Instructions : Clone your GitHub Classroom repository for this assignment. Follow all steps below to complete the programming assignment portion of Homework 2. Push your changes to GitHub and check that all tests are passing in Actions.

After completing each portion of the assignment, compile your code with the command *sh compile.sh* and check correctness with *make test* from within the *build* directory. Note, if your local computer uses an alternative to *make* you will need to compile manually. **Do not change compile.sh as this could cause GitHub Actions to fail.**

If you get confused with GitHub/CMake logistics, refer back to Homework 0 here.

1 Virtual Memory

The goal of this portion of the programming assignment is to convert a virtual address to the corresponding physical address. You will accomplish this with five helper methods, all within the file `translation.cpp`. Helper classes and functions are provided for you in the files `src.hpp` and `src.cpp` within the submodule folder. You will need to use these provided classes/methods to complete this assignment. Brief descriptions of each are provided below:

Listing 1: TLB* tlb

```
int tlb->k; // dimension of the k-way associated cache
bool tlb->lookup(int index, int tag, PTE** entry);
void tlb->add_entry(int index, int tag, PTE* entry);

bool lookup(int idx, int tag, PTE** entry)
{
    // If an entry is located at `idx` and `tag`:
    //     *entry holds the located PTE* entry
    //     return true
    // Otherwise:
    //     *entry = NULL
    //     return false
}

void add_entry(int index, int tag, PTE* entry)
{
    // Will add given page table entry to the TLB at the given index, tag
}
```

Listing 2: PageTable* table

```
table->lookup(int VPN, PTE** entry);

void lookup(int VPN, PTE** entry)
{
    // Returns entry at index VPN
}
```

Listing 3: PTE* entry

```
1  int entry->PFN; //the entry's physical frame number
2  int entry->valid_bit; //1 if entry is valid within page table, 0 otherwise
3  int entry->protect_bit; //1 if page is protected, 0 otherwise
4  int entry->present_bit; //1 if page is present in main memory, 0 otherwise
```

Listing 4: Miscellaneous Methods

```
1  segmentation_fault(); //Throws a segmentation fault exception
2  protection_fault(); //Throws a protection fault exception
3  page_fault(); //Throws a page fault exception
4  tlb_miss(); //Throws a TLB miss exception
```

Listing 5: Defined Variables

```
1  #define BITS_PER_BYTE //Number of bits per byte
2  #define TLB_MISS //String thrown in tlb_miss() function
```

1. **Finding VPN and Offset:** Update the method `split_virtual_address(···)`. This method is passed the following parameters:

   ```
   int virtual_address; //the virtual address of the instruction
   int page_size; //the number of bytes per page
   int* VPN; //pointer to which method should return VPN
   int* offset; //pointer to which method should return offset
   ```

   Given the virtual address and page size, return the VPN and offset of the virtual address. Assume the number of bits within a virtual address is:

   ```
   sizeof(int) * BITS_PER_BYTE
   ```

2. **Finding TLB tag and index:** Update the method `split_VPN(···)`. This method is passed the following parameters:

   ```
   int VPN; //virtual page number
   int k; //number of sets in the TLB (k-way associative)
   int* index; //pointer to which method should return TLB index
   int* tag; //pointer to which method should return TLB tag
   ```

   Given the VPN and value of k, return the TLB index and tag.

3. **Checking TLB for PTE:** Update the method `TLB_lookup(···)`. This method is passed the following parameters:

   ```
   TLB* tlb; //pointer to a k-way associative TLB cache object
   int VPN; //virtual page number
   ```

   Call `tlb->lookup(···)` (defined in Listing 1) to search the TLB for the given VPN. If the VPN is not in the TLB, throw a TLB miss exception (Listing 4). If a table entry exists in the TLB but it cannot be accessed, throw a protection fault (Listing 4). Otherwise, return the associated physical frame number.

4. **Search the Page Table for the given VPN:** Update the method `table_lookup(···)`. This method is passed the following parameters:

Cont.

```
PageTable* table; //pointer to the page table
TLB* tlb; //pointer to a k–way associative TLB cache object
int VPN; //virtual page number
```

Call `table->lookup(VPN)` (Listing 2) to get the page table entry associated with VPN. If it is not valid, throw a segmentation fault (Listing 4). If you cannot access the entry, throw a protection fault (Listing 4). If the entry is not present in the page table, throw a page fault (Listing 4). Otherwise, add the entry to the TLB using `tlb->add_entry(···)` (Listing 1). If no fault is thrown, the method should return the associated physical frame number.

5. **Form physical address:** Update the method `get_physical_address(···)`. This method is passed the following parameters:

```
int PFN; //physical frame number
int offset; //offset of address location within page/frame
int page_size; //number of bytes per page
```

Return the physical address. Assume the physical address contains the following number of bits:

```
sizeof(int) * BITS_PER_BYTE
```

6. **Translate the virtual to physical address:** Update the method `virtual_to_physical(···)`. This method is passed the following parameters:

```
int virtual_address; //virtual address of instruction
int page_size; //number of bytes per page
TLB* tlb; //pointer to a k–way associate TLB cache object
PageTable* table; //pointer to the page table object
```

Using all of the methods that you have completed above, convert a given virtual address to the associated physical address. First look for the VPN in the TLB, catching any fault that occurs. If a TLB miss exception is thrown (e.g. you catch the string defined for you (Listing 5)), find the PFN in the Page Table instead. **You will need to use a try-catch statement to catch TLB miss exceptions (and continue to throw any other fault with 'throw'). This is a C++ method, but doesn't require knowledge of C++. Information on try/catch/throw is available here.**

2 Replacement Algorithms

For this part of the programming assignment, you will write three different frame replacement algorithms. To do this, you will edit the file `replacement.cpp`. Each method is passed a FrameList object along with a pointer to a FrameList. The FrameList object has the following methods:

```
FrameList* next; //next frame in the linked list
int idx; //Index indicating when associated page was last accessed.
         //(higher index: more recently accessed)
         //Assume this index is unique
int clock_bit; //bit used for the clock algorithm (0 or 1)
```

The FrameList* linked list is ordered by arrival, with the head of the list arriving least recently and the tail arriving most recently. Complete the following methods, each returning a pointer to the FrameList* object selected for removal. The methods each return an integer of the number of FrameList* objects accessed throughout the method.

1. **FIFO Replacement:** Update the method `fifo(···)`. Implement the first in, first out frame replacement algorithm. Return the number of FrameList objects accessed during this algorithm, along with the FrameList to be removed.

Cont.

2. **Least Recently Used Replacement:** Update the method `lru(···)`. Implement the least recently used frame replacement algorithm. Return the number of FrameList objects accessed during this algorithm, along with the FrameList to be removed.

3. **Approximate Least Recently Used:** Update the method `clock_lru(···)`. Implement the clock algorithm for approximating LRU. Return the number of FrameList objects accessed during this algorithm, along with the FrameList to be removed.

3 Check for Correctness

To check that your code is working, do the following:

1. Make sure all tests are passing locally

```
sh compile.sh
cd build
make test
```

2. Push all changes to GitHub

3. Check that your GitHub actions are passing

The End.