Escola Politécnica da USP
Departamento de Engenharia de Computação e Sistemas Digitais
PCS3732 – Laboratório de Processadores
Prof. Bruno Abrantes Basseto

Simple MFQS
Escalonador de Prioridades Dinâmicas em Processador ARMv4

Integrates	**Nº USP**
Dênio Araujo de Almeida	10309013
Francisco Cavalheiro Mariani	11803701
Lucas von Ancken Garcia	11257592

Universidade de São Paulo
18 de agosto de 2023

Sumário

1. Resumo

Esse relatório detalha o desenvolvimento do projeto quadrimestral da disciplina de Laboratório de Processadores (PCS3732). São apresentados os resultados do projeto Simple MFQS, o qual objetiva oferecer uma implementação básica de um algoritmo de escalonamento dinâmico de prioridades em um processador ARM.

O mecanismo de escalonamento implementado pela equipe para essa finalidade foi o algoritmo MFQS (*Multi-Level Feedback Queue Scheduler*), cujas características gerais são apresentadas na Seção 2.2.

Para uma demonstração do escalonador desenvolvido, utilizou-se a placa Evaluator7T, disponibilizada aos integrantes da equipe no oferecimento da disciplina. Tal placa, desenvolvida para fins didáticos, dispõe de um processador ARMv4 , e unidades externas que permitem funcionalidades como controle de interrupções e acesso aos dispositivos de I/O. Especificações referentes a Evaluator-7T podem ser consultadas em [1].

O resultado final da codificação do escalonador proposto pode ser consultado no repositório GitHub do projeto [2]. Detalhes quanto a essa implementação são dados na Seção 3. A Seção 4 expõe brevemente possíveis melhorias a serem feitas futuramente, e a Seção 5 apresenta as conclusões sobre o projeto.

2. Proposta do Projeto

A seguir, descreve-se a proposta do projeto, comentando sobre os principais objetivos, sobre o escalonador escolhido e sobre as decisões de projeto e hipóteses simplificadoras.

2.1. Principais Objetivos

O objetivo do projeto consiste na implementação de um escalonador de threads com prioridades que atenda aos seguintes requisitos funcionais:

1. **Controlar o tempo de execução das threads com interrupções periódicas**: o escalonador deve ser capaz de interromper periodicamente a execução de uma thread (ou seja, preempta-lá), de forma a garantir um nível de *fairness* entre todas as threads.

2. **Permitir um sistema de prioridades dinâmico entre threads:** o escalonador deveria ser capaz de suportar uma hierarquia de prioridades (i.e. certas threads devem ser executadas na frente de outras), e a prioridade de uma thread deve poder mudar com o tempo.

3. **Evitar que threads sofram starvation**: considerando o sistema de prioridades, é possível que threads de prioridades baixas sejam raramente executadas, ou sofram de starvation. O escalonador deveria ter um mecanismo para evitar isso.

4. **Permitir que threads concedam voluntariamente tempo de execução (yield)**: uma outra forma de threads deixarem de usar a CPU é através da sua concessão voluntária de tempo de execução. Imaginou-se esse requisito para simular threads que na vida real correspondem aos de operações I/O, que usam a CPU em bursts;

5. **Permitir que threads sejam finalizadas e retiradas de execução (halt):** mais uma vez, imaginando um caso mais real de sistema operacional, em algum momento as threads são finalizadas. Assim, eles devem ter algum mecanismo para serem terminados.

Considerando esses requisitos, optou-se por implementar o escalonador MFQS. Esse escalonador é usado por inúmeros sistemas operacionais modernos, e também é razoavelmente simples de ser implementado, mesmo que de forma embrionária.

2.2. Escalonador MFQS

O algoritmo MFQS (Multilevel Feedback Queue Scheduler), faz a alocação de processos em uma estrutura de fila multinível. Nesta estrutura, cada nível agrupa processos com mesma prioridade de execução. Um esquema dessa fila pode ser visto na Figura 1.

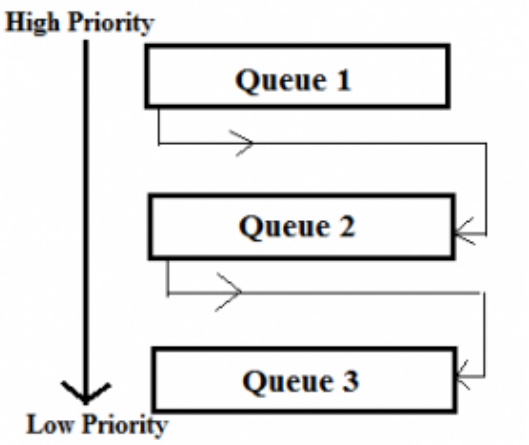

Figura 1 - Esquema da fila multinível do escalonador MFQS

O *feedback*, que está no nome desse escalonador, se refere à mudança dinâmica da prioridade dos processos de acordo com o comportamento deles ao longo do tempo. Processos que permanecem por muito tempo em uma certa prioridade sofrem *downgrade* para uma prioridade menor, e processos sofrendo de starvation ganham um *upgrade* para uma prioridade maior.

Em detalhes, as regras do algoritmo são:

1. A thread a ser executada deve ser aquela na frente da fila não vazia mais prioritária.

2. Uma thread pode ser executada por até certo tempo, determinado em cada fila. Caso ela exceda esse tempo, ela é retirada da fila atual e colocada na seguinte fila menos prioritária.

2a. Se uma thread excede o tempo de execução da última fila, ela é colocada no final dessa mesma fila. Ou seja, a última fila segue uma lógica de round-robin.

3. Se uma thread permanecer muito tempo em uma dada fila sem ser executada, ela é retirada da fila atual e colocada na seguinte fila mais prioritária. Esse processo de **aging** acontece em todas as threads que não estão sendo executadas no momento.

4. Uma thread em execução pode ceder o seu tempo de execução remanescente em uma certa fila. Nesse caso, elas retornam para o final da mesma fila, sem perder prioridade.

Desse modo, o MFQS busca alcançar dois principais objetivos:

1. **Tornar o SO mais responsivo**: manter a prioridade de processos que abandonam a execução em CPU para operações de I/O (processos iterativos);
2. **Diminuir o tempo médio de turnaround**: procura-se priorizar a execução de processos mais curtos e penalizar processos mais longos.

Para um entendimento mais aprofundado do funcionamento deste algoritmo, bem como de seus objetivos, recomenda-se a leitura do documento disponibilizado em [3].

2.3 Decisões de Projeto e Hipóteses Simplificadoras

Devido à complexidade envolvida no desenvolvimento de funcionalidades de manipulação de **processos reais** em um sistema operacional, optou-se por simplificar o funcionamento do escalonador MFQS deste projeto para um **mecanismo de escalonamento de *threads***, ao invés de processos propriamente dito.

Além disso, escolheu-se por desenvolver uma versão do algoritmo MFQS que utilize o (já mencionado) mecanismo de aging. Essa, no entanto, não é a única maneira de evitar o *starvation* de threads. Uma segunda estratégia (*boost-up*) envolve redefinir todas as prioridades depois de um certo período de tempo, de forma que todas as threads sejam colocadas na fila mais prioritária. Esse mecanismo tem ainda o benefício de evitar deadlocks de recursos entre threads, o que não é verdade para o mecanismo de aging.

Uma versão de um escalonamento MFQS usando *boost-up* foi desenvolvida em um projeto de escopo semelhante, conduzido por Gabriel Zambelli, João Victor Degelo, e Luiz Fernando Motta [4]. Recomenda-se fortemente a leitura do material criado por eles.

3. Implementação da Solução

Nesta seção, são apresentados os aspectos mais relevantes sobre a implementação da solução do projeto. Detalhes adicionais a respeito do funcionamento do sistema desenvolvido podem ser obtidas a partir da consulta ao código-fonte do projeto, disponibilizado em [2].

3.1. Interrupções

O algoritmo MFQS é tanto baseado na preempção forçada de threads, quanto na concessão voluntária do processador por parte da thread em execução (no caso de um *yield*, por exemplo). Esse primeiro caso implica na criação de interrupções periódicas, que interrompam a execução da thread atual e verifiquem se uma outra thread deve ser escalonada no seu lugar. Para tanto, utilizou-se um dos timers da placa Evaluator-7T.

No início do código, é feita a configuração necessária para um serviço de preempção de threads. O timer da placa é ligado, com um *reset value* de TIME_LIMIT, equivalente a 25 milhões de ciclos de clock da placa, ou então 0.5 segundos. Esse período de 0.5 segundos corresponde a um *quantum*, e é a unidade de tempo fundamental do escalonador.

O vetor de interrupções da placa também é configurado de forma que o timer gere uma interrupção IRQ. No tratamento dessas interrupções IRQ, rotinas do escalonador e do despachador são executadas, de forma que a thread atual pode (ou não) ser substituída por uma nova thread de acordo com o algoritmo MFQS.

3.2. Thread Control Block

Na visão do escalonador e do despachador, cada thread é representada pelo sua Thread Control Block (ou TCB). A TCB deve conter todo o contexto necessário para a correta execução do programa (como os valores dos registradores), assim como informações usadas pelo próprio escalonador para determinar *quando* a thread deve ser despachada.

No caso específico do MFQS, a TCB deve conter ao menos duas informações essenciais:

1. **Slots de execução:** quantos *quanta* a thread pode ainda executar em uma determinada prioridade, antes de sofrer downgrade para uma prioridade inferior.
2. **Age:** a quantos *quanta* a thread não é executada na sua prioridade atual. Isso será usado para a thread ganhar um *upgrade* para uma prioridade superior.

Assim, na implementação do MFQS feita para esse projeto, optou-se por usar uma *struct* do C para representar cada TCB, com todos os seus atributos. Essa *struct* pode ser vista na Figura 2.

```c
typedef struct {
    uint32_t regs[17];   // contexto (17 registradores)
    uint32_t tid;        // identificador da thread
    uint32_t priority;   // prioridade atual da thread
    uint32_t exc_slots;  // número de execuções restantes
    uint32_t age;        // tempo de espera da thread na fila (aging)
    uint32_t cpu_time;   // quantidade total de execuções de slots de tempo da thread
} tcb_t;
```

Figura 2 - struct usada para representar as TCBs das threads

Percebe-se que, além dos registradores, do *exc_slots* e do *age*, a TCB também outras informações: *thread id* (ou TID) é um identificador único da thread, *priority* corresponde à prioridade atual da thread (i.e. em qual fila ela está), e *cpu_time* é um atributo de depuração, que guarda por quantos *quanta* a thread executou ao todo, em todas as prioridades.

3.3. Fila Multinível

Com o formato da TCB definido, fez-se necessário implementar a principal estrutura de dados usada no algoritmo de escalonamento do MFQS: a fila multinível. Essa é responsável por determinar a próxima thread a ser despachada, a prioridade relativa entre diferentes threads, e parâmetros que ditam quando uma *thread* deve perder ou ganhar prioridade.
A fila multinível deve ser composta por ao menos duas filas.
Assim como a TCB, é necessário que cada fila tenha ao menos dois atributos essenciais:

3. **Quanta limit:** o número máximo de quanta de execução permitido a uma thread na fila, antes que ela sofra um downgrade para uma prioridade inferior.
4. **Age limit:** o número máximo permitido para o *age* de uma thread nessa fila, antes que ela ganhe um upgrade para uma prioridade superior.

Geralmente, esses atributos devem mudar de acordo com a prioridade da fila. Para uma boa implementação do algoritmo, como explicado em [3], filas mais prioritárias devem ter um quanta_limit menor, e age limit deve seguir uma proporcionalidade com o quanta_limit.

Nesse projeto, optou-se por implementar cada fila como uma lista duplamente ligada de TCBs. Essa estrutura de dados foi escolhida, em vez de um vetor, pois é mais flexível para retirar nós arbitrários da estrutura, o que é necessário durante o término de threads. A *struct* de cada fila tem um atributo *head* que aponta para a cabeça da fila, e os atributos quanta_limit e age_limit, como exibido na Figura 3.

```c
typedef struct {
    node_t* head;   // cabeca da fila
    uint32_t quanta_limit;
    uint32_t age_limit;
} queue_t;
```

Figura 3 - struct usada para representar cada fila dentro da fila multinível do MFQS

Já a fila multinível foi implementada como uma *struct* que contém um vetor de filas, e um atributo *next_thread* que aponta para a TCB da próxima thread a ser despachada, como pode ser visto na Figura 4. Especificamente, decidiu-se que há 3 filas no vetor de filas.

```c
typedef struct {
    queue_t* queues[NUM_OF_QUEUES];   // filas com prioridades
    tcb_t* next_thread;   // proxima thread a ser despachada
} multiqueue_t;
```

Figura 4 - struct usada para a fila multinível do MFQS

3.4. Escalonador MFQS

O escalonador MFQS implementado é apenas um conjunto de funções - invocadas periodicamente, ou durante uma chamada de sistema - que atualizam as estruturas de dados já mencionadas, de forma a seguir as regras do algoritmo indicadas na Seção 2.2. O escalonador também controla a atualização da variável global *current_tcb*, um ponteiro para a thread atualmente em execução.

Essa suíte de funções pode ser encontrada nos arquivos scheduler.c e scheduler.h.

Em alto nível, a implementação do escalonador pode ser condensada em quatro principais responsabilidades:

1. **Atualização das informações nas TCBs**, como os slots de execução da thread atual, e o age de todas as threads que não estão sendo executadas.
1. **Ajuste Dinâmico de Prioridade das Threads:** isso envolve o movimento das threads entre os três níveis de prioridade existentes, de acordo com o estado de cada thread.
2. **Escolha da Próxima Thread a ser Escalonada:** isso implica a busca pela thread localizada na parte frontal da fila de mais alta prioridade que não esteja vazia.
3. **Adição e remoção de threads da Fila Multinível**, quando uma thread é criada ou terminada, respectivamente.

Sempre que um quantum termina (i.e. sempre que uma interrupção IRQ é gerada pelo timer), todas as TCBs são atualizadas: a thread em execução tem o seu *exc_slots* decrescido, e todas as outras threads tem o seu *age* acrescido. Baseado nisso, as threads são elevadas para uma prioridade superior (se o age delas ultrapassa o age_limit da fila atual), ou rebaixada para uma prioridade inferior (se o exc_slots dela naquela fila chegou a 0).

A função em alto nível que realiza esse processo pode ser vista na Figura 5 abaixo:

```
int mfqs_update_threads(int yield) {
    current_tcb→cpu_time++; // Incrementa o tempo de cpu (qtde total de execuções de slots de tempo)

    // Tratamento para caso em que não há mais threads a serem executadas
    if (!multi_queue.next_thread) {
        update_next_thread(&multi_queue);
        if (!multi_queue.next_thread) return 0;
        else return 1;
    }

    int remaining_slots = --current_tcb→exc_slots;

    if (remaining_slots && yield) {
        // Mantém a thread na msm fila, caso um yield tenha sido executado
        keep_thread_on_same_queue(current_tcb, &multi_queue);
    }
    else if (!remaining_slots) {
        //  Diminui a prioridade da thread que acabou de ser executada
        downgrade_thread(current_tcb, &multi_queue);
    }

    // Faz o aging de todas as threads, exceto a que executou agora
    age_all_threads(&multi_queue);
    current_tcb→age = 0;

    // Atualiza a proxima thread a ser executada
    update_next_thread(&multi_queue);

    // Checa se precisa salvar contexto
    return (current_tcb == multi_queue.next_thread) ? 0 : 1;
}
```

Figura 5 - função mfqs_update_threads(), responsável por atualizar as TCBs de todas as threds no final de cada quanta.

Logo depois, checa-se também se a próxima thread a ser executada de acordo com o MFQS (isso é, a primeira thread da primeira fila não vazia) mudou. Se sim, é necessário que o despachador salve o contexto da thread atual, peça para o escalonador atualizar a variável current_tcb, e coloque o contexto da nova thread para ser executado.

3.5. Despachador

O despachador implementado tem apenas duas funções simples:

1. Salvar o contexto da thread em execução, quando sabe-se que ela deve ser preemptada em favor de outra thread.
2. Recuperar o contexto da próxima thread a ser executada, elencada pelo escalonador.

Ambas essas funções requerem código explicitamente em assembly, para que seja possível controlar o conteúdo dos registradores do sistema diretamente. O funcionamento do escalonador pode ser encontrado no arquivo kernel.s.

Um ponto importante a ser enfatizado é o valor armazenado na posição da TCB correspondente ao *Program Counter* durante esse processo.

Quando a origem da troca de contexto decorre de uma interrupção IRQ, como ao final de um quantum, o valor salvo na posição da TCB correspondente ao PC é "LR - 4". Porém, quando uma troca de contexto é desencadeada por uma interrupção SWI, como é o caso da chamada de sistema *yield* (Seção 3.9.2), o valor salvo nessa posição é dado diretamente pelo valor do LR. Essa discrepância é devida às características do pipeline do processador ARMv4.

3.6. Inicialização do Sistema

Assim que a placa Evaluator-7T é ligada, executa-se uma função de *boot* responsável por criar a estrutura de fila multinível mencionada anteriormente na Seção 3.3. Nesse processo, são fixados os parâmetros de cada fila: o limite de *quanta* de execução, e o limite de *age*.

Além disso, para fins de demonstração, a estrutura de multinível também é inicializada com algumas *threads,* dotadas de prioridades escolhidas arbitrariamente. Para tanto, são criadas em tempo de execução, por intermédio da função *malloc*, as TCBs das threads a serem executadas. A criação com alocação dinâmica de memória dessas estruturas implica, consequentemente, que elas são armazenadas no *heap*.

3.7. Criação de Threads

O processo previamente mencionado de criação das threads é desempenhado por meio de chamadas da função *create_tcb()*, cuja implementação é apresentada na Figura 5 a seguir.

```c
// Função para criar um elemento tcb_t
tcb_t* create_tcb(uint32_t tid, uint32_t priority, uint32_t exc_slots, uint32_t age, uint32_t entry_point) {
    tcb_t* new_tcb = (tcb_t*)malloc(sizeof(tcb_t));
    uint8_t* stack = (uint8_t*)malloc(sizeof(uint8_t) * 4096);

    for (int i = 0; i < 17; ++i) {
        new_tcb->regs[i] = 0; // Inicializa os registradores
    }

    new_tcb->regs[13] = (uint32_t)(stack + 4096);
    new_tcb->regs[14] = (uint32_t)halt;
    new_tcb->regs[15] = entry_point;

    new_tcb->tid = tid;
    new_tcb->priority = priority;
    new_tcb->exc_slots = exc_slots;
    new_tcb->age = age;
    new_tcb->cpu_time = 0;

    return new_tcb;
}
```

Figura 6 - Implementação da função create_tcb()

Como aspectos mais relevantes de implementação dessa função, merecem destaque:

1. Definição da função a ser executada pela thread: O argumento "uint32_t entry_point" recebe o endereço da função a ser executada pela thread e repassá-lo como valor inicial Program Counter na TCB (registrador R14 no ARMv4).

2. Inicialização da *stack* de execução: Conforme pode ser notado no código apresentado, cria-se dinamicamente, para cada thread, uma pilha de execução no *heap*. Tal pilha é criada com 4096 posições de memória. O último endereço desse fragmento de memória define o valor inicial para o registrador *stack pointer* (registrador R13 no ARMv4).

3. Todas as threads criadas tem o valor do registrador *lr* apontando para a função halt() (Seção 3.9.1). Isso garante que, ao fim da função da thread, ela será terminada e excluída do escalonamento.

3.8. Thread de Sistema Operacional

Um caso interessante de considerar no funcionamento do MFQS é o que deve acontecer quando não há mais threads a serem escalonadas; ou seja, a fila multinível (Seção 3.3) está completamente vazia. Nesse caso, optou-se por implementar uma thread especial para ser executada nesse cenário, chamada de *thread de sistema operacional*.

Essa thread é criada na inicialização do sistema (Seção 3.6), junto com as filas do algoritmo. Caso, ao final de algum quanta, a fila multinível esteja completamente vazia, a variável *current_tcb* é atualizada para essa thread, e o despachador carrega o contexto dela.

Semelhantemente às outras threads, essa thread de SO também tem sua execução interrompida ao final de cada *quanta*. No entanto, diferentemente das outras threads, ela não participa do processo de escalonamento; ela nunca faz parte da fila multinível.

Dessa forma, garante-se dois objetivos fundamentais. Primeiramente, o processador executa alguma tarefa arbitrária enquanto checa periodicamente se há novas threads na fila; de fato, essa estratégia de *thread de SO* é análoga a forçar o processador a fazer um busy wait do escalonamento. Por esse motivo, o programa a ser executado pela thread de SO pode ser algo extremamente simples, como um loop infinito sem instruções.

Segundamente, como a thread de SO não participa das estruturas de dados do escalonador, ela é apenas executada se a fila multinível estiver vazia. Semelhantemente, assim que uma nova thread surgir na fila multinível, ela obrigatoriamente para de ser executada.

3.9. Chamadas de sistema

De modo a permitir a interação entre o "sistema operacional" da solução disposta e os programas de usuário a serem executados nas threads, criou-se uma pequena suíte de funções que funcionam como chamadas de sistema.

Essas funções disparam interrupções SWI que, durante o seu tratamento, vão executar tarefas diretamente relacionadas às estruturas de dados do sistema operacional. Exemplos dessas chamadas podem ser vistos abaixo:

1. **getpid( )**: retorna o identificador *TID* da thread que faz a chamada;
2. **get_priority( )**: retorna a prioridade da *thread* que faz a chamada, podendo ser 0, 1 ou 2. Quanto maior o valor numérico da prioridade, menor a prioridade;
3. **get_cpu_time( )**: retorna número total de quanta pelo qual a thread já foi executada

Há ainda duas chamadas de sistema particularmente importantes que devem ser mencionadas: halt() e yield().

3.9.1. Finalização de Threads: halt()

A chamada de sistema halt() é chamada para terminar a execução da thread que a chamar. Essa chamada pode ser invocada explicitamente no programa de uma thread. Além disso, toda thread é criada com a posição da função halt() no seu LR, de forma que ao realizar um *return* ao fim da sua execução, a thread é auto-deletada.

Assim como as outras chamadas de sistema, halt() apenas invocará uma interrupção SWI, como pode ser visto na Figura 7.

```
// Chamada de sistema halt()
// Usada para terminar a execucao de uma thread
void __attribute__((naked)) halt(void) {
      asm volatile("mov r0, #5 \n\t"
                   "swi #0     \n\t");
}
```

Figura 7 - Implementação da função halt(), evidenciando a instrução swi.

O tratamento dessa SWI vai retirar a thread que chamou a função halt() (i.e. a thread atual) da fila multinível usada pelo escalonador. Dessa forma, garante-se que ela não é mais escalonada, efetivamente terminando a thread.

Como passo final, o tratamento dessa SWI deve chamar o escalonador para atualizar a fila multinível, e o despachador para carregar o contexto da próxima thread a ser executada.

3.9.2. Concessão de Tempo de Execução: yield()

Um aspecto do escalonamento MFQS é que a thread em execução pode voluntariamente ceder o controle do processador, e permitir que a thread seguinte execute. Isso é particularmente útil se a thread está "bloqueada", esperando por algum recurso indisponível no momento. Nesse caso, a thread que cedeu o seu tempo de execução não deve perder prioridade, e deve ser colocada ao fim da fila que já estava.

No entanto, mesmo que a thread tenha sido preemptada em favor de outra, os seus slots de execução não são reiniciados; ou seja, a thread pode executar naquela fila por apenas o número de quantas que ela cedeu. Esse mecanismo é importante para evitar que uma thread explore a concessão para se manter indefinidamente em uma mesma prioridade, o que poderia prejudicar a execução de outras *threads*.

Uma thread pode ceder o controle do processador usando a chamada de sistema *yield()*. Essa chamada invocará uma interrupção SWI que, quando tratada, vai atualizar a fila da thread, e colocá-la no final da mesma. Além disso, o tratamento SWI chamará o despachador para carregar o contexto da próxima thread a ser executada.

Finalmente, para fins de demonstração do projeto, escolheu-se por também desenvolver uma rotina que simule uma chamada de yield() assim que uma interrupção de botão da Evaluator-7T é gerada.

Para tanto, foi necessário implementar uma rotina em *assembly* que realiza o tratamento de interrupções de botão geradas na placa Evaluator-7T para essa finalidade. Nela, as mesmas funções chamadas para implementação do serviço da chamada de sistema *yield* também são utilizadas.

3.10 Manipulação dos Dispositivos de I/O

Para a configuração das interrupções IRQ geradas pelo timer, envolvidas na temporização do escalonamento, e para o controle dos valores mostrados nos LEDs e displays, destaca-se que também foram implementadas rotinas em C específicas para o cumprimento dessas responsabilidades. Em ambos os casos, tornou-se necessário utilizar ponteiros para o acesso às posições de memória que mapeiam os registradores de controle da placa Evaluator-7T.

4. Testes e Resultados

Após desenvolver o escalonador MFQS, foi necessário validar o seu funcionamento. Para isso, precisou-se de uma maneira de checar a correta execução de várias threads, na sequência prevista pelo algoritmo do MFQS. Felizmente, a placa Evaluator-7T tem alguns dispositivos de I/O que foram úteis para esse propósito.

Durante a etapa de inicialização descrita na Seção 3.6, são criadas várias threads a serem executadas. Todas essas threads tem exatamente o mesmo código, que pode ser visto abaixo na Figura 8:

```
int user_thread(void) {
    int pid = getpid();
    int time_to_halt = 8;
    int cpu_time = 0;

    for(;;) {
        int priority = get_priority();
        updateLed(priority);
        updateDisplay(pid);

        cpu_time = get_cpu_time();

        if (cpu_time > time_to_halt) {
            halt();
        }
    }
}
```

Figura 8 - Implementação da função user_thread(), usada para testar o escalonador.

Nesse código, é feito uso de rotinas que permitem o acesso aos *displays* e LEDs da placa Evaluator-7T. Tais dispositivos externos são usados para exibir, respectivamente, o identificador e a prioridade da *thread* em execução. Dessa forma, torna-se possível visualizar, ao longo da execução, a troca de contexto realizada pelo algoritmo MFQS. Uma demonstração da execução do Simple MFQS pode ser encontrada nesse link.

Percebe-se que, de fato, o escalonador funcionou como esperado, realizando o manejamento das filas de acordo com o estado de cada TCB. Também nota-se que a função de halt() terminou os processos devidamente, e a thread de SO (que faz o L no display de 7 segmentos) foi escalonada ao fim de todas as threads.

5. Possíveis Melhorias

Ao longo do desenvolvimento do projeto, muitas funcionalidades foram discutidas e até rascunhadas mas, devido a restrições de tempo, não foram finalizadas. Reconhece-se que elas são possíveis melhorias ao projeto, caso uma versão melhorada dele viesse a ser implementada. Dentre elas, mencionamos:

- **A chamada de sistema fork()**, que faz uma cópia da thread em execução. A implementação dessa função requer mudanças na estrutura das TCBs, assim como da maneira que o stack de cada thread funciona (mais detalhes aqui). Essas mudanças já foram feitas em uma branch do repositório, mas não foram extensivamente testadas.
- **Um escalonador que contemple threads bloqueadas** por motivos de espera por recursos do sistema. Quando uma thread se tornasse bloqueada, ela seria retirada da fila multinível, e armazenada em uma segunda estrutura de dados com esse objetivo. Posteriormente, quando o recurso se tornasse disponível, ela retornaria para o escalonamento multinível.

6. Conclusões

Conclui-se que foi possível atingir o objetivo principal do projeto: o desenvolvimento e validação de uma prova de conceito de um escalonador MFQS. Mesmo assim, ainda há muito espaço para melhoria, seja na implementação do escalonador em si, seja na adição de mais *features* relacionadas diretamente (ou indiretamente) ao tópico de escalonamento de threads. De forma geral, esse projeto possibilitou ao grupo um entendimento prático de questões fundamentais para o funcionamento de sistemas operacionais.

Referências

[1] ARM DEVELOPER HUB. Documentation – Arm Developer. Disponível em: https://developer.arm.com/documentation/dui0134/latest/
Acesso em: 18 ago. 2023.

[2] Simple MFQS. Disponível em:
https://github.com/LucasVon0645/PCS3732-PrioritySchedulingProject
Acesso em: 18 ago. 2023.

[3] H. ARPACI-DUSSEAU, R. Scheduling: The Multi-Level Feedback Queue. [s.l.] UW-Madison, [s.d.]. Disponível em:
https://pages.cs.wisc.edu/~remzi/OSTEP/cpu-sched-mlfq.pdf
Acesso em: 18 ago. 2023.

[4] Projeto Lab-Proc. Disponível em:
https://github.com/GabZamba/Projeto-LabProc
Acesso em: 18 ago. 2023.