CS2106 Operating Systems
Tutorial 9 **Suggested Solutions**

Virtual Memory Management

1. [TLB + Paging + Virtual Paging] We have learnt the idea of paging, translation look aside buffers (TLBs) and virtual memory in the last two lectures. In this question, we are going to put them all in the same scenario and study the interaction.

   Here is the basic setup. Note: To make the question easier to comprehend, the scale is vastly reduced compared to the real world counterparts.

   - Page Size = Frame Size = 4KB
   - TLB = 2 entries (0..1)
   - Page Table = 8 entries (i.e. 8 pages, 0..7)
   - Physical Frame = 8 frames (0..7)
   - Swap Page (Virtual page in hard disk) = 16 pages (0..15)

   Below is a snapshot of process P at time T:

   TLB (should have the same fields as the PTE, not shown for simplicity)**:**

Page No.	Frame No.
3	7
0	4

   Page Table:

Page No	Frame No/ Swap Page No	Memory Resident?	Valid?
0	4	1	1
1	1	1	1
2	1	0	1
3	7	1	1
4	2	1	1
5	15	0	1
6	---	0	0
7	---	0	0

   Although the content / layout of the physical memory frames and hard disk swap pages can be deduced from the above, you are encouraged to sketch the RAM and hard disk content to aid understanding.

   a. Below is the skeleton of the access algorithm for this setup. Give the missing algorithm for the OS **TLB fault** and **page fault** handlers.

Algorithm for accessing virtual address VA:

1. [HW] VA is decomposed into <Page#, Offset> **2. [HW] Search TLB for <Page#>:** **a. If TLB miss: Trap to OS { *TLB Fault* }** **3. [HW] Is <Page#> memory resident?** **a. Non-memory resident: Trap to OS { *Page Fault* }** 4. **[HW] Use <Frame#><Offset> to access physical memory.**

b. Using (a), walkthrough the following access **in sequence.** For simplicity, only the page number is given. If TLB/ Page Replacement is needed, just pick the entry with the smallest page number.

i. Access Page 3
ii. Access Page 1
iii. Access Page 5

(Optional) Think about how a dynamically allocated new page fit into this scheme.

c. If we have the following hardware specification:
- TLB access time = 1ns
- Memory access time = 30ns
- Hard disk access time (per page) = 5ms

Give the best and worst memory access scenarios and the respective memory access speed. For simplicity, you can give an approximate answers.

d. If 2-Level paging is used, is there a worse scenario compared to (c)?

ANS:

a.

Algorithm for accessing virtual address VA:
1. [HW] VA is decomposed into <Page#, Offset> **2. [HW] Search TLB for <Page#>:** **a. If TLB miss: Trap to OS { *TLB Fault* }** **3. [HW] Is <Page#> memory resident?** **a. Non-memory resident: Trap to OS { *Page Fault* }** 4. **[HW] Use <Frame#><Offset> to access physical memory.**

Algorithm for TLB Fault Handler, given <Page #>:
1. [OS] Access full table of the process (e.g. in the PCB of the process), <Page#> is the index **2. [OS] Check whether valid bit is set, if not segmentation fault.** 3. **[OS] Load the relevant PTE into TLB.** 4. **[OS] Return from trap.**

Algorithm for Page Fault Handler, given <Page #>:

1. **[OS] Global/Local replacement, Replacement algorithm applicable here**
2. **[OS] Write out the page to be replaced if needed.**
3. **[OS] Locate the page in secondary swap pages.**
4. **[OS] Load the page into a physical frame.**
5. **[OS] Update page table entries.**
6. **[OS] Update TLB**
7. **[OS] Return from Trap**

Note: This algorithm assumes that OS handles the TLB fault (commonly known as **software managed TLB**). Some architectures provide additional hardware assistance for this (i.e. "TLB fault handler" above will be handled by hardware instead of OS). Essentially, the base address of the full-page table is stored in a dedicated register. During a TLB-Miss, the processor will automatically retrieve the missing page table entry from memory as it is just a simple offset from the base of page table. This is commonly known as **hardware managed TLB.**

b.

i. Access Page 3: TLB-Hit

ii. Access Page 1: TLB-Miss, Memory Resident, i.e. TLB handler executed

TLB:

Page No.	Frame No.
3	7
1	**1**

Page Table

Page No	Frame No/ Swap Page No	Memory Resident?	Valid?
0	4	1	1
1	1	1	1
2	1	0	1
3	7	1	1
4	2	1	1
5	15	0	1
6	---	0	0
7	---	0	0

iii. Access Page 5: TLB Miss, Page Fault, i.e. both TLB and Page Fault handler executed.

TLB:

Page No.	Frame No.

3	7
5	**4**

Page Table

Page No	Frame No/ Swap Page No	Memory Resident?	Valid?
0	**any swap page**	**0**	1
1	1	1	1
2	1	0	1
3	7	1	1
4	2	1	1
5	**4**	**1**	1
6	---	0	0
7	---	0	0

(Dynamically allocated new page) Major points:
- Page table update (valid bit)
- Swap page allocation? / Frame allocation? / TLB update?

Consider the possible approaches: e.g. whether we should immediately move the new page into RAM and/or update TLB?

c. Can be easily calculated as overhead + actual memory access. The actual memory access is always 30ns.

Overhead in best scenario:
= TLB-Hit
= 1ns

Total = 31ns

Minimal overhead in worst scenario:
= TLB access + Full table access + Service page fault
= 1ns + 30ns + 5ms + 5ms
= 10,000,031ns (10.000,031ms)

Note that the above is a crude approximation for the overhead. The major component is the two disk accesses for swapping in/out the memory page, which dwarfs all other overheads. Also, the above ignore the overhead of re-accessing TLB and/or page table. So, it should be taken as the minimal overhead.

d.
Since the smaller page tables are stored in separate pages. The even worse scenario is when a particular smaller page table is needed, it is not in physical memory, i.e. page fault. Hence, there is a possibility of servicing **two page faults** under the 2-level paging scheme.

2. [Adapted from AY1920S2 Final – Page Table Structure] The Linux machines in the SoC cluster used in the labs have 64-bit processors, but the virtual addresses are 48-bit long (the highest 16 bits are not used). The physical frame size is 4KiB. The system uses a hierarchical direct page table structure, with each table/directory entry occupying 64 bits (8 bytes) regardless of the level in the hierarchy.

Assume Process *P* runs the following simple program on one of the lab machines:

```
1.  #include <stdio.h>
2.  #include <stdlib.h>



3.  #define ONE_GIG 1<<30
4.  #define NUM_PAGES 1<< 18



5.  void main(){
6.    char* array = malloc(ONE_GIG);
7.    int i;



8.    for (i=0; i<NUM_PAGES; i++)
          array[i<<12]='a';
9.    printf("0x%lX\n", array);
10.   // point of interest
11.   free(array);
12. }
```

At line 9, the program prints out the value of variable **`array`** in hexadecimal:

`0x7F9B55226010`

Consider the point in time when the process execution reaches the point of interest (line 10 in the listing). Answer the following questions:

a. If we use a single-level page table, how much memory space is needed just to store the page table for process *P*?
b. How many levels are there in the page-table structure for process *P*?

c. How many **entries in the root page directory** are holding information related to process *P*'s **dynamically allocated data**?

d. How many **physical frames** are holding information related to process *P*'s dynamically allocated data in the **second level** of the hierarchical page-table structure (next after the root)?
e. How many **physical frames** are holding information related to process *P*'s dynamically allocated data in the **penultimate level** of the page-table structure (i.e., the level before the last one).
f. How many **physical frames** are holding information related to process *P*'s dynamically allocated data in the **last level** of the page-table structure?

ANS:

a. The maximum memory space size for *P* is 2^{48} bytes.
Splitting this memory space into 4KiB page gives us $2^{48} / 2^{12} = 2^{36}$ pages
So, a single level page table has 2^{36} PTEs, each at 8 bytes => page table is 2^{36} x 8 bytes = 2^{39} bytes (512 GiB!)
b. If we are keeping the "page directory" at each level to a single page, the branching factor is

2^{12} / 8 bytes = 2^9 = 512

i.e. each directory can points to 512 next level directories. [Note: we assume that a memory pointer is the same size as a PTE for simplicity].

This tell us that we use 9-bit of the virtual address for each level. Since the last 12 bits of the virtual address is used as the page offset. So, we are left with 48-12 = 36 bit of address:

ceiling(36 bit address / 9-bit per level) = 4 levels.

Note that this is considering the entire memory space for process *P*, i.e. not restricted to just the dynamically allocated space used by "**`array`**".

With the above information, we can now look at the address of array closer. The virtual address has 5 components:

Root Level Idx	Second Level Idx	Penultimate Level Idx	Last Level Idx	Page Offset
9 bits	9 bits	9 bits	9 bits	12 bits

Give address of array = **`0x7F9B55226010`**

Root Level Idx									Second Level Idx								Pen.Level Idx								Last Level Idx								Page Offset 12bits											
0	1	1	1	1	1	1	1	1	0	0	1	1	0	1	1	0	1	0	1	0	1	0	1	0	0	0	1	0	0	0	1	0	0	1	1	0	0	0	0	0	0	0	1	0	0	0	0

Component	Effective Value

Page Offset	$0x010 = 16_{10}$
Last Level Idx	$0x26 = 38_{10}$
Pen.Level Idx	$0xA9 = 169_{10}$
Second Level Idx	$0x6D = 109_{10}$
Root Level Idx	$0xFF = 255_{10}$

It makes more sense to discuss (c) to (f) in reverse order.

c. **`Array`** span across $2^{18} + 1$ pages (1 GiB data / 4 KiB page = $2^{30} / 2^{12} = 2^{18}$ pages + 1 due to page misalignment).

   Since we have a branching factor of 2^9 (see (b)), we will need only 1 entry after 2 levels. So there is only 1 entry needed at the root level (highest level).

d. 1 page (only 2 entries required!)

e. 2 pages for 2nd level page table entries
   ceiling(513 last level page table pages / 2^9) = 2 pages

   From the component value table, you can see that the first last level index is at position 38, i.e. the first (512-38) = 474 entries is in the first page and the rest (38+1) entries spill over to the next page.

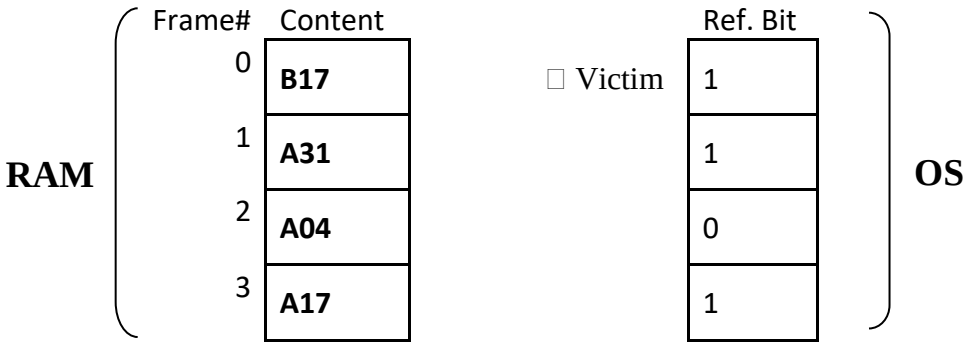
f. It would seems that we have 2^{18} last level PTEs for the **`array`**: allocated 1GiB split into 4KiB pages = $2^{30} / 2^{12} = 2^{18}$ pages. However, the printed address shows that the **`array`** is unfortunately not allocated at the page boundary (start of a page should have 0 at the 12 least significant bits of the virtual address). From the components value table, you can see that the first byte of array is +16 offset from the top of the page.

   This gives us **$2^{18}+1$** pages in total.

   We need $2^{18} / 2^9 + 1 = 2^9 + 1 = 513$ frames to hold $(2^{18} + 1)$ PTE entries.

3. [Adapted from final exam AY2018/19 S1 – Page Replacement]

Below is a snapshot of the memory frames in RAM on a system with virtual memory. The memory pages in the frame is shown as **<Process><Page#>**, e.g. **B17** means Page **17** of Process **B**.



The OS maintains additional information shown on the right to perform **second chance** page replacement algorithm on the memory frames.

a. Suppose **Process A access Page 8** (i.e. A08) now, answer the following:
   i. Which **memory frame** will be replaced?
   ii. Give the steps OS takes to update the affected page table entries.

b. Continuing from (a), if **Process B access Page 13** (i.e B13) now, answer the following:
   i. Which memory frame will be replaced?
   ii. If OS keeps an **inverted page table**, give the content of the inverted page table **after** the replacements (after a. and c.).
   iii. Give the steps OS takes to update the affected page table entries with the help of inverted page table.

ANS:

a. Frame 2 is replaced (first 0 that is encountered in second chance algorithm)

b. The OS must discover that Process A has a page in frame 2, and that page's number. Without an inverted page table, this procedure will be very slow. The OS will then mark A's page table entry for page 4 as invalid, and page 8 will be marked as valid, with the mapped frame number being 2.

   i. Frame 0 is replaced (circular wraparound to the top).
   ii. Frames 0 to 3 mapped to B13, A31, A08, A17 respectively
   iii. The inverted page table immediately shows that Process B has page 17 in frame 0. B's page table entry for 17 will be marked as invalid, and page 13 will be marked as valid.

For your own practice

1. [Page Table Structure] Given the following information:
    - Virtual Memory Address Space = 32 bits
    - Physical memory = 512MB
    - Page Size = 4 KB
    - PTE Size = 4 bytes

   Suppose there are 4 processes, each using 512MB virtual memory. Answer the following:
    a. Direct Paging is used. What is the total space needed for all page tables used?
    b. Two-levels paging is used. What is the total space needed for this scheme?
    c. Inverted table is used. What is the total space needed? You can assume each inverted table entry takes 8 bytes.

ANS:

a.
Number of page table entries = $2^{32} / 2^{12} = 2^{20}$
Size of page table = $2^{20} * 4 = 4MB$
Each process has its own page table, so total is 4MB * 4 = 16MB

b.
The page table will be split into portions (smaller page table), each portion should fit inside a page (4KB). So, each portion can hole 4KB / 4 byte = 1K (2^{10}) PTE entries

Page table is split into $2^{20} / 2^{10} = 2^{10}$ portions
 Page directory has 2^{10} entries.

Assuming the same size of entry is used in page directory, then a page directory is $2^{10} * 4$ =
2^{12} (4KB) in size.

A process uses only 512MB 2^{29} (512MB) / 2^{12} (4KB) = 2^{17} pages
So, $2^{17} / 2^{10} = 2^{7} = 128$ page table portions (smaller page table) is used.

Overhead for a single process = 1 page directory + 128 page table portions
= 4KB + 128 * 1024 * 4
= 516KB
Total overhead = 516 KB * 4

c.
Number of physical frames is 2^{29} (512MB) / $2^{12} = 2^{17}$ frames
A single frame table is used system wide, overhead = $2^{17} * 8$ bytes = 2^{20} = 1MB

Q: Why do you think the inverted table entry takes up more space than a page table entry?