FreeBSD Scheduler with Multi-Level Feedback Queuing

Harmohan Singh (W784066)
Dake Zhao
Satyakumar Vemuri (W0756992)
ABSTRACT

The project’s main objective is to alter the existing scheduler in the FreeBSD Operating System, by enabling Multi-Level Feedback Queuing and thereby making the scheduling more efficient. The scheduler was implemented on a FreeBSD OS installed on VMware and the results were captured using the Kernel Trace Dump. Performance of both the schedulers is compared using the result procured form the implementation.

Introduction

The FreeBSD time-share-scheduling algorithm is based on multilevel feedback queues. The system adjusts the priority of a thread dynamically to reflect resource requirements (e.g., being blocked awaiting an event) and the amount of resources consumed by the thread (e.g., CPU time). Threads are moved between run queues based on changes in their scheduling priority (hence the word feed-back in the name multilevel feedback queue). When a thread other than the currently running thread attains a higher priority (by having that priority either assigned or given when it is awakened), the system switches to that thread immediately if the current thread is in user mode. Otherwise, the system switches to the higher-priority thread as soon as the current thread exits the kernel. The system tailors this short-term scheduling algorithm to favor interactive jobs by raising the scheduling priority of threads that are blocked waiting for I/O for one or more seconds and by lowering the priority of threads that accumulate significant amounts of CPU time.

Short-term thread scheduling is broken into two parts. The next section describes when and how a thread’s scheduling priority is altered; the section after that describes the management of the run queues and the interaction between thread scheduling and context switching.

The original FreeBSD scheduler dynamically updates the priority of each process based on its current service level. CPU hogs become less likely to run, while I/O
bound jobs become more likely to run. Implement Multi Level Feed Back Queue Algorithm for Process Scheduling in FreeBSD which maintains multiple queues based on service criteria.

**The Original FreeBSD Scheduler**

All threads that are runnable are assigned a scheduling priority that determines in which run queue they are placed. In selecting a new thread to run, the system scans the run queues from highest to lowest priority and chooses the first thread on the first nonempty queue. If multiple threads reside on a queue, the system runs them round robin—that is, it runs them in the order that they are found on the queue, with equal amounts of time allowed. If a thread blocks, it is not put back onto any run queue. If a thread uses up the time quantum (or time slice) it is allowed, it is placed at the end of the queue from which it came, and the thread at the front of the queue is selected to run. The shorter the time quantum, the better the interactive response.

However, longer time quanta provide higher system throughput because the system will have less overhead from doing context switches, and processor caches will be flushed less often. The time quantum FreeBSD uses is 0.1 second. This value was empirically found to be the longest quantum that could be used without loss of the desired response for interactive jobs such as editors. Perhaps surprisingly, the time quantum has remained unchanged over the past 20 years. Although the time quantum was originally selected on centralized timesharing systems with many users, it is still correct for decentralized workstations today. While workstation users expect a response time faster than that anticipated by the timesharing users of 20 years ago, the shorter run queues on the typical workstation make a shorter quantum unnecessary.

**The Multi-level Feedback Queue**

Figure 1 illustrates a Multi-Level Feedback Queue consisting of five queues (Q1-Q5) that make up the operating system’s “ready queue” of processes available for CPU-execution. Processing bursts enter in Queue 1, which is both the highest-
priority queue, and also the queue that allocates the shortest burst of CPU time to processes (in this example, 16ms). A processing burst in Q1 either completes its burst, exiting the ready queue to a wait queue, or completes and exits the system entirely, or is incomplete and “falls down”, to Q2. Q2 only receives CPU processing time when Q1 is empty. Q3 receives processing time only when both Q1 and Q2 are empty.

Similarly, the last queue, which is also the lowest priority queue, only receives CPU time when all higher-priority queues are empty. This creates an opportunity for processes in Q5 to receive no CPU time when the workload is high (as a percentage of 100% CPU processing capability). In the MFLQ, CPU time is allocated to queues on a non-interruptible basis for normal processing. Interrupts and critical services can interrupt current processing. For example, if queues Q1, Q2, and Q3 are currently empty, the CPU is allocated to processing work in Q4. If a new processing burst then becomes ready for execution in Q1, it will not be serviced until the current burst being processed in Q4 either completes or the entire Q4 time quantum is consumed. For either of those events, the system will check Q1 (and Q2 and Q3 as appropriate) for work in those higher-priority queues. This then results in a context-switch to service the new processing burst that arrived in Q1.
Calculations of Thread Priority

A thread’s scheduling priority is determined directly by two values associated with the thread structure: kg_estcpu and kg_nice. The value of kg_estcpu provides an estimate of the recent CPU utilization of the thread. The value of kg_nice is a user-settable weighting factor that ranges numerically between -20 and 20. The normal value for kg_nice is zero. Negative values increase a thread’s priority, whereas positive values decrease its priority.

A thread’s user-mode scheduling priority is calculated after every four clock ticks (typically 40 milliseconds) that it has been found running by this equation:

\[ kg_{\text{user\_pri}} = PRI\_\text{MIN\_TIMESHARE} + \left[ \frac{kg_{\text{estcpu}}}{4} + 2 \times kg_{\text{nice}} \right] \]

Values less than PRI_MIN_TIMESHARE (160) are set to PRI_MIN_TIMESHARE (see table 1); values greater than PRI_MAX_TIMESHARE (223) are set to PRI_MAX_TIMESHARE. This calculation causes the priority to decrease linearly based on recent CPU utilization. The user-controllable kg_nice parameter acts as a limited weighting factor. Negative values retard the effect of heavy CPU utilization by offsetting the additive term containing kg_estcpu. Otherwise, if we ignore the second term, kg_nice simply shifts the priority by a constant factor.

The CPU utilization, kg_estcpu, is incremented each time that the system clock ticks and the thread is found to be executing. In addition, kg_estcpu is adjusted once per second via a digital decay filter. The decay causes about 90 percent of the CPU usage accumulated in a one-second interval to be forgotten over a period of time that is dependent on the system load average. To be exact, kg_estcpu is adjusted according to

\[ kg_{\text{estcpu}} = \frac{(2 \times \text{load})}{(2 \times \text{load} + 1)} kg_{\text{estcpu}} + kg_{\text{nice}}, \]

where the load is a sampled average of the sum of the lengths of the run queue and of the short-term sleep queue over the previous one-minute interval of system operation.

To understand the effect of the decay filter, consider the case where a single compute-bound thread monopolizes the CPU. The thread’s CPU utilization will accumulate clock
ticks at a rate dependent on the clock frequency. The load average will be effectively 1, resulting in a decay of

\[ \text{kg\_estcpu} = 0.66 \times \text{kg\_estcpu} + \text{kg\_nice}. \]

If we assume that the thread accumulates \( Ti \) clock ticks over time interval \( i \) and that \( \text{kg\_nice} \) is zero, then the CPU utilization for each time interval will count into the current value of \( \text{kg\_estcpu} \) according to

\[
\begin{align*}
\text{kg\_estcpu} &= 0.66 \times T_0 \\
\text{kg\_estcpu} &= 0.66 \times (T_1 + 0.66 \times T_0) = 0.66 \times T_1 + 0.44 \times T_0 \\
\text{kg\_estcpu} &= 0.66 \times T_2 + 0.44 \times T_1 + 0.30 \times T_0 \\
\text{kg\_estcpu} &= 0.66 \times T_3 + \ldots + 0.20 \times T_0 \\
\text{kg\_estcpu} &= 0.66 \times T_4 + \ldots + 0.13 \times T_0.
\end{align*}
\]

Thus, after five decay calculations, only 13 percent of \( T_0 \) remains present in the current CPU utilization value for the thread. Since the decay filter is applied once per second, about 90 percent of the CPU utilization is forgotten after five seconds.

Threads that are runnable have their priority adjusted periodically as just described. However, the system ignores threads that are blocked awaiting an event: these threads cannot accumulate CPU usage, so an estimate of their filtered CPU usage can be calculated in one step. This optimization can significantly reduce a system’s scheduling overhead when many blocked threads are present. The system recomputes a thread’s priority when that thread is awakened and has been sleeping for longer than one second. The system maintains a value, \( \text{kg\_slptime} \), that is an estimate of the time a thread has spent blocked waiting for an event. The value of \( \text{kg\_slptime} \) is set to zero when a thread

<table>
<thead>
<tr>
<th>Range</th>
<th>Class</th>
<th>Thread type</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-63</td>
<td>ITHD</td>
<td>bottom-half kernel (interrupt)</td>
</tr>
<tr>
<td>64-127</td>
<td>kern</td>
<td>top-half kernel</td>
</tr>
<tr>
<td>128-159</td>
<td>realtime</td>
<td>realtime user</td>
</tr>
<tr>
<td>160-223</td>
<td>timeshare</td>
<td>time-sharing user</td>
</tr>
<tr>
<td>224-255</td>
<td>idle</td>
<td>idle user</td>
</tr>
</tbody>
</table>

\[ \text{kg\_estcpu} = 0.66 \times T_0 \]
calls sleep() and is incremented once per second while the thread remains in a SLEEPING or STOPPED state. When the thread is awakened, the system computes the value of kg_estcpu according to

\[ kg_{estcpu} = \left\lfloor \frac{(2 \times \text{load})^{k_{estcpu}}}{(2 \times \text{load} + 1)} \right\rfloor \times kg_{estcpu}, \]

and then recalculates the scheduling priority using this equation. This analysis ignores the influence of kg_nice; also, the load used is the current load average rather than the load average at the time that the thread blocked.

**MLFQ Solution Implementation**

**Basic Design Rules**

- If Priority(A) > Priority(B), A will run (and B won’t).
- If Priority(A) = Priority(B), both A and B will be run in round-robin fashion.
- When a job enters the system, it is placed at the highest priority (the topmost queue).
- If a job uses up an entire time slice while running, its priority is reduced (i.e., it moves down one queue).
- If a job gives up the CPU before the time slice is up, it stays at the same priority level.
- After some time period S, move all the jobs in the system to the topmost queue.

**Algorithm Used**: 

1. Round-robin queue, Q = T. If a job runs to the completion of its timeslice, move it down to queue 2.

2. Round-robin queue, Q = 2 × T. If a thread in this queue gets no service for three quanta, move it up to queue 1. If a job runs to the completion of its timeslice, move it down to queue 3.

3. Round-robin queue, Q = 4 × T. If a thread in this queue gets no service for three quanta, move it up to queue 2. If a job runs to the completion of its timeslice, move it down to queue 4.
4. A FIFO queue. Jobs in this queue run to completion (they can be preempted by jobs arriving in better-priority queues). That is, $Q = \infty$. Jobs enter this queue at the tail, and until the job at the front of the queue completes, no other job in this queue will run. If a thread gets no CPU service for $120 \times T$ ticks, move it up to the tail of queue

**Simulation Results**

**Legend**

Mi_switch $\rightarrow$ context switching function  
PID $\rightarrow$ Process ID  
Csh $\rightarrow$ shell process  
$T$ $\rightarrow$ time stamp  
$Q$ $\rightarrow$ quantum

---

2117 mi_switch:: QUEUE 2 - PID 498 HAS PROCESS NAME csh  
2116 mi_switch:: QUEUE 2 - THREAD OF PID 498 READY TO CONTEXT SWITCH  
2115 mi_switch:: QUEUE 2 - PID 498 HAS PROCESS NAME csh  
2114 mi_switch:: QUEUE 2 - THREAD OF PID 498 HAS PRIORITY 104  
2113 mi_switch:: THREAD OF PID 36 HAS PRIORITY 36  
2112 mi_switch:: QUEUE 1 - THREADMOVED DOWN TO QUEUE 2  
2111 mi_switch:: QUEUE 1 - PID 518 HAS PROCESS NAME a.out  
2110 mi_switch:: QUEUE 1 - THREAD OF PID 518 HAS USED ITS TIME  
QUANTUM => $Q = T$

2109 mi_switch:: QUEUE 1 - THREAD OF PID 518 HAS PRIORITY 189  
2108 mi_switch:: THREAD OF PID 3 HAS PRIORITY 76  
2107 mi_switch:: THREAD OF PID 4 HAS PRIORITY 76  
2106 mi_switch:: THREAD OF PID 36 HAS PRIORITY 36  
2105 mi_switch:: QUEUE 1 - THREADMOVED DOWN TO QUEUE 2  
2104 mi_switch:: QUEUE 1 - PID 519 HAS PROCESS NAME a.out  
2103 mi_switch:: QUEUE 1 - THREAD OF PID 519 HAS USED ITS TIME  
QUANTUM => $Q = T$

2102 mi_switch:: QUEUE 1 - THREAD OF PID 519 HAS PRIORITY 189  
2101 mi_switch:: THREAD OF PID 2 HAS PRIORITY 76  
2100 mi_switch:: THREAD OF PID 36 HAS PRIORITY 36  
2099 mi_switch:: QUEUE 1 - THREADMOVED DOWN TO QUEUE 2  
2098 mi_switch:: QUEUE 1 - PID 528 HAS PROCESS NAME a.out  
2097 mi_switch:: QUEUE 1 - THREAD OF PID 528 HAS USED ITS TIME  
QUANTUM => $Q = T$

2096 mi_switch:: QUEUE 1 - THREAD OF PID 528 HAS PRIORITY 187  
2095 mi_switch:: THREAD OF PID 36 HAS PRIORITY 36  
2094 mi_switch:: QUEUE 1 - THREADMOVED DOWN TO QUEUE 2  
2093 mi_switch:: QUEUE 1 - PID 529 HAS PROCESS NAME a.out  
2092 mi_switch:: QUEUE 1 - THREAD OF PID 529 HAS USED ITS TIME  
QUANTUM => $Q = T$

2091 mi_switch:: QUEUE 1 - THREAD OF PID 529 HAS PRIORITY 187  
2090 mi_switch:: THREAD OF PID 36 HAS PRIORITY 36  
2089 mi_switch:: QUEUE 1 - THREADMOVED DOWN TO QUEUE 2  
2088 mi_switch:: QUEUE 1 - PID 530 HAS PROCESS NAME a.out
2087 mi_switch:: QUEUE 1 - THREAD OF PID 530 HAS USED ITS TIME QUANTUM -- Q = T
2086 mi_swch:: QUEUE 1 - THREAD OF PID 530 HAS PRIORITY 188
2085 mi_switch:: THREAD OF PID 51 HAS PRIORITY 104
2084 mi_switch:: THREAD OF PID 36 HAS PRIORITY 36
2083 calwaittime:: QUEUE 2 - THREAD OF PID 38 HAS WAITED FOR 1T
2082 calwaittime:: QUEUE 2 - THREAD OF PID 498 HAS WAITED FOR 5T
2081 calwaittime:: QUEUE 2 - THREAD OF PID 506 HAS WAITED FOR 1T
2080 calwaittime:: QUEUE 2 - THREAD OF PID 507 HAS WAITED FOR 1T
2079 calwaittime:: QUEUE 2 - THREAD OF PID 508 HAS WAITED FOR 5T
2078 calwaittime:: QUEUE 2 - THREAD OF PID 509 HAS WAITED FOR 1T
2077 calwaittime:: QUEUE 2 - THREAD OF PID 510 HAS WAITED FOR 3T
2076 calwaittime:: QUEUE 2 - THREAD OF PID 511 HAS WAITED FOR 3T
2075 calwaittime:: QUEUE 2 - THREAD OF PID 512 HAS WAITED FOR 3T
2074 calwaittime:: QUEUE 2 - THREAD OF PID 513 HAS WAITED FOR 4T
2073 calwaittime:: QUEUE 2 - THREAD OF PID 514 HAS WAITED FOR 5T
2072 calwaittime:: QUEUE 2 - THREAD OF PID 515 HAS WAITED FOR 5T
2071 calwaittime:: QUEUE 2 - THREAD OF PID 516 HAS WAITED FOR 1T
2070 calwaittime:: QUEUE 2 - THREAD OF PID 517 HAS WAITED FOR 5T
2069 calwaittime:: TIME TO MOVE THREAD OF PID 518 TO QUEUE 1
2068 calwaittime:: QUEUE 2 - THREAD OF PID 518 HAS WAITED FOR 6T
2067 calwaittime:: TIME TO MOVE THREAD OF PID 519 TO QUEUE 1
2066 calwaittime:: QUEUE 2 - THREAD OF PID 519 HAS WAITED FOR 6T
2065 calwaittime:: QUEUE 2 - THREAD OF PID 524 HAS WAITED FOR 1T
2064 calwaittime:: QUEUE 2 - THREAD OF PID 525 HAS WAITED FOR 1T
2063 calwaittime:: QUEUE 2 - THREAD OF PID 526 HAS WAITED FOR 5T
2062 calwaittime:: QUEUE 2 - THREAD OF PID 527 HAS WAITED FOR 5T
2061 calwaittime:: TIME TO MOVE THREAD OF PID 528 TO QUEUE 1
2060 calwaittime:: QUEUE 2 - THREAD OF PID 528 HAS WAITED FOR 6T
2059 calwaittime:: TIME TO MOVE THREAD OF PID 529 TO QUEUE 1
2058 calwaittime:: QUEUE 2 - THREAD OF PID 529 HAS WAITED FOR 6T
2057 calwaittime:: TIME TO MOVE THREAD OF PID 530 TO QUEUE 1
2056 calwaittime:: QUEUE 2 - THREAD OF PID 530 HAS WAITED FOR 6T
2055 calwaittime:: QUEUE 2 - THREAD OF PID 531 HAS WAITED FOR 1T
2054 calwaittime:: QUEUE 2 - THREAD OF PID 532 HAS WAITED FOR 1T
2053 calwaittime:: QUEUE 2 - THREAD OF PID 533 HAS WAITED FOR 1T
2052 calwaittime:: QUEUE 2 - THREAD OF PID 534 HAS WAITED FOR 2T
2051 calwaittime:: QUEUE 2 - THREAD OF PID 535 HAS WAITED FOR 4T
2050 calwaittime:: QUEUE 2 - THREAD OF PID 536 HAS WAITED FOR 5T
2049 calwaittime:: QUEUE 2 - THREAD OF PID 537 HAS WAITED FOR 1T
2048 calwaittime:: QUEUE 2 - THREAD OF PID 538 HAS WAITED FOR 2T
2047 calwaittime:: QUEUE 2 - THREAD OF PID 539 HAS WAITED FOR 2T
2046 calwaittime:: QUEUE 2 - THREAD OF PID 540 HAS WAITED FOR 2T
2045 calwaittime:: QUEUE 2 - THREAD OF PID 541 HAS WAITED FOR 2T
2044 calwaittime:: QUEUE 2 - THREAD OF PID 542 HAS WAITED FOR 2T
2043 calwaittime:: QUEUE 2 - THREAD OF PID 543 HAS WAITED FOR 3T
2042 calwaittime:: QUEUE 2 - THREAD OF PID 544 HAS WAITED FOR 3T
2041 calwaittime:: QUEUE 2 - THREAD OF PID 545 HAS WAITED FOR 3T
2040 calwaittime:: QUEUE 2 - THREAD OF PID 546 HAS WAITED FOR 3T
2039 calwaittime:: QUEUE 2 - THREAD OF PID 547 HAS WAITED FOR 4T
2038 calwaittime:: QUEUE 2 - THREAD OF PID 548 HAS WAITED FOR 4T
2037 calwaittime:: QUEUE 2 - THREAD OF PID 549 HAS WAITED FOR 5T
2036 calwaittime:: QUEUE 2 - THREAD OF PID 550 HAS WAITED FOR 3T
2035 calwaittime:: QUEUE 2 - THREAD OF PID 551 HAS WAITED FOR 4T
2034 calwaittime:: QUEUE 2 - THREAD OF PID 552 HAS WAITED FOR 2T
2033 calwaittime:: QUEUE 2 - THREAD OF PID 553 HAS WAITED FOR 2T
2032 calwaittime:: QUEUE 2 - THREAD OF PID 554 HAS WAITED FOR 2T
calwaittime:: QUEUE 2 - THREAD OF PID 555 HAS WAITED FOR 2T
2030 calwaittime:: QUEUE 2 - THREAD OF PID 556 HAS WAITED FOR 3T
2029 calwaittime:: QUEUE 2 - THREAD OF PID 557 HAS WAITED FOR 3T
2028 calwaittime:: QUEUE 2 - THREAD OF PID 558 HAS WAITED FOR 4T
2027 calwaittime:: QUEUE 2 - THREAD OF PID 559 HAS WAITED FOR 4T
2026 calwaittime:: QUEUE 2 - THREAD OF PID 560 HAS WAITED FOR 4T
2025 calwaittime:: QUEUE 2 - THREAD OF PID 561 HAS WAITED FOR 4T
2024 calwaittime:: QUEUE 2 - THREAD OF PID 562 HAS WAITED FOR 4T
2023 calwaittime:: QUEUE 2 - THREAD OF PID 563 HAS WAITED FOR 5T
2022 calwaittime:: QUEUE 2 - THREAD OF PID 564 HAS WAITED FOR 5T
2021 calwaittime:: QUEUE 2 - THREAD OF PID 567 HAS WAITED FOR 5T
2020 calwaittime:: TIME TO CALCULATE WAIT TIME

mswitch:: QUEUE 2 - PID 524 HAS PROCESS NAME a.out
2019 mswitch:: QUEUE 2 - THREAD OF PID 524 READY TO CONTEXT SWITCH
2018 mswitch:: QUEUE 2 - PID 524 HAS PROCESS NAME a.out
2017 mswitch:: QUEUE 2 - THREAD OF PID 524 HAS PRIORITY 187
2016 mswitch:: THREAD OF PID 52 HAS PRIORITY 80
2015 mswitch:: THREAD OF PID 50 HAS PRIORITY 68
2014 mswitch:: THREAD OF PID 9 HAS PRIORITY 68
2013 mswitch:: THREAD OF PID 36 HAS PRIORITY 36
2012 mswitch:: THREAD OF PID 36 HAS PRIORITY 36
2011 mswitch:: QUEUE 2 - PID 525 HAS PROCESS NAME a.out
2010 mswitch:: QUEUE 2 - THREAD OF PID 525 READY TO CONTEXT SWITCH
2009 mswitch:: QUEUE 2 - PID 525 HAS PROCESS NAME a.out
2008 mswitch:: QUEUE 2 - THREAD OF PID 525 HAS PRIORITY 188
2007 mswitch:: THREAD OF PID 36 HAS PRIORITY 36
2006 mswitch:: QUEUE 2 - PID 531 HAS PROCESS NAME a.out
2005 mswitch:: QUEUE 2 - THREAD OF PID 531 READY TO CONTEXT SWITCH
2004 mswitch:: QUEUE 2 - PID 531 HAS PROCESS NAME a.out
2003 mswitch:: QUEUE 2 - THREAD OF PID 531 HAS PRIORITY 187
2002 mswitch:: QUEUE 2 - THREAD OF PID 36 HAS PRIORITY 36
2001 mswitch:: QUEUE 2 - PID 532 HAS PROCESS NAME a.out
2000 mswitch:: QUEUE 2 - THREAD OF PID 532 READY TO CONTEXT SWITCH
1999 mswitch:: QUEUE 2 - PID 532 HAS PROCESS NAME a.out
1998 mswitch:: QUEUE 2 - THREAD OF PID 532 HAS PRIORITY 188
1997 mswitch:: THREAD OF PID 36 HAS PRIORITY 36
1996 mswitch:: QUEUE 2 - PID 533 HAS PROCESS NAME a.out
1995 mswitch:: QUEUE 2 - THREAD OF PID 533 READY TO CONTEXT SWITCH
1994 mswitch:: QUEUE 2 - PID 533 HAS PROCESS NAME a.out
1993 mswitch:: QUEUE 2 - THREAD OF PID 533 HAS PRIORITY 187
1992 mswitch:: THREAD OF PID 36 HAS PRIORITY 36
1991 mswitch:: QUEUE 1 - THREAD MOVED DOWN TO QUEUE 2
1990 mswitch:: QUEUE 1 - PID 506 HAS PROCESS NAME a.out
1989 mswitch:: QUEUE 1 - THREAD OF PID 506 HAS USED ITS TIME

QUANTUM -: Q = T
1988 mswitch:: QUEUE 1 - THREAD OF PID 506 HAS PRIORITY 188
1987 mswitch:: THREAD OF PID 3 HAS PRIORITY 76
1986 mswitch:: THREAD OF PID 4 HAS PRIORITY 76
1985 mswitch:: THREAD OF PID 36 HAS PRIORITY 36
1984 mswitch:: QUEUE 1 - THREAD MOVED DOWN TO QUEUE 2
1983 mswitch:: QUEUE 1 - PID 507 HAS PROCESS NAME a.out
1982 mswitch:: QUEUE 1 - THREAD OF PID 507 HAS USED ITS TIME

QUANTUM -: Q = T
1981 mswitch:: QUEUE 1 - THREAD OF PID 507 HAS PRIORITY 188
1980 mswitch:: THREAD OF PID 2 HAS PRIORITY 76
1979 mswitch:: THREAD OF PID 36 HAS PRIORITY 36
1978 mswitch:: QUEUE 1 - THREAD MOVED DOWN TO QUEUE 2
1977 mswitch:: QUEUE 1 - PID 509 HAS PROCESS NAME a.out
Some of the functions changed/Modified:

```c
int runq_traverse(void) [sched_bsd4.c]
ke = runq_choose(&runq);
```

This function is responsible for maintaining the run queue in which processes and threads are put.

```c
if(ke->ke_thread->belongsto == 0)
{ // Performing check to add thread in the queue 1
    if(thread_pri >= 160 && thread_pri <= 175) {
        q_bit = 1; CTR2(KTR_PRJ1, "runq_add:: PROCESS %d HAS PRIORITY: %d",process_id,thread_pri);
        q_bit = 1; CTR1(KTR_PRJ1, "runq_add:: PROCESS %d IS ON RUNQUEUE: %d",process_id,pri);
        ke->ke_thread->belongsto = 1;
    }
}
```

This snippet of code puts the thread into the right runqueue of the 4 queues of the User TIMESHARE Class.

```c
void experiment(void); // Function that performs experiments.
```

The function to perform experiments

```c
Int runq_traverse(void);
// Function to check if there is any higher priority thread.
```

```c
void calwaittime(void);
// Function to calculate the thread's wait time.
void td_moveup(struct thread *td, int newpri); // Function to move
```

How to build, compile and run the FreeBSD
1) Put kern_thread.c into /sys/kern/
2) Put kern_sync.c into /sys/kern/
3) Put kern_switch.c into /sys/kern/
4) Put ktr.h /src/sys/sys
5) Put proc.h into /sys/sys

Change to /usr/src

Compile the kernel

#make buildkernel KERNCONF=GENERIC

Install the kernel

#make installkernel KERNCONF=GENERIC

**Conclusion**

The Multi-Layer Queue (MLQ) scheduling strategy utilizes multiple separate process queues, where each queue is scheduled individually. The Multi-Level Feedback Queue (MLFQ) improves on this concept by dynamically allowing processes to move between queues. Each queue is allocated a different quantum of CPU time, from the first queue to the last. So moving a process between queues is analogous to changing a process’s priority. Thus, ensuring that interactive and I/O bound processes receive CPU time maximizes system performance and overall best use of system resources (CPU, memory, and I/O).