Gabriel Bodenmüller

Escalonamento

Trabalho de curso submetido a Universidade do Vale do Itajaí - UNIVALI -, para obtenção de nota na disciplina de Sistemas Operacionais

Universidade do Vale do Itajaí - UNIVALI

Faculdade de Engenharia de Computação

Bacharelado

Orientador: Felipe Viel

Itajaí - SC

2023

Sumário

1 Introdução

Um sistema operacional é uma camada de software que atua como intermediário entre o hardware de um computador e os programas que são executados nele. Ele é responsável por gerenciar recursos do sistema, como memória, processamento, armazenamento e entrada/saída, permitindo que os programas executem suas tarefas de forma eficiente e segura.

O estudo de sistemas operacionais é fundamental para entender como as diversas partes do sistema trabalham juntas, desde a inicialização do computador até a execução dos programas. Além disso, conhecer os principais conceitos e funcionalidades dos sistemas operacionais é importante para otimizar o desempenho do computador e garantir a segurança dos dados armazenados.

Este relatório tem como objetivo apresentar uma implementação de dois algoritmos de escalonadores de tarefas (taks). Os escalonadores são o Round-Robin (RR), o Round-Robin com prioridade (RR_p) e FCFS (First Come, First Served).

2 Enunciado

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):

- 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.

- 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.

- 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.

- 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.

- 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.

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:

- 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.

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

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

- 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.

- 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.

3 Desenvolvimento

3.1 First-Come, First-Served (FCFS)

O escalonador FCFS é um algoritmo simples que segue a abordagem "primeiro a chegar, primeiro a ser atendido"para o agendamento de tarefas. O código de implementação do escalonador FCFS é composto pelos arquivos "schedule_fcfs.h", "list.h", "task.h"e "CPU.h". Ele contém as seguintes funções principais:

add_FCFS(char name, int priority, int burst): Essa função é responsável por adicionar uma nova tarefa à lista de tarefas do escalonador FCFS. Ela recebe o nome, a prioridade e o tempo de execução da tarefa como parâmetros. A função aloca memória para uma nova tarefa, define seus atributos e insere a tarefa na lista de tarefas usando a função insert do módulo list.h.

schedule_FCFS(): Essa função implementa o algoritmo FCFS para agendar e executar as tarefas da lista. Ela começa verificando se a lista de tarefas está vazia. A função percorre a lista de tarefas usando um ponteiro current_task. Para cada tarefa na lista, a função exibe o nome da tarefa e inicia a execução.

Dentro do loop principal, a função executa a tarefa em fatias de tempo (quantum). Ela verifica se o tempo restante de execução da tarefa é menor que o quantum e, se for, define a fatia de tempo como o tempo restante. A função então chama a função run do módulo CPU.h para executar a tarefa por essa fatia de tempo. Em seguida, o tempo restante da tarefa é decrementado pela fatia de tempo.

Após a execução da tarefa, o ponteiro current_task é atualizado para apontar para a próxima tarefa na lista. O processo se repete até que todas as tarefas tenham sido executadas e então a função exibe uma mensagem informando que todas as tarefas foram executadas.

A implementação do escalonador FCFS fornece uma maneira básica e eficiente de agendar tarefas em uma ordem sequencial, onde as tarefas são executadas na ordem em que foram adicionadas à lista. No entanto, ele não leva em consideração a prioridade das tarefas nem realiza qualquer tipo de reordenação com base em prioridades ou tempos de espera.

Segue abaixo a saida do console para o escalonador FCFS:

```
Executando tarefa: T6
Running task = [T6] [40] [50] for 10 units.
Running task = [T6] [40] [40] for 10 units.
Running task = [T6] [40] [30] for 10 units.
Running task = [T6] [40] [20] for 10 units.
Running task = [T6] [40] [10] for 10 units.
Executando tarefa: T5
Running task = [T5] [40] [50] for 10 units.
Running task = [T5] [40] [40] for 10 units.
Running task = [T5] [40] [30] for 10 units.
Running task = [T5] [40] [20] for 10 units.
Running task = [T5] [40] [10] for 10 units.
Executando tarefa: T4
Running task = [T4] [40] [50] for 10 units.
Running task = [T4] [40] [40] for 10 units.
Running task = [T4] [40] [30] for 10 units.
Running task = [T4] [40] [20] for 10 units.
Running task = [T4] [40] [10] for 10 units.
Executando tarefa: T3
Running task = [T3] [40] [50] for 10 units.
Running task = [T3] [40] [40] for 10 units.
Running task = [T3] [40] [30] for 10 units.
Running task = [T3] [40] [20] for 10 units.
Running task = [T3] [40] [10] for 10 units.
Executando tarefa: T2
Running task = [T2] [40] [50] for 10 units.
Running task = [T2] [40] [40] for 10 units.
Running task = [T2] [40] [30] for 10 units.
Running task = [T2] [40] [20] for 10 units.
Running task = [T2] [40] [10] for 10 units.
Executando tarefa: T1
Running task = [T1] [40] [50] for 10 units.
Running task = [T1] [40] [40] for 10 units.
Running task = [T1] [40] [30] for 10 units.
Running task = [T1] [40] [20] for 10 units.
Running task = [T1] [40] [10] for 10 units.
Todas as tarefas foram executadas.
```

Figura 1 – Saida do FCFS

3.2 Round Robin (RR)

O escalonador Round Robin (RR) é um algoritmo de escalonamento de processos amplamente utilizado em sistemas operacionais. Ele garante um compartilhamento justo de tempo de CPU entre os processos, permitindo que cada processo execute por um curto período.

add_RR(char name, int priority, int burst): Essa função é responsável por adicionar uma nova tarefa à lista de tarefas do escalonador RR. Ela recebe como parâmetros o nome da tarefa, prioridade e duração, dentro da função, é criada uma nova estrutura 'Task' e preenchida com as informações fornecidas. Em seguida, a tarefa é inserida na lista de tarefas usando a função 'insert', que adiciona o nó na posição correta de acordo com a prioridade.

schedule_RR(): Essa função é responsável por executar o escalonamento das tarefas usando o algoritmo RR. Ela percorre a lista de tarefas do escalonador RR e executa cada tarefa pelo tempo do quantum, definido como 'QUANTUM'. Se a duração da tarefa for maior que o quantum, ela é executada pelo tempo do quantum e a duração é reduzida. Caso contrário, a tarefa é executada pelo tempo restante e removida da lista. A função utiliza as funções auxiliares 'run', que simula a execução da tarefa na CPU, e 'delete', que remove a tarefa da lista.

É importante destacar que a eficiência do escalonador RR pode ser afetada pelo tamanho do quantum. Quantum muito curtos podem causar um alto custo de troca de contexto, enquanto quantum muito longos podem resultar em um menor compartilhamento de tempo de CPU.

Segue abaixo a saída do console para o escalonador RR:

```
Running task = [T6] [40] [50] for 10 units.
Running task = [T5] [40] [50] for 10 units.
Running task = [T4] [40] [50] for 10 units.
Running task = [T3] [40] [50] for 10 units.
Running task = [T2] [40] [50] for 10 units.
Running task = [T1] [40] [50] for 10 units.
```

Figura 2 – Saida do RR

3.3 Round Robin com Prioridade(RR_P)

O escalonador Round Robin com prioridade (RR_P) é um algoritmo de escalonamento de processos que combina a abordagem Round Robin com o conceito de prioridade. Ele garante um compartilhamento justo de tempo de CPU entre os processos, levando em consideração a prioridade de cada processo. Neste relatório, discutiremos a implementação do escalonador RR_P, detalhando as principais funções e estruturas utilizadas.

add_RR_P(char name, int priority, int burst): Essa função é responsável por adicionar uma nova tarefa à lista de tarefas do escalonador RR_P. Ela recebe como parâmetros o nome da tarefa, prioridade e duração. Dentro da função, é criada uma nova estrutura Task e preenchida com as informações fornecidas. Em seguida, a tarefa é inserida na lista de tarefas usando a função insert, que adiciona o nó na posição correta de acordo com a prioridade.

swap e quick_sort: Essas funções são responsáveis pela ordenação da lista de tarefas por prioridade. A função swap troca a posição de duas tarefas na lista.

A função quick_sort implementa o algoritmo Quick Sort para ordenar a lista de tarefas com base na prioridade. A função partition é usada internamente pela função quick_sort para particionar a lista.

schedule_RR_P(): Essa função é responsável por executar o escalonamento das tarefas usando o algoritmo RR_P. Ela itera sobre a lista de tarefas do escalonador RR_P e realiza a ordenação das tarefas por prioridade usando o Quick Sort. Em seguida, ela executa cada tarefa pelo tempo do quantum (definido como QUANTUM) ou pelo tempo restante da tarefa, caso seja menor que o quantum. Após a execução, a tarefa é removida da lista usando a função delete.

Segue abaixo a saída do console para o escalonador RR:

```
Running task = [T6] [5] [50] for 10 units.
Running task = [T5] [4] [50] for 10 units.
Running task = [T4] [2] [50] for 10 units.
Running task = [T3] [3] [50] for 10 units.
Running task = [T2] [2] [50] for 10 units.
Running task = [T1] [1] [50] for 10 units.
Running task = [T6] [5] [30] for 10 units.
Running task = [T5] [4] [30] for 10 units.
Running task = [T4] [2] [30] for 10 units.
Running task = [T3] [3] [30] for 10 units.
Running task = [T2] [2] [30] for 10 units.
Running task = [T1] [1] [30] for 10 units.
Running task = [T6] [5] [10] for 10 units.
Running task = [T5] [4] [10] for 10 units.
Running task = [T4] [2] [10] for 10 units.
Running task = [T3] [3] [10] for 10 units.
Running task = [T2] [2] [10] for 10 units.
Running task = [T1] [1] [10] for 10 units.
```

Figura 3 – Saida do RR_P

4 Códigos Implementados

Códigos First-Come, First-Served (FCFS)

```c
void add_FCFS(char *name, int priority, int burst)
{
    Task *newTask = malloc(sizeof(Task)); // Allocate memory for a new task
    newTask->name = name;                 // Set the name of the task
    newTask->tid = ++tid_counter;         // Assign a unique task ID
    newTask->priority = priority;         // Set the priority of the task
    newTask->burst = burst;               // Set the burst time of the task
    insert(&task_list_fcfs, newTask);     // Insert the task into the task list
}

// Function to schedule tasks using the FCFS algorithm
void schedule_FCFS()
{
    if (task_list_fcfs == NULL)
    {
        printf("A lista de tarefas esta vazia.\n");
        return;
    }

    struct node *current_task = task_list_fcfs; // Pointer to traverse the task li
    while (current_task != NULL)
    {
        printf("Executando tarefa: %s\n", current_task->task->name);

        int remaining_burst = current_task->task->burst; // Remaining burst time o
        while (remaining_burst > 0)
        {
            int slice = (remaining_burst < QUANTUM) ? remaining_burst : QUANTUM;
            // Run the task for a time slice (QUANTUM) or the remaining burst time
            run(current_task->task, slice);
            remaining_burst -= slice; // Decrement the remaining burst time
        }

        current_task = current_task->next; // Move to the next task in the list
```

```
    }

    printf("Todas as tarefas foram executadas.\n");
}
```

Códigos Round Robin (RR)

```
void add_RR(char *name, int priority, int burst)
{
    Task *newTask = malloc(sizeof(Task));
    newTask->name = name;
    newTask->tid = ++tid_counter_rr;
    newTask->priority = priority;
    newTask->burst = burst;
    insert(&task_list_rr, newTask);
}

// Schedule tasks using the Round Robin (RR) scheduling algorithm
void schedule_RR()
{
    struct node *current_task = task_list_rr;
    while (current_task != NULL)
    {
        Task *task = current_task->task;
        if (task->burst > QUANTUM)
        {
            // Execute the task for the duration of the quantum
            run(task, QUANTUM);
            task->burst -= QUANTUM;
        }
        else
        {
            // Execute the remaining part of the task
            run(task, task->burst);
            // Remove the task from the list
            struct node *next_task = current_task->next;
            delete(&task_list_rr, task);
            current_task = next_task;
```

```
            continue; // Continue directly to avoid advancing current_task
        }
        current_task = current_task->next;
    }
}
```

Códigos Round Robin com Prioridade (RR_P)

```
void add_RR_P(char *name, int priority, int burst) {
    Task *new_task = malloc(sizeof(Task));
    if (new_task == NULL) {
        // Handle memory allocation error
        return;
    }
    new_task->name = name;
    new_task->tid = ++tid_counter_p;
    new_task->priority = priority;
    new_task->burst = burst;
    insert(&TASK_LIST_p, new_task);
}

// Swap the positions of two tasks in the list
void swap(struct node *a, struct node *b) {
    Task *temp = a->task;
    a->task = b->task;
    b->task = temp;
}

// Sort the task list by priority using the Quick Sort algorithm
void quick_sort(struct node *low, struct node *high) {
    if (high != NULL && low != high && low != high->next) {
        struct node *pivot = partition(low, high);
        quick_sort(low, pivot->prev);
        quick_sort(pivot->next, high);
    }
}

// Partition the list for Quick Sort
```

```
struct node *partition(struct node *low, struct node *high) {
    Task *pivot = high->task;
    struct node *i = low->prev;

    for (struct node *j = low; j != high; j = j->next) {
        if (j->task->priority < pivot->priority) {
            i = (i == NULL) ? low : i->next;
            swap(i, j);
        }
    }

    i = (i == NULL) ? low : i->next;
    swap(i, high);
    return i;
}

// Execute the Round Robin with priority (RR_P) scheduling
void schedule_RR_P() {
    struct node *temp;
    while (TASK_LIST_p != NULL) {
        temp = TASK_LIST_p;
        quick_sort(temp, NULL);
        while (temp != NULL) {
            if (temp->task->burst > QUANTUM) {
                run(temp->task, QUANTUM);
                temp->task->burst -= QUANTUM;
            } else {
                run(temp->task, temp->task->burst);
                delete(&TASK_LIST_p, temp->task);
            }
            temp = temp->next;
        }
    }
}
```

5 Conclusão

Concluindo, o relatório apresentou uma visão dos principais temas estudados na disciplina de sistemas operacionais.

Foi possível perceber a importância dos sistemas operacionais para o funcionamento dos computadores e como eles atuam como intermediários entre o hardware e os programas que são executados neles. Além disso, conhecer os principais conceitos e funcionalidades dos sistemas operacionais é fundamental para otimizar o desempenho do computador e garantir a segurança dos dados armazenados.

Ao longo do relatório, foram apresentados exemplos de sistemas operacionais mais comuns e suas respectivas características, além de algumas tendências futuras na área. É importante destacar que os sistemas operacionais estão em constante evolução, buscando sempre oferecer novas funcionalidades e melhorias de desempenho.

Por fim, pode-se concluir que o estudo de sistemas operacionais é fundamental para todos que trabalham ou pretendem trabalhar na área de tecnologia da informação, sendo uma disciplina que proporciona uma compreensão mais aprofundada sobre o funcionamento dos computadores e sua interação com os programas que são executados neles.