    - The steps are:
      1. Reference
         - Try to look for 'M' in page table
      2. Trap
         - Since 'M' is not in the page table, a trap is created by the operating system (OS)
      3. Page is on backing store
         - The OS locates 'M' in the backing store
           - Backing store is the hard drive (HDD)
      4. Bring in missing page
         - 'M' is copied into physical memory
      5. Reset page table
         - The page table is updated with new information

                - New information is 'M' and its location
            6. Restart instruction
                - When the instruction is restarted, it will be properly executed this time
    - Note: It is the job of the operating system to handle page faults

- Free Frame List
    - When a page fault occurs, the operating system must bring the desired page from secondary storage into main memory
    - Most operating systems maintain a free frame list
        - This is a pool of free frames for satisfying such requests
        - The list contains all free frames in physical memory
        - New data from the backing store is copied into these free frames
            - Then, the page table is updated with information that contains the translation of logical to physical address
    - Zero-fill-on-demand
        - The content of the frames zeroed-out before being allocated
    - When a system starts up, all available memory is placed on the free frame list

- Performance Of Demand Paging
    - Moving page(s) into main memory takes time
    - There are 3 major activities:
        1. Service the interrupt
        2. Read the page
        3. Restart the process
    - Page fault rate: 0 <= p <= 1
        - If the page fault rate is very low, then the system will run quickly
        - A high page fault rate slows down the system
    - Effective Access Time (EAT)
        - EAT = (1 - p) * memory access +
          p * (page fault overhead + swap page out
          + swap page in)
            - If `p` is 1, then `1 -p` is cancelled out
                - Only the second part needs to be calculated
                    - This is the page fault overhead of swap page out and swap page in
            - If `p` is 0, then there is no fault
                - EAT = Memory access time
    - Demand Paging Example
        - For memory access time = 200ns, and average page fault service time is 8ms
        - EAT = (1 - p) * 200ns + p * 8000000ns
        - For p = 0, E = 200ns
        - For p = 1 / 10000, EAT = 8us --> Slowdown by a factor of 40
            - This demonstrates why it is important to not have a page fault, because it dramatically slows down the system

            - Even a small value of `p` can drastically slow down the system

- Basic Page Replacement
    - When a page fault occurs, we need to:
        - Find the location of the desired page on disk
        - Then, 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 and write victim frame to disk
        - Bring the desired page into the (newly) free frame and update the page and frame tables
        - Restart the instruction that caused the trap

- Page Replacement
    - Assuming that the physical memory is full, the page replacement steps are:
        1. Swap out victim page
            - This is the page that will be rewritten
        2. Change to invalid
            - The old page is no longer in physical memory, so its valid-invalid bit is set to 'i'
        3. Swap desired page in
        4. Reset page table for new page

- Page & Frame Replacement Algorithms
    - FIFO
        - First in, first out
    - OPT
        - Optimal algorithm
            - This is not possible to implement in reality, but good to have for reference
            - Other algorithms are compared to the optimal algorithm
    - LRU
        - Least recently used
    - LFU
        - Least frequently used
    - MFU
        - Most frequently used
    - Depending on the scenario, some algorithms work better than others
    - Page replacement algorithms move pages from secondary storage into main memory

- 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
        - These numbers represent data
            - i.e. Processes
    - 3 frames

    - 3 pages can be in memory at a time per process
  - In the FIFO algorithm, the first frame is used for the replacement
    - This algorithm is very simple to implement, but it is not effective
  - i.e. FIFO Diagram

Reference string (part 1)

7	0	1	2	0	3	0	4	2	3	0	3	2	1	2
7	7	7	2		2	2	4	4	4	0			0	0
	0	0	0		3	3	3	2	2	2			1	1
		1	1		1	0	0	0	3	3			3	2
*	*	*	*		*	*	*	*	*	*			*	*

Hit

Reference string (part 2)

0	1	7	0	1
		7	7	7
		1	0	0
		2	2	1
		*	*	*

Hit

  - 15 page faults in total
    - Count the asterisks in the diagram above
  - Adding more frames can cause more page faults
    - This is known as Belady's Anomaly

- FIFO: Belady's Anomaly
  - Belady's anomaly states that adding more frames can cause more page faults
    - Thus, a bad replacement choice increases the page-fault rate and slows down process execution
  - i.e. Table of Page Faults

Number Of Frames	Number Of Page Faults
1	12
2	12
3	9
4	10

```
| 5          | 5            |
| 6          | 5            |
| 7          | 5            |
|------------|--------------|
```

- The reference string for this table is:
  1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5
- According to the table above, increasing the number of frames from 3 to 4 also increases the number of page faults from 9 to 10

- Optimal Algorithm
    - Replace page that will not be used for longest period of time
    - How do you know this?
        - We don't, because we can't see the future
    - Used for measuring how well your algorithm performs
        - This algorithm yields the best possible result
            - No algorithm can be better than the optimal algorithm
    - i.e. Optimal Algorithm Diagram

```
 Reference string (part 1)
| 7 | 0 | 1 | 2 | 0 | 3 | 0 | 4 | 2 | 3 | 0 | 3 | 2 | 1 | 2 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 7 | 7 | 7 | 2 |   | 2 |   | 2 |   |   | 2 |   |   | 2 |   |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
|   | 0 | 0 | 0 |   | 0 |   | 4 |   |   | 0 |   |   | 0 |   |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
|   |   | 1 | 1 |   | 3 |   | 3 |   |   | 3 |   |   | 1 |   |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| * | * | * | * |   | * |   | * |   |   | * |   |   | * |   |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
 Hit

 Reference string (part 2)
| 0 | 1 | 7 | 0 | 1 |
|---|---|---|---|---|
|   |   | 7 |   |   |
|---|---|---|---|---|
|   |   | 0 |   |   |
|---|---|---|---|---|
|   |   | 1 |   |   |
|---|---|---|---|---|
|   |   | * |   |   |
|---|---|---|---|---|
 Hit
```

- Only 9 page faults
    - This is the best possible outcome

- Least Recently Used (LRU) Algorithm
    - Replace page that has not been used in the most amount of time
        - Associate time of last use with each page
        - The idea behind LRU is that if a page has not been recently

used, then it won't be used again in the future
- i.e. LRU Diagram

  Reference string (part 1)

7	0	1	2	0	3	0	4	2	3	0	3	2	1	2
7	7	7	2		2		4	4	4	0			1	
	0	0	0		0		0	0	3	3			3	
		1	1		3		3	2	2	2			2	
*	*	*	*		*		*	*	*	*			*	

  Hit

  Reference string (part 2)

0	1	7	0	1
1		1		
0		0		
2		7		
*		*		

  Hit
- 12 page faults
  - Better than FIFO, but worse than OPT
  - Generally good algorithm, and frequently used

- LRU Algorithm
  - There are two ways to implement the LRU algorithm
    - Counter Implementation
      - Every page entry has a counter
      - Every time a page is referenced, copy the clock into the counter
        - In this way, we always have the "time" of the last reference to each page.
      - When a page needs to be changed, look at the counters to find smallest value
    - Stack Implementation
      - Keep a stack of page numbers in a double link form
      - Whenever a page is referenced, it is removed from the stack and put on the top

- Stack Implementation
  - i.e. Stack Diagram

```
4   7   0   7   1   0   1   2   1   2   7   1   2
                                    |   |
```

```
                                           |   |
                                           a   b
     | 2 |        | 7 |
     |---|        |---|
     | 1 |        | 2 |
     |---|        |---|
     | 0 |        | 1 |
     |---|        |---|
     | 7 |        | 0 |
     |---|        |---|
     | 4 |        | 4 |
     |---|        |---|
     stack        stack
     before       after
     'a'          'b'
```

- The least recently used page is always at the bottom
- LRU and OPT are cases of stack algorithms that don't have Belady's Anomaly

- Approximaion Of LRU: Clock Algorithm
  - LRU needs special hardware
  - Reference bit algorithm
    - With each page associate a bit, initially = 0
      - When page is referenced bit set to 1
    - 8 bit shift registers contain the history of page use for the last eight time periods
      - If the shift register contains 00000000, for example, then the page has not been used for eight time periods
  - Second chance algorithm
    - If page to be replaced has reference bit = 0
      - Then, replace it
    - If reference bit = 1, then set reference bit to 0, leave page in memory, replace next page, subject to same rules

- Second Chance (Clock) Page Replacement Algorithm
  - Looks like a circular queue of pages
  - If next victim's reference bit is 1, then it is set to 0
  - If next victim's reference bit is 0, it is replaced

- Counting Algorithms
  - Other page replacement algorithms require a counter
    - Keep a counter of the number of references that have been made to each page
  - Least frequently used (LFU) algorithm
    - Replaces page with smallest count
    - The rationale is that if a page has not been used for an extended period of time, it may not be used again
      - Thus, it is better to replace it
  - Most frequently used (MFU) algorithm
    - Based on the argument that the page with the smallest count

was probably just brought in, and has yet to be used
    - Thus, it will be used again, so it is better to keep it in memory

- Page Buffering Algorithms
  - Algorithms can be improved by always keeping a pool of free frames
    - Read page into free frame and select victim to evict and add to free pool
  - Possibly, keep list of modified pages. Whenever the paging device is idle, a modified page is selected and is written to secondary storage
  - Possibly, keep free frame contents intact and note what is in them
    - If referenced again before reused, no need to load contents again from disk
      - This is used to improve performance, and is heavily used in operating systems
  - All of these algorithms have OS guessing about future page access
    - There is no guarantee about future page access, only probability

- Thrashing & Working Set Model
  - Thrashing occurs when a process spends more time paging than executing
    - The memory becomes as slow as the hard drive (HDD)
    - Trashing may significantly decrease performance, and waste time moving data between secondary and primary storage
  - One way to prevent trashing is through the use of a working set model
  - Working set model
    - Basis: Locality
    - Working set window (WSW)
      - A time frame
      - If the frame is inside the WSW, then it is not removed
        - Even if the frame is marked as a victim
    - Working set (WS)
      - A set of pages referenced in the time frame
    - Working set size (WSS)
      - Number of pages in WS
  - Page replacement can be determined by working set model
  - A working set model can prevent thrashing

- Trashing & Working Set
  - How does the system detect thrashing?
    - The system can detect thrashing by evaluating the level of CPU utilization compared with the level of multi-programming
      - If the CPU is extremely busy, then something is not

correct, because a few threads should not take up all the processing time
        - This implies that there is thrashing
  - Is it possible for a process to have two working sets, one for representing data and another representing code?
    - Yes
      - In fact, many processors provide 2 TLBs for this very reason
      - As an example, the code being accessed by a process may retain the same working set (WS) for a long time
        - However, the data the code accesses may change, thus reflecting a change in the WS for the data accesses

- End
  - Operating systems are among the most complex pieces of software ever developed!