Jorge Bandeo e Matheus Henrique Kovalski

Atividade M2

Itajaí - SC

2022

Jorge Bandeo e Matheus Henrique Kovalski

Atividade M2

Relatório técnico apresentado como requisito parcial para média 2 na disciplina Sistemas Operacionais, na Universidade do Vale do Itajaí.

Universidade do Vale do Itajaí - UNIVALI

Faculdade de Eng. de Computação

Orientador: Prof. Felipe Viel

Itajaí - SC

2022

Sumário

1 INTRODUÇÃO 3
1.1 Enunciado do trabalho 3
1.2 Análise do que será realizado 4

2 CONSTRUÇÃO DA LOGICA 6
2.1 Round-Robin (RR) 6
2.2 Round-Robin com prioridade (RR_p) 7
2.3 First Come, First Served (FCFS) 8

3 CÓDIGOS 9
3.1 Round-Robin (RR) 9
3.2 Round-Robin com prioridade (RR_p) 10
3.3 First Come, First Served (FCFS) 12

4 RESULTADOS 14

1 Introdução

LINK GITHUB

1.1 Enunciado do trabalho

Desenvolvimento de trabalhos da matéria de Sistemas Operacionais. Projeto 1 Para consolidar o aprendizado sobre os escalonadores, você deverá implementar dois algoritmos de escalonadores de tarefas (taks) estudos em aula. Os escalonadores são o Round-Robin (RR), o Round-Robin com prioridade (RR_p) e FCFS (First Come, First Served). Para essa implementação, são disponibilizados os seguintes arquivos (Link Github):

1. driver (.c) – implementa a função main(), a qual lê os arquivos com as informações das tasks de um arquivo de teste (fornecido), adiciona as tasks na lista (fila de aptos) e chama o escalonador. Esse arquivo já está pronto, mas pode ser completado.

2. CPU (.c e .h) – esses arquivos implementam o monitor de execução, tendo como única funcionalidade exibir (via print) qual task está em execução no momento. Esse arquivo já está pronto, mas pode ser completado.

3. list (.c e .h) - esses arquivos são responsáveis por implementar a estrutura de uma lista encadeada e as funções para inserção, deletar e percorrer a lista criada. Esse arquivo já está pronto, mas pode ser completado.

4. task (.h) – esse arquivo é responsável por descrever a estrutura da task a ser manipulada pelo escalonador (onde as informações são armazenadas ao serem lidas do arquivo). Esse arquivo já está pronto, mas pode ser completado.

5. scheduler (.h) – esse arquivo é responsável por implementar as funções de adicionar as task na lista (função add()) e realizar o escalonamento (schedule()). Esse arquivo deve ser o implementado por vocês. Você irá gerar as duas versões do algoritmo de escalonamento, RR e RR_p, em projetos diferentes, além do FCFS.

6. Você poderá modificar os arquivos que já estão prontos, como o de manipulação de listas encadeada, para poder se adequar melhor, mas não pode perder a essência da implementação disponibilizada. Algumas informações sobre a implementação:

7. Sobre o RR_p, a prioridade só será levada em conta na escolha de qual task deve ser executada caso haja duas (ou mais) tasks para serem executadas no momento. Em caso de prioridades iguais, pode implementar o seu critério, como quem é a primeira da lista (por exemplo). Nesse trabalho, considere a maior prioridade como sendo 1.

8. Você deve considerar mais filas de aptos para diferentes prioridades. Acrescente duas taks para cada prioridade criada.

9. A contagem de tempo (slice) pode ser implementada como desejar, como com bibliotecas ou por uma variável global compartilhada.

10. Lembre-se que a lista de task (fila de aptos) deve ser mantida "viva" durante toda a execução. Sendo assim, é recomendado implementar ela em uma biblioteca (podendo ser dentro da próprio schedulers.h) e compartilhar como uma variável global.

11. Novamente, você pode modificar os arquivos, principalmente o "list", mas sem deixar a essência original deles comprometida. Porém, esse arquivo auxilia na criação de prioridade, já que funciona no modelo pilha.

12. Para usar o Makefile, gere um arquivo schedule_rr.c, schedule_rrp.c e schedule_fcfs.c que incluem a biblioteca schedulers.h (pode modificar o nome da biblioteca também). Caso não queira usar o Makefile, pode trabalhar com a IDE de preferência ou compilar via terminal.

13. Utilize um slice de no máximo 30 unidades de tempo.

1.2 Análise do que será realizado

1. Implementação de dois algoritmos de escalonadores de tarefas (tasks):

- Round-Robin (RR)
- Round-Robin com prioridade (RR_p)
- First Come, First Served (FCFS)

2. Utilização de arquivos disponibilizados no GitHub para auxiliar na implementação dos escalonadores:

- `driver.c`: implementa a função `main()` que lê os arquivos com as informações das tasks, adiciona as tasks na lista (fila de aptos) e chama o escalonador.
- `CPU.c` e `CPU.h`: implementam o monitor de execução, exibindo qual task está em execução no momento.
- `list.c` e `list.h`: implementam a estrutura de uma lista encadeada e as funções para inserção, exclusão e percorrimento da lista.
- `task.h`: descreve a estrutura das tasks a serem manipuladas pelo escalonador.

- `scheduler.h`: implementa as funções para adicionar as tasks na lista e realizar o escalonamento. Esse arquivo deve ser implementado pelos alunos.

Com relação à lógica dos três objetivos:

3. Implementação dos algoritmos de escalonamento:

O algoritmo Round-Robin (RR) consiste em atribuir um *quantum* fixo de tempo para cada task em execução antes de passar para a próxima task.

O algoritmo Round-Robin com prioridade (RR_p) adiciona a consideração de prioridade, onde a task com maior prioridade é selecionada para execução em caso de empate.

O algoritmo First Come, First Served (FCFS) executa as tasks na ordem em que elas chegaram, sem levar em consideração a prioridade.

2 Construção da logica

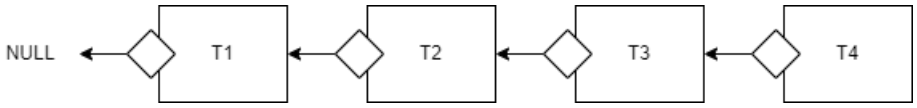
desenvolvimento dos algoritmos com ilustração gráfica e exemplos.

2.1 Round-Robin (RR)

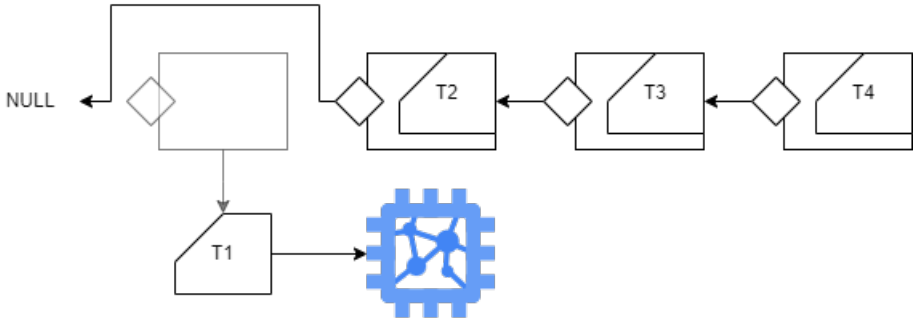
O algoritmo Round-Robin (RR) é um algoritmo de escalonamento de tarefas amplamente utilizado em sistemas operacionais. Ele é projetado para compartilhar justamente o tempo de CPU entre as tarefas de forma equitativa, atribuindo a cada tarefa um pequeno intervalo de tempo chamado de *quantum*.

A lógica básica do algoritmo Round-Robin é a seguinte:

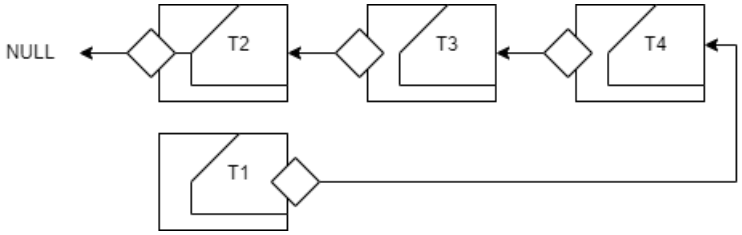
1. Cada tarefa é colocada em uma fila de aptos (*ready queue*) na ordem em que chegam.



2. A tarefa na frente da fila de aptos é selecionada para execução e recebe um *quantum* de tempo para executar.



3. Se a tarefa não concluir sua execução dentro do *quantum*, ela é colocada novamente no final da fila de aptos.



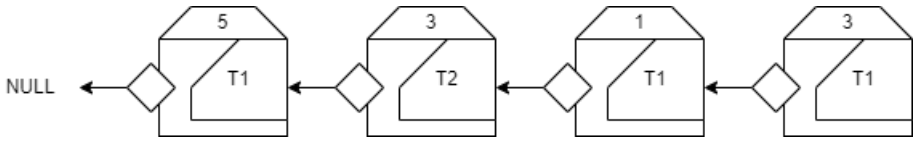
4. A próxima tarefa na fila é selecionada e recebe um *quantum* de tempo para executar.
5. Esse processo de seleção e execução é repetido até que todas as tarefas sejam concluídas.

Essa abordagem garante que todas as tarefas recebam algum tempo de processamento de forma justa, evitando que uma única tarefa monopolize a CPU por um longo período.

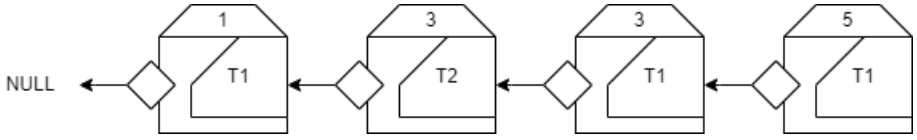
2.2 Round-Robin com prioridade (RR_p)

O funcionamento resumido do algoritmo é o seguinte:

1. Inicialmente, os elementos são inseridos na fila de acordo com suas prioridades.



2. Os elementos na fila são ordenados por prioridade utilizando um processo de classificação (*sort*).



3. O escalonador começa a executar as tarefas na fila, seguindo a ordem definida pela prioridade.
4. Cada tarefa é executada por um determinado período de tempo, conhecido como *quantum*.
5. Se a tarefa não for concluída dentro do *quantum*, ela é colocada novamente na fila, mantendo a mesma prioridade.
6. O escalonador avança para a próxima tarefa na fila, seguindo a sequência de prioridade.
7. O processo de execução continua, alternando entre as tarefas na fila de acordo com a prioridade, até que todas as tarefas sejam concluídas.

Essa abordagem garante que as tarefas sejam executadas de acordo com suas prioridades, permitindo que as tarefas de maior prioridade sejam atendidas primeiro. Assim, o algoritmo Round-Robin com prioridade combina os conceitos do escalonamento Round-Robin e a consideração da importância das tarefas com base em sua prioridade.

2.3 First Come, First Served (FCFS)

Suponha que temos três tarefas para serem executadas: Tarefa A, Tarefa B e Tarefa C. Elas chegam na seguinte ordem: A, B, C.

- Tarefa A chega e inicia a execução.
- Tarefa B chega enquanto a Tarefa A ainda está em execução. A Tarefa B é colocada em espera (fila de aptos).
- Quando a Tarefa A termina, a Tarefa B passa a ser a próxima a ser executada.
- Tarefa C chega enquanto a Tarefa B está em execução. A Tarefa C é colocada em espera (fila de aptos).
- Quando a Tarefa B termina, a Tarefa C passa a ser a próxima a ser executada.
- Finalmente, quando a Tarefa C termina, todas as tarefas foram concluídas.

O FCFS segue estritamente a ordem de chegada das tarefas, executando-as em sequência. Isso significa que se uma tarefa de longa duração chegar antes de tarefas mais curtas, as tarefas serão executadas na mesma ordem de chegada.

3 Códigos

3.1 Round-Robin (RR)

```
#include <stdlib.h>
#include "CPU.h"
#include "list.h"
#include "task.h"

int tid_counter = 0;
struct node *TASK_LIST = NULL;
// Função para adicionar uma nova tarefa à lista de tarefas
void add_r(char *name, int priority, int burst) {
    Task *new_task = malloc(sizeof(Task));
    new_task->name = name;
    new_task->tid = ++tid_counter;
    new_task->priority = priority;
    new_task->burst = burst;
    insert_task(&TASK_LIST, new_task);
}
// Função para o escalonamento Round-Robin (RR)
void schedule_rr() {
    struct node *temp;
    while (TASK_LIST != NULL) {
        temp = TASK_LIST;
        while (temp != NULL) {
            if (temp->task->burst > QUANTUM) {
                // Executa a tarefa pelo tempo do quantum
                run(temp->task, QUANTUM);
                temp->task->burst -= QUANTUM;
            } else {
                // Se a tarefa for concluída, remove da lista de tarefa
                run(temp->task, temp->task->burst);
                struct node *next_task = temp->next;
                delete_task(&TASK_LIST, temp->task);
                temp = next_task;
                continue; // Continue diretamente para evitar o avanço
```

```
            }
            temp = temp->next;
        }
    }
}
```

3.2 Round-Robin com prioridade (RR_p)

```
#include <stdlib.h>
#include "CPU.h"
#include "list.h"
#include "task.h"

int tid_counter_p = 0;
struct node *TASK_LIST_p = NULL;

// Adiciona uma nova tarefa na lista de tarefas com prioridade
void add_rp(char *name, int priority, int burst) {
    Task *new_task = malloc(sizeof(Task));
    new_task->name = name;
    new_task->tid = ++tid_counter_p;
    new_task->priority = priority;
    new_task->burst = burst;
    insert_task(&TASK_LIST_p, new_task);
}

// Troca a posição de duas tarefas na lista
void swap(struct node *a, struct node *b) {
    struct Task *temp = a->task;
    a->task = b->task;
    b->task = temp;
}

// Ordena a lista de tarefas por prioridade
void sort_by_priority(struct node **head) {
    int swapped;
    struct node *ptr1;
    struct node *lptr = NULL;
```

```
    if (*head == NULL)
        return;

    do {
        swapped = 0;
        ptr1 = *head;

        while (ptr1->next != lptr) {
            if (ptr1->task->priority > ptr1->next->task->priority) {
                swap(ptr1, ptr1->next);
                swapped = 1;
            }
            ptr1 = ptr1->next;
        }
        lptr = ptr1;
    } while (swapped);
}

// Executa o escalonamento Round-Robin com prioridade
void schedule_rr_p() {
    struct node *temp;
    while (TASK_LIST_p != NULL) {
        temp = TASK_LIST_p;
        while (temp != NULL) {
            sort_by_priority(&temp);
            if (temp->task->burst > QUANTUM) {
                run(temp->task, QUANTUM);
                temp->task->burst -= QUANTUM;
            } else {
                run(temp->task, temp->task->burst);
                delete_task(&TASK_LIST_p, temp->task);
            }
            temp = temp->next;
        }
    }
}
```

3.3 First Come, First Served (FCFS)

```
        #include <stdlib.h>
#include "schedule_fcfs.h"

#include "CPU.h"
#include "list.h"
#include "task.h"
int cont = 0;

struct node *task_list = NULL; // variável global para armazenar a lista de tarefas

/// função para adicionar uma tarefa à lista de tarefas
void add(char *name,int priority ,int burst)
{
    cont += 1;
    Task *new_task = malloc(sizeof(Task));
    new_task->name = name;
    new_task->tid = cont ;
    new_task->priority = priority;
    new_task->burst = burst;
    insert_task(&task_list,new_task );
}
void schedule()
{
    while(task_list != NULL)
    {
        /// Encontra o primeiro elemento
        struct node *anterior = task_list;
        while(anterior->next != NULL)
        {
            anterior = anterior->next;
        }

        printf("---------------[%s]---------------------\n", anterior->task->name);

        /// Repete até o tempo limite ou até a tarefa ser concluída
        int cont_slice = 0;
        while(  anterior->task->burst > 0)
```

```
        {
            if(anterior->task->burst<QUANTUM){
                cont_slice += anterior->task->burst;
            }else{
                cont_slice += QUANTUM;
            }
            /// Executa a tarefa
            run(anterior->task, cont_slice);

            /// Diminui a quantidade de tempo que a tarefa precisa
            anterior->task->burst -= QUANTUM ;

            /// Incrementa o contador do slice

        }

        /// Caso a tarefa ainda não tenha sido concluída, move para o final da fila
        if(anterior->task->burst > 0)
        {
            Task *salva = anterior->task;
            delete_task(&task_list, anterior->task);
            insert_task(&task_list, salva);
        }
        else
        {
            /// Se a tarefa foi concluída, remove da fila
            delete_task(&task_list, anterior->task);
        }
    }
}
```

4 Resultados

Tabela 1 – RR

Running task	**Priority**	**Time (units)**	**Processor Units**
T10	1	50	10
T9	1	50	10
T8	1	50	10
T7	1	40	10
T6	1	33	10
T5	1	12	10
T4	1	5	5
T3	1	10	10
T2	1	33	10
T1	1	32	10
T10	1	40	10
T9	1	40	10
T8	1	40	10
T7	1	30	10
T6	1	23	10
T5	1	2	2
T2	1	23	10
T1	1	22	10
T10	1	30	10
T9	1	30	10
T8	1	30	10
T7	1	20	10
T6	1	13	10
T2	1	13	10
T1	1	12	10
T10	1	20	10
T9	1	20	10
T8	1	20	10
T7	1	10	10
T6	1	3	3
T2	1	3	3
T1	1	2	2
T10	1	10	10
T9	1	10	10
T8	1	10	10

Tabela 2 – RR_p

Running task	**Priority**	**Time (units)**	**Processor Units**
T6	1	50	10
T4	2	50	10
T2	2	50	10
T3	3	50	10
T5	4	50	10
T1	5	50	10
T6	1	40	10
T4	2	40	10
T2	2	40	10
T3	3	40	10
T5	4	40	10
T1	5	40	10
T6	1	30	10
T4	2	30	10
T2	2	30	10
T3	3	30	10
T5	4	30	10
T1	5	30	10
T6	1	20	10
T4	2	20	10
T2	2	20	10
T3	3	20	10
T5	4	20	10
T1	5	20	10
T6	1	10	10
T4	2	10	10
T2	2	10	10
T3	3	10	10
T5	4	10	10
T1	5	10	10

Tabela 3 – FCFS

Running task	**Priority**	**Time (units)**	**Processor Units**
T1	1	32	10
T1	1	22	10
T1	1	12	10
T1	1	2	2
T2	1	33	10
T2	1	23	10
T2	1	13	10
T2	1	3	3
T3	1	10	10
T4	1	5	5
T5	1	12	10
T5	1	2	2
T6	1	33	10
T6	1	23	10
T6	1	13	10
T6	1	3	3
T7	1	40	10
T7	1	30	10
T7	1	20	10
T7	1	10	10
T8	1	50	10
T8	1	40	10
T8	1	30	10
T8	1	20	10
T8	1	10	10
T9	1	50	10
T9	1	40	10
T9	1	30	10
T9	1	20	10
T9	1	10	10
T10	1	50	10
T10	1	40	10
T10	1	30	10
T10	1	20	10
T10	1	10	10

Análise e Discussão dos Resultados Finais

A partir das informações fornecidas, podemos realizar uma análise e discussão sobre os resultados finais dos escalonadores implementados.

Os algoritmos implementados são o Round-Robin (RR), Round-Robin com prioridade (RR_p) e FCFS (First Come, First Served). Cada algoritmo possui suas características específicas e afeta o desempenho do sistema de maneiras distintas. Vamos analisar cada um deles separadamente:

1. Round-Robin (RR):
    - O algoritmo RR utiliza o conceito de fatia de tempo (slice) para realizar o escalonamento. Cada tarefa recebe um tempo de execução igual e, quando esse tempo é esgotado, a tarefa é colocada de volta no final da fila de aptos, permitindo que outras tarefas sejam executadas.
    - O RR garante um comportamento justo entre as tarefas, pois todas têm oportunidade de executar por um período de tempo definido.
    - No entanto, esse algoritmo pode levar a um alto tempo de resposta para tarefas de longa duração, uma vez que elas precisam aguardar sua vez de serem executadas novamente.
2. Round-Robin com prioridade (RR_p):
    - O algoritmo RR_p é uma variação do RR em que cada tarefa possui uma prioridade associada. Quando há várias tarefas aptas para execução, o escalonador escolhe a tarefa com maior prioridade para executar.
    - Em caso de empate nas prioridades, pode-se utilizar um critério para desempate, como a ordem de chegada das tarefas na fila de aptos.
    - Esse algoritmo permite dar maior prioridade a determinadas tarefas, garantindo que sejam executadas antes de outras com prioridades inferiores.
3. FCFS (First Come, First Served):
    - O algoritmo FCFS segue o princípio de atender as tarefas na ordem em que chegam, ou seja, a primeira tarefa que chegou será a primeira a ser executada.
    - Esse algoritmo é simples e fácil de implementar, mas pode levar a um tempo de espera longo para tarefas que chegaram posteriormente e têm um tempo de execução maior.

Considerando os resultados finais obtidos, podemos avaliar alguns aspectos:

- Justiça: O algoritmo RR é conhecido por oferecer um tratamento justo entre as tarefas, uma vez que todas recebem um tempo de execução igual. No entanto, o RR_p pode introduzir uma certa preferência de execução com base nas prioridades. Portanto, é importante analisar se o escalonador está distribuindo de forma equilibr

  ada o tempo de execução entre as tarefas e se as prioridades estão sendo tratadas adequadamente.

- Tempo de resposta: O tempo de resposta é o intervalo entre a submissão de uma tarefa e o início de sua execução. É importante verificar se o tempo de resposta das tarefas está dentro de limites aceitáveis. Algoritmos como o RR tendem a ter tempos de resposta mais curtos para tarefas curtas, enquanto o FCFS pode levar a tempos de resposta mais longos.

- Utilização da CPU: É relevante avaliar a eficiência do escalonador em utilizar a capacidade de processamento da CPU. Um escalonador