**Prashant Yadav**

**CS 4348.002 – Program 4 Report**

**Problem statement –**

The assigned task was to simulate page fault process for following eviction policies.

* Least recently Used eviction policy(LRU)
* First in first out eviction policy(FIFO)
* Random eviction policy
* Clock eviction policy

Each simulation consists of varying working set i.e. cache size from 2 to 20. We have to do 1000 such simulations. At the end average out faults over 1000 simulations. Draw a chart between working set size and number of faults for each eviction policy to understand performance of each policy.

**Approach to solution –**

I went through slides and book to understand each type of eviction policy. I understood the circumstances which results into page faults. When cache becomes full and process requests for a memory address which is not into the cache, OS tries to bring that into cache but there is not enough space, that is when OS implements various eviction policies to make place for new memory addresses. I have implemented each of mentioned eviction policies using two basic data structures –

*//This is a representation of a memory location*

*struct memory\_location{*

*int address;*

*struct memory\_location \*next;*

*struct memory\_location \*prev;*

*int use; //Only Used in Clock policy scheduling.*

*};*

*//This structure is representation of a cache.*

*struct cache{*

*int cache\_size; //It shows current cache size.*

*struct memory\_location \*start; //Dummy starting memory location.*

*struct memory\_location \*end; //Dummy end memory location.*

*};*

1. **LRU Implementation –** As per this policy the memory location which is least recently used is removed when cache becomes full to accommodate new memory location.
2. **FIFO Implementation –** In case of FIFO eviction policy when cache becomes full, addresses which came first will be removed to accumulate new memory locations.
3. **RANDOM implementation –** In case of random eviction policy a memory location has been chosen randomly to remove.
4. **CLOCK implementation –** Clock implementation is quite like FIFO, the only difference here is that it maintains a use bit to signify if that memory location is in use or not. It locates a memory location by iterating over cache in a clock fashion, which has use bit to 0. While iterating it set use bit for all intermediate memory locations to 1.

**Solution description –**

After understanding each type of eviction policies, I implemented them in C. I used psudo code provided in the problem description to generate 1000 random memory locations which are used to simulate page fault process for all four eviction policies where working set varies from 2 to 20 for each simulation.

for(working\_set=2;working\_set<=20;working\_set++){

initialize\_cache();

faults[working\_set-2][0]+=lru\_policy\_faults(working\_set); //Simulate LRU eviction policy

faults[working\_set-2][1] += fifo\_policy\_faults(working\_set); //Simulate FIFO eviction policy

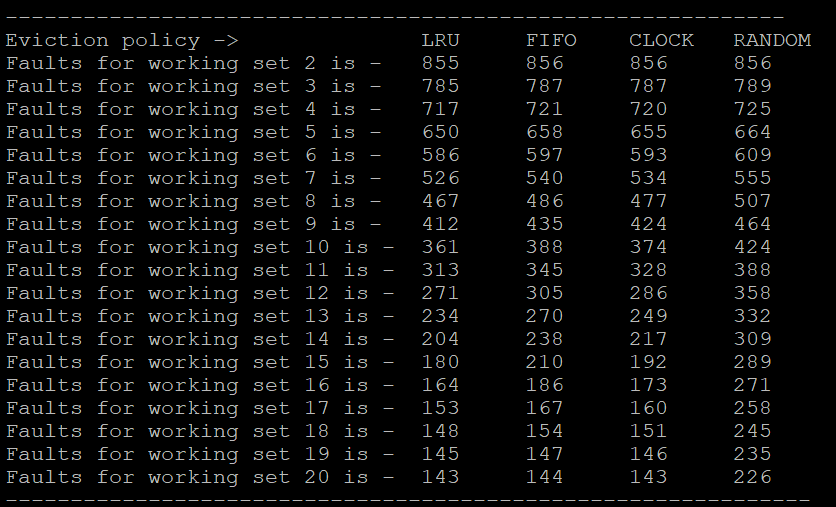
faults[working\_set-2][2] += clock\_policy\_faults(working\_set); //Simulate CLOCK eviction policy

faults[working\_set-2][3] += random\_policy\_faults(working\_set);//Simulate RANDOM eviction policy

}

Above snippet of code shows variation of working\_set size from 2-20. Count of fault count for each type of cache is maintained in “faults” matrix. Later we normalize these fault count over 1000 experiments.

Here is the snapshot of execution –



Here is the plot of fault count over working set size for each type of eviction policy(see next page).

Figure