Implementing dynamic Load Balancer in XV6

High Level Overview:

- We want each core to have their own process queue.
- Now, we also want to track for how many times a core has been idle, we will use this to migrate process from one core to another.
- We shall call the Load Balancer function on fixed intervals of time.

Changes in proc.h file:

- In cpu struct we need to add a field “int idle_ticks” this will count the number of times cpu has been idle (How?).
- In proc struct we add a field “int core_id” this will determine which process has been allocated to which core. This helps us to remove an overhead of keeping a Queue for each core.

Changes in proc.c file:

How do we allocate a process to a particular core?

- In allocproc() function we make this changes.

```
// **Assign core_id to initprocess**
acquire(&core_assign_lock);
p->core_id = next_core;
next_core = (next_core + 1) % ncpu;
release(&core_assign_lock);
```

  This ensures that each core will get processes in Round Robin manner which ensures that no core is allocated extra number of processes. And this also ensures that all processes created will have a core Id allocated to them (even one created through “fork ()”).

How to execute processes and how to calculate idle time of a core?

- Now to execute a process we need to make a little change in scheduler function which ensures that process core Id is equal to current core id.

```
// Adding extra condition for core_id
if (p->core_id != cpuid())
  continue;
```

- Now to calculate idle time of a core. It’s obvious that if a core is not executing any process and there is no process to choose from proc array then whenever scheduler function will be called it will loop through proc array without finding a

process to execute. So, we just increment idle time of a core whenever it does not find a process to execute from proc array.

How to initialize the idle time of a core to zero?

- For this we will need to make changes in “mpmain ()” function in “main.c” file. What this function exactly do is set up the CPU.

```
struct cpu *c = mycpu();
c->idle_ticks = 0;
c->lb_ticks_since_last = 0;
```

How/Where do we call the Load Balancer function?

I thought of two places from where we call call the load balancer function on was that we call it from the scheduler function itself and the other option is to call it from “trap” function in “trap.c”.

- In our case I am going with “trap” function one. Now in trap function “T_IRQ0 + IRQ_TIMEE” section indicates hardware interrupts. We will be using this section to call our Load Balancer function periodically.
- We also ensure that only Core 0 calls the Load Balancer function this ensures that no Race condition occur.

```
// **Load Balancer Integration Start**
{
    struct cpu *c = mycpu();  // Retrieve current CPU structure
    c->lb_ticks_since_last++; // Increment the load balancer tick counter

    if(c->lb_ticks_since_last >= 1000){
        c->lb_ticks_since_last = 0;  // Reset the counter
        // **Ensure Only Core 0 Invokes the Load Balancer**
        // So that no race condition comes.
        if(cpuid() == 0){
          load_balancer();   // Invoke the load balancer
        }
    }
}
// **Load Balancer Integration End**
```

How do we exactly implement Load Balancer function?

The Location of this function is in “proc.c” file. Algorithm logic is straight forward:

- First, we calculate the average idle time of the system. This gives us an overall idea of our system. Now, after this we calculate threshold, this just helps us to decide whether there is imbalance in the core load.
- After this we calculate the most overloaded and most underloaded core in the system.
- After this we check if the difference between most and least loaded cores is greater than the threshold. If yes, then we will balance the load.
- How do you balance the load? – Just transfer one process from overloaded core to underloaded core.
- After balancing the load, we are going to reset idle time of each core. Why? Just think of this case, what happen if at starting a core was idle, then most of the time this core idle time will always be greater than other core idle time even if this core will start executing CPU bound processes. So, it is necessary to reinitialize all the core idle time to zero after each load balancer function call.

```
void load_balancer(void){
  const double threshold_percentage = 0.2;
  int total_idle_time = 0;
  for(int i=0; i<ncpu; i++){
    total_idle_time += cpus[i].idle_ticks;
  }
  double average_idle_time = total_idle_time/(double)ncpu;
  double threshold = average_idle_time*threshold_percentage;
  int overloaded = -1, underloaded = -1;
  long int max_idle_time = -2147483648;
  long int min_idle_time = 2147483647;
  for (int i=0; i<ncpu; i++){
    int idle_time = cpus[i].idle_ticks; // Busy time
    if (idle_time < min_idle_time){
      min_idle_time = idle_time;
      overloaded = i;
    }
    if (idle_time > max_idle_time){
      max_idle_time = idle_time;
      underloaded = i;
    }
  }
  if (overloaded != -1 && underloaded != -1 && (max_idle_time - min_idle_time > threshold)){
    acquire(&ptable.lock);
    for (struct proc *p = ptable.proc; p < &ptable.proc[NPROC]; p++){
      if (p->state == RUNNABLE && p->core_id == overloaded){
        p->core_id = underloaded;
        cprintf("Load Balancer: Migrated Process %d to Core %d\n", p->pid, underloaded);
        cprintf("Idle time of Core 0 : %d, Idle time of Core 1 : %d\n", cpus[0].idle_ticks, cpus[1].idle_ticks);
        break; // Migrate only one process per balance operation
      }
    }
    release(&ptable.lock);
  }
}
```

Conclusion:

We implemented one CPU, and one I/O bound to test our load balance function. We also implemented “ps” syscall to check status of process.