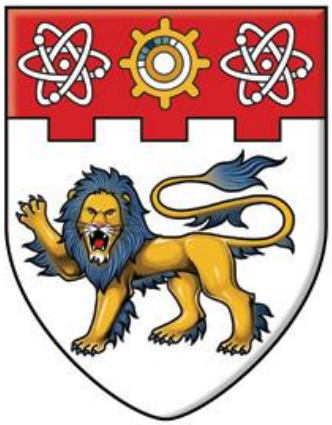

NANYANG TECHNOLOGICAL UNIVERSITY

COURSE CODE	**:**	**CZ2005**
COURSE NAME	**:**	**OPERATING SYSTEMS**
NAME	**:**	**HTET NAING**
MATRICULATION NO.	**:**	**U1620683D**
LAB GROUP	**:**	**SSP4**
COURSEWORK	**:**	**LAB 3 ASSIGNMENT**

Contents

Implementation of code fragments of the following:

Analysis of test program output to verify the solution

Implementation of the function: ***int VpnToPhyPage(int vpn)***

```
//------------------------------------------------------------------------
// VpnToPhyPage
//      Gets a phyPage for a vpn, if exists in ipt.
//------------------------------------------------------------------------

int VpnToPhyPage(int vpn)
{
  //your code here to get a physical frame for page vpn
  //you can refer to PageOutPageIn(int vpn) to see how an entry was created in ipt

  // loop to find the correct physiclal page with pid and vpn
  // pid vpn -> physical frame no
  // else return -1

  for (int i = 0; i < NumPhysPages; i++){
    if ((memoryTable[i].valid) && (memoryTable[i].pid == currentThread->pid) && (memoryTable[i].vPage == vpn)){
        return i;
    }
  }
  return -1;
}
```

Figure 1: Getting a physical page for a virtual page number if it exists in IPT

In the above figure, **memoryTable** refers to the Inverted Page Table (IPT). Since the number of entries in the IPT is identical to the number of physical pages, "**NumPhysPages"** is used as the limit for termination in the for-loop condition.

In each iteration, it searches for a required physical page in IPT. First, it checks if a particular entry in the table is valid. If it is invalid, the conditional check in if statement fails, and it will just continue to check next entry in the table. Otherwise, it proceeds to verify whether the process id which is running on the current thread is equivalent to the process id of the entry. If they are identical, it proceeds to check if the page number, "**vpn**" is equal to the virtual page number in the entry. If all above-mentioned conditions are fulfilled, it means that page hit occurs in IPT and the method will return the index **"i"** for the physical page entry. Otherwise, it will return **"-1"** indicating that page fault occurs in IPT and no entry that matches the above conditions can be found.

Implementation of the function: ***void InsertToTLB(int vpn, int phyPage)***

```
//------------------------------------------------------------------------
// InsertToTLB
//      Put a vpn/phyPage combination into the TLB. If TLB is full, use FIFO
//    replacement
//------------------------------------------------------------------------

void InsertToTLB(int vpn, int phyPage)
{
  int i = 0; //entry in the TLB

  //your code to find an empty in TLB or to replace the oldest entry if TLB is full

  bool thereIsEmpty = false;
  // declare a static FIFO pointer
  static int FIFOPointer = 0;

  // loop to find invalid entry in TLB (using TLBSize)
  for (i = 0; i < TLBSize; i++){
    if (!machine->tlb[i].valid){
        thereIsEmpty = true;
        break;
    }
  }
  if (!thereIsEmpty){
    i = FIFOPointer;
  }

  // if an entry is just inserted, then the entry next to it is the oldest entry
  FIFOPointer = (i + 1) % TLBSize;

  // return an invalid entry or the oldest entry
```

Figure 2: Putting a virtual page and its associated physical page into the TLB with respect to FIFO

The implementation of the method **InsertToTLB** can be seen in the above figure. The method will be called when there is a TLB Miss (i.e. the required entry is not found in the TLB). In each iteration of the for-loop, it searches for an empty slot for a new entry that contains the values for a virtual page and its associated physical page to be put into the TLB.

Firstly, it checks if a particular entry in the table is invalid. If it is invalid, the entry is empty and can be used to store the new entry. Otherwise, it will just continue to check the next entry in the table. However, if the entire TLB table is occupied (i.e. the for-loop has iterated to its completion), the FIFO replacement will be practised to replace the oldest entry with the new entry.

In order to keep record of which entry in the TLB is the oldest, a static variable "**FIFOPointer**" is implemented. If an entry is just inserted, then the entry next to it is the oldest entry. Accordingly, the "**FIFOPointer**" will be set to the index position of the oldest entry. Therefore, in the case where the TLB table is full, the new entry will be inserted at the entry as indicated by the "**FIFOPointer**".

Implementation of the function: ***int lruAlgorithm(void)***

```
//---------------------------------------------------------------------
// lruAlgorithm
//      Determine where a vpn should go in phymem, and therefore what
// should be paged out. This lru algorithm is the one discussed in the
// lectures.
//---------------------------------------------------------------------

int lruAlgorithm(void)
{
  //your code here to find the physical frame that should be freed
  //according to the LRU algorithm.
  int phyPage=0;
  // look for a free frame (in memoryTable)
  // see the lastUsed field
  int last = memoryTable[0].lastUsed;

  for (int i = 0; i < NumPhysPages; i++){
    if (!memoryTable[i].valid){ // free frame
        phyPage = i;
        break;
    }
    if (memoryTable[i].lastUsed < last){
        last = memoryTable[i].lastUsed;
        phyPage = i;
    }
  }

  return phyPage;
}
```

Figure 3: Implementing the Least Recently Used Algorithm that determines which physical page should be paged out

In the above figure, the method "**lruAlgorithm**" will return the **phyPage** that indicates one of the two positions. The first is the index position where the new entry should be inserted in the case where an empty slot is found. The second is the index position where the new entry should replace the old entry that is least recently used in the case where IPT is fully occupied.

At every iteration of the for-loop, it will check if the entry in the memory table is invalid (i.e. empty). Once an empty slot is found, it will break out of the for-loop and return the **phyPage** that stores the corresponding index.

The variable "**last**" will keep track of which entry has the smallest **lastUsed** value. It will be initialised to the **lastUsed** value of the first entry in the IPT. At the completion of all the iterations (i.e. no empty slot is found yet), the **phyPage** is already set to the index position that indicates the

entry with the smallest **LastUsed** value. Eventually, the method will return the above-mentioned **phyPage**.

Experimental result table

Tick	VPN	PID	IPT[0]	IPT[1]	IPT[2]	IPT[3]	TLB[0]	TLB[1]	TLB[2]	PhyPageOut
10	0	0	0,0,0,0	0,0,0,0	0,0,0,0	0,0,0,0	0,0,0	0,0,0	0,0,0	
13	9	0	0,0,12,1	0,0,0,0	0,0,0,0	0,0,0,0	0,0,1	0,0,0	0,0,0	
15	26	0	0,0,12,1	0,9,15,1	0,0,0,0	0,0,0,0	0,0,1	9,1,1	0,0,0	
20	1	0	0,0,12,1	0,9,19,1	0,26,17,1	0,0,0,0	0,0,1	9,1,1	26,2,1	
26	0	0	0,0,12,1	0,9,25,1	0,26,17,1	0,1,22,1	1,3,1	9,1,1	26,2,1	
28	10	0	0,0,28,1	0,9,25,1	0,26,17,1	0,1,22,1	1,3,1	0,0,1	26,2,1	26
41	9	0	0,0,40,1	0,9,25,1	0,10,28,1	0,1,22,1	1,3,1	0,0,1	10,2,1	
42	26	0	0,0,40,1	0,9,42,1	0,10,28,1	0,1,22,1	9,1,1	0,0,1	10,2,1	
47	0	0	0,0,40,1	0,9,46,1	0,10,28,1	0,26,44,1	9,1,1	26,3,1	10,2,1	
59	0	1	0,0,49,1	0,9,46,1	0,10,28,1	0,26,44,1	9,1,0	26,3,0	0,0,0	
62	9	1	0,0,49,1	0,9,46,1	1,0,61,1	0,26,44,1	0,2,1	26,3,0	0,0,0	26
64	26	1	0,0,49,1	0,9,46,1	1,0,61,1	1,9,64,1	0,2,1	9,3,1	0,0,0	
69	1	1	0,0,49,1	1,26,66,1	1,0,61,1	1,9,68,1	0,2,1	9,3,1	26,1,1	
74	0	1	1,1,71,1	1,26,66,1	1,0,61,1	1,9,73,1	1,0,1	9,3,1	26,1,1	
117	0	0	1,1,71,0	1,26,66,0	1,0,76,0	1,9,73,0	1,0,0	0,2,0	26,1,0	
120	9	0	0,0,119,1	1,26,66,0	1,0,76,0	1,9,73,0	0,0,1	0,2,0	26,1,0	
122	10	0	0,0,119,1	0,9,121,1	1,0,76,0	1,9,73,0	0,0,1	9,1,1	26,1,0	
123	26	0	0,0,119,1	0,9,121,1	0,10,123,1	1,9,73,0	0,0,1	9,1,1	10,2,1	
125	0	0	0,0,119,1	0,9,121,1	0,10,124,1	0,26,124,1	26,3,1	9,1,1	10,2,1	

Legends:

IPT entry : "pid,vpn,lastUsed,valid"

TLB entry : "vpn,phy,valid"

Experimental details and screenshot of the summarized output

(a) Page Size : 128 bytes (3456 / 27 = 128)
(b) No. of Physical Frames : 4
(c) TLB Size : 3
(d) Pages Used : 0,1,9,10,26

(e) No. of Pages Used : 5
(f) No. of Page Faults : 14
(g) No. of Page Out : 2
(h) No. of TLB Miss : 19

```
Ticks: total 127, idle 0, system 70, user 57
Disk I/O: reads 0, writes 0
Console I/O: reads 0, writes 0
Paging: faults 14, outs 2, tlb miss: 19
Network I/O: packets received 0, sent 0
```

Figure 4: Output summary

Analysis of the test program output

Initially, all entries in both IPT and TLB are invalid, meaning that they are empty and ready to store any incoming entry.

Tick 10: First, all the valid bits of the entries in both IPT and TLB are set as 0, meaning that they all are invalid/empty. It causes TLB miss in TLB, followed by Page fault in IPT. Therefore, the required frame should be paged in and updated accordingly in both TLB and IPT. Hence, as highlighted in the above experimental result table, the first empty slots,

IPT[0] will be updated with pid = 0 and vpn = 0

TLB[0] will be updated with vpn = 0 and phyPage = 0

Tick 13: The similar incident (as in Tick 10) happens for the virtual page number 9. The program finds out that there are invalid (empty) entries in the second rows of both TLB and IPT tables. As highlighted in the above experimental result table,

IPT[1] will be updated with pid = 0 and vpn = 9

TLB[1] will be updated with vpn = 9 and phyPage = 1

Tick 15: The program carries out the similar task (as completed in Tick 10 and 13). As highlighted in the above table,

IPT[2] will be updated with pid = 0 and vpn = 26

TLB[2] will be updated with vpn = 26 and phyPage = 2

Tick 20: The program checks if the virtual page number 1 is present in the TLB. Since it is not found there, TLB miss occurs. Since the valid bit for IPT[3] is still “0”, the required page has to be

paged in and updated accordingly in both TLB and IPT tables. One important thing to note here is that although the program is able to find the empty slot in IPT which is IPT[3], the TLB table is fully occupied currently. Therefore, when deciding which entry to be replaced, the FIFO replacement policy is practised and thus, since the TLB[0] is the oldest entry in the TLB table, it will be selected to be updated with new values. As highlighted in the above experimental result table,

IPT[3] will be updated with pid = 0 and vpn = 1

TLB[0] will be updated with vpn = 1 and phyPage = 3

Tick 26: The program checks if there is the entry that matches the virtual page number 0 in TLB. However, in **Tick 20**, the entry that has "vpn 0" has just been replaced with the entry that has "vpn 1". So, no match can be found for the virtual page number 0, resulting in TLB miss. The program checks if there is the required entry in IPT. As it turns out, IPT[0] has the required entry with the value "vpn 0". Hence, it will be used to update the TLB with the new values. When searching for which entry to be replaced (as TLB is full), FIFO policy is used and the oldest entry,

TLB[1] will be updated with vpn = 0 and phyPage = 0

Tick 28: The program checks if there is the entry that matches the virtual page number 10 in TLB. Since it cannot be found in both TLB and IPT, TLB miss and page fault occur in TLB and IPT respectively. The required entry has to be paged in. However, since now all the entries in the IPT are occupied, the LRU algorithm is practised and among all the lastUsed values of the entries in the IPT table, **IPT[2]** has the least value. From the output, the page in **IPT[2]** has been modified and hence it is a dirtied page. Thus, it has to be paged out from IPT and replaced with the new entry (that has been paged in). Lastly, according to the FIFO policy, the oldest entry, TLB[2] will also be chosen to be replaced.

IPT[2] will be updated with pid = 0 and vpn = 10

TLB[1] will be updated with vpn = 10 and phyPage = 2

Tick 41: The program checks if there is the virtual page number 9 in the TLB table. Previously, TLB[1] has the entry with the value vpn 9 but it has been replaced with vpn 0 in Tick 26. So, as usual, TLB miss occurs. It proceeds to look for the required entry in the IPT and the match is found in IPT[1]. The following values will be fetched from IPT[1] and update the oldest entry, **TLB[0].**

TLB[0] will be updated with vpn = 9 and phyPage = 1

Tick 42: The program searches if there is an entry with the values that match the virtual page number 26 in TLB. The entry with the "vpn 26" has been replaced and paged out in **Tick 28**. Now when the program is trying to find it in the TLB again, it cannot be found. Consequently, both TLB

miss and Page fault occur. The least recently used entry in the IPT that is **IPT[3]** with the lastUsed value 22, will be replaced. Moreover, the oldest entry in the TLB table, **TLB[1]** will be replaced too.

IPT[3] will be updated with pid = 0 and vpn = 26

TLB[1] will be updated with vpn = 26 and phyPage = 3

Tick 47: The program checks if there is the virtual page number 0 in the TLB and since it cannot be found, TLB miss occurs. It proceeds to check if there is an entry with the vpn 0 in IPT. Then it has found a match in IPT[0]. The required value will be fetched from IPT[0] and the oldest entry is TLB[2]. Therefore,

TLB[2] will be updated with vpn = 0 and phyPage = 0

Tick 49: A "**context switch**" from the main thread to the "userprogram 1" occurs. When the context switch happened, the TLB entries' valid bit were all set to invalid (i.e valid bit will become "0"). The process is known as "TLB Flush". Since TLB does not contain the process id but rather references to virtual and physical page numbers, the above process is necessary in ensuring that the new process that is going to run does not refer to the old virtual-to-physical address translation in TLB used by the previous process.

Tick 59: Now, the new process, "userprogram 1", is running. The entry with the virtual page number 0 is being searched in the TLB. Since no match is found, it leads to TLB miss and subsequently Page fault. Since all TLB entries are now invalid (they can be considered as empty), the first entry TLB[0] will be used to store the new entry. Since IPT[2] has the least lastUsed values among all the entries in IPT, it will be replaced with the new entry that is paged in.

IPT[2] will be updated with pid = 1 and vpn = 0

TLB[0] will be updated with vpn = 0 and phyPage = 2

Tick 62: The userprogram 1 is still running. The program searches for the corresponding entry with the virtual page number 9 in the TLB. Since it cannot be found both in TLB and IPT, it results in both TLB miss and Page fault. Since the IPT table is fully occupied, the IPT[3] with the least lastUsed value (i.e. 46) will be replaced with the new entry. The page in IPT[3] with the virtual page number 26 has been modified and it is considered a dirty page since TLB[1] is using it for the address translation. Hence, paging out for the dirty page in IPT[3] occurs here as well. As for the TLB table, the next empty entry is TLB[1].

IPT[3] will be updated with pid = 1 and vpn = 9

TLB[1] will be updated with vpn = 9 and phyPage = 3

Tick 64: The userprogram 1 is still running. The program looks for the corresponding entry with the virtual page number 26 in the TLB. Since in Tick 62, the entry that had the reference to vpn 26 has been updated with new values, now it cannot be found in the TLB and leads to TLB miss and subsequently page fault in the IPT. The entry TLB[2] is still marked as invalid (i.e. empty), hence it will be used to store the new entry. In IPT, the IPT[1] that has the least lastUsed value will be replaced according to LRU algorithm.

IPT[1] will be updated with pid = 1 and vpn = 26

TLB[2] will be updated with vpn = 26 and phyPage = 1

Tick 69: The userprogram 1 is still running. The program checks if an entry with the virtual page number 1 is in TLB. Since it is missing in both TLB and IPT, it result in TLB miss, followed by Page fault. The required entry is paged in. Since the oldest entry in TLB is TLB[0] and the least recently used entry in IPT is IPT[0],

IPT[0] will be updated with pid = 1 and vpn = 1

TLB[0] will be updated with vpn = 1 and phyPage = 0

Tick 74: The userprogram 1 is still running. The program searches for an entry with the virtual page number 0 in TLB. Since it cannot be found in TLB, TLB miss occurs. Then, it checks if the required entry is in IPT. As IPT[2] has "**vpn 0**" and Page hit occurs, the required entry will be fetched from IPT[2]. After that, the oldest entry in TLB that is:

TLB[1] will be updated with vpn = 0 and phyPage = 2

Tick 86: The userprogram 1 has finished and been put to sleep. After that, a "**context switch**" from the "userprogram 1" to the main thread occurs. When the context switch happened, the program will carry out TLB flushing, a process that is explained in details in **Tick 49**. Hence, all valid bits in TLB have been marked as invalid (i.e. become "0"). One more important thing to note here is that once the context switch is completed, the user program thread #1 is deleted. Consequently, all valid bits for the userprogram 1 (which is "pid 1") in IPT have also been marked as invalid because the program thread #1 is deleted and all references to it in the IPT is considered invalid**.**

Tick 117: Now the main thread is running. Presently, all the valid bits of the entries in both IPT and TLB are set as 0, meaning that they are invalid/empty. It causes TLB miss in TLB, followed by Page fault in IPT. Therefore, the required frame should be paged in and updated accordingly in both TLB and IPT. Hence, as highlighted in the above experimental result table, the first empty slots,

IPT[0] will be updated with pid = 0 and vpn = 0

TLB[0] will be updated with vpn = 0 and phyPage = 0

Tick 120: The similar incident (as in Tick 117) happens for the virtual page number 9. The program finds out that there are invalid (empty) entries in the second rows of both TLB and IPT tables. As highlighted in the above experimental result table,

IPT[1] will be updated with pid = 0 and vpn = 9

TLB[1] will be updated with vpn = 9 and phyPage = 1

Tick 122: The similar incident (as in Tick 117) happens for the virtual page number 10. The program finds out that there are invalid (empty) entries in the third rows of both TLB and IPT tables. As highlighted in the above experimental result table,

IPT[2] will be updated with pid = 0 and vpn = 10

TLB[2] will be updated with vpn = 10 and phyPage = 2

Tick 123: The program checks if the virtual page number 26 is present in the TLB. Since it is not found there, TLB miss occurs. Since the valid bit for IPT[3] is still "0", the required page has to be paged in and updated accordingly in both TLB and IPT tables. One important thing to note here is that although the program is able to find the empty slot in IPT which is IPT[3], the TLB table is fully occupied currently. Therefore, when deciding which entry to be replaced, the FIFO replacement policy is practised and thus, since the TLB[0] is the oldest entry in the TLB table, it will be selected to be updated with new values. As highlighted in the above experimental result table,

IPT[3] will be updated with pid = 0 and vpn = 26

TLB[0] will be updated with vpn = 26 and phyPage = 3

Tick 125: The program checks if there is the entry that matches the virtual page number 0 in TLB. Since no match can be found for the virtual page number 0, it results in TLB miss. The program checks if there is the required entry in IPT. As it turns out, IPT[0] has the required entry with the value "vpn 0" and it is a Page hit. Hence, it will be used to update the TLB with the new values. When searching for which entry to be replaced (as TLB is full), FIFO policy is used and the oldest entry,

TLB[1] will be updated with vpn = 0 and phyPage = 0