Operating Systems
Lab 3 Assignment/Report

Course Name: Operating Systems

Course Code: CZ2005

Name: Aaron Gangemi

Matriculation Number: N1903373E

Lab Group Number: SS8

Due Date: Tuesday, 29th October 2019

Table of Contents

Implementation of Code Fragments:

VpnToPhyPage:

```
43 int VpnToPhyPage(int vpn)
44 {
45   int phyPage = 0;
46   int i;
47   bool entryFound = false;
48   printf("VPNToPhyPage:\n");
49   for(i = 0; i < NumPhysPages; i++)
50   {
51         if(memoryTable[i].valid && memoryTable[i].pid == currentThread->pid && memoryTable[i].vPage == vpn)
52         {
53                 phyPage = i;
54                 printf("Valid Entry - index: %d\n",i);
55                 entryFound = true;
56                 break;
57         }
58   }
59   if(!entryFound)
60   {
61         phyPage = -1;
62         printf("No valid entry was found\n");
63   }
64   printf("---IPT Table---\n");
65   for(int j = 0; j < NumPhysPages;j++)
66   {
67         printf("IPT[%d]: pid: %d vpn: %d Last Used: %d valid: %d\n",j, memoryTable[j].pid, memoryTable[j].vPage,memoryTable[j].lastUsed,memoryTable[j].valid);
68   }
69   printf("---TLB Table---\n");
70   for(int j = 0; j < TLBSize;j++)
71   {
72         printf("TLB[%d]: vpn: %d phys: %d valid: %d\n",j,machine->tlb[j].virtualPage,machine->tlb[j].physicalPage,machine->tlb[j].valid);
73   }
74   return phyPage;
75   //your code here to get a physical frame for page vpn
76   //you can refer to PageOutPageIn(int vpn) to see how an entry was created in ipt
77 }
78
```

The above code segment is the implementation of the VpnToPhyPage function. The purpose of this function is to take a virtual page number (“vpn”) as a parameter and find the index of the physical page (“phyPage”) for a virtual page number, if the appropriate conditions are met. Initially, a Boolean variable “entryFound” has been set to false. This variable indicates if a physical page entry has been found in the IPT matching the required criteria. The memoryTable in the function refers to the inverted page table (IPT). The size of the IPT is defined by the constant “NumPhysPages”, therefore, “NumPhysPages” has been defined as the limit which will terminate the for loop. The for loop is then used to iterate through the inverted page table and run several checks. The first is to check that the entry in the memoryTable is valid. The algorithm then proceeds to check whether the current threads process id is equal to the process id of the entry in the IPT. After both checks are run, the algorithm moves on to determine whether the virtual page number passed into the function for conversion to a physical page is the same as the virtual page number in the IPT entry. If all 3 conditions are satisfied, the algorithm will enter the IF statement. The variable phyPage will be assigned to the physical page entry’s index in the IPT. The Boolean flag/variable “entryFound” will be set to true to indicate that an entry has been found and the for loop will be exited. However, if a physical page entry has not been found and the variable “entryFound” is still set to false and the index of the physical page will be set to -1 indicating that a page fault has occurred due to the IPT not being able to meet the above requirements. The index of the physical page entry is then returned.

InsertToTLB:

```
79 //----------------------------------------------------------------------
80 // InsertToTLB
81 //      Put a vpn/phyPage combination into the TLB. If TLB is full, use FIFO
82 //    replacement
83 //----------------------------------------------------------------------
84
85 static int FIFOPointer = 0;
86 void InsertToTLB(int vpn, int phyPage)
87 {
88   int i = 0;
89   int j;
90   int physPage;
91   bool found = false;
92   //your code to find an empty in TLB or to replace the oldest entry if TLB is full
93   while(i < TLBSize && !found)
94   {
95         if(!machine->tlb[i].valid)
96         {
97                 physPage = i;
98                 found = true;
99                 break;
100        }
101        i++;
102  }
103  if(!found)
104  {
105        i = FIFOPointer;
106  }
107  FIFOPointer = (i + 1) % TLBSize;
```

The above image is the implementation of the InsertToTLB() method. This method starts by using the while loop to iterate through the TLB, to search for an invalid entry. The new entry consists of the virtual page number and the physical page number, both of which are passed into the method. The method first searches through the TLB table to determine if any entry in the TLB is invalid. If an entry is invalid, then the index is stored, and the Boolean flag “found” is set to true. The new entry will be paged in for the invalid entry. The “found” flag is used to indicate whether an invalid page has been found in the page table. If an invalid entry was not able to be found, then the static variable “FIFOPointer” is assigned to the index. The FIFO implementation is used when the TLB is completely occupied by valid entries. The FIFOPointer points to the oldest entry in the TLB and when the next value is inserted, the entry being pointed to by the FIFOPointer, also known as the oldest entry is paged out for the new entry and the FIFOPointer is updated to the next entry which is deemed as the oldest.

LRU Algorithm:

```
247 //---------------------------------------------------------------------
248 // lruAlgorithm
249 //      Determine where a vpn should go in phymem, and therefore what
250 // should be paged out. This lru algorithm is the one discussed in the
251 // lectures.
252 //---------------------------------------------------------------------
253
254 int lruAlgorithm(void)
255 {
256   //your code here to find the physical frame that should be freed
257   int phyPage = 0;
258   int i = 0;
259   int lastPageUsed;
260   bool invalidPageFound = false;
261   printf("LRU Algorithm\n");
262   while(i < NumPhysPages && !invalidPageFound)
263   {
264         if(!memoryTable[i].valid)
265         {
266                 phyPage = i;
267                 invalidPageFound = true;
268         }
269         i++;
270   }
271   if(!invalidPageFound)
272   {
273         lastPageUsed = memoryTable[0].lastUsed;
274         for(i = 0; i < NumPhysPages;i++)
275         {
276                 if(memoryTable[i].lastUsed < lastPageUsed)
277                 {
278                         lastPageUsed = memoryTable[i].lastUsed;
279                         phyPage = i;
280                 }
281         }
282   }
283   return phyPage;
284 }
```

The above image is the implementation of the LRU Algorithm. The algorithm begins by searching through the IPT for an invalid entry. If an entry is invalid, then the algorithm will assign the phyPage to that index and the algorithm can use that slot to insert the next entry. However, if no invalid entry can be found, then the algorithm searches for the index position which contains the entry which was the least recently used in the IPT. To do this, the algorithm stores the first entries least recently used value. It then iterates through the other entries in the IPT specifically. It retrieves those entries last used values and checks it against the “lastPageUsed” value, which stores the least recently used value. If the next entry’s last used value is seen to be less than the “lastPageUsed” value, then the lastPageUsed is set to the lastUsed value of that entry and the phyPage is assigned to the index where the entry was found.

Explanation of how the program works:

The program contains a Translation Look-aside buffer (TLB) and an Inverted Page Table (IPT). Each table will contain entries of pages. Initially, every entry in both tables are invalid. This means that they are both not occupied, and the necessary pages will need to be paged in. The program will recognize the first few entries as invalid and empty which causes both page faults and TLB misses. Therefore, the starting TLB and IPT entries will be updated. Following this, the program begins by searching through the TLB for a match on the given virtual page number. If a TLB hit is found, then the TLB and IPT will be successfully updated. If a match cannot be found in the TLB, then a TLB miss will occur. At this point, the IPT will be iterated over and checked for the required entry. If the required entry is found in the IPT, then the TLB will be updated based on the fetched values from the IPT hit. However, if the entry is still not found in the IPT, then a page fault occurs, and the entry is paged in. If all entries in the TLB are occupied, the program uses a FIFO implementation to determine which entry should be paged out. The program calculates which entry in the TLB is the oldest and pages out this entry. This is replaced with the paged in entry. If the IPT is completely occupied, then the lruAlgorithm() is run to replace the least recently used page in the IPT.

Lab Results and Values:

As defined in machine.h:

```
35 #define NumPhysPages     4
36 #define MemorySize       (NumPhysPages * PageSize)
37 #define TLBSize          3
```

- Number of Pages used by Test Program = 5 (0,9,26,1,10) -> Shown from VPN Column
- TLB Size = 3
- Number of Physical Frames = 4
- Page Size = 128 bytes
- Number of TLB Miss = 19
- Number of Page Faults = 14
- Number of Page outs = 2 (Tick 28 and Tick 62)

Analysis of Output:

Results Table:

The IPT entry format is: (pid, vpn, Last Used, valid). The TLB entry format is: (vpn, phys, valid)

Tick	VPN	pid	IPT[0]	IPT[1]	IPT[2]	IPT[3]	TLB[0]	TLB[1]	TLB[2]	Page Out
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	

Explanation of TLB and IPT entries:

Tick 10: //Main Thread

Initially, all entries in the TLB and IPT are set to 0. This means that each entry is invalid, and the first page should be paged in due to a TLB miss and a page fault from the IPT. The highlighted entries in the first row demonstrate the column that will be updated. Therefore, in IPT[0], the pid will be set to 0 and the VPN will be set to 0. In TLB[0], the pid will be set to 0 and the entry is set to be valid.

Tick 13:

In Tick 13, the program goes to the next index of the IPT and TLB. The program recognizes that the entry is invalid, depicted by the final 0 of each highlighted cell in the second row. This means that there is a TLB miss and a page fault.

IPT[1]: VPN is updated to be 9, entry is set to valid, pid set to 0.

TLB[1]: VPN is update to 9, physical page paged in is 1, entry is set to valid.

Tick 15:

Like Tick 10 and 13, the program finds that the entire entry of the TLB and IPT is zeroes. Therefore, a page fault and TLB miss occurs because the entries are invalid.

IPT[2]: VPN is set to 26, pid is set to and entry is set to valid.

TLB[2]: VPN is set to 26, physical page paged in is 2, and entry is set to valid.

Tick 20:

In Tick 20, when the inverted page table is iterated over, the program finds that IPT[3]'s entry is all zero's. Therefore, the IPT will be updated like the previous ticks. However, the VPN of Tick 20 is 1. The TLB will cause a TLB miss as 1 is not found in the TLB. Given that the TLB at this point is completely occupied by previous entries, we must use the first-in-first-out implementation which will page out the oldest entry, which as shaded in, is TLB[0].

IPT[3]: VPN is set to 1, pid remains as 0 and entry is set to valid

TLB[0]: VPN changes from 0 to 1, physical page paged in is 3, entry stays valid

Tick 26:

Like tick 20, the program will find that searching for Virtual Page Number 0 will result in a TLB miss because the VPN does not appear in the TLB now. The IPT will then be iterated over and the algorithm will get a hit because the VPN 0 appears in IPT[0]. IPT[0]'s values are fetched and the TLB is updated accordingly.

TLB[1]: VPN changes from 9 to 0, physical page paged out is 1, physical page paged in is 0, entry remains valid.

Tick 28:

In Tick 28, the VPN is 10. The algorithm will search through the TLB and recognize that a TLB miss occurs because the VPN 10 is not in the TLB. It then searches through the IPT and will find there are no matches, so a page fault occurs. Therefore, the FIFO policy will page out the oldest entry which in this case, as highlighted, is TLB[2]. One thing to notice is that, in the IPT of tick 28, all entries are now occupied by previous entries. The lruAlgorithm() will also be applied for the IPT as well for this tick. Hence, IPT[2] being the entry, which is the least recently used, and will be paged out.

TLB[2]: VPN 26 paged out for VPN 10, physical page 2 remains and entry is valid.

IPT[2]: VPN 26 paged out for VPN 10, last Used time updated to 28, entry is valid, pid remains.

Tick 41:

In Tick 41, the algorithm looks through the TLB and won't find a match for VPN 9. This results in a TLB miss. From here, the algorithm then searches through the IPT and finds a match for VPN 9. Therefore, TLB[0] will be updated with the values from IPT[1].

TLB[0]: VPN is updated to 9, physical page mapped to is 1, and entry is valid.

Tick 42:

In Tick 42, the algorithm, like it has done in previous ticks, will search through the TLB and IPT and find that both a page fault and TLB miss occur due to the Virtual Page Number 26 not appearing in either table. Therefore, the least recently used algorithm will page out the oldest entry which is IPT[3] and the FIFO policy will page out TLB[1], as it is the oldest entry.

TLB[1]: VPN 0 now becomes VPN 26, physical page 0 paged out for page 3, entry is still valid.

IPT[3]: VPN 1 now becomes VPN 26, last used time is 44, entry is still valid and pid remains.

Tick 47:

In Tick 47, the algorithm searches through the TLB and finds there are no results for VPN 0. This means there is a TLB miss. The algorithm then searches through the IPT and finds a hit in IPT[0]. This data is then used to update TLB[2] as it is the oldest entry.

TLB[2]: VPN becomes 0, and physical page becomes 0.

Tick 49:

At Tick 49, a context switch occurs from the main thread to the "userprogram1" thread. By doing this, the valid entries in the TLB will be set to invalid. This is represented by the final value of each TLB entry becoming a 0. This will impact future entries as the new process will not refer to previous entries that were updated or inserted in the previous main thread.

Tick 59: //Now in "userprogram1" thread

Although a context switch has occurred and as mentioned in tick 49, the new thread ensures that the old thread does not refer to previous entries. The algorithm will attempt to search for VPN 0 in the TLB, however each entry is invalid so TLB[0] will be paged. In the IPT, each entry is still valid and given that the algorithm will not look at entries which were modified by the other thread, it will not find 0 in the IPT, causing a page fault. Therefore, the least recently used algorithm is used to page out IPT[2].

```
== Tick 49 ==
        interrupts: on -> off
Time: 49, interrupts off
Pending interrupts:
In mapcar, about to invoke 804b454(982c098)
Interrupt handler timer, scheduled at 96
In mapcar, about to invoke 804b454(982c580)
Interrupt handler console read, scheduled at 110
End of pending interrupts
        interrupts: off -> on
Reading VA 0x44, size 4
        Translate 0x44, read: phys addr = 0x44
        value read = 0000000c
At PC = 0x44: SYSCALL
Exception: syscall
SyscallJoin(), initiated by user program main #0.
        interrupts: on -> off
Sleeping thread main #0
Switching from thread main #0 to thread userprogram #1
        interrupts: off -> on
```

TLB[0]: VPN becomes 0, physical page pages in 2 and entry is reset to valid.

IPT[2]: pid is now set to 1 due to context switch, VPN is changed to 0, last used time changed to 61 and entry remains valid.

Tick 62:

In Tick 62, the VPN number 9 is searched for in the TLB. It does not exist so a TLB miss occurs. The algorithm then looks for VPN 9 is the IPT. Although 9 appears to be in the IPT, it is a reference to the previous main thread before the context switch, so the algorithm will result in a page fault. Therefore, it takes the oldest entry in the TLB using the FIFO implementation which will paged out TLB[1]. In the IPT, IPT[3] is invalid so this entry will be paged out using the least recently used algorithm.

TLB[1]: VPN 26 is now VPN 9, physical page 3 remains and the entry now becomes valid.

IPT[3]: The pid changes to 1 because in the "userprogram1" thread and VPN 26 is updated to VPN 9.

Tick 64:

Tick 64 works like the previous tick. It searches for VPN 26 in the TLB and finds that the table is fully occupied, and the VPN does not exist. It also searches through the IPT and recognizes that VPN 26 does not exist, causing both a TLB miss and a page fault to occur. It then uses the lruAlgorithm() to page out the oldest entry in the IPT which is IPT[1]. In the TLB, the final entry is invalid so this entry will be paged out.

IPT[1]: pid changes from 0 to 1 because it is in the "userprogram1" thread, the VPN 9 is updated to VPN 26.

TLB[2]: VPN 0 is updated to VPN 26; physical page is updated to 1 and entry is now valid.

Tick 69:

In Tick 69, the algorithm searches through the TLB for VPN 1 and results in a TLB miss. From here, it searches through the IPT which results in a page fault. The FIFO policy which has been implemented many times throughout this algorithm is used to update TLB[0] because it is the oldest entry. In the IPT, the lruAlgorithm() is in place to find the least recently used entry which is IPT[0].

TLB[0]: VPN 0 is updated to 1, physical page 2 becomes 0, entry remains valid.

IPT[0]: pid is now 1 because it is the "userprogram1" thread1, the VPN is updated to 1 and entry remains valid.

Tick 74:

In Tick 74, the algorithm searches for VPN 0 in the TLB which results in a TLB miss. It then looks through the IPT which get a hit. Therefore, TLB[1] is updated accordingly based on IPT[2]'s values.

TLB[1]: VPN 9 is updated to 0, physical page now becomes 2 and entry is now invalid.

Tick 86: //Context switch to main thread:

```
== Tick 86 ==
        interrupts: on -> off
Time: 86, interrupts off
Pending interrupts:
In mapcar, about to invoke 804b454(9ac7098)
Interrupt handler timer, scheduled at 96
In mapcar, about to invoke 804b454(9ac7580)
Interrupt handler console read, scheduled at 110
End of pending interrupts
        interrupts: off -> on
        interrupts: on -> off
Finishing thread userprogram #1
Sleeping thread userprogram #1
Switching from thread userprogram #1 to thread main #0
Now in thread main #0
Deleting thread userprogram #1
thread main acquires semaphore done
        interrupts: off -> on
```

A context switch back to the main thread occurs as seen in the image to the left. "Userprogram1" is finished and now put to sleep. Any entries identified with the previous thread is now set to invalid. This will be depicted as all entries with pid = 1 will have 0 to identify the entry as invalid.

Tick 117: //Now in the main thread.

Tick 117 is the first entry in the main thread. Due to the thread change, all entries are set to invalid meaning this thread cannot reference entries that have been modified from the previous thread. Therefore, after searching the TLB and IPT, the result will be a TLB miss and a page fault. Given that each entry is invalid, the first entry of both the TLB and IPT will be updated.

IPT[0]: pid is now 0, VPN is also 0 and entry is now valid.

TLB[0]: VPN is updated to 0, physical page is also 0 and entry is valid.

Tick 120:

Tick 120 is like the previous tick. It will look through the TLB and IPT. It will then find that both tables contain invalid entries and update those entries.

IPT[1]: pid is now set to 0, VPN is set to 9, entry is now valid.

TLB[1]: VPN is now 9, physical page is now 1, and entry is valid.

Tick 122:

Tick 122 is like the previous 2 ticks. The algorithm recognizes that after searching for VPN 10, that both the IPT and TLB contain invalid entries. These entries are paged out and updated. The TLB is now completely occupied.

TLB[2]: VPN is updated to 10, physical page is 2 and entry is set to valid.

IPT[2]: pid is set to 0, VPN is updated to 10 and entry is set to valid.

Tick 123:

Tick 123:

Searches for VPN 26 in the completely occupied TLB and does not find a match which causes a TLB miss. It then searches through the IPT for VPN 26 and finds that the IPT still contains an invalid entry due to the context switch. The invalid entry will be updated, and the FIFO implementation will be used on TLB[0] as it is the oldest entry.

TLB[0]: VPN is updated to 1, physical page is updated to 3, entry remains valid.

IPT[3]: pid changes to 0, VPN is now 26, and entry is now valid.

Tick 125:

In Tick 125, it searches for VPN 0 in the TLB and finds that no match exists, so it searches through the IPT and finds a match in IPT[0]. Therefore, the oldest entry TLB[1] is updated based on the fetched values from IPT[0].

TLB[1]: VPN becomes 0. //Values are outside of table.