You only need to have the option selected in the memory of the computer eg A game option that is selected is in memory, don’t need options not currently running.

**Background**

**Virtual memory (logical memory):** Separation of logical and physical.

**Advantages:**  
Only part of the program needs to be in memory for execution.

Can share address spaces by several processes.

More efficient process creation.

Allows more programs to run concurrently.

Less I/O needed to swap programs into memory. (Broken programs into smaller pieces [pages])

**Virtual address space:** logical view of how process is stored in memory

MMU (memory management unit) maps logical to physical.

Start at address 0, with contiguous addressing until end of space.

Physical memory organised into page frames.

Can be implemented into:

**Demand paging**: Pages are even in size.

**Demand segmentation**: Pages are uneven in size.

**Page table describes:** If a page is in memory and where it is, and it also describes if it is not in memory.  
A memory map will show where to find it in storage/hard disk, or physical disk (this is probs the page table).

**Virtual address space**

Broken into stack…heap…data…code… It has some max and min space. Stack **grows** **down** from max, and heap **grows** **up** from min.

Can have shared libraries between processes.

**Demand Paging**

We have learnt that a process is a .exe, and is loaded into memory. If not enough memory, can swap down to disk.

**Swapping** is same principle, however don’t need to page in **whole process**, instead **page in and out** pages that you need (swapping in and out).

**Lazy swapper:** Only **swaps pages** into memory if its **needed**.

**Basic Concepts**

Need ability to detect if pages needed are not in memory.

Extra functionality is needed.

**Valid-Invalid Bit**

Invalid and valid bit can now be used to detect if a frame is valid/in memory (in a valid location).

If has a i (invalid), is a **page fault**.

**Page fault**

The OS looks at another table to decide if its **actually invalid** (aborts process), or if the memory is just not free (**valid process**).   
1.In that case, it must find some valid memory (free frame) to put this page in (via scheduled disk operation).   
2.After success, sets bit to valid (v).  
3.Then restarts instruction that caused the page fault

**Aspects of Demand Paging**

**Pure demand paging:** Page table consists of many invalid pages (eg if you keep on clicking on program over and over). Where you start process with **no** pages in memory (**extreme case**).

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | 0 | 1 | 2 | 3 |
| 0 | a | b | c | D |
| 1 | e | f | G | h |
| 2 | i | k | k | L |
| 3 | m | n | o | p |

How we see a 2D Array

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| a | b | c | d | e | f | g | h | i | j | k | l | m | n | o | P |

This is how its actually like in memory. It is row by row.

**This is called locality of reference:** Page a bunch from array into memory (row by row). When memory full, page fault occurs. So if you have enough memory allocated for one row.

**This guesses that a page might access other pages eg a accesses b, so it gets a-d pages when a is called. (maybe???)**

-As a programmer you have an influence on how many page faults occur (eg if you swap it to do cols then rows, you will run into more page faults)

**NB example**

**For(rows)  
 for(cols)  
 array[rows][cols]**

**Swapping around the for loops is what changes speed. You want rows as outer loop.**

**Hardware demands:**

1.Page tables have valid/invalid bits.

2.Secondary memory space (to do swaps **[swap space])**

3.Processor must have ability to restart instruction.

**Demand Paging Optimiszations (you can just read through this – probs not NB)**

**What happens if no free frame?**

You have to figure out which page to swap out. You can use **Page replacements. The type of algorithm (maybe how to do) is a good exam question.**

**Page replacement**

**Prevent over-allocation** of memory allocated by modifying page-faults to include page replacement.

You want to keep track of what is and Is not modified. **Only modified pages are written to disk.** You can keep track of modified (**dirty) bits** to reduce overhead of page transfers.

**Basic page replacement**

1. Ind location of desired page on disk
2. Find free frame  
   -IF there is free frame, use it  
   -Else, use replacement algorithm to sleect a **victim frame   
   --**Write victim frame to disk if dirty
3. Bring desired page into newly free frame; update the page and frame tables
4. Continue the process by restarting the instruction that caused the trap.

**Page and Frame Replacement Algorithms**

**Frame Allocation algorithm:** Determines **how many** frames to **give to each process** and **which frames** to **replace.**

**FIFO (NBNB in exam!)**

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Page Requests | 9 | 7 | 8 | 8 | 7 | 7 | 4 | 3 | 9 |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  | FIFO |  |  |  |
| Page Tables | 9 | 9 (page 9 is still in table from prev step) | 9 | 9 | 9 | 9 | 4 | 4 | 4 |  |  |  |  |  |
|  |  | 7 | 7 | 7 | 7 | 7 | 7 | 3 | 3 |  |  |  |  |  |
|  |  |  | 8 | 8 | 8 | 8 | 8 | 8 | 9 |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| Page Faults: | 1 | 2 | 3 |  |  |  | 4 | 5 | 6 |  |  |  |  |  |

\* \* \* Notice at stage 8 and 7, that no page faults occur due to already being there

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Page Requests | 9 | 7 | 8 | 8 | 4 | 3 | 9 | 7 | 8 |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  | FIFO |  |  |  |
| Page Tables | 9 | 9 | 9 | 9 | 4 | 4 | 4 | 7 | 7 |  |  |  |  |  |
|  |  | 7 | 7 | 7 | 7 | 3 | 3 | 3 | 8 |  |  |  |  |  |
|  |  |  | 8 | 8 | 8 | 8 | 9 | 9 | 9 |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| Page Faults: | 1 | 2 | 3 |  | 4 | 5 | 6 | 7 | 8 |  |  |  |  |  |

Now we are using an Optimal paging algorithm (theoretical algorithm, does not work in reality as no OS can know what’s going to happen in future). This is used as a gold standard to compare how well other algorithms perform.

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Page Requests | 9 | 7 | 8 | 8 |  | 4 | 3 | 9 | 7 | 8 |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  | OPTIMAL |  |  |  |
| Page Tables | 9 | 9 | 9 | 9 |  | 9 | 9 | 9 | 9 | 9 |  |  |  |  |  |
|  |  | 7 | 7 | 7 |  | 7 | 7 | 7 | 7 | 7 |  |  |  |  |  |
|  |  |  | 8 | 8 |  | 4 | 3 | 3 | 3 | 8 |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| Page Faults: | 1 | 2 | 3 |  |  | 4 | 5 |  |  | 6 |  |  |  |  |  |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Pages** | **Loaded** | **Last Ref** | **R** | **M** |
| A | 194 | 221 | 0 | 0 |
| B | 138 | 270 | 0 | 1 |
| C | 214 | 272 | 1 | 1 |
| D | 96 | 386 | 1 | 1 |
|  |  |  |  |  |

**Loaded**: Time when page was loaded in table

**Last Reference:** Each time you access an address, you indicate that time (so time when last accessed at).

**R:** Bit that changes as soon as something is referenced (changes to 1). At some point changes to 0 (not used for a while, depending on algorithm).

**M:** Modified bit, if a change has been made to frame and must be saved to disk.

We can see that D is the oldest page. Then B, then A, then C.

So even though D is the oldest, it is the most recently accessed page.

**Here are the different methods:**

**FIFO:** Victim Frame Candidate **is** D– as D is the oldest.

**LRU (least recently used):** Oldest Last referenced page, where R=0.  
-A

**2nd Chance:** FIFO, but you take into the reference bit into consideration. So go to D, but as R=1, then change D R->0, and take 2nd oldest as victim(B).

(**Below also important for tests)**

**Enhanced second chance:** Takes into R and M. As it is more efficient to take out a non modified page as no writing to disk is needed (where m=0). But in addition, you don’t want to take out a recently referenced page. So you look at decimal values of R and M.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Pages** | **Loaded** | **Last Ref** | **R** | **M** | The extra col is the binary 00, 01, 11 and 11. |
| A | 194 | 221 | 0 | 0 | 0 |
| B | 138 | 270 | 0 | 1 | 1 |
| C | 214 | 272 | 1 | 1 | 3 |
| D | 96 | 386 | 1 | 1 | 3 |
|  |  |  |  |  |  |

As we can see, 0 is smallest so we take out A.

**Note for this type of table:**

The last Ref is a waste of time, as you have to update the table every time it is accessed. Instead, there are different ways to see when it was last referenced.

Stack Approach **(Need to know this as well for exam [LRU slide it refers to])**

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Requests | 6 | 0 | 9 | 7 | 1 | 1 | 5 | 7 |  |
|  |  |  |  |  |  |  | a | b |  |

This is the stack (at a)

|  |
| --- |
|  |
|  |
| 5 |
| 1 |
| 7 |
| 9 |
| 0 |
| 6 |

Now accessing 7 (at b) – using doubly linked list implementation.

|  |
| --- |
|  |
|  |
| 7 |
| 5 |
| 1 |
| 9 |
| 0 |
| 6 |

Stack Approach (Get rid of the original, and add new eg 6 on top)

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Requests | 6 | 0 | 5 | 6 | 1 | 1 | 5 | 7 |  |
|  |  |  |  |  |  |  | a | b |  |

This is the stack (at a)

|  |
| --- |
|  |
|  |
|  |
|  |
| 5 |
| 1 |
| 6 |
| 0 |

Now accessing 7 (at b)

|  |
| --- |
|  |
|  |
|  |
| 7 |
| 5 |
| 1 |
| 6 |
| 0 |

Page fault occurs: Traps to OS (where OS must handle error). This occurs when you eg Don’t have enough memory for page.

**Thrashing**

For a process to execute successfully, the pages of the process must be in the memory of the computer. Some of the unused pages can be on disk.

Now, if the page fault rate is high, then it occurs where **we use a page, put the page on disk, then take that page back from disk (This is thrashing)**. This normally happens when you run low on memory, or too many processes for RAM.

1.This cause low CPU utilization, as no progress made during this.  
2. OS thinking it needs to increase the degree of multiprogramming (which actually just increases issue)  
3. Another process added to the system (From above)

Solution:

OS forces x amount of processes to sleep, while having certain processes run to completion.

**Working set**

Hypothetical size of memory each process needs to run without causing too many page faults.

You want the working set to be well established, and not too high or low.

**Best Solution to Reduce Thrashing: Increase Ram of Computer**

The order in which you process data, such as doing col then row, can cause n^2 page faults, compared to row then col, which would only be n faults.