# Problem Statement 3

## Group Number #1

## Group Members: Akanksha Chaudhari, Parmesh Mathur, Shounak Naik

- The memory subsystem [with TLB, L1 Cache, L2 Cache and Main Memory] has following configuration:
- TLB: Conventional Hierarchical TLB [8-way set associative L1 TLB with 16 entries [Random replacement] and 4-way set-associative L2 TLB with 32 entries [LRU square
- matrix replacement]. Flushing of non-shared entries happens at context switching.].
- L1 Cache: 4KB, 32B, 4 Way set associative way-halting cache. The cache follows Write back and Look through. It follows LRU counters as a replacement policy.
- L2 Cache: 32KB, 64B, 16 Way set associative cache. The cache follows Write through and look aside. It follows FIFO as a replacement policy.
- Main Memory with Memory Management: 32MB Main memory with Second Chance as replacement policy. The memory management scheme used is Pure Paging.
- Thrashing mechanism to implement: Page fault frequency.

Assumptions made:

1. Virtual Memory contains the complete program. Hard disk is big enough to hold all the programs and No network file is accessed.
2. First 2 blocks of the process (assume the page size and frame size is the same and is 512B) will be pre-paged into main memory before a process starts its execution.
3. All other pages are loaded on demand [Assume the system supports dynamic loading. Also assume that the system has no dynamic linking support].
4. Main memory follows Global replacement. Lower limit number of pages and upper limit number of pages per process should be strictly maintained.
5. Page tables [and segment tables wherever applicable] of all the processes reside in main memory and will not be loaded into cache memory levels.

---

## Paging Hierarchy

We have implemented a three level hierarchy.

The 32 bits address is split as follows:
**9 7 7 9**

1. **9 bits** for block offset as the page size is 512B.

2. Since each of our page table entries is 4B long, the page table holds 128 entries, thus to identify an entry in the page table , we need **7 bits**.
3. Similarly a page directory can hold 128 entries to page tables. Thus another **7 bits** are required to identify the page table.
4. Remaining **9 bits** are split in a customized way as follows:
    - Since all of the traces use a limited series of virtual addresses namely 0x10xxxxxx and 0x7fxxxxxx, we have chosen to not allocate any pages for the remaining virtual addresses.
    - This cuts down the number of hierarchies from 4 to 3 as only 1/128 addresses are accessed.
    - Thus the outermost page directories only hold 4 entries ( 2 for 0x10xxxxxx and 2 for 0x7fxxxxxx). The lower two levels are implemented as normal page tables and normal page directories.
    - Logically, this cuts down our address split to: **2 7 7 9**

A page table entry holds:
**Page number** (Ideally the page table is indexed based on page number but we have chosen to specifically store it for ease of accessing), **physical block address** (which can either be an address to a block [for a page table] or the address to a page table[for page directory]), and also holds a **valid bit** and a **shared bit**.

---

## Process Control Information

A prepraging function is called at the start of the program to populate the first 2 pages of the processes which are READY.
There are 5 processes in the system . A random number is generated between 200 and 300 and after those many virtual addresses read and processed , a context switch happens.

A random number again indicates if the current virtual address is a read or a write access.
The 7f series from the input files is taken as instructions and the 10 series is data.

Each process has some information stored related to it :
The process_state which can be either READY, RUNNING or WAITING.
The pid
The input file name
Page directory base address. (For paging)
The total number of pages for that process.

Each process has information about all types of accesses of the memory(both misses and hits), caches, tlbs. It also has information about the page fault frequency.

## Main memory

Addressing:
Physical address = 25 bits
Frame offset = 9 bits
Frame number = (25 - 9) = 16 bits
Therefore, the total number of frames: $2^{16}$ = 65536

Here, since the page tables and the frame table reserved a space of 1024 frames each, the effective physical memory allowed for usage by the processes is cut down by 2048 frames (becomes 63488 blocks).

Virtual address = 32 bits
Page offset = 9 bits because page size is 512 bytes
Page number = (32 - 9) = 23 bits
Total number of possible pages: 2^(23) = 8388608 pages

Here, since we have a reduced number of possible addresses from the processor, the number of possible logical addresses is also tangibly smaller in size (1/128th), This brings down the total number of possible pages to 65536 pages.

---

## L1 cache

Way halting cache.
A 4KB, 4 way set associative cache with each entry of 32B. Thus the number of entries would be 128. And the number of sets would be 32.

L1 cache is separated as data and instruction equally with 16 sets of data and 16 sets of instruction. In our implementation we have used 2 different caches of 2KB sizes

To address an entry with the 25 bit Physical address:
Being a block of 32B, we need 5 bits byte offset.
4 bits set-index is there (because 16 sets are there)
Rest 16 bits are tag.
Out of those tag bits , lower 4 bits are halt tag bits and the rest 12 are main tag bits.

There will be 4 halt tag arrays of size 16. (way0, way1,way2,way3). These arrays will have entries corresponding to the lower 4 bits of the tag (16 bits).

Decoding the set-index from the Physical Address takes significant time in systems.
Thus the lower 4 bits from the tag from the physical address will be checked against all entries of way0 , way1, way2, way3.If no entries from a particular way match , that way is blocked for all the 16 sets.(This is done to save power)
Once we get the set index , many of the ways are already disabled and the data that we need is returned.

The replacement policy of this cache is LRU and this is done using LRU Counters.

---

## L2 cache

512 lines are there because the cache is 32KB . Each set has 16 lines. Therefore 32 sets are there.

The cache follows Write through and look aside. We have not implemented the look aside because that would require parallel processing . We call the L2_cache search , if it's a miss then we call the main_memory search(making the implementation look through).

It follows FIFO as a replacement policy.

Each entry has a 64B data . Additionally each entry has  14 tag bits and 1 valid/invalid bit and 5 bits for fifo_replacement policy.

To address an entry with the 25 bit Physical address:
Being a block of 64B, we need 6 bits byte offset.
5 bits set-index
Rest 14 bits are tag

---

## TLB

L1 TLB 8-way set-associative L1 TLB with 16 entries.Thus 2 sets
L2 TLB 4-way set-associative L2 TLB with 32 entries.Thus 8 sets

**TLB 1 entry** consists of:
22-bit page no. (32-bit virtual address - 9-bit page offset-1-bit-set_index)
16-bit frame no. (25-bit physical address - 9-bit frame offset)
1-bit valid/invalid bit

1-bit shared bit

**TLB 2 entry** consists of:

20-bit page no. (32-bit virtual address - 9-bit page offset-3-bit-set_index)
16-bit frame no. (25-bit physical address - 9-bit frame offset)
1-bit valid/invalid bit
1-bit shared bit.

**The 2 TLBs are inclusive.**
If there is a TLB1 miss and a TLB2 hit, TLB1 is updated with the found entry in the TLB2.
If there is a TLB2 miss also, an exception is raised and the kernel performs page walk and updates both the TLB.

The replacement policy for TLB1 is random replacement and TLB2 is LRU and we have implemented a square matrix method to maintain an LRU entry.

At context switch, Both the TLB's are flushed.The entries with the shared bit as 1 are retained.

---

**We have maintained the number of hits and misses at all levels of memory.**

**Shared paging** is implemented by fixing a certain set of virtual addresses to be shared pages across processes. Namely 7FFF7xxx is reserved for shared pages. These pages are stored in main memory.

**Thrashing** : If the thrashing frequency is greater than 1% of the memory access, we swap out that process. (Changing the state of the process to WAITING). No separate space is maintained. Changing the state to WAITING ensures that the process isn't executed until the page fault frequency falls back below 1%.

---

**A list of files in the directory:**

The directory has the following header files:

- tlb.h
  ADT's related to the TLB implementation.

- cache.h
  ADT's related to the cache implementation.

- mainmemory.h
  ADT's related to the main memory implementation.

- pagetable.h
  ADT's related to the page table's implementation.

- processes.h
  ADT's related to implementation of individual processes (trace files, in this case).

- driver.c
  The module that contains the main function, and calls other functions when required.

- l1_cache_functions.c
  The module that contains the functions implemented as part of the L1 (data and instruction cache).

- l2cache.c
  The module that contains the functions implemented as part of the L2 cache.

- mainmemory.c
  The module that contains the functions implemented as part of the memory.

- Pagetable.c
  This module contains the functions required for the paging hierarchy implementation

- Processes.c
  The module that contains the functions implemented as part of the process control

The directory also includes:
- makefile

Input files
- Process_files.txt: contains number of processes and the process trace input files

---

**Compilation and Execution of the Program** can be done by simply running the makefile in the terminal, given it is opened in the path where the directory has been extracted.

```
$ make
```

Similarly, the program can be executed by running the driver.out file that is created on compiling the modules.

```
$ ./driver.out
```

---

## Known bugs

We were unable to debug the code for the main memory and the page tables. This was predominantly due to the increased use of pointers, due to the size of functions and the structures involved. Since a lot of structures had to be allocated and freed on the fly, it was difficult to track their (lack of) existence.

The debugging of all the modules was done on their own, with dummy inputs, to test initialization and deconstruction functions. The TLB's, L1 cache and L2 cache were working well when integrated, but were not tested with the main memory.

The main memory was tested on a toy piece of code with 20 entries and worked fine, but failed to do so in the actual input file as the limits were not tested.