```
 1 #include "copyright.h"
 2 #include "tlb.h"
 3 #include "syscall.h"
 4 #include "machine.h"
 5 #include "thread.h"
 6 #include "system.h"
 7 #include "utility.h"
 8
 9 //----------------------------------------------------------------------
10 // UpdateTLB
11 //      Called when exception is raised and a page isn't in the TLB.
12 // Figures out what to do (get from IPT, or pageoutpagein) and does it.
13 //----------------------------------------------------------------------
14
15 void UpdateTLB(int possible_badVAddr) {
16     int badVAddr;
17     unsigned int vpn;
18     int phyPage;
19
20     if (possible_badVAddr) // get the bad address from the correct location
21         badVAddr = possible_badVAddr; // fault in kernel
22     else
23         badVAddr = machine->registers[BadVAddrReg]; // fault in userprog
24
25     vpn = (unsigned) badVAddr / PageSize;
26
27     if ((phyPage = VpnToPhyPage(vpn)) != -1)
28         InsertToTLB(vpn, phyPage);
29     else {
30         if (vpn >= currentThread->space->numPages && !GetMmap(vpn))
31             machine->RaiseException(AddressErrorException, badVAddr);
32         else
33             InsertToTLB(vpn, PageOutPageIn(vpn));
34     }
35 }
36
37 //----------------------------------------------------------------------
38 // VpnToPhyPage
39 //      Gets a phyPage for a vpn, if exists in ipt.
40 //----------------------------------------------------------------------
41
42 int VpnToPhyPage(int vpn) {
43     //your code here to get a physical frame for page vpn
44     //you can refer to PageOutPageIn(int vpn) to see how an entry was created in
   ipt
45
46     //Implementing the hash function to retrieve the list of entries
47     IptEntry *iptPtr = hashIPT(vpn, currentThread->pid);
48
49     //while traversing to the next node in the linked list
50     while (iptPtr = iptPtr->next) {
51         //if the node matches, return the physical page
52         if (iptPtr->vPage == vpn && iptPtr->pid == currentThread->pid)
53             return iptPtr->phyPage;
54     }
```

```
 55     return -1;
 56
 57 }
 58
 59 //----------------------------------------------------------------------
 60 // InsertToTLB
 61 //      Put a vpn/phyPage combination into the TLB. If TLB is full, use FIFO
 62 //  replacement
 63 //----------------------------------------------------------------------
 64
 65 void InsertToTLB(int vpn, int phyPage) {
 66     int i = 0; //entry in the TLB
 67
 68     //your code to find an empty in TLB or to replace the oldest entry if TLB is
    full
 69
 70     static int FIFOPointer = 0;
 71
 72     //Traverse through the TLB and proceed if there are invalid entry.
 73     while (i < TLBSize) {
 74         if (!machine->tlb[i].valid) {
 75             break;
 76         }
 77         i++;
 78     }
 79
 80     //After traversing the TLB, if i equals to TLBSize, set i to the entry which
    pointed by the FIFOPointer.
 81     if (i == TLBSize) {
 82         i = FIFOPointer;
 83     }
 84
 85     //Move the FIFOPointer to the next entry.
 86     FIFOPointer = (i + 1) % TLBSize;
 87
 88     // copy dirty data to memoryTable
 89     if (machine->tlb[i].valid) {
 90         memoryTable[machine->tlb[i].physicalPage].dirty = machine->tlb[i].dirty;
 91         memoryTable[machine->tlb[i].physicalPage].TLBentry = -1;
 92     }
 93
 94     //update the TLB entry
 95     machine->tlb[i].virtualPage = vpn;
 96     machine->tlb[i].physicalPage = phyPage;
 97     machine->tlb[i].valid = TRUE;
 98     machine->tlb[i].readOnly = FALSE;
 99     machine->tlb[i].use = FALSE;
100     machine->tlb[i].dirty = memoryTable[phyPage].dirty;
101
102     //update the corresponding memoryTable
103     memoryTable[phyPage].TLBentry = i;
104     DEBUG('p', "The corresponding TLBentry for Page %i in TLB is %i ", vpn, i);
105     //reset clockCounter to 0 since it is being used at this moment.
106     //for the implementation of Clock algorithm.
107     memoryTable[phyPage].clockCounter = 0;
```

```
108 }
109
110 //----------------------------------------------------------------------
111 // PageOutPageIn
112 //      Calls DoPageOut and DoPageIn and handles IPT and memoryTable
113 // bookkeeping. Use clock algorithm to find the replacement page.
114 //----------------------------------------------------------------------
115
116 int PageOutPageIn(int vpn) {
117     int phyPage;
118     IptEntry *iptPtr;
119
120     //increase the number of page faults
121     stats->numPageFaults++;
122     //call the clock algorithm, which returns the freed physical frame
123     phyPage = clockAlgorithm();
124     //Page out the victim page to free the physical frame
125     DoPageOut(phyPage);
126     //Page in the new page to the freed physical frame
127     DoPageIn(vpn, phyPage);
128
129     //make an entry in ipt
130     iptPtr = hashIPT(vpn, currentThread->pid);
131     while (iptPtr->next) iptPtr = iptPtr->next;
132     iptPtr->next = new IptEntry(vpn, phyPage, iptPtr);
133     iptPtr = iptPtr->next;
134
135     //update memoryTable for this frame
136     memoryTable[phyPage].valid = TRUE;
137     memoryTable[phyPage].pid = currentThread->pid;
138     memoryTable[phyPage].vPage = vpn;
139     memoryTable[phyPage].corrIptPtr = iptPtr;
140     memoryTable[phyPage].dirty = FALSE;
141     memoryTable[phyPage].TLBentry = -1;
142     memoryTable[phyPage].clockCounter = 0;
143     memoryTable[phyPage].swapPtr = currentThread->space->swapPtr;
144
145
146     return phyPage;
147 }
148
149 //----------------------------------------------------------------------
150 // DoPageOut
151 //      Actually pages out a phyPage to it's swapfile.
152 //----------------------------------------------------------------------
153
154 void DoPageOut(int phyPage) {
155     MmapEntry *mmapPtr;
156     int numBytesWritten;
157     int mmapBytesToWrite;
158
159     if (memoryTable[phyPage].valid) {           // check if pageOut possible
160         if (memoryTable[phyPage].TLBentry != -1) {
161             memoryTable[phyPage].dirty =
162                     machine->tlb[memoryTable[phyPage].TLBentry].dirty;
```

```
163             machine->tlb[memoryTable[phyPage].TLBentry].valid = FALSE;
164         }
165         if (memoryTable[phyPage].dirty) {        // pageOut is necessary
166             if ((mmapPtr = GetMmap(memoryTable[phyPage].vPage))) { // it's mmaped
167                 DEBUG('p', "mmap paging out: pid %i, phyPage %i, vpn %i\n",
168                       memoryTable[phyPage].pid, phyPage, memoryTable[phyPage].
    vPage);
169                 if (memoryTable[phyPage].vPage == mmapPtr->endPage)
170                     mmapBytesToWrite = mmapPtr->lastPageLength;
171                 else
172                     mmapBytesToWrite = PageSize;
173                 numBytesWritten = mmapPtr->openFile->
174                         WriteAt(machine->mainMemory + phyPage * PageSize,
    mmapBytesToWrite,
175                                 (memoryTable[phyPage].vPage - mmapPtr->beginPage)
     * PageSize);
176                 ASSERT(mmapBytesToWrite == numBytesWritten);
177             } else { // it's not mmaped
178                 DEBUG('p', "paging out: pid %i, phyPage %i, vpn %i\n",
179                       memoryTable[phyPage].pid, phyPage, memoryTable[phyPage].
    vPage);
180                 numBytesWritten = memoryTable[phyPage].swapPtr->
181                         WriteAt(machine->mainMemory + phyPage * PageSize,
    PageSize,
182                                 memoryTable[phyPage].vPage * PageSize);
183                 ASSERT(PageSize == numBytesWritten);
184             }
185         }
186         delete memoryTable[phyPage].corrIptPtr;
187         memoryTable[phyPage].valid = FALSE;
188     }
189 }
190
191 //----------------------------------------------------------------------
192 // DoPageIn
193 //      Actually pages in a phyPage/vpn combo from the swapfile.
194 //----------------------------------------------------------------------
195
196 void DoPageIn(int vpn, int phyPage) {
197     MmapEntry *mmapPtr;
198     int numBytesRead;
199     int mmapBytesToRead;
200
201     if ((mmapPtr = GetMmap(vpn))) { // mmaped file
202         DEBUG('p', "mmap paging in: pid %i, phyPage %i, vpn %i\n",
203               currentThread->pid, phyPage, vpn);
204         if (vpn == mmapPtr->endPage)
205             mmapBytesToRead = mmapPtr->lastPageLength;
206         else
207             mmapBytesToRead = PageSize;
208         numBytesRead =
209                 mmapPtr->openFile->ReadAt(machine->mainMemory + phyPage *
    PageSize,
210                                           mmapBytesToRead,
211                                           (vpn - mmapPtr->beginPage) * PageSize);
```

```
212         ASSERT(numBytesRead == mmapBytesToRead);
213     } else { // not mmaped
214         DEBUG('p', "paging in: pid %i, phyPage %i, vpn %i\n", currentThread->pid,
215               phyPage, vpn);
216         numBytesRead = currentThread->space->swapPtr->ReadAt(machine->mainMemory
    +
217                                                              phyPage * PageSize,
218                                                              PageSize,
219                                                              vpn * PageSize);
220         ASSERT(PageSize == numBytesRead);
221     }
222 }
223
224 //----------------------------------------------------------------------
225 // clockAlgorithm
226 //      Determine where a vpn should go in phymem, and therefore what
227 // should be paged out. This clock algorithm is a variant of the one
228 // discussed in the lectures.
229 //----------------------------------------------------------------------
230
231 int clockAlgorithm(void) {
232     int phyPage;
233
234     //your code here to find the physical frame that should be freed
235     //according to the clock algorithm.
236
237     static int clockPointer = 0;
238
239     while (true) {
240         printf("********************clock\n");
241         if (!memoryTable[clockPointer].valid)
242             break;
243         if (!memoryTable[clockPointer].dirty && memoryTable[clockPointer].c
    lockCounter == OLD_ENOUGH)
244             break;
245         if (memoryTable[clockPointer].dirty && memoryTable[clockPointer].c
    lockCounter == OLD_ENOUGH + DIRTY_ALLOWANCE)
246             break;
247
248         //increase the clockCounter of an entry
249         memoryTable[clockPointer].clockCounter++;
250
251         //proceed to the next entry
252         clockPointer = (clockPointer + 1) % NumPhysPages;
253     }
254     //if any of the above 3 if conditions are met, then physical should be freed
255     phyPage = clockPointer;
256     printf("Phypage = %d ", phyPage);
257
258     //increment the clockPointer
259     clockPointer = (clockPointer + 1) % NumPhysPages;
260
261     return phyPage;
262 }
263
```

```
264 //----------------------------------------------------------------------
265 // GetMmap
266 //      Return an MmapEntry structure corresponding to the vpn.  Returns
267 // 0 if does not exist.
268 //----------------------------------------------------------------------
269
270 MmapEntry *GetMmap(int vpn) {
271     MmapEntry *mmapPtr;
272
273     mmapPtr = currentThread->space->mmapEntries;
274     while (mmapPtr->next) {
275         mmapPtr = mmapPtr->next;
276         if (vpn >= mmapPtr->beginPage && vpn <= mmapPtr->endPage)
277             return mmapPtr;
278     }
279     return 0;
280 }
281
282 //----------------------------------------------------------------------
283 // PageOutMmapSpace
284 //      Pages out stuff being mmaped (or just between beginPage and
285 // endPage.
286 //----------------------------------------------------------------------
287
288 void PageOutMmapSpace(int beginPage, int endPage) {
289     int vpn;
290     int phyPage;
291
292     for (vpn = beginPage; vpn <= endPage; vpn++) {
293         if ((phyPage = VpnToPhyPage(vpn)) == -1)
294             continue;
295         DoPageOut(phyPage);
296     }
297 }
298
```