NANYANG TECHNOLOGICAL UNIVERSITY
SCHOOL OF COMPUTER SCIENCE AND ENGINEERING

NANYANG
TECHNOLOGICAL
UNIVERSITY

CE2005: Operating Systems

Lab Report

SEP1

Submitted By:

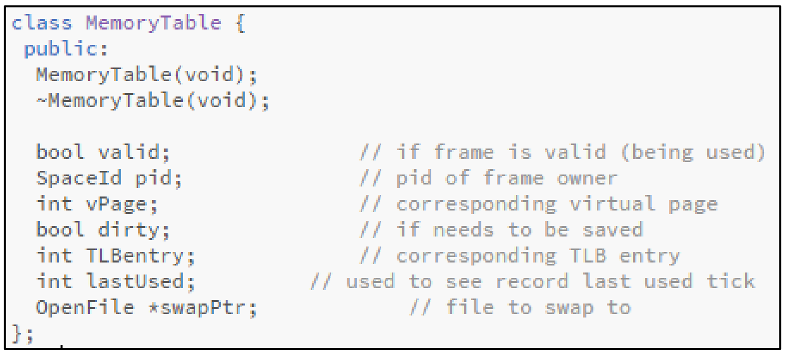
Contents

Introduction

For this lab experiment, we are required to complete a virtual memory implementation. This implementation would include an Inverted Page Table (IPT) and a Translation Look Aside Buffer (TLB).

Inverted Page Table (IPT):

Inverted Page Table is a table that maps physical frames to virtual pages. This helps to eliminate unnecessary storage of individual page tables for each running process.

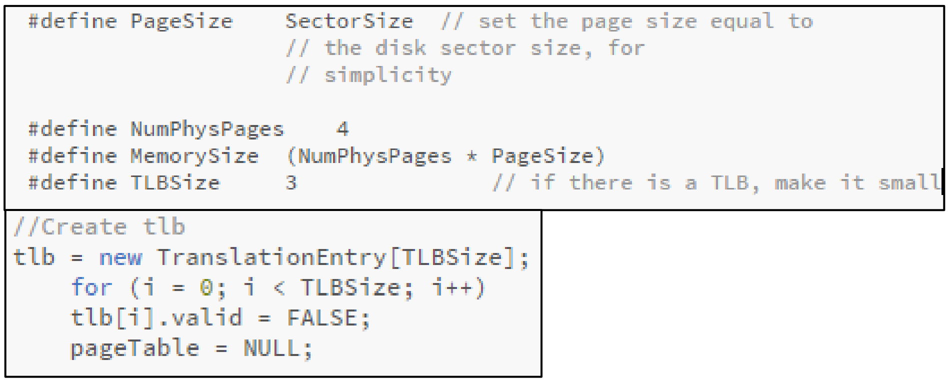


```
class MemoryTable {
 public:
  MemoryTable(void);
  ~MemoryTable(void);

  bool valid;                // if frame is valid (being used)
  SpaceId pid;               // pid of frame owner
  int vPage;                 // corresponding virtual page
  bool dirty;                // if needs to be saved
  int TLBentry;              // corresponding TLB entry
  int lastUsed;          // used to see record last used tick
  OpenFile *swapPtr;             // file to swap to
};
```

Figure 1.1, Code snippet of IPT creation

Translation Look Aside Buffer (TLB):

Translation Look Aside buffer is a type of memory cache. This cache is used to help reduce the amount of time taken to access a memory. It stores the latest translation of the virtual memory to physical memory into a cache. After which, at any level of processing we can always call back the buffer to ask for a translation. A TLB of size 3 is created per machine and the valid bit is initialised to FALSE for all entries.



```
 #define PageSize     SectorSize  // set the page size equal to
                      // the disk sector size, for
                      // simplicity

 #define NumPhysPages    4
 #define MemorySize   (NumPhysPages * PageSize)
 #define TLBSize      3               // if there is a TLB, make it small
```

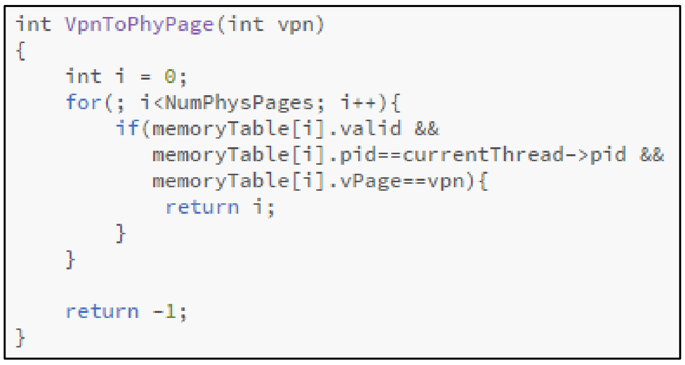
```
//Create tlb
tlb = new TranslationEntry[TLBSize];
    for (i = 0; i < TLBSize; i++)
    tlb[i].valid = FALSE;
    pageTable = NULL;
```

Figure 1.2, Code snippet of TLB creation

Implementation

int VpnToPhyPage (int vpn)

This function takes in a virtual page number in the Inverted Page Table. This function attempts to find the associated physical frame according to the parsed in virtual page number.



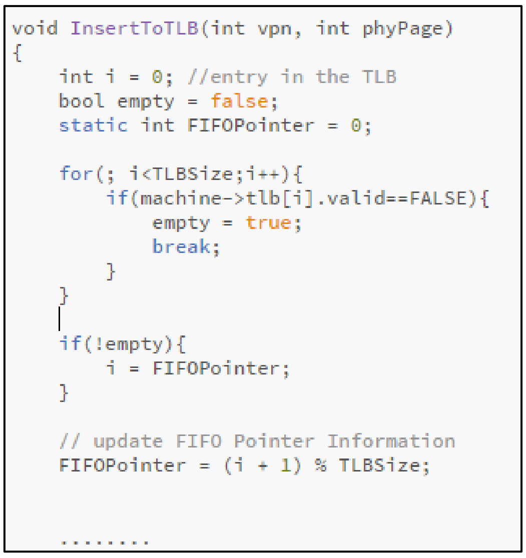
```
int VpnToPhyPage(int vpn)
{
    int i = 0;
    for(; i<NumPhysPages; i++){
        if(memoryTable[i].valid &&
           memoryTable[i].pid==currentThread->pid &&
           memoryTable[i].vPage==vpn){
            return i;
        }
    }

    return -1;
}
```

Figure 1.3, Code snippet for method int VpnToPhyPage (int vpn)

In this function, it loops through the memory table to find the corresponding physical frame to the virtual page. To do so, a check is done at each entry of the memory table to see if the entry is valid, and if both the process id and virtual page number matches. If it matches, the index of the entry in the memory table would be returned (mapping of that virtual page is found). If nothing corresponds in the memory table, this means that no mapping of that virtual page exists and -1 is returned.

void InsertToTLB (int vpn, int phyPage)

This function takes in the virtual page number and a physical page number. The function helps to store the mapping between the virtual page number and physical page number into the Translation Look Aside Buffer (TLB).



```
void InsertToTLB(int vpn, int phyPage)
{
    int i = 0; //entry in the TLB
    bool empty = false;
    static int FIFOPointer = 0;

    for(; i<TLBSize;i++){
        if(machine->tlb[i].valid==FALSE){
            empty = true;
            break;
        }
    }

    if(!empty){
        i = FIFOPointer;
    }

    // update FIFO Pointer Information
    FIFOPointer = (i + 1) % TLBSize;

    ........
```

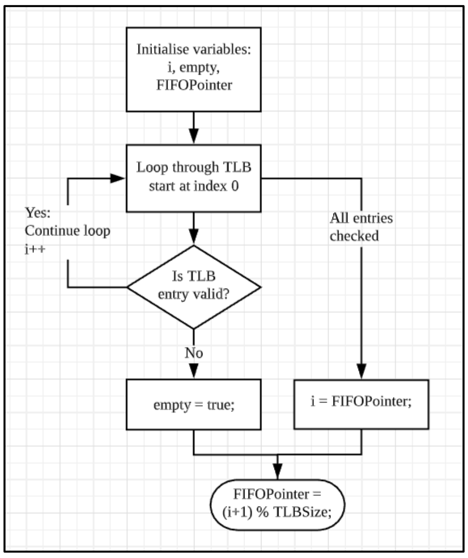

Initialise variables: i, empty, FIFOPointer
Loop through TLB start at index 0
Yes: Continue loop i++
All entries checked
Is TLB entry valid?
No
empty = true;
i = FIFOPointer;
FIFOPointer = (i+1) % TLBSize;

Figure 1.4, Code snippet for method void InsertToTLB (int vpn, int phyPage) and also code flow chart

In this function, we first look through the existing TLB. To do so, a loop starting from index 0 (int ***i***) is used to help search through the TLB. If the entry is valid, it means that the entry has already been allocated to others. If so, it will continue to search through the TLB, until an entry that is not valid is found, which means that an available entry is found. In this case, ***i*** would be pointing to that available entry and a Boolean (bool ***empty***) will be set to TRUE. If all entries in the TLB has been checked and there is no available entry, ***i*** will point to the value of the ***FIFOPointer***. After all the searching, the ***FIFOPointer*** value will be updated to point to the next entry of ***i***, this is to ensure that the ***FIFOPointer*** is always pointing to the oldest entry in the TLB.

```
//update the TLB entry
machine->tlb[i].virtualPage  = vpn;
machine->tlb[i].physicalPage = phyPage;
machine->tlb[i].valid        = TRUE;
machine->tlb[i].readOnly     = FALSE;
machine->tlb[i].use          = FALSE;
machine->tlb[i].dirty        = memoryTable[phyPage].dirty;
```

Figure 1.5, code snippet of TLB entry being updated

Next, the TLB entry at index ***i*** (the first available entry or the oldest unavailable entry) will be updated with the corresponding virtual page number, physical page number and the valid bit will be set to TRUE to indicate that this entry has been allocated.

int lruAlgorithm (void)

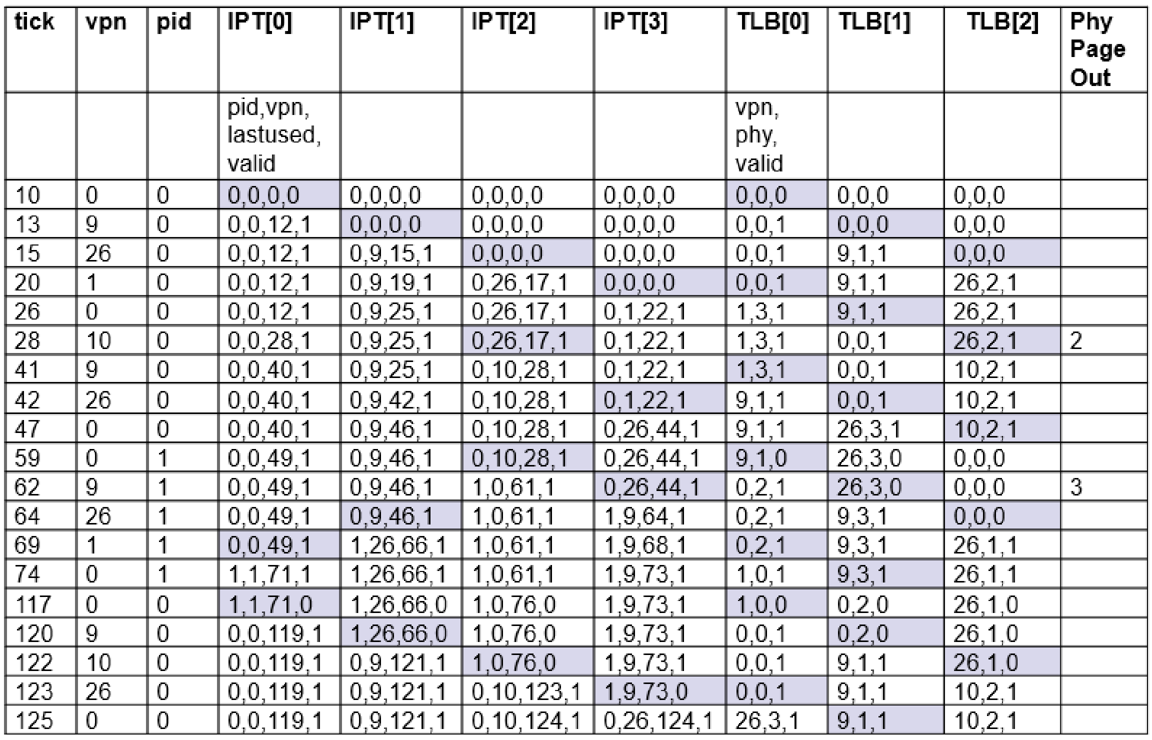
This function is used to determine which physical page should be paged out, in order to do that, Least Recently Used (LRU) algorithm is used.

```
int lruAlgorithm(void)
{
    int leastUsedPage = 0; //Track least used page
    int i=0;
    for(; i<NumPhysPages;i++){
        if(!memoryTable[i].valid){
            return i;
        }
        else if(memoryTable[i].lastUsed < memoryTable[leastUsedPage].lastUsed){
            //If current entry's lastUsed smaller than leastUsedPage, current page is
            lastUsed
            leastUsedPage = i;
        }
    }

    return leastUsedPage;
}
```

Figure 1.6, code snippet of int lruAlgorithm (void)

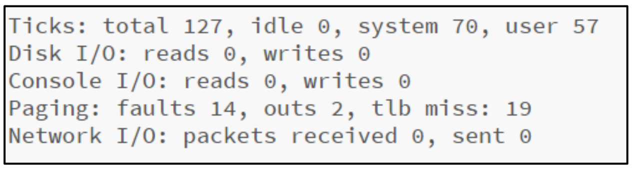
In this function, it loops through the memory table starting an index 0 (int ***i***), to find an invalid entry that can be paged out. At the same time, if the entry is valid, a comparison between the leastUsedPage's lastUsed value (a variable used find the least accessed physical page) and the current entry's lastUsed value is done to see which lastUsed value is smaller. If the current entry's lastUsed value is smaller, it means that the current entry is least accessed and the current leastUsedPage value is updated ***i***.

Analysis



tick	vpn	pid	IPT[0]	IPT[1]	IPT[2]	IPT[3]	TLB[0]	TLB[1]	TLB[2]	Phy Page Out
			pid,vpn, lastused, valid				vpn, phy, valid			
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	2
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	3
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	1,9,73,1	1,0,0	0,2,0	26,1,0	
120	9	0	0,0,119,1	1,26,66,0	1,0,76,0	1,9,73,1	0,0,1	0,2,0	26,1,0	
122	10	0	0,0,119,1	0,9,121,1	1,0,76,0	1,9,73,1	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	

Figure 1.7, result for IPT & TLB experiment



```
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 1.8, summary of events

There was a total of 127 ticks. 19 of which were page misses, 14 of which were page faults, and were 2 page out operations.

<u>Tick 10</u>

Both Inverted Page Table (IPT) and Translation Lookaside Buffer (TLB) have their valid bits set to 0, which means they are available to store mappings. An attempt was made by ***Process 0*** to read the memory. It needs to find the translation for virtual page 0 to physical page 0. However, no such translation was found, hence both page fault and no TLB entry exception occurs.

<u>After tick 10 is completed, the following IPT and TLB values are:</u>

IPT [0]: pid = 0, vpn = 0, lastused = 10, valid = 1

TLB [0]: vpn = 0, phypage = 0, valid =1

<u>Tick 13</u>

The same events occur here as compared to **[Tick 10]**, where by the only difference is that the translation is done for virtual page 9 to physical page 1.

<u>After tick 13 is completed the following IPT and TLB values are:</u>

IPT [1]: pid = 0, vpn = 9, lastused = 13, valid = 1

TLB [1]: vpn = 9, phypage = 1, valid =1

Tick 15

An attempt was made by ***Process 0*** to write the memory. A translation was done for virtual page 26 to physical page 2.

After tick 15 is completed the following IPT and TLB values are:

IPT [2]: pid = 0, vpn = 26, lastused = 15, valid = 1

TLB [2]: vpn = 26, phypage = 2, valid= 1

Tick 20

At tick 20, an attempt was made by ***Process 0*** to read the memory. A translation was attempted for virtual page 1 to physical page 3. However, all entries in the TLB are not valid. The FIFOPointer will help to resolve this issue. The FIFOPointer is always pointing to the oldest entry in the TLB. Hence, the translation is stored in TLB [0].

After tick 20 is completed the following IPT and TLB values are:

IPT [3]: pid = 0, vpn = 1, lastused = 20, valid =1

TLB [0]: vpn = 1, phypage =3, valid = 1

Tick 26

At tick 26, an attempt was made by ***Process 0*** to read the memory. For this instance, the virtual page 0 is found in the IPT table. However, inside the TLB, the translation was not found, therefore a TLB entry exception was thrown. Hence, a translation was inserted into the TLB.

After tick 26 is completed the TLB values are:

TLB [1]: vpn = 0, phypage = 0, valid = 1

Tick 28

At tick 28, an attempt was made by ***Process 0*** to read the memory. It tries to find the translation for virtual page 10 in the TLB. However, the entry was not found in the TLB nor IPT. In this instance, the required entry needs to be paged in, but the IPT table is fully occupied. To resolve this, the LRU algorithm first searches for an invalid entry in the memory table. If there is an invalid entry, the entry would be replaced, otherwise the last used page in the IPT table will be replaced with the new physical page. Additionally, Process 1 is loaded into the ready list for execution.

After tick 28 is completed the following IPT and TLB values are:

IPT [2]: pid = 0, vpn = 10, lastused = 28, valid =1

TLB [2]: vpn = 10, phypage =2, valid = 1

Tick 41

At tick 41, an attempt was made by ***Process 0*** to read the memory. It tries to find the translation for virtual page 9 in the TLB. However, at tick 26, TLB [1] has the entry with the virtual page 9 but was replaced with virtual page 0. As a result, the entry was not found in the TLB, it then proceeds to find it in the IPT. The mapping translation was then found in IPT [1].

After tick 26 is completed the TLB values are:

TLB [0]: vpn = 9, phypage =1, valid = 1

Tick 42

At tick 42, an attempt was made by ***Process 0*** to write the memory. It tries to find the translation for virtual page 26 in the TLB. However, the entry was not found in the TLB nor IPT. In this instance, the required entry needs to be paged in, but the IPT table is now fully occupied. Therefore, finding the least used entry and oldest entry the respective IPT and TLB entries will be updated.

After tick 28 is completed the following IPT and TLB values are:

IPT [3]: pid = 0, vpn = 26, lastused = 42, valid =1

TLB [1]: vpn = 26, phypage = 3, valid = 1

Tick 47

At tick 47, an attempt was made by Process 0 to read the memory. It tries to translate the virtual page 0 to its counterpart. The translation is found in IPT [0].

After tick 47 is completed the following TLB values are:

TLB [2]: vpn = 0, phypage = 0, valid = 1

Tick 49 *

In this tick, the main thread (***Process 0***) is context switched out to userprogram1(***Process 1***). When this occurs, entries inside the TLB will become invalid. This is also known as TLB Flushing. Flushing is done so that the new process doesn't refer to the old virtual to physical translations in TLB.

Tick 59

At this point, ***Process 1*** attempts to read the memory. It tries to find the translation for virtual page 0 inside the TLB. Since all TLB entries are set to invalid, TLB [0] would be used to store the new entry. Using the FIFOPointer, the new IPT entry would be stored in IPT [2].

After tick 59 is completed the following IPT and TLB values are:

IPT [2]: pid = 1, vpn = 0, lastused = 59, valid =1

TLB [0]: vpn = 0, phypage = 2, valid = 1

Tick 62

In this tick, ***Process 1*** attempts to read the memory. It tries to find the translation for virtual page 9. The translation was not found in both the IPT and TLB. Since the IPT table is also full, it attempts to find the leastusedpage. As such, IPT [3] was selected and the leastused page was replaced with the new entry. As IPT [3] with virtual page 26 was modified, the page is to be considered as dirty, therefore paging it out. For the TLB the next empty entry is TLB [1].

After tick 62 is completed the following IPT and TLB values are:

IPT [3]: pid = 1, vpn = 0, lastused = 59, valid =1

TLB [1]: vpn = 0, phypage = 2, valid = 1

Tick 64

At this tick, ***Process 1*** attempts to write the memory. It tries to find the translation for virtual page 26. But previously on Tick 62, the reference for virtual number 26 was overridden. Thus, the translation cannot be found. The same occurs for the search in IPT. As TLB [2] is marked invalid in [Tick 49], it will be used to store the new entry. Using the LRU algorithm, IPT [1] has the leastlastused value, thus this value would be replaced.

After tick 64 is completed the following IPT and TLB values are:

IPT [1]: pid = 1, vpn = 26, lastused = 64, valid =1

TLB [2]: vpn = 26, phypage = 1, valid = 1

<u>Tick 69</u>

At this tick, ***Process 1*** attempts to read the memory. It tries to find the translation for virtual page 1. As translations were not found in both IPT and TLB. The required entry is paged in, as the oldest entry for TLB is TLB [0] and the least used entry for IPT is IPT [0].

<u>After tick 69 is completed the following IPT and TLB values are:</u>

IPT [0]: pid = 1, vpn = 1, lastused = 69, valid =1

TLB [0]: vpn = 1, phypage = 0, valid = 1

<u>Tick 74</u>

In this tick, ***Process 1*** attempts to read the memory. It tries to translate the virtual page 0 to its counterpart. The translation is found in IPT [2].

<u>After tick 74 is completed the following TLB values are:</u>

TLB [1]: vpn = 0, phypage = 2, valid = 1

<u>Tick 76 *</u>

In this tick, the main thread (***Process 0***) is loaded into the ready list for execution.

<u>Tick 86 *</u>

In this tick, the userprogram1(***Process 1***) is deleted and context switched out to the main thread (Process 0). When this occurs, entries inside the TLB will become invalid again (TLB Flushing). Also, all mapping of Process 1 inside the IPT will become invalid as a Process 1 is deleted.

<u>Tick 117</u>

At this point, ***Process 0*** attempts to read the memory. It tries to find the translation for virtual page 0 inside the TLB. Since all IPT and TLB entries are set to invalid, IPT [0] and TLB [0] would be used to store the new entry.

<u>After tick 117 is completed the following IPT and TLB values are:</u>

IPT [0]: pid = 0, vpn = 0, lastused = 117, valid =1

TLB [0]: vpn = 0, phypage = 0, valid = 1

<u>Tick 120</u>

At this point, ***Process 0*** attempts to read the memory. It tries to find the translation for virtual page 9 inside the TLB. However, it is not found in IPT nor TLB, therefore the translation is updated in IPT [1] and TLB [1] respectively.

<u>After tick 120 is completed the following IPT and TLB values are:</u>

IPT [1]: pid = 0, vpn = 9, lastused = 120, valid =1

TLB [1]: vpn = 9, phypage = 1, valid = 1

<u>**Tick 122**</u>

At this point, ***Process 0*** attempts to read the memory. It tries to find the translation for virtual page 10 inside the TLB. However, it is not found in IPT nor TLB, therefore the translation is updated in IPT [2] and TLB [2] respectively.

<u>After tick 122 is completed the following IPT and TLB values are:</u>

IPT [2]: pid = 0, vpn = 10, lastused = 122, valid =1

TLB [2]: vpn = 10, phypage = 2, valid = 1

<u>**Tick 123**</u>

At this point, ***Process 0*** attempts to write the memory. A translation was attempted for virtual page 26. However, all entries in the TLB are not valid. As such, the FIFOPointer will help to resolve this issue. As the FIFOPointer is always pointing to the oldest entry in the TLB. Hence, the translation is stored in TLB [0].

<u>After tick 123 is completed the following IPT and TLB values are:</u>

IPT [3]: pid = 0, vpn = 26, lastused = 123, valid =1

TLB [0]: vpn = 26, phypage =3, valid = 1

<u>**Tick 125**</u>

At this point, ***Process 0*** attempts to write the memory. For this instance, the virtual page 0 is found in the IPT table. However, inside the TLB, the translation was not found, therefore a TLB entry exception was thrown. Hence, a translation was inserted into the TLB.

<u>After tick 125 is completed the following TLB values are:</u>

TLB [1]: vpn = 0, phypage = 0, valid = 1

Conclusion

Through this experiment, we learned how Inverted Page Table and Translation Lookaside Buffer helps to mitigate unnecessary storage of individual page tables and reduce the amount of time taken to access the memory. However, using such implementation comes with its limitations. An example would be page fault occurrence. This limitation is resolved with the LRU algorithm and FIFO algorithm. The LRU algorithm helps to resolve the page fault in IPT. And the FIFO algorithm helps to resolve the page fault in TLB. To conclude, this implementation helps to improve CPU and memory efficiency.