WIKI

Design

MLFQ Implementation

- The 3-Level MLFQ scheduler aims to manage processes with different priorities in xv6. It consists of three priority levels: L0, L1, and L2.
- Processes in L0 have the highest priority, followed by L1 and L2.
    - The scheduler will first run processes in the highest priority queue if any are runnable.
    - Time quantums for L0, L1, and L2 are set at 4, 6, and 8 ticks, respectively.
    - L0 and L1 use round-robin scheduling, while L2 uses priority-based scheduling with a range of 0 to 3.

Preventing Starvation

- To prevent starvation, a priority boosting mechanism is used. When the global tick counter reaches 100 ticks, all processes are readjusted to L0, and their priority values are reset.
- Additionally, the scheduler allows processes to lock the scheduler and prioritize their execution using the `schedulerLock()` and `schedulerUnlock()` system calls.
    - These system calls enable processes to override the MLFQ scheduler when necessary, ensuring that critical processes receive the necessary CPU time.

Implementation

- Note that most of the codes that I have added are marked with comments like this. The files that I have modified to implement mlfq are proc.c, param.h, defs.h, sysproc.c, syscall.h, syscall.c, usyss.S, user.h, proc.h, trap.c, Makefile and traps.h

```
//IMAN S
//IMAN E

IMAN S marks the start of the part of code that I added and IMAN E
marks the end of the code snippet that I added.
```

1. *Struct and Variables:* To implement the MLFQ scheduler, several fields are added to the `struct proc` in `proc.h`:

```
  int level;               // Level of the process in MLFQ (0, 1, or 2)
  int priority;            // Priority value for processes in L2 (0 to 3)
  int time_quantum;        // Time quantum assigned to the process
  int time_quantum_ticks;  // Ticks used in the current time quantum
  int scheduler_locked;    // Flag for scheduler lock (0 or 1)
```

2. *Helper Functions:* Two helper functions, `next_process()` and `next_process_L2()`, are implemented in `proc.c` to manage MLFQ scheduling. These functions help find the next process to run in L0/L1 and L2, respectively.
3. *Scheduler:* The `scheduler()` function in `proc.c` is updated to implement the MLFQ scheduling logic, priority boosting, and interaction between the `schedulerLock()` and `schedulerUnlock()` functions.

```
// MLFQ scheduling
if (c->proc == 0) { // If CPU is idle
    next = 0;
    // Find a process in L0
    next = next_process(ptable.proc, 0);

    if (!next) {
        // Find a process in L1
        next = next_process(ptable.proc, 1);
    }

    if (!next) {
        // Find a process in L2
        next = next_process_L2();
    }

    if (next) {
        // Run the process
```

```
            c->proc = next;
            switchuvm(next);
            next->state = RUNNING;
            swtch(&(c->scheduler), next->context);
            switchkvm();

            // Process is done running for now.
            // It should have changed its p->state before coming back.
            c->proc = 0;

            // Update process's time_quantum_ticks, level, and time_quantum
            next->time_quantum_ticks = 0;
            if (next->level < 2) {
                next->level++;
                if (next->level == 1) {
                    next->time_quantum = L1_QUANTUM;
                } else {
                    next->time_quantum = L2_QUANTUM;
                }
            }
        }
    }
```

- The scheduler divides processes into three different priority levels, L0, L1, and L2. L0 is the highest priority level and L2 is the lowest priority level.
- When a process is ready to run, the scheduler looks for processes in L0 first. If there is a process in L0, it is chosen to run immediately. If there are no processes in L0, the scheduler looks for processes in L1, and then in L2.
- When a process completes its time quantum (the amount of time it is allowed to run on the CPU), it is moved to a lower priority queue. If a process blocks or yields, it remains in the same priority queue. If a process is in L2, it remains there until it completes its execution.
- The scheduler also periodically boosts the priority of all processes to prevent starvation and ensure that all processes get a chance to run.
- In summary, the MLFQ scheduling is a way to manage multiple processes in an operating system by dividing them into different priority levels and choosing the next process to run based on its priority level and time spent waiting for CPU time. It ensures that interactive processes are responsive while also allowing batch processes to run eventually.

4. *System Calls:* System calls `yield()`, `getLevel()`, `setPriority()`, `schedulerLock()`, and `schedulerUnlock()` are implemented as per the given specifications, and their corresponding trap calls are added to `syscall.c` and `usys.S`.

Some of the significant parts I have added in xv6 public file

1. param.h

```
//defined to keep up with the specifications of L0, L1 and L2
//having 4 ticks, 6 ticks and 8 ticks
#define L0_QUANTUM 4
#define L1_QUANTUM 6
#define L2_QUANTUM 8
```

2. proc.c

```
//Functions that I have added

//Used in L0 and L1 to find the next process using round robin process
struct proc* next_process(struct proc *start, int level);

//used in L2 for priority based scheduling
struct proc* next_process_L2();

int setPriority(int pid, int priority);

//Unlike in the specification, changed the return value of these two
//functions from void to int.
int schedulerLock(int password);
int schedulerUnlock(int password);
```

3. trap.c

```
//Handling trap requirements for 129 and 130
void
idtinit(void)
{
  //IMAN s
  // Set up interrupt gates for interrupts 129 and 130
  SETGATE(idt[129], 1, SEG_KCODE << 3, vectors[129], DPL_USER);
  SETGATE(idt[130], 1, SEG_KCODE << 3, vectors[130], DPL_USER);
  //IMAN E
  lidt(idt, sizeof(idt));
}

//IMAN S
  else if(tf->trapno == T_SCHEDULERLOCK){
    if(myproc()->killed)
      exit();
    myproc()->tf->eax = schedulerLock(myproc()->tf->ebx);
  }
  else if(tf->trapno == T_SCHEDULERUNLOCK){
    if(myproc()->killed)
      exit();
```

```
    myproc()->tf->eax = schedulerUnlock(myproc()->tf->ebx);
  }
//IMAN E
```

Result

- After compiling and executing the modified xv6, the 3-Level MLFQ scheduler works as expected. Processes are scheduled according to their levels and priorities, with priority boosting occurring at 100 global ticks to prevent starvation. Processes can lock the scheduler to prioritize their execution using the `schedulerLock()` and `schedulerUnlock()` system calls.

```
Booting from Hard Disk..xv6...
cpu0: starting 0
sb: size 1000 nblocks 941 ninodes 200 nlog 30 logstart 2 inodestart 32 bmap start 58
init: starting sh
$ mlfq_test
MLFQ test start
[Test 1] default
Process 4
L0: 2120
L1: 2613
L2: 95267
L3: 0
L4: 0
Process 5
L0: 3524
L1: 3235
L2: 93241
L3: 0
L4: 0
Process 6
L0: 3410
L1: 4067
L2: 92523
L3: 0
L4: 0
Process 7
L0: 4605
L1: 5001
L2: 90394
L3: 0
L4: 0
[Test 1] finished
done
```

Trouble Shooting

During the implementation, there were a few challenges:

1. *Distinguishing between global ticks and time quantum ticks:* There was an initial confusion between the global tick counter and the time quantum ticks for each

process. To resolve this, I used a global tick counter in `trap.c` and added a `time_quantum_ticks` field to the `struct proc` to keep track of ticks used in the current time quantum for each process.

2. *Implementing priority-based scheduling in L2:* Implementing the priority-based scheduling for L2 required careful consideration of the selection algorithm. The use of a helper function, `next_process_L2()`, helped manage the complexity and ensured that processes with the same priority were scheduled using FCFS.
3. *Handling scheduler lock and unlock:* Ensuring the correct implementation of the `schedulerLock()` and `schedulerUnlock()` functions was crucial. Initially, the `schedulerUnlock()` function did not completely follow the specifications, but after reviewing the requirements, the function was updated to comply with the given specifications.