

**UNIVERSIDADE DE SÃO PAULO**

**Escola Politécnica**



**PCS 3732 – Laboratório de Microprocessadores**

**Projeto - Escalonador de Threads MLFQ**

**Professor Bruno Abrantes Basseto**

**Gabriel Zambelli Scalabrini  
João Victor Texeira Degelo  
Luiz Fernando de Almeida Mota**

**Nº USP: 11803715  
Nº USP: 11803479  
Nº USP: 11807790**

**SÃO PAULO, 08/07/2023**

# 1. Introdução

O projeto desenvolvido pelo grupo consiste em um escalonador de threads implementado mesclando a linguagem de programação C e a linguagem de montagem do processador ARMv4. O escalonamento entre as threads foi realizado por meio do MultiLevel Feedback Queue (MLFQ), uma das abordagens mais conhecidas de escalonamento, que será melhor detalhada a seguir. A implementação deste algoritmo foi fortemente baseada no capítulo *Scheduling: The Multi-Level Feedback Queue* do livro *Three Easy Pieces*, presente [neste link](#). Além disso, foi implementada uma interface para o usuário contendo funções muito semelhantes às da biblioteca Pthreads, também da linguagem C. O código é open-source e está localizado no [GitHub](#).

A verificação do funcionamento deste projeto foi realizada por meio tanto do software de emulação e virtualização de código QEMU, como também pela placa de desenvolvimento ARM Evaluator 7T. Devido às limitações do QEMU, não foi possível simular o funcionamento de um temporizador, nem da sua correspondente interrupção IRQ (Interrupt Request), ambos sendo essenciais para o funcionamento do escalonamento por MLFQ. Porém, ele foi importante para testar as primeiras versões do código, que implementaram um escalonamento Round-Robin simples através de chamadas `yield()`, e funções auxiliares mais específicas.

O escalonador desenvolvido neste projeto apresenta ao todo quatro prioridades, de 0 a 3, sendo 0 a mínima (final), e 3 a máxima (inicial). A fila de cada uma das prioridades suporta até 10 threads, entretanto este valor é customizável, alterando os valores `BUFFER_SIZE` e `SCHEDULER_SIZE` no arquivo `buffer.h`. É importante ter em mente, porém, que o tamanho do escalonador é igual à quantidade de prioridades possíveis, e o campo que designa a prioridade de cada thread, na estrutura de dados `tcb_t` no mesmo arquivo, apresenta apenas 2 bits, sendo portanto necessário alterar o tamanho deste valor, de modo a suportar o número total de prioridades.

## 2. Escalonamento por MultiLevel Feedback Queue

O MultiLevel Feedback Queue (MLFQ) é um algoritmo de escalonamento bastante conhecido e utilizado, apresentando uma solução aos problemas que surgem em outros algoritmos de escalonamento, como Round-Robin (RR), e Shortest Task Comes First (STCF ou SJF). Algoritmos como o Priority Round-Robin possibilitam a ocorrência de starvation, que ocorre quando um processo tem sua execução bloqueada por um grande período de tempo, algo que não é desejável. Isto pode ocorrer, por exemplo, se as filas com prioridades

maiores estiverem cheias, resultando na não execução de processos em filas mais baixas. Outro problema que o MLFQ soluciona é um que surge com o algoritmo Shortest Task Comes First. Nele, é necessário o conhecimento prévio do tempo de execução total de cada tarefa, de modo que o algoritmo consiga ordená-las corretamente, entretanto isso não é algo fácil de ser determinado pelo Sistema Operacional. O MLFQ, por sua vez, consegue, dinamicamente, determinar a prioridade dos processos, e por consequência seus tempos de execução, sendo uma boa solução para este problema.

Existem diversas formas de se implementar um MultiLevel Feedback Queue, cada uma apresentando uma estratégia ou solução diferente para os problemas apontados acima. Este projeto aplica o algoritmo de decréscimo e aumento periódico de prioridade (priority decrease and boost-up), que será melhor detalhado a seguir. Um outro exemplo de estratégia é o algoritmo de envelhecimento, que realiza o escalonamento por Round-Robin, porém, à medida que um processo espera na fila sem ser executado, sua prioridade aumenta, desta forma garantindo a sua posterior execução, e solucionando o problema do starving. Esta estratégia foi desenvolvida pelo grupo do Dênio Almeida, Lucas Garcia, e Francisco Mariani, que auxiliaram também neste projeto, propondo soluções e melhorias ao longo das recorrentes conversas. O projeto deles está disponível no [GitHub](#), e recomenda-se fortemente a consulta.

O algoritmo de MLFQ por decréscimo e aumento periódico de prioridade apresenta múltiplas filas de execução, com prioridades decrescentes, e escalona os processos com base na prioridade. Para processos com mesma prioridade, o escalonamento é um Round-Robin simples, e processos com prioridades inferiores são escalonados somente quando não há nenhum processo com prioridade superior. Assim que um novo processo é criado, ele detém a maior prioridade, porém, após a execução por 1 quantum, seja ao ser interrompido pelo temporizador após este período, ou realizar múltiplas execuções e devoluções por meio do yield(), sua prioridade é decrementada. Além disso, após um determinado tempo, todos os processos são movidos à fila de prioridade máxima (boost-up).

Esta abordagem soluciona o problema do starvation, uma vez que garante a execução de um processo em uma prioridade inferior ao realizar, periodicamente, o boost-up, movendo-o para a fila de máxima prioridade. Além disso, este algoritmo melhora a responsividade do sistema às ações do usuário, visto que todo novo processo, neste caso sendo um click do mouse ou teclado, é alocado na fila de máxima prioridade, o que garante a sua rápida execução.

É importante também citar que uma boa prática ao se implementar estratégias como esta é apresentar filas com quantums maiores quanto menor for a prioridade. Em outras palavras, processos na prioridade máxima apresentam um quantum bem menor (p.e. 20ms) do que processos na prioridade mínima (p.e. 400ms). Isto é importante porque processos que consomem bastante processamento costumam ter sua prioridade rapidamente diminuída, por normalmente executarem até serem interrompidos pelo temporizador, porém garante justiça ao reservar um período de tempo (quantum) maior para a sua execução, de modo que facilite o término da tarefa correspondente. Além disso, para um processos variáveis, que alternam entre períodos de rápida execução e períodos de maior processamento, este escalonamento é justo por conta do boost-up periódico de prioridade, que faz com que o processo seja dinamicamente alocado à fila mais adequada para si, isto é, prioridades maiores na rápida execução, e menores no processamento extenso.

Resumindo, o algoritmo de escalonamento MLFQ por decréscimo e aumento periódico de prioridade apresenta as seguintes regras:

1. Se um processo A possuir prioridade maior do que um processo B, então A executa e B não;
2. Se dois ou mais processos apresentam a mesma prioridade, então eles executam em Round-Robin;
3. Todo novo processo criado apresentará a prioridade máxima;
4. Quando o tempo total de execução de um processo superar o de um quantum, a sua prioridade é diminuída.
  - a. Isto ocorre seja em uma única execução (interrompida por temporizador), ou ao longo de múltiplas execuções (retornos sucessivos através de yield);
5. Após um período de tempo T, todos os processos são movidos à máxima prioridade (boost-up).

A estratégia de MLFQ consegue não somente trazer uma maior justiça ao escalonamento de processos, eliminando completamente a ocorrência de starving, como também não demanda de um conhecimento a priori da natureza do processo, como é o caso do escalonamento SJF, determinando-a dinamicamente ao longo da execução. A comprovação de que este é um bom algoritmo está em sua utilização em sistemas operacionais como o BSD Unix, Solaris, e alguns sistemas Windows.

### 3. API para o usuário baseada no Pthreads

Para facilitar a utilização deste escalonador pelo usuário, foi implementada uma API (Application Programming Interface) muito semelhante à da biblioteca Pthreads, uma das mais utilizadas para a criação e gerenciamento de threads. Foram desenvolvidas versões das funções `pthread_create()`, `pthread_join()`, `pthread_exit()`, e também versões do `pthread_mutex_lock()` e `pthread_mutex_unlock()`, baseadas no código fornecido pelo ARM Developer Guide sobre como [implementar um mutex](#). Abaixo estão listadas as funções, e uma breve explicação sobre seu funcionamento:

- **`void thread_create(uint32_t *threadId, void *(*routine)(void *), void *args);`**
  - Pthreads: `int pthread_create (pthread_t *__newthread, const pthread_attr_t *__attr, void *(*__start_routine) (void *), void *__arg);`
  - Cria uma nova thread, adicionando-a na fila de execução de máxima prioridade. Recebe como parâmetro a rotina a ser executada, e os parâmetros a serem passados para esta rotina, e retorna ao chamante o id da thread criada por meio do ponteiro `threadId`, de modo que ele possa ser utilizado, por exemplo, em um `thread_join`.
    - É importante que a rotina receba e retorne um ponteiro para void para que o gerenciador de threads funcione corretamente, caso contrário podem ocorrer problemas imprevisíveis.
- **`void thread_exit(void *returnPointer);`**
  - Pthreads: `void pthread_exit (void *__retval) __attribute__((__noreturn__));`
  - Realiza a destruição da thread atual, salvando o seu retorno, caso válido, em uma estrutura interna e dinamicamente alocada do gerenciador de thread, permitindo que possa ser acessada pela função `thread_join()`.
    - É importante que este ponteiro contendo o retorno da função, se não for nulo, esteja apontando para um endereço dinamicamente alocado (`malloc`), caso contrário o valor retornado poderá ser perdido.
- **`void thread_join(uint32_t threadId, void **threadReturn);`**
  - Pthreads: `int pthread_join (pthread_t __th, void **__thread_return);`
  - Faz com que a thread atual aguarde o término da execução da thread com o identificador passado. Caso esta thread tenha apresentado algum valor de retorno, ele será devolvido pelo ponteiro `threadReturn`.
- **`void thread_mutex_lock(void *mutex);`**
  - Pthreads: `int pthread_mutex_lock (pthread_mutex_t *__mutex);`
  - Realiza o travamento de um mutex simples definido no endereço passado como parâmetro.

- **void thread\_mutex\_unlock(void \*mutex);**
  - Baseada na função: int pthread\_mutex\_unlock (pthread\_mutex\_t \*\_\_mutex);
  - Realiza o destravamento de um mutex simples definido no endereço passado como parâmetro.
- **void thread\_yield(void);**
  - Devolve o controle da execução para o escalonador de threads, que seleciona a próxima a ser despachada.
- **uint8\_t get\_tid(void);**
  - Informa o identificador da thread atual.

Com esta interface de funções torna-se possível desenvolver e testar uma gama muito maior de programas. A figura a seguir mostra uma comparação resumida entre as chamadas às funções criadas pelo grupo, e as funções da biblioteca Pthreads:

<pre> void thread_create(     uint32_t *threadId,     ThreadProperties *threadProperties,     void *(*routine)(void *),     void *args);  void thread_exit(void *returnPointer);  void thread_join(     uint32_t threadId,     void **threadReturn);  void thread_yield(void);  void thread_mutex_lock(void *mutex);  void thread_mutex_unlock(void *mutex); </pre>	<pre> int pthread_create(     pthread_t *__restrict __newthread,     const pthread_attr_t *__restrict __attr,     void *(*__start_routine)(void *),     void *__restrict __arg)     __THROWNL __nonnull((1, 3));  void pthread_exit(void *__retval)     __attribute__((__noreturn__));  int pthread_join(     pthread_t __th,     void **__thread_return);  int pthread_yield(void) __THROW;  int pthread_mutex_lock(pthread_mutex_t *__mutex)     __THROWNL __nonnull((1));  int pthread_mutex_unlock(pthread_mutex_t *__mutex)     __THROWNL __nonnull((1)); </pre>
---	---

**Comparação entre API desenvolvida (esquerda), e API da Pthreads (direita)**

## 4. Teste do funcionamento do escalonador

### Teste 1: Escalonamento entre 2 threads de curta, e 1 thread de longa execução

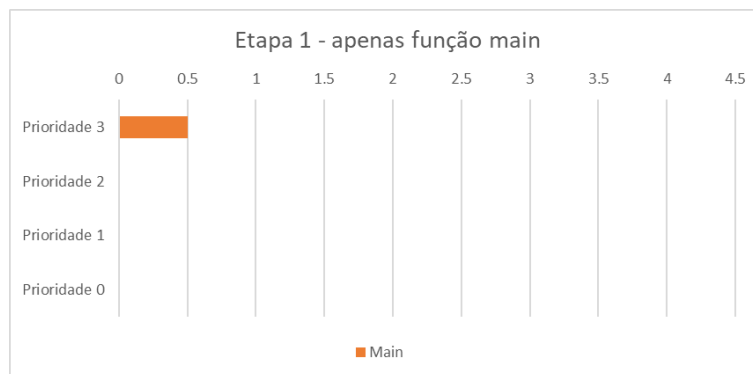
Para validar o correto funcionamento do escalonador MLFQ, foi construído um programa bem simples que utiliza a interface discutida, que está disponível no arquivo main.c no repositório, e no qual é realizada a criação de três threads muito simples. Conforme dito anteriormente, as filas apresentam quantums de 1, 2, 3 e 4 segundos, e o período de boost-up é de 30 segundos, de modo a facilitar a visualização.

As threads 1 e 3 chamam o `thread_yield()` após um tempo curto de execução, simulando um programa voltado a inputs do usuário, e que portanto apresenta um baixo processamento da CPU. É esperado que estas threads permaneçam por algumas iterações nas prioridades superiores, até serem diminuídas após a passagem de 1 quantum.

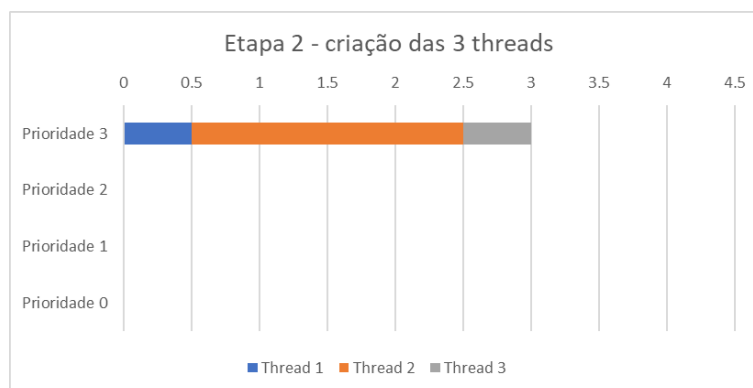
A thread 2, por sua vez, consiste em um loop eterno, sem nenhuma chamada a qualquer outra função, simulando um programa que utiliza grandes quantidades de processamento. Por conta disso, sua execução deverá ser interrompida pelo temporizador, após a passagem de 1 quantum, e sua prioridade deve diminuir.

Abaixo encontram-se diagramas ilustrando cada uma das etapas esperadas para o programa descrito acima, considerando o correto funcionamento do escalonador MLFQ.

Etapa 1: Inicialmente há apenas a thread 0, que contém a função main:

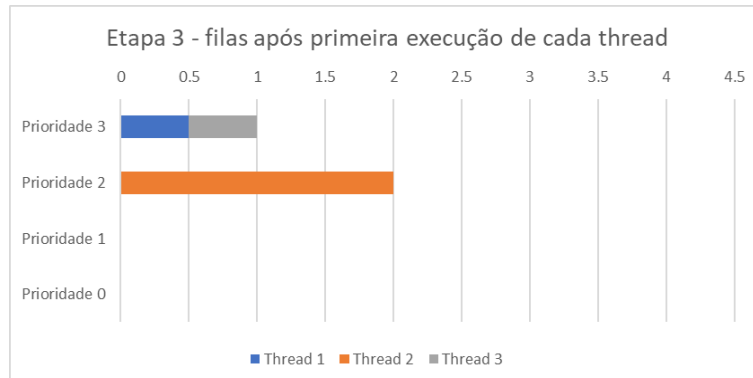


Etapa 2: Após a criação das três threads pela função main, todas elas estarão na fila de máxima prioridade. A thread 0, como terminou sua execução, é removida da fila:

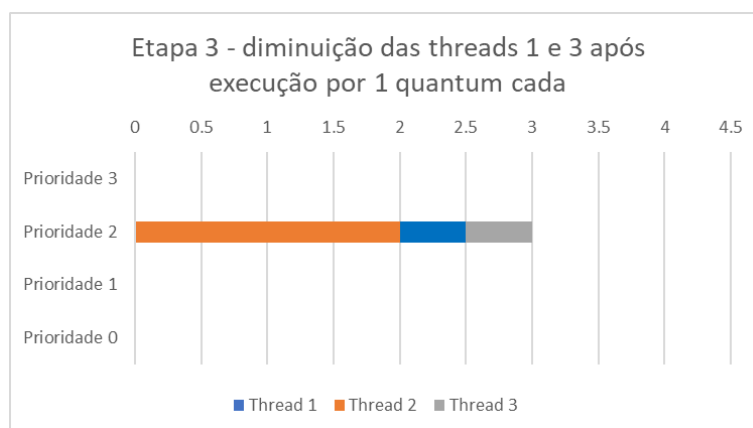


Etapa 3: A thread 1 executa por um curto período de tempo, antes de chamar a função `thread_yield()`. A thread 2 executa até ser interrompida pelo temporizador, e tem sua prioridade diminuída. A thread 3, assim como a 1, executa por um curto período, e devolve

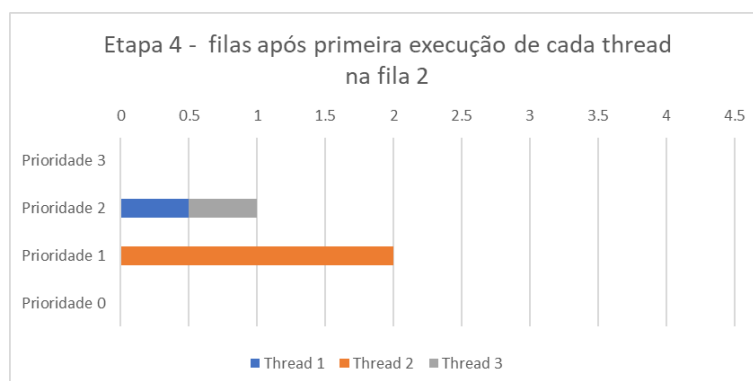
o controle ao escalonador. A figura abaixo mostra as filas após a primeira execução de cada uma das threads:



Etapa 4: As threads 1 e 3 alternam continuamente suas execuções, até estourarem, cada uma, o tempo total de execução nesta fila, determinado pelo quantum. Em seguida, suas prioridades são diminuídas:

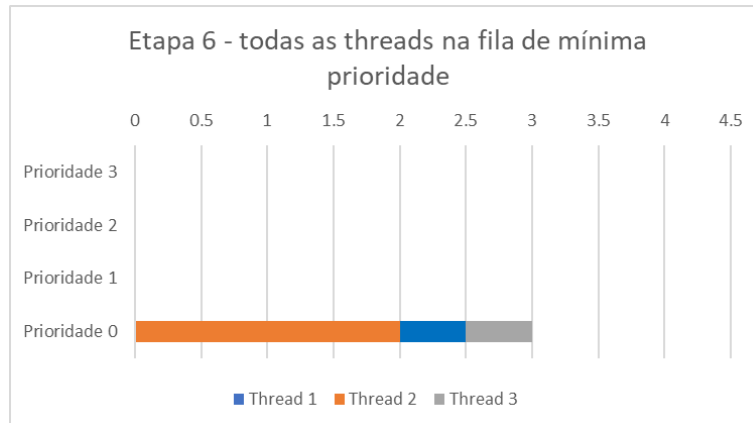


Etapa 5: A thread 2 executa novamente até ser interrompida por timer, o que diminui sua prioridade. Mais uma vez, as threads 1 e 3 alternam continuamente suas execuções, e, após executarem por 1 quantum, são movidas à prioridade inferior

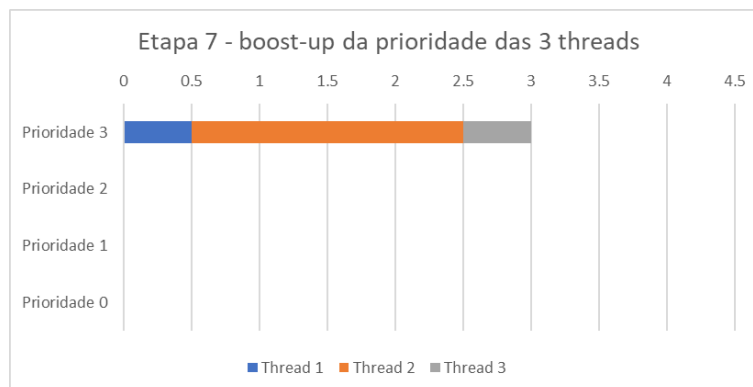




Etapa 6: O passo anterior se repete, até todas as threads se encontrarem na prioridade mínima. Nela, as três threads alternarão continuamente suas execuções.



Etapa 7: Após um determinado período de tempo, ocorre o boost-up, e todas as threads retornam à prioridade máxima, repetindo novamente todo o processo a partir da etapa 3.



No repositório, dentro da pasta `src/test`, o arquivo `mlfq.c` apresenta o código correspondente ao teste acima, que consiste na criação das três threads na função `main`, e o escalonamento delas em seguida.

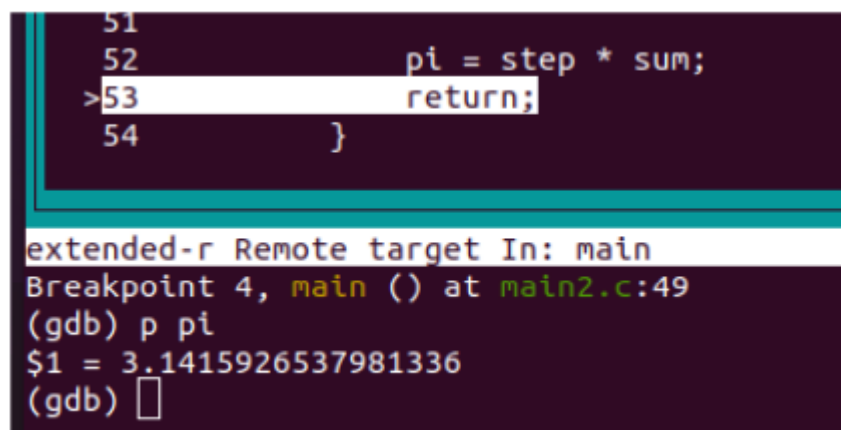
A simulação deste algoritmo foi realizada via a placa de desenvolvimento Evaluator 7T, que apresenta um processador ARMv4. Nela, o display de 7 segmentos mostra o identificador da thread que está sendo executada no momento, e os leds mostram a prioridade desta thread. Para facilitar a verificação, as filas apresentam quântums de 1, 2, 3 e 4 segundos (ao contrário de 20ms, 40ms, etc.), e o período de boost-up foi definido como 30 segundos. Como se observa [neste vídeo](#), o resultado foi bastante compatível com o esperado.

## Teste 2: Cálculo do valor de Pi utilizando multithreading

Na disciplina de Organização e Arquitetura de Computadores, foi analisado um código que realiza o cálculo do valor de Pi através de multithreading, utilizando a biblioteca Pthreads. Dadas as semelhanças entre ambas as APIs, conforme discutido anteriormente, decidiu-se realizar a verificação do funcionamento correto das funções desenvolvidas utilizando uma versão adaptada do arquivo, pi.c, que encontra-se na pasta src/test.

No arquivo original, havia um tratamento dos argumentos passados à função main, entretanto este não será realizado aqui, e portanto as linhas correspondentes foram removidas. Fora isso, as únicas alterações necessárias se resumiram a alterar os tipos de alguns dados, de modo a realizar a chamada correta das funções desenvolvidas.

Após a execução do código, foi obtido o resultado esperado, conforme mostra a figura abaixo:



```
51
52         pi = step * sum;
>53     return;
54     }
```

```
extended-r Remote target In: main
Breakpoint 4, main () at main2.c:49
(gdb) p pi
$1 = 3.1415926537981336
(gdb) □
```

Resultado do cálculo de Pi por multithreading

## 5. Conclusão

O desenvolvimento e implementação do escalonador MultiLevel Feedback Queue (MLFQ) utilizando o algoritmo de diminuição de prioridade e boost-up periódico possibilitou o aprendizado dos principais conceitos relacionados ao gerenciamento de processos em sistemas computacionais. Os resultados obtidos comprovaram as vantagens deste algoritmo em relação aos outros algoritmos comentados, como o Round-Robin e o Shortest Task Comes First, resolvendo os problemas de starving, e a determinação dinâmica da prioridade e do tempo de execução de cada processo.