Assignment #1

Group Members/Contribution

Chirag Krishnaswamy (2016B4A70752G) - L1 cache, L2 cache, Cache policies
Ishan Sang (2017A7PS0069G) - TLB, Scheduler
Richi Dubey (2017A7PS0099G) - Memory, Page Tables, Segment Tables
Milan Vipin Kumar Jain (2017A7PS0954G) - Write Buffer, Cache Controller, Cache policies

Files List and Description

```
.
├── cache
│   ├── cache_controller.c
│   ├── cache_controller.h
│   ├── FIFO
│   │   ├── fifo.c
│   │   ├── fifo.h
│   │   └── fifo.o
│   ├── L1_CACHE.c
│   ├── L1_CACHE.h
│   ├── L2_CACHE.c
│   ├── L2_CACHE.h
│   ├── write_buffer.c
│   └── write_buffer.h
├── Makefile
├── memory
│   ├── memory.c
│   ├── memory.h
│   ├── pagetable.c
│   ├── pagetable.h
│   ├── segtable.c
│   └── segtable.h
├── processes
│   ├── APSI.txt
│   ├── CC1.txt
│   ├── LI.txt
│   ├── M88KSIM.txt
│   └── VORTEX.txt
├── README.pdf
├── scheduler
│   ├── scheduler.c
│   └── scheduler.h
├── tlb
    ├── tlb.c
    └── tlb.h
```

How to compile/execute

‘make’ automatically compiles everything needed to run the program, runs the program and then cleans up the object files after successful execution
When the program starts, the user needs to enter the number of processes he/she wants to simulate in the program.

Structure of program

Code broken into five modules : scheduler, main memory, cache, cache controller and tlb.

SCHED : Selects process (and gets instruction address).
TLB : Saves page number to frame number translation on each page access.
CACHE CTRL : Acts as an interface between MEMORY and CACHE.
MEMORY : Contains Page-table, Segment-Table and Frames (to be allotted/replaced)
CACHE : Saves most recently used (at two level) based on principle of locality

Flow of control through the program :
SCHED -> TLB -> CACHE CTRL -> MEMORY | CACHE

Things not done yet

- Having separate memory locations allocated to page tables, so that the outermost page table for processes do not get swapped out.
- L1 cache not split into instruction and data cache

Cache

Tag and Index calculation

The three portions of an address in a set-associative or direct-mapped cache are given by a three tuple, **<tag, index, offset>**. The index is used to select the set, then the tag is used to choose the block by comparison with the blocks in the selected set. The block offset is the address of the desired data within the block.

L1 cache

configuration : cache size = 8KB; block size = 16B; assoc = 4 way (“way” = blocks per row)

block in cache = (cache size)/(block size) = (8KB) / (16B) = 512 blocks
rows in cache = (# block in cache) / assoc = (512 blocks) / (4 blocks / row) = 128 rows
rows in cache = 128 rows = 2^7 rows => **7 bit index**

block size = 16B = 2^4 bytes => **4 bit block offset**

Memory size = 64MB = 2^26 bytes = 26-bit address space
Assume 32-bit space (although we get 26-bit memory)
=> tag = 32 - (7 + 4) = **21 bit tag**
NOTE : 1st 6 bits (of 21 bits) would always be 0 in our case

Physical address format (assume 32-bit address space) (address.png)
000000 <TAG [25..11] > <INDEX [10..4] > <BLOCK_OFFSET [3..0] >

L2 cache

configuration : cache size = 32KB; block size = 32B; assoc = 16 way (“way” = blocks per row)

block in cache = (cache size)/(block size) = (32KB) / (32B) = 1024 blocks
rows in cache = (# block in cache) / assoc = (1024 blocks) / (16 blocks / row) = 64 rows
rows in cache = 64 = 2^6 => **6 bit index**

block size = 32B = 2^5 bytes => **5 bit block offset**

Memory size = 64MB = 2^26 bytes = 26-bit address space
Assume 32-bit space (although we get 26-bit memory)
=> tag = 32 - (6 + 5)= **21 bit tag**
NOTE : 1st 6 bits (of 21 bits) would always be 0 in our case

Physical address PADDR format (assume 32-bit address space) (address.png)
000000 <TAG [25..11] > <INDEX [10..5] > <BLOCK_OFFSET [4..0] >

Structure

L1 and L2 cache share structural similarities, differing only in their replacement policy. The data structure maintained for implementing the required replacement policy is highlighted below :

```
L1_CACHE_ENTRY : {int V, int D, int tag}
L1_CACHE_SET   : {L1_CACHE_ENTRY rows[]}
L1_CACHE       : {L1_CACHE_SET sets[], int n_sets, int n_rows, QUEUE fifo[][]
                  int hit, int misses}

L2_CACHE_ENTRY : {int V, int D, int tag, int counter}
L2_CACHE_SET   : {L2_CACHE_ENTRY rows[]}
L2_CACHE       : {L2_CACHE_SET sets[], int n_sets, int n_rows, int hits,
                  int misses}
```

NOTE : `QUEUE` data structure is a linked list implementation with insert(), remove() and finsert() [front-insert] functionality.

API Design

L1 and L2 cache share functional similarities as well. However, the implementation of some of these functionality differs significantly owing to the different replacement policies.
Common Functions :

- *`init()`* : Initializes `CACHE` data structures (`sets[], n_sets, n_rows` etc.).
- *`lookup()`* : Returns pointer to `ENTRY` (if present) without updating replacement data structures.
- *`extract()`* : Changes parameters to reflect the <tag, index, offset> triple of the corresponding physical address (see tag and index calculation section).

Common Functions with different Implementation :

- *`read(), write()`* : Returns pointer to `ENTRY` (if present). *`write()`* also sets the “dirty bit”. Counters in `L2_CACHE` updated to reflect the “most recently accessed”.
- *`insert(), remove()`* : Returns pointer to updated `ENTRY` and updates replacement data structure (**`fifo[]`** in `L1_CACHE` & **`counter`** in `L2_CACHE`)

Exclusive functions :

- *`counter_update()`* : (only in L2) updates counters of at a given row.
- *`lru()`* : (only for L2) returns pointer to the next `ENTRY` being replaced in `L2_CACHE`.
- *`fifo()`* : (only for L1) returns pointer to the next `ENTRY` being replaced in `L1_CACHE`.

Cache Controller

This controls the entire process of looking for data in L1, L2 and main memory. Maintains the inclusiveness of cache and handles the policies such as write buffer, write back, look through and look aside.

API Design

- *`cache_controller()`* : This function handles the coordination of the entire cache and memory together which includes loading of blocks from main memory to L2, L2 to L1 and all the policies of cache
- *`invalidate_cache()`* : After a frame is removed from main memory it is necessary to invalidate all it's cache entries from L1 cache, push any change from write buffer to L2, push from L2 to main memory, invalidate in L2 and finally update the memory frame going to be replaced
- *`load_to_L2()`* : This loads a block from L2 to L1 and handles calling of function to replace a block in L1 or L2 if needed
- *`load_to_L1()`* : This loads a blocks in L1 and may call function to replace a block in L1
- *`L1_replacer()`*: Whenever an entry is going to be replaced in L1, it will update any entries corresponding to it in write buffer to L2
- *`L2_replacer()`*: Whenever an entry is to be replaced in L2, this ensures it will invalidate it's sub-blocks if present in L1 and push all the changes if there in write buffer to L2 and now push this (going to be replaced) block of L2 to main memory
- *`delete_update()`*: Deletes an entry from write_buffer and updates it to L2
- *`invalidate_in_L1()`:* To invalidate an entry of L1 cache

Write Buffer

In order to avoid repetitive writing to L2, we store the modified blocks in the write buffer till the bus is free and we can update the write buffer. For our implementation we can set the write buffer size by `SIZE_OF_BUFFER` which is set as 8 initially to store 8 blocks.

Structure

`write_buffer : { char address_buffer[SIZE_OF_BUFFER][27] , int number_of_elements }`
address_buffer - we store the byte addresses of all the blocks
number_of_elements - currently the number of elements in write_buffer

API Design

- *`initialize_wb()`* : initializes write buffer
- *`search_write_buffer()`* : returns index to the entry for a particular PADDR if present
- *`add_write_buffer()`*: checks if there is space in write buffer else creates space in write buffer
- *`add_one_entry_write_buffer()`*: adds an entry in the write buffer after space is there for sure
- *`delete_write_buffer()`* : Deletes an entry from write buffer
- *`update_buffer_to_L2()`* : Takes one empty from write buffer and updates the corresponding bigger block to L2

Extras

- *`convert_to_PADDR()`*: converts binary address to unsigned integer

Logical to Physical Address Translation

Memory subsystem has following programs :

The call to get physical address goes as follows :
scheduler -> segtable -> pagetable -> memory

Segment Table

Structure

`SEGENTRY : { base, limit, valid}`
base: Outer Page Table Base Address of this segment[26 bit]
limit: Limit of this segment [29 bits]
valid: Valid/Invalid bit (v/i = 0 -> requested page not in memory) [1 bit]

`SEGTABLE :  { SEGENTRY *segentry, segtable_size}`
***segentry**: pointer to the segment table entry
segtable_size: size of the segment table (maximum #entries which can be present in the segment table; **4** in our case)

API Design

- *`initialize_segtable()`* : initializes segment table structure
- *`get_page_table_address()`* : returns the outer page table base address of the segment

Page Table

Structure

PAGEENTRY : {frame, valid}
`frame: stores the frame address pointed by the entry [16 bits]`
valid: Valid/Invalid bit (v/i = 0 -> requested page not in memory) [1 bit]

PAGETABLE: { PAGEENTRY **entry, base, pagetable_size, num_pagetable}
**entry: pointer to the beginning of page table
pagetable_size: size of the page table (maximum #entries which can be present in the page table; **256** in our case)
num_pagetable : number of total page table : 256*8+8+ for our case.

API Design

- initialize_pagetable*( )* : initializes page table structure
- get_nextaddress*( )* : takes the current base and offset and returns the frame address for the corresponding entry. The FINAL variable(in the parameter) denotes whether we are looking for one more page table base address or the data location from the memory.
- Get_frame_address(): Scheduler calls get_frame_address(...) with the outermost Page Table base address and this function returns the final frame of the memory by iteratively calling get_nextaddress two times (as there is three level paging in the system).
- Print_pagetable() : prints the page table at that instant

Main Memory

Structure

`FRAME : {valid, dirty, owner, refer}`
valid: Valid/Invalid bit (v/i = 0 -> requested page not in memory) [1 bit]
dirty : marked dirty, needs to be written back to secondary storage[1 bit]
owner : stores pid of the process that owns this frame [3 bits]
refer : reference bit which shows if the frame has been accessed for read/write after last second chance replacement search. [1 bit]

`MEMORY: { FRAME *frame,memory_size}`
***frame**: pointer to the beginning of frames
memory_size: size of the memory (maximum #frame which can be present in the memory; 65536 in our case, as memory size is 64 MB and 1 frame is 1K in size, so 64K frames)

Variable **lastfifo** : int variable that stores the current point where to start our search for while finding a frame to replace with second chance replacement scheme(Cycles from num_processes to MEMORY_SIZE, starts from num_processes as the beginning of the memory is saved for segment tables of the process and should not be replaced)

API Design

- `initialize_memory()` : initializes main memory
- `find_free_frame()` : returns the physical address of a frame that can be allocated to the calling process. There is a chance that there might be no free frame left, in which case find_free_frame calls replace_sec_chance to get a frame with the second chance replacement policy.

- `load_memory()`: loads the frame in the memory from secondary storage
- `replace_sec_chance()`: Returns physical address of a frame after searching according to the second chance replacement algorithm.
- `read_memory()`: Reads the frame from memory. This function has to be called every time thereafter the user is trying to read that frame as this function makes sure that the frame is still belonging to the process(accounting for the fact that the frame might have been allocated to some other process while doing a second chance replacement). In case the process does not own the frame anymore, it finds a new frame again with find_free_frame.
- `process_dies()`: Frees all the frames owned by a dying process. Whenever a process finishes execution, all the frames allocated to the process are made invalid by calling the function process_dies() and all the dirty frames are written back to main memory.
- `write_memory()` : Writes into the frame in the main memory by giving its frame address and marks the frame dirty. The write_memory function is used to indicate that the value has to be written back to secondary storage, thus making the frame dirty and reluctant to be allocated with a second chance replacement.

TLB

Structure

`TLB_ENTRY : { pid, page_number, frame_number, valid, used, counter }`
pid: process ID [3 bit]
page_number: Page Number [22 bit]
frame_number: Frame Number [16 bit]
valid: Valid/Invalid bit (v/i = 0 -> requested page not in memory) [1 bit]
used: TLB entry used value (used = 0 -> TLB entry hasn't been used yet) [1 bit]
counter: LRU counter value (ranging between 0-31) [5 bit]

`TLB :  { TLB_ENTRY *entry, TLB_size, entries_used, hits, misses }`
***entry**: pointer to the first TLB_ENTRY of the TLB
TLB_size: size of the TLB (maximum #entries which can be present in the TLB; **32** in our case)
entries_used: #entries in the TLB that have been filled up
hits: total #hits in TLB yet
misses: total #misses in TLB yet

API Design

- *`initialize_TLB()`* : initializes TLB structure
- *`show_TLB()`* : prints the whole TLB at that instant
- *`get_TLB_slot()`* : returns the index where the next TLB entry should be inserted according to LRU replacement policy
- *`update_TLB_counters()`* : updates the counters of each used entry of TLB
- *`update_TLB()`* : updates the values of the latest entry and then updates counters
- *`invalidate_process_TLB()`* : called upon process termination - makes all entries corresponding to the dying process invalid
- *`find()`* : finds whether the required page#->frame# translation is already there in the TLB

Scheduler

Structure

`PCB: { fp, pid, stbr, stlr, ptbr, ptlr, state, num_pages_prepaged, batch_size }`
fp: file pointer from where the process instructions are to be read
pid: process ID [3 bit]
stbr: segment table base register [26 bit]
stlr: segment table length register [26 bit]
ptbr: outermost page table base register [26 bit]
ptlr: outermost page table length register [26 bit]
state: current state of process (state = 0 -> process not running) [1 bit]
num_pages_prepaged: #pages to be pre-paged for this process
batch_size: #addresses to be read for this process in one batch

`INFO: { running }`
running: #processes running at the moment

API Design

- *`initialize_PCB()`* : initializes PCB structure
- *`get_instruction()`* : fetches the next instruction for the processor
- *`main()`* : runs all the processes using RoundRobin scheduling policy