Memory Manager

Generated by Doxygen 1.8.11

Contents

Chapter 1

Class Index

1.1 Class List

Here are the classes, structs, unions and interfaces with brief descriptions:

Chapter 2

File Index

2.1 File List

Here is a list of all files with brief descriptions:

Chapter 3

Class Documentation

3.1 MemoryManager Class Reference

A memory management unit.

`#include <memory.h>`

Public Member Functions

- MemoryManager ()
  Constructor.
- char ReadMemory (int addr)
  Read a value from memory.
- int TranslateAddress (int addr)
  Translate a virual address (P, d) to a physical address (f, d). Doesn't implement any p.
- void PrintPageTable ()
  Print the page table.
- void PrintTLB ()
  Print the TLB.
- void PrintInversePageTable ()
  Print Inverse page table.
- void PrintAll ()
  Print TLB and Page Table.
- void PrintStats ()
  Print statistics for page faults and hit rate.

3.1.1 Detailed Description

A memory management unit.

Definition at line 251 of file memory.h.

3.1.2 Constructor & Destructor Documentation

3.1.2.1 MemoryManager::MemoryManager ( )

Constructor.

Definition at line 291 of file memory.cpp.

3.1.3 Member Function Documentation

3.1.3.1 void MemoryManager::PrintAll ( )

Print TLB and Page Table.

Definition at line 432 of file memory.cpp.

3.1.3.2 void MemoryManager::PrintInversePageTable ( )

Print Inverse page table.

Definition at line 437 of file memory.cpp.

3.1.3.3 void MemoryManager::PrintPageTable ( )

Print the page table.

Definition at line 424 of file memory.cpp.

3.1.3.4 void MemoryManager::PrintStats ( )

Print statistics for page faults and hit rate.

Definition at line 441 of file memory.cpp.

3.1.3.5 void MemoryManager::PrintTLB ( )

Print the TLB.

Definition at line 428 of file memory.cpp.

3.1.3.6 char MemoryManager::ReadMemory (int *addr*)

Read a value from memory.

Parameters

int	Virtual address to read from.

Return values

char	value from mem[addr]

Definition at line 299 of file memory.cpp.

3.1.3.7 int MemoryManager::TranslateAddress (int *addr*)

Translate a virual address (P, d) to a physical address (f, d). Doesn't implement any p.

Parameters

int	Virtual address to translate.

Definition at line 403 of file memory.cpp.

The documentation for this class was generated from the following files:

- src/memory.h
- src/memory.cpp

3.2 MemoryPairAddress_t Struct Reference

`#include <memory.h>`

Public Attributes

- int P
- int d

3.2.1 Detailed Description

Definition at line 181 of file memory.h.

3.2.2 Member Data Documentation

3.2.2.1 int MemoryPairAddress_t::d

Definition at line 183 of file memory.h.

3.2.2.2 int MemoryPairAddress_t::P

Definition at line 182 of file memory.h.

The documentation for this struct was generated from the following file:

- src/memory.h

3.3 PageTable Class Reference

Page table holding page/frame pairs.

`#include <memory.h>`

Public Member Functions

- PageTable ()

  Constructor for PageTable object.
- int LookupPage (int pagenum)

  Lookup a page number and return the corresponding frame.
- int LookupPage_no_LRU (int pagenum)

  Lookup a page number, but don't update LRU calculations.
- void SetPageToFrame (int pagenum, int framenum)

  Set a page table entry to a given frame.
- bool PageIsValid (int pagenum)

  Determines if a page is loaded into physical memory.
- void PrintPageTable ()

  Print out the page table.
- void PrintInversePageTable ()

  Print the Inverse Page table.
- int GetLRUPage ()

  Get the LRU page.
- void UpdateLRUList (int last_used)

  Update the LRU list.
- void PageOut_table (int pagenum)

  Page out the table.

3.3.1 Detailed Description

Page table holding page/frame pairs.

Definition at line 102 of file memory.h.

3.3.2 Constructor & Destructor Documentation

3.3.2.1 PageTable::PageTable ( )

Constructor for PageTable object.

Definition at line 85 of file memory.cpp.

3.3.3 Member Function Documentation

3.3.3.1 int PageTable::GetLRUPage ( )

Get the LRU page.

Return values

int	The integer value of the LRU page

Definition at line 185 of file memory.cpp.

3.3.3.2 int PageTable::LookupPage (int *pagenum*)

Lookup a page number and return the corresponding frame.

Parameters

int	page

Return values

int	frame

Definition at line 93 of file memory.cpp.

3.3.3.3 int PageTable::LookupPage_no_LRU (int *pagenum*)

Lookup a page number, but don't update LRU calculations.

Parameters

int	Page to Lookup

Return values

int	Frame at

Definition at line 101 of file memory.cpp.

3.3.3.4 bool PageTable::PageIsValid (int *pagenum*)

Determines if a page is loaded into physical memory.

Parameters

int	Page number to check

Return values

bool	True if in memory (hit), False if not (miss)

Definition at line 135 of file memory.cpp.

3.3.3.5 void PageTable::PageOut_table (int *pagenum*)

Page out the table.

Parameters

int	The page to pageout.

Definition at line 118 of file memory.cpp.

3.3.3.6 void PageTable::PrintInversePageTable ( )

Print the Inverse Page table.

Definition at line 157 of file memory.cpp.

3.3.3.7 void PageTable::PrintPageTable ( )

Print out the page table.

Definition at line 141 of file memory.cpp.

3.3.3.8 void PageTable::SetPageToFrame (int *pagenum,* int *framenum*)

Set a page table entry to a given frame.

Definition at line 109 of file memory.cpp.

3.3.3.9 void PageTable::UpdateLRUList (int *last_used*)

Update the LRU list.

Parameters

int	The latest used element

Definition at line 174 of file memory.cpp.

The documentation for this class was generated from the following files:

- src/memory.h
- src/memory.cpp

3.4 PhysicalMemory Class Reference

Imitates a physical memory.

`#include <memory.h>`

Public Member Functions

- PhysicalMemory ()
  Constructor. Initializes memory to zero.
- int FindFirstFrame ()
  Finds the first available frame in the memory.
- char GetMemoryContents (int frame, int offset)
  Gets the byte at position (f, d)
- bool isFull ()
  Returns true/false if the memory is full/empty.
- void PageIn (int frame, char pagein[FRAME_SIZE])
  Pages a page into frame f.
- void PageOut (int frame)
  Page out a frame.

3.4.1 Detailed Description

Imitates a physical memory.

Definition at line 41 of file memory.h.

3.4.2 Constructor & Destructor Documentation

3.4.2.1 PhysicalMemory::PhysicalMemory ( )

Constructor. Initializes memory to zero.

Definition at line 7 of file memory.cpp.

3.4.3 Member Function Documentation

3.4.3.1 int PhysicalMemory::FindFirstFrame ( )

Finds the first available frame in the memory.

Return values

int	Integer position of the first available frame.

Definition at line 27 of file memory.cpp.

3.4.3.2 char PhysicalMemory::GetMemoryContents (int *frame*, int *offset*)

Gets the byte at position (f, d)

Parameters

int	Frame #
int	Offset in bytes

Return values

char	Byte at (f, d)

Definition at line 37 of file memory.cpp.

3.4.3.3 bool PhysicalMemory::isFull ( )

Returns true/false if the memory is full/empty.

Return values

bool	True if memory is full, False otherwise

Definition at line 17 of file memory.cpp.

3.4.3.4 void PhysicalMemory::PageIn (int *frame*, char *pagein[FRAME_SIZE]*)

Pages a page into frame f.

Parameters

int	Frame # to page into
char[FRAME_SIZE]	Contents of the frame

Definition at line 50 of file memory.cpp.

3.4.3.5 void PhysicalMemory::PageOut (int *frame*)

Page out a frame.

Parameters

int	Frame to page out

Definition at line 61 of file memory.cpp.

The documentation for this class was generated from the following files:

- src/memory.h
- src/memory.cpp

3.5 TLBReturnData_t Struct Reference

`#include <memory.h>`

Public Attributes

- int frame
- int entry

3.5.1 Detailed Description

Definition at line 198 of file memory.h.

3.5.2 Member Data Documentation

3.5.2.1 int TLBReturnData_t::entry

Definition at line 200 of file memory.h.

3.5.2.2 int TLBReturnData_t::frame

Definition at line 199 of file memory.h.

The documentation for this struct was generated from the following file:

- src/memory.h

3.6 TranslationLookasideBuffer Class Reference

`#include <memory.h>`

Public Member Functions

- TranslationLookasideBuffer ()

  Constructor for the TLB.
- bool isFull ()

  Determines whether the TLB is full.
- TLBReturnData_t LookupTLBFrame (int pagenum)

  Searches the TLB for the frame.
- int UpdateTLB (int pagenum, int framenum)

  Update the TLB with a new page/frame combination.
- void PrintTLB ()

  Print the TLB.

3.6.1 Detailed Description

Definition at line 203 of file memory.h.

3.6.2 Constructor & Destructor Documentation

3.6.2.1 TranslationLookasideBuffer::TranslationLookasideBuffer ( )

Constructor for the TLB.

Definition at line 198 of file memory.cpp.

3.6.3 Member Function Documentation

3.6.3.1 bool TranslationLookasideBuffer::isFull ( )

Determines whether the TLB is full.

Return values

bool	True/false depending on status of TLB

Definition at line 206 of file memory.cpp.

3.6.3.2 TLBReturnData_t TranslationLookasideBuffer::LookupTLBFrame (int *pagenum*)

Searches the TLB for the frame.

Return values

TLBReturnData_t	Returns frame number, or -1 if a TLB miss

Definition at line 213 of file memory.cpp.

3.6.3.3 void TranslationLookasideBuffer::PrintTLB ( )

Print the TLB.

Definition at line 261 of file memory.cpp.

3.6.3.4 int TranslationLookasideBuffer::UpdateTLB (int *pagenum*, int *framenum*)

Update the TLB with a new page/frame combination.

Parameters

int	Page number to cache
int	Frame number to cache

Return values

int	The index into which page/frame combo was hashed

Definition at line 227 of file memory.cpp.

The documentation for this class was generated from the following files:

- src/memory.h
- src/memory.cpp

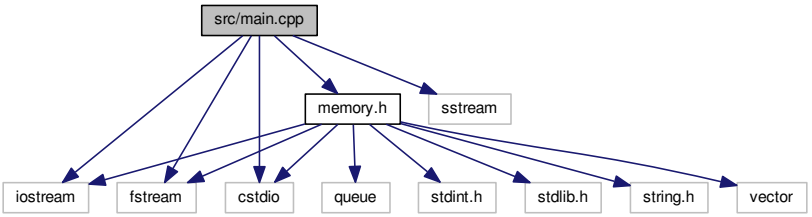
Chapter 4

File Documentation

4.1 src/main.cpp File Reference

```
#include <iostream>
#include <fstream>
#include <sstream>
#include <cstdio>
#include "memory.h"
```

Include dependency graph for main.cpp:



Macros

- #define INPUT_FN "addresses2.txt"

Functions

- void RunPhysicalMemoryTests ()
- void RunAddressConversionTests ()
- void RunPagingTests ()
- void RunFullMemoryTests ()
- void ExecuteFromFile ()
- int main ()

4.1.1 Macro Definition Documentation

4.1.1.1 #define INPUT_FN "addresses2.txt"

Definition at line 8 of file main.cpp.

4.1.2 Function Documentation

4.1.2.1 void ExecuteFromFile ( )

Definition at line 22 of file main.cpp.

4.1.2.2 int main ( )

Definition at line 17 of file main.cpp.

4.1.2.3 void RunAddressConversionTests ( )

Definition at line 92 of file main.cpp.

4.1.2.4 void RunFullMemoryTests ( )

Definition at line 50 of file main.cpp.

4.1.2.5 void RunPagingTests ( )

Definition at line 68 of file main.cpp.

4.1.2.6 void RunPhysicalMemoryTests ( )

Definition at line 102 of file main.cpp.

4.2 main.cpp

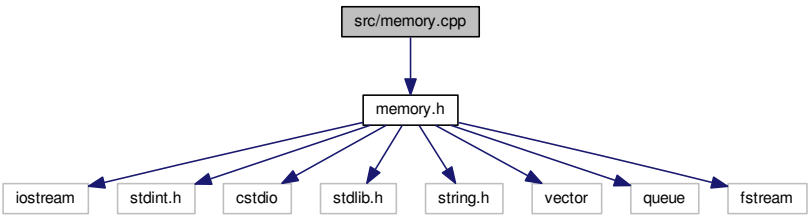
```
00001 #include <iostream>
00002 #include <fstream>
00003 #include <sstream>
00004 #include <cstdio>
00005 #include "memory.h"
00006 using namespace std;
00007
00008 #define INPUT_FN "addresses2.txt"
00009
00010 void RunPhysicalMemoryTests();
00011 void RunAddressConversionTests();
00012 void RunPagingTests();
00013 void RunFullMemoryTests();
00014
00015 void ExecuteFromFile();
00016
00017 int main() {
00018     ExecuteFromFile();
00019     return 0;
00020 }
00021
00022 void ExecuteFromFile() {
00023     MemoryManager mmu;
00024
00025     std::ifstream infile(INPUT_FN);
00026     std::string line;
00027     while(std::getline(infile, line)) {
00028         std::istringstream iss(line);
00029         int addr;
00030
00031         if(!(iss >> addr)){
00032             cout << "I/O ERROR: Couldn't parse " << iss << endl;
00033             exit(EXIT_FAILURE);
00034         }
00035         cout << "Dereferencing address {" << addr << "}..." << endl;
00036         int contents = mmu.ReadMemory(addr);
00037         cout << "... found data [" << contents << "]" << endl << endl;
00038     }
00039
00040     cout << endl << "Contents of TLB:" << endl;
00041     mmu.PrintTLB();
00042     cout << endl << "Contents of Page Table:" << endl;
00043     mmu.PrintPageTable();
00044     cout << endl << "Contents of Page Table (Inverse):" << endl;
00045     mmu.PrintInversePageTable();
00046     cout << endl << "Memory Access Statistics: " << endl << endl;
00047     mmu.PrintStats();
00048 }
00049
00050 void RunFullMemoryTests() {
00051     MemoryManager mmu;
00052     // Fill the memory with the first 8 frames
00053
00054     for(int i = 1; i < 4095-255; i = i + 256) {
00055         mmu.ReadMemory(i);
00056         mmu.PrintAll();
00057     }
00058     mmu.ReadMemory(3555);
00059     mmu.PrintAll();
00060     mmu.ReadMemory(5);
00061     mmu.PrintAll();
00062     mmu.ReadMemory(2050);
00063     mmu.PrintAll();
00064     mmu.ReadMemory(2050);
00065     mmu.PrintAll();
00066 }
00067
00068 void RunPagingTests() {
00069     MemoryManager mmu;
00070     char result;
00071     int addr;
00072
00073     addr = 1;
00074     result = mmu.ReadMemory(addr);
00075     printf("Retrieved {%d} from virtual address {%d}\n\n", result, addr);
00076     // The page table should have frame 0 assigned to page 0
00077     // mmu.PrintPageTable();
00078
00079     addr = 513;
00080     result = mmu.ReadMemory(addr);
00081     printf("Retrieved {%d} from virtual address {%d}\n\n", result, addr);
00082     // mmu.PrintPageTable();
00083
00084     addr = 515;
```

```
00085     result = mmu.ReadMemory(addr);
00086     printf("Retrieved {%d} from virtual address {%d}\n\n", result, addr);
00087     // result = mmu.ReadMemory(515);
00088     mmu.PrintPageTable();
00089
00090 }
00091
00092 void RunAddressConversionTests(){
00093     MemoryPairAddress_t m1 = ConvertAddressFormat(2096);
00094     PrintMemoryPairAddress(m1);
00095
00096     m1 = ConvertAddressFormat(4095);
00097     PrintMemoryPairAddress(m1);
00098
00099     m1 = ConvertAddressFormat(6);
00100     PrintMemoryPairAddress(m1);
00101 }
00102 void RunPhysicalMemoryTests(){
00103     PhysicalMemory pmem;
00104     pmem.GetMemoryContents(0, 100);
00105     pmem.GetMemoryContents(1, 100);
00106     pmem.GetMemoryContents(2, 100);
00107     pmem.GetMemoryContents(3, 100);
00108
00109     pmem.GetMemoryContents(5, 100);
00110     pmem.GetMemoryContents(6, 100);
00111     pmem.GetMemoryContents(7, 100);
00112     pmem.GetMemoryContents(1, 100);
00113     pmem.GetMemoryContents(2, 100);
00114     pmem.GetMemoryContents(3, 100);
00115     pmem.GetMemoryContents(4, 100);
00116
00117     pmem.GetMemoryContents(5, 100);
00118     pmem.GetMemoryContents(6, 100);
00119
00120     // pmem.GetMemoryContents(0, 300);
00121
00122     // printf("Least used: %d\n", pmem.FindLRUFrame());
00123     printf("Full?: %d\n", pmem.isFull());
00124     printf("First available frame: %d\n", pmem.FindFirstFrame());
00125
00126 }
```

4.3 src/memory.cpp File Reference

`#include "memory.h"`

Include dependency graph for memory.cpp:



Functions

- MemoryPairAddress_t ConvertAddressFormat (int addr)

  Convert a base-10 address to (P, d) format.
- void PrintMemoryPairAddress (MemoryPairAddress_t mempair)

4.3.1 Function Documentation

4.3.1.1 MemoryPairAddress_t ConvertAddressFormat (int *addr*)

Convert a base-10 address to (P, d) format.

Parameters

int	base-10 address to translate

Return values

MemoryPair↩ Address_t	(P, d) pair corresponding to the address

Definition at line 457 of file memory.cpp.

4.3.1.2 void PrintMemoryPairAddress (MemoryPairAddress_t *mempair*)

Definition at line 465 of file memory.cpp.

4.4 memory.cpp

```
00001 #include "memory.h"
00002 using namespace std;
00003 /*=======================================
00004 =            Physical Memory            =
00005 =======================================*/
00006
00007 PhysicalMemory::PhysicalMemory() {
00008     for(int i = 0; i < n_frames; i++) {
00009         for(int j = 0; j < frame_size; j++) {
00010             memory[i][j] = 0x00;
00011         }
00012         occupied[i] = 0x00;
00013     }
00014 }
00015
00016 // Determine if the memory is full
00017 bool PhysicalMemory::isFull() {
00018
00019     for(int i = 0; i < n_frames; i++) {
00020         if(occupied[i] == 0) return false;
00021     }
00022     return true;
00023
00024 }
00025
00026 // Find the first available frame
00027 int PhysicalMemory::FindFirstFrame() {
00028     for(int i = 0; i < n_frames; i++) {
00029         if(occupied[i] == 0) {
00030             return i;
00031         }
00032     }
00033     return -1;
00034 }
00035
00036 // Return the contents of memory at a give frame and offset
00037 char PhysicalMemory::GetMemoryContents(int frame, int offset) {
00038     if(frame >= n_frames) {
00039         fprintf(stderr, "%s %d\n", "MEM_ERROR1: invalid frame #: ",frame);
00040         exit(EXIT_FAILURE);
00041     }
```

```
00042     if(offset >= frame_size) {
00043         fprintf(stderr, "%s %d\n", "MEM_ERROR2: invalid offset #: ",offset);
00044         exit(EXIT_FAILURE);
00045     }
00046 
00047     return memory[frame][offset];
00048 }
00049 
00050 void PhysicalMemory::PageIn(int frame, char pagein[
      FRAME_SIZE]) {
00051     if(frame >= N_FRAMES) {
00052         fprintf(stderr, "%s %d\n", "MEM_ERROR3: invalid frame #: ", frame);
00053         exit(EXIT_FAILURE);
00054     }
00055     for(int i = 0; i < FRAME_SIZE; i++) {
00056         memory[frame][i] = pagein[i];
00057     }
00058     occupied[frame] = 1;
00059 }
00060 
00061 void PhysicalMemory::PageOut(int frame) {
00062     if(frame < 0 || frame >= N_FRAMES) {
00063         fprintf(stderr, "MEM_ERROR4: invalid frame # %d\n",frame);
00064         exit(EXIT_FAILURE);
00065     }
00066     for(int i = 0; i < FRAME_SIZE; i++) {
00067         memory[frame][i] = 0x00;
00068     }
00069     occupied[frame] = 0;
00070 }
00071 /*=====  End of Physical Memory  ======*/
00072 
00073 
00074 
00075 
00076 
00077 
00078 
00079 
00080 /*==================================
00081 =            Page Table            =
00082 ==================================*/
00083 
00084 // Page Table Constructor
00085 PageTable::PageTable() {
00086     for(int i = 0; i < pgtable_entries; i++) {
00087         pgtable[i] = -1;
00088         valid[i] = 0;
00089     }
00090 }
00091 
00092 // Lookup a page number and return the corresponding frame
00093 int PageTable::LookupPage(int pagenum) {
00094     if(pagenum >= pgtable_entries) {
00095         fprintf(stderr, "%s %d\n", "PT_ERROR: invalid page #: ", pagenum );
00096         exit(EXIT_FAILURE);
00097     }
00098     UpdateLRUList(pagenum);
00099     return pgtable[pagenum];
00100 }
00101 int PageTable::LookupPage_no_LRU(int pagenum){
00102     if(pagenum >= pgtable_entries) {
00103         fprintf(stderr, "%s %d\n", "PT_ERROR: invalid page #: ", pagenum );
00104         exit(EXIT_FAILURE);
00105     }
00106     return pgtable[pagenum];
00107 }
00108 // Set a value of the page table
00109 void PageTable::SetPageToFrame(int pagenum, int framenum) {
00110     if(pagenum >= pgtable_entries || framenum >= FRAME_SIZE) {
00111         fprintf(stderr, "PT_ERROR: Invalid Page/Frame Combination: Page: %d , Frame: %d\n", pagenum,
      framenum);
00112         exit(EXIT_FAILURE);
00113     }
00114     pgtable[pagenum] = framenum;
00115     valid[pagenum] = 1;
00116 }
00117 
00118 void PageTable::PageOut_table(int pagenum) {
00119     if(pagenum >= pgtable_entries) {
00120         fprintf(stderr, "PT_ERROR: Invalid Page #: %d", pagenum);
00121         exit(EXIT_FAILURE);
00122     }
00123     pgtable[pagenum] = -1;
00124     valid[pagenum] = 0;
00125 
00126     int lrusize = LRU_list.size();
```

```
00127     for(int i = 0; i < lrusize; i++) {
00128         if (LRU_list[i] == pagenum) {
00129             LRU_list.erase(LRU_list.begin() + i);
00130             break;
00131         }
00132     }
00133 }
00134
00135 bool PageTable::PageIsValid(int pagenum) {
00136     if(valid[pagenum] == 1) {
00137         return true;
00138     } else return false;
00139 }
00140
00141 void PageTable::PrintPageTable() {
00142     printf("\n\t\tPAGE TABLE\n");
00143     printf("\t%s  %s  %s\n","[Page #]", "[Frame #]", "[Valid?]");
00144     printf("\t%s","-----------------------------\n");
00145     for(int i = 0; i < PAGE_TABLE_ENTRIES; i++) {
00146         if (pgtable[i] != -1)
00147              printf("\t%d\t     %d\t        %d\n", i, pgtable[i], valid[i]);
00148         else
00149             printf("\t%d\t     %s\t        %d\n", i, "-", valid[i]);
00150     }
00151     printf("\tLRU List (old->new): ");
00152     for(uint32_t i = 0; i < LRU_list.size(); i++) {
00153         printf("{%d}", LRU_list[i]);
00154     }
00155     printf("\n");
00156 }
00157 void PageTable::PrintInversePageTable() {
00158     printf("\n\t\tINVERSE PAGE TABLE\n");
00159     printf("\t%s  %s  %s\n","[Frame #]", "[Page #]", "[Valid?]");
00160     printf("\t%s","-----------------------------\n");
00161     for(int i = 0; i < N_FRAMES; i++) { // i is the frame number
00162         for(int j = 0; j < PAGE_TABLE_ENTRIES; j++) { // j is the page number
00163             if(i == pgtable[j] && valid[j]) {
00164                 if (valid[i] != -1)
00165                      printf("\t%d\t     %d\t        %d\n", i, j, valid[j]);
00166                 else
00167                     printf("\t%d\t     %s\t        %d\n", i, "-", valid[j]);
00168             }
00169         }
00170     }
00171     printf("\n");
00172 }
00173
00174 void PageTable::UpdateLRUList(int last_used) {
00175     int size = LRU_list.size();
00176     for(int i = 0; i < size; i++) {
00177         if(LRU_list.at(i) == last_used){
00178             LRU_list.erase(LRU_list.begin() + i);
00179             break;
00180         }
00181     }
00182     LRU_list.push_back(last_used);
00183 }
00184
00185 int PageTable::GetLRUPage() {
00186     return LRU_list[0];
00187 }
00188
00189 /*=====  End of Page Table  ======*/
00190
00191
00192
00193
00194 /*================================================
00195 =            TranslationLookasideBuffer          =
00196 ================================================*/
00197
00198 TranslationLookasideBuffer::TranslationLookasideBuffer
      () {
00199     for(int i = 0; i < TLB_ENTRIES; i++) {
00200         pagecol[i] = -1;
00201         framecol[i] = -1;
00202         occupied[i] = 0;
00203     }
00204 }
00205
00206 bool TranslationLookasideBuffer::isFull() {
00207     for(int i = 0; i < TLB_ENTRIES; i++) {
00208         if(occupied[i] == 0) return false;
00209     }
00210     return true;
00211 }
00212
```

```
00213 TLBReturnData_t TranslationLookasideBuffer::LookupTLBFrame
      (int pagenum) {
00214     TLBReturnData_t tlbdata;
00215     for(int i = 0; i < TLB_ENTRIES; i++) {
00216         if(pagecol[i] == pagenum) {
00217             tlbdata.frame = framecol[i];
00218             tlbdata.entry = i;
00219             return tlbdata;
00220         }
00221     }
00222     tlbdata.frame = -1;
00223     tlbdata.entry = -1;
00224     return tlbdata;
00225 }
00226
00227 int TranslationLookasideBuffer::UpdateTLB(int pagenum, int framenum) {
00228     bool page_in_tlb = false;
00229     int i;
00230     for(i = 0; i < TLB_ENTRIES; i++) {
00231         page_in_tlb = (pagecol[i] == pagenum);
00232         if(page_in_tlb) break;
00233     }
00234     if(page_in_tlb) {
00235         return i;
00236     } else if(!this->isFull()) {
00237         int next_empty = -1;
00238         for(int i = 0; i < TLB_ENTRIES; i++) {
00239             if (occupied[i] == 0) {
00240                 next_empty = i;
00241                 break;
00242             }
00243         }
00244         pagecol[next_empty] = pagenum;
00245         framecol[next_empty] = framenum;
00246         occupied[next_empty] = 1;
00247         FIFO_tlb.push(next_empty);
00248         return next_empty;
00249     } else if(!page_in_tlb) {
00250         int index_to_replace = FIFO_tlb.front();
00251         FIFO_tlb.pop();
00252         pagecol[index_to_replace] = pagenum;
00253         framecol[index_to_replace] = framenum;
00254         FIFO_tlb.push(index_to_replace);
00255         return index_to_replace;
00256     }
00257     return -1;
00258 }
00259
00260
00261 void TranslationLookasideBuffer::PrintTLB() {
00262     printf("\n\t\t\tTLB\n");
00263     printf("\t%s    %s    %s    %s\n","[TLB #]", "[Page #]","[Frame #]", "[Valid?]");
00264     printf("\t%s","-----------------------------------------\n");
00265     for(int i = 0; i < TLB_ENTRIES; i++) {
00266         if (pagecol[i] != -1)
00267              printf("\t%d\t     %d\t         %d\t        %d\n", i, pagecol[i], framecol[i],occupied[i]);
00268         else
00269             printf("\t%d\t     %s\t         %s\t        %d\n", i, "-", "-",occupied[i]);
00270     }
00271     printf("\tFIFO List (top->bottom): ");
00272     std::queue<int> qcopy = FIFO_tlb;
00273     for(uint32_t i = 0; i < qcopy.size(); i++) {
00274         printf("{%d}", qcopy.front());
00275         qcopy.pop();
00276     }
00277     printf("\n");
00278 }
00279 /*=====  End of TranslationLookasideBuffer  ======*/
00280
00281
00282
00283
00284
00285
00286
00287 /*====================================
00288 =            MemoryManager           =
00289 ====================================*/
00290
00291 MemoryManager::MemoryManager() {
00292     backend_store_filename = (char*) malloc(sizeof(char) * BACKEND_FN_CHARS);
00293     strcpy(backend_store_filename, BACKEND_FN);
00294     total_accesses = 0;
00295     page_faults = 0;
00296     tlb_hitrate = 0;
00297 }
00298
```

```
00299 char MemoryManager::ReadMemory(int addr) {
00300     if (addr > VIRTUAL_ADDRESS_MAX || addr < 0) {
00301         fprintf(stderr, "SEGFAULT at address %d\n", addr);
00302         exit(EXIT_FAILURE);
00303     }
00304     total_accesses += 1;
00305     MemoryPairAddress_t mempair_virtual =
      ConvertAddressFormat(addr);
00306
00307     TLBReturnData_t frame_from_tlb = tlb.LookupTLBFrame(mempair_virtual.
      P);
00308
00309     if(frame_from_tlb.frame != -1 ){
00310         // If the frame was found in the TLB
00311         //      1. Get the contents in memory
00312         //      2. Update the page table LRU
00313         tlb_hitrate += 1;
00314         printf(" ---> Virtual Address {%d} contained in page {%d}, frame {%d} found at TLB Entry {%d}\n",
      addr, mempair_virtual.P, frame_from_tlb.entry, frame_from_tlb.frame);
00315         page_table.UpdateLRUList(mempair_virtual.P);
00316         char contents = physical_memory.GetMemoryContents(frame_from_tlb.frame, mempair_virtual.
      d);
00317         return contents;
00318
00319     } else if (page_table.PageIsValid(mempair_virtual.P)) {
00320         printf(" ---> Virtual address {%d} contained in page {%d} is not in the TLB\n",addr,
      mempair_virtual.P);
00321         // If the page is valid:
00322         //      1. Lookup the frame in the page table
00323         //      2. Access the memory at (frame, d)
00324         //      3. Update the TLB
00325
00326         int frame = page_table.LookupPage(mempair_virtual.P);
00327         char contents = physical_memory.GetMemoryContents(frame, mempair_virtual.
      d);
00328         printf(" ---> Virtual address {%d} is contained in page {%d}, frame {%d}\n", addr, mempair_virtual.
      P, frame);
00329         int tlbval = tlb.UpdateTLB(mempair_virtual.P, frame);
00330         printf(" ---> TLB now has page {%d}, frame {%d} at index {%d}\n", mempair_virtual.
      P, frame, tlbval);
00331         return contents;
00332     } else {
00333         printf(" ---> Virtual address {%d} contained in page {%d} is not in the TLB\n",addr,
      mempair_virtual.P);
00334         printf(" ---> Virtual address {%d} contained in page {%d} causes a page fault\n",addr,
      mempair_virtual.P);
00335         page_faults += 1;
00336         if(physical_memory.isFull()) {
00337             printf(" ---> Physical Memory Full! Taking corrective action...\n");
00338             // If the page is invalid and the memory is full:
00339             //      1. Load the page from the memory file
00340             //      2. Find the LRU Page and extract its frame
00341             //      3. PageOut() the page
00342             //      4. PageIn() the desired page in the right frame
00343             //      5. Update the page table and LRU list
00344             //      6. Update the TLB
00345             // (1) Load page from memory
00346
00347             char* page_to_load = new char[PAGE_SIZE];
00348             FileSeek(mempair_virtual.P, page_to_load);
00349
00350             // (2) find the LRU page
00351             int lru_page = page_table.GetLRUPage();
00352             int target_frame = page_table.LookupPage_no_LRU(lru_page);
00353
00354             // (3) Pageout() the corresponding frame
00355             physical_memory.PageOut(target_frame);
00356             page_table.PageOut_table(lru_page);
00357             printf(" ---> Paging out LRU page {%d}\n", lru_page);
00358
00359             // (4) PageIn() the desired frame
00360             physical_memory.PageIn(target_frame, page_to_load);
00361             // (5) update the page table
00362             page_table.SetPageToFrame(mempair_virtual.P, target_frame);
00363             printf(" ---> Paging in page {%d} to frame {%d}\n", mempair_virtual.
      P, target_frame);
00364             page_table.UpdateLRUList(mempair_virtual.P);
00365             int tlbval = tlb.UpdateTLB(mempair_virtual.P, target_frame);
00366             printf(" ---> TLB now has page {%d}, frame {%d} at index {%d}\n", mempair_virtual.
      P, target_frame, tlbval);
00367             delete[] page_to_load;
00368             return physical_memory.GetMemoryContents(target_frame, mempair_virtual.
      d);
00369         } else {
00370             // If the page is invalid and the memory isn't full:
00371             //      1. Load the page from the memory file
00372             //      2. Find the first avaialable frame
```

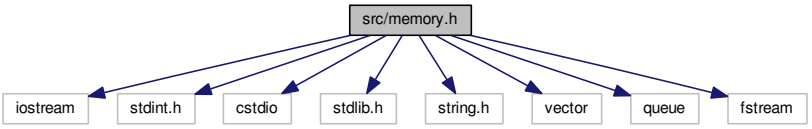
```
00373             //      3. PageIn() the data
00374             //      4. Update the page table for this page with the frame found in (2)
00375             //      5. Update the TLB
00376 
00377             // (1): Load page from memory
00378             char* page_to_load = new char[PAGE_SIZE];
00379             FileSeek(mempair_virtual.P, page_to_load);
00380 
00381             // (2): Find the first available frame
00382             int available_frame = physical_memory.FindFirstFrame();
00383 
00384             // (3): PageIn the data
00385             physical_memory.PageIn(available_frame, page_to_load);
00386             printf(" ---> Page {%d} paged into frame {%d}\n",mempair_virtual.P, available_frame);
00387 
00388             // (4): Update the page table
00389             page_table.SetPageToFrame(mempair_virtual.P, available_frame);
00390             page_table.UpdateLRUList(mempair_virtual.P);
00391             int tlbval = tlb.UpdateTLB(mempair_virtual.P, available_frame);
00392             printf(" ---> TLB now has page {%d}, frame {%d} at index {%d}\n", mempair_virtual.
      P, available_frame, tlbval);
00393             delete[] page_to_load;
00394             return physical_memory.GetMemoryContents(available_frame, mempair_virtual.
      d);
00395 
00396         }
00397     }
00398     cout << "Memory Manager control flow failed (?!)" << endl;
00399     exit(EXIT_FAILURE);
00400     return 0xAA;
00401 }
00402 
00403 int MemoryManager::TranslateAddress(int addr) {
00404     return 0;
00405 }
00406 
00407 void MemoryManager::FileSeek(int fpage, char* dest) {
00408     ifstream infs;
00409     // uint32_t buffer[PAGE_SIZE];
00410 
00411     infs.open(BACKEND_FN, ios::binary);
00412     if(infs.is_open()) {
00413         infs.seekg(fpage*PAGE_SIZE*4); // 4 bytes/uint32_t
00414         // infs.read((char*) buffer, PAGE_SIZE*4);
00415         for(int i = 0; i < PAGE_SIZE; i++) {
00416             infs.read(dest+i, 1);
00417         }
00418         infs.close();
00419     } else {
00420         fprintf(stderr, "%s %s\n", "Couldn't open ", BACKEND_FN);
00421     }
00422 }
00423 
00424 void MemoryManager::PrintPageTable() {
00425     page_table.PrintPageTable();
00426 }
00427 
00428 void MemoryManager::PrintTLB() {
00429     tlb.PrintTLB();
00430 }
00431 
00432 void MemoryManager::PrintAll() {
00433     this->PrintTLB();
00434     this->PrintPageTable();
00435 }
00436 
00437 void MemoryManager::PrintInversePageTable(){
00438     page_table.PrintInversePageTable();
00439 }
00440 
00441 void MemoryManager::PrintStats() {
00442     cout << "\t" << "Page Faults/Accesses: \t" << page_faults << "/" << total_accesses << endl;
00443     cout << "\t" << "TLB Hits/Accesses: \t\t" << tlb_hitrate << "/" << total_accesses << endl;
00444 }
00445 /*======  End of MemoryManager  ======*/
00446 
00447 
00448 
00449 
00450 
00451 
00452 /*=======================================
00453 =            Helper Functions           =
00454 =======================================*/
00455 
00456 
00457 MemoryPairAddress_t ConvertAddressFormat(int addr) {
```

```
00458     MemoryPairAddress_t mempair;
00459     uint16_t addr_16 = (uint16_t) addr;
00460     mempair.d = addr_16 & 0x00FF;
00461     mempair.P = (addr_16 & 0x0F00) >> 8;
00462     return mempair;
00463 }
00464
00465 void PrintMemoryPairAddress(MemoryPairAddress_t mempair) {
00466     printf("(%d,%d)\n", mempair.P, mempair.d);
00467 }
00468
00469 /*======  End of Helper Functions  ======*/
```

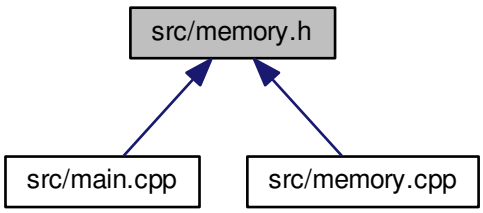
4.5 src/memory.h File Reference

```
#include <iostream>
#include <stdint.h>
#include <cstdio>
#include <stdlib.h>
#include <string.h>
#include <vector>
#include <queue>
#include <fstream>
```

Include dependency graph for memory.h:



This graph shows which files directly or indirectly include this file:



Classes

- class PhysicalMemory

  Imitates a physical memory.
- class PageTable

Page table holding page/frame pairs.

- struct MemoryPairAddress_t
- struct TLBReturnData_t
- class TranslationLookasideBuffer
- class MemoryManager

  A memory management unit.

Macros

- #define ENABLE_LRU
- #define FRAME_SIZE 256
- #define PAGE_SIZE 256
- #define N_FRAMES 8
- #define PAGE_TABLE_ENTRIES 16
- #define BACKEND_FN "BACKING_STORE.bin"
- #define BACKEND_FN_CHARS 18
- #define VIRTUAL_ADDRESS_MAX 4095
- #define TLB_ENTRIES 4

Functions

- MemoryPairAddress_t ConvertAddressFormat (int addr)

  Convert a base-10 address to (P, d) format.
- void PrintMemoryPairAddress (MemoryPairAddress_t mempair)

4.5.1 Macro Definition Documentation

4.5.1.1 #define BACKEND_FN "BACKING_STORE.bin"

Definition at line 21 of file memory.h.

4.5.1.2 #define BACKEND_FN_CHARS 18

Definition at line 22 of file memory.h.

4.5.1.3 #define ENABLE_LRU

Definition at line 14 of file memory.h.

4.5.1.4 #define FRAME_SIZE 256

Definition at line 15 of file memory.h.

4.5.1.5 #define N_FRAMES 8

Definition at line 17 of file memory.h.

4.5.1.6 #define PAGE_SIZE 256

Definition at line 16 of file memory.h.

4.5.1.7 #define PAGE_TABLE_ENTRIES 16

Definition at line 19 of file memory.h.

4.5.1.8 #define TLB_ENTRIES 4

Definition at line 26 of file memory.h.

4.5.1.9 #define VIRTUAL_ADDRESS_MAX 4095

Definition at line 24 of file memory.h.

4.5.2 Function Documentation

4.5.2.1 MemoryPairAddress_t ConvertAddressFormat (int *addr*)

Convert a base-10 address to (P, d) format.

Parameters

int	base-10 address to translate

Return values

MemoryPair↩ Address_t	(P, d) pair corresponding to the address

Definition at line 457 of file memory.cpp.

4.5.2.2 void PrintMemoryPairAddress (MemoryPairAddress_t *mempair*)

Definition at line 465 of file memory.cpp.

4.6 memory.h

```
00001 #ifndef __MEMORY_H_
00002 #define __MEMORY_H_
00003
00004 #include <iostream>
00005 #include <stdint.h>
```

```
00006 #include <cstdio>
00007 #include <stdlib.h>
00008 #include <string.h>
00009 #include <vector>
00010 #include <queue>
00011 #include <fstream>
00012 using namespace std;
00013
00014 #define ENABLE_LRU
00015 #define FRAME_SIZE 256
00016 #define PAGE_SIZE 256
00017 #define N_FRAMES 8
00018
00019 #define PAGE_TABLE_ENTRIES 16
00020
00021 #define BACKEND_FN "BACKING_STORE.bin"
00022 #define BACKEND_FN_CHARS 18
00023
00024 #define VIRTUAL_ADDRESS_MAX 4095
00025
00026 #define TLB_ENTRIES 4
00027
00028
00029 // Forward Declarations
00030
00031 class PhysicalMemory;
00032 class PageTable;
00033 class TranslationLookasideBuffer;
00034 class MemoryManager;
00035
00036
00041 class PhysicalMemory {
00042
00043 public:
00047     PhysicalMemory();
00048
00053     int FindFirstFrame();
00054
00062     char GetMemoryContents(int frame, int offset);
00063
00068     bool isFull();
00069
00076     void PageIn(int frame, char pagein[FRAME_SIZE]);
00077
00083     void PageOut(int frame);
00084
00085 private:
00086     static const int n_frames = N_FRAMES;
00087     static const int frame_size = FRAME_SIZE;
00088
00089     char memory[n_frames][frame_size];
00090
00094     char occupied[n_frames];
00095
00096 };
00097
00102 class PageTable {
00103
00104 public:
00105
00109     PageTable();
00110
00116     int LookupPage(int pagenum);
00117
00123     int LookupPage_no_LRU(int pagenum);
00127     void SetPageToFrame(int pagenum, int framenum);
00128
00134     bool PageIsValid(int pagenum);
00135
00139     void PrintPageTable();
00140
00144     void PrintInversePageTable();
00145
00151     int GetLRUPage();
00152
00157     void UpdateLRUList(int last_used);
00158
00165     void PageOut_table(int pagenum);
00166
00167 private:
00168     static const int pgtable_entries = PAGE_TABLE_ENTRIES;
00172     int pgtable[PAGE_TABLE_ENTRIES];
00173     int valid[PAGE_TABLE_ENTRIES];
00174     std::vector<int> LRU_list;
00175 };
00176
00181 typedef struct {
```

```
00182     int P;
00183     int d;
00184 } MemoryPairAddress_t;
00185
00191 MemoryPairAddress_t ConvertAddressFormat(int addr);
00192 void PrintMemoryPairAddress(MemoryPairAddress_t mempair);
00193 /*
00194     @class TranslationLookasideBuffer
00195     @brief A TLB used as a cache for memory
00196 */
00197
00198 typedef struct {
00199     int frame;
00200     int entry;
00201 } TLBReturnData_t;
00202
00203 class TranslationLookasideBuffer {
00204
00205 public:
00209     TranslationLookasideBuffer();
00210
00215     bool isFull();
00216
00221     TLBReturnData_t LookupTLBFrame(int pagenum);
00222
00230     int UpdateTLB(int pagenum, int framenum);
00231
00235     void PrintTLB();
00236 private:
00237     int pagecol[TLB_ENTRIES];
00238     int framecol[TLB_ENTRIES];
00239     int occupied[TLB_ENTRIES];
00240
00244     std::queue<int> FIFO_tlb;
00245 };
00246
00251 class MemoryManager {
00252 public:
00257     MemoryManager();
00258
00265     char ReadMemory(int addr);
00266
00273     int TranslateAddress(int addr);
00274
00278     void PrintPageTable();
00279
00283     void PrintTLB();
00284
00288     void PrintInversePageTable();
00289
00293     void PrintAll();
00294
00298     void PrintStats();
00299 private:
00300     char* backend_store_filename;
00301
00302     PageTable page_table;
00303     PhysicalMemory physical_memory;
00304     TranslationLookasideBuffer tlb;
00305
00306     uint32_t total_accesses;
00307     uint32_t page_faults;
00308     uint32_t tlb_hitrate;
00309
00317     void FileSeek(int fpage, char* dest);
00318 };
00319
00320
00321 #endif
```

Index