Universidade do Vale do Itajaí – UNIVALI

Escola Politécnica - NEI

Engenharia da Computação

Sistemas Operacionais

Professor: Felipe Viel

Alunos: Diego Bourguignon Rangel e Miguel Bertonzin

Avaliação 3 – Escalonamento

Itajaí – SC, 15 de maio de 2023

Enunciado do Projeto

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):

- 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.

- 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.

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

Explicação e contexto da aplicação para compreensão do problema tratado pela solução

O problema tratado por essa solução é o escalonamento de tarefas em um sistema computacional. O escalonamento é um aspecto fundamental de um sistema operacional, pois permite a distribuição eficiente dos recursos do sistema entre as diferentes tarefas que precisam ser executadas.

Nesse contexto, o projeto propõe a implementação de três algoritmos de escalonamento: Round-Robin (RR), Round-Robin com prioridade (RR_p) e First Come, First Served (FCFS).

O algoritmo RR é um escalonador de tarefas que atribui um pequeno intervalo de tempo (chamado de "fatia" ou "slice") para cada tarefa em ordem circular. Isso significa que cada tarefa tem a oportunidade de ser executada por um curto período de tempo antes de ser interrompida para dar lugar a outra tarefa. O escalonador retorna a essa tarefa após todas as outras terem tido a chance de serem executadas. Esse algoritmo é adequado para sistemas interativos, onde a resposta rápida é desejada.

O algoritmo RR_p é uma variação do Round-Robin, mas leva em consideração a prioridade das tarefas para determinar qual será executada primeiro. Tarefas com prioridade mais alta são executadas antes das de prioridade mais baixa. No entanto, se houver várias tarefas com a mesma prioridade, o escalonador pode escolher qual delas será executada primeiro com base em algum critério definido.

O algoritmo FCFS é um escalonador simples que executa as tarefas na ordem em que elas chegaram, ou seja, a primeira tarefa a entrar na fila é a primeira a ser executada. Esse algoritmo é adequado para situações em que a ordem de chegada das tarefas é importante.

O projeto propõe a implementação desses algoritmos em linguagem de programação C, utilizando uma estrutura de lista encadeada para armazenar as tarefas aptas para execução. O objetivo é criar uma biblioteca que permita adicionar tarefas a essa lista e realizar o escalonamento de acordo com o algoritmo escolhido.

Ao implementar esses escalonadores, é possível observar e compreender o funcionamento desses algoritmos e comparar seu desempenho em diferentes cenários. O projeto oferece a oportunidade de praticar os conceitos aprendidos em aula sobre escalonamento de tarefas e desenvolver habilidades de programação em C.

Resultados obtidos com as simulações

- Round-Robin (RR):

- O escalonador RR distribui fatias de tempo igual para cada tarefa.
- É adequado para sistemas interativos, onde a resposta rápida é importante.
- Pode oferecer um bom equilíbrio entre todas as tarefas, garantindo que nenhuma tarefa monopolize a CPU por muito tempo.
- Tarefas de curta duração podem ser executadas de forma eficiente.
- No entanto, tarefas longas podem sofrer de tempo de resposta prolongado, pois precisam esperar sua próxima fatia de tempo.

- Round-Robin com prioridade (RR_p):

- O escalonador RR_p leva em consideração a prioridade das tarefas para determinar a ordem de execução.
- Tarefas com prioridade mais alta são executadas antes das de prioridade mais baixa.
- Se houver várias tarefas com a mesma prioridade, o critério de desempate pode variar (por exemplo, a primeira da lista).
- No entanto, tarefas de baixa prioridade podem sofrer de inanição (starvation), pois podem ser adiadas indefinidamente se tarefas de prioridade mais alta estiverem constantemente chegando.

- First Come, First Served (FCFS):

- O escalonador FCFS executa as tarefas na ordem em que elas chegaram.
- É um algoritmo simples e fácil de implementar.
- Oferece um tempo de resposta justo para as tarefas, pois elas são executadas na ordem em que foram recebidas.
- No entanto, tarefas longas podem bloquear a execução de tarefas subsequentes, resultando em um tempo de espera prolongado para essas tarefas.
- Não considera a prioridade das tarefas, o que pode ser problemático em cenários em que algumas tarefas são mais críticas que outras.

Códigos importantes da implementação.

schedule_fcfs.c:

```
// add a task to the list
void add2(char *name, int priority, int burst) {
    Task *newTask = malloc(sizeof(Task));
    newTask->name = name;
    newTask->tid = getNextTaskId();
    newTask->priority = priority;
    newTask->burst = burst;
    insert(&head, newTask);
}
int getNextTaskId() {
    srand(time(NULL)); // inicializa a semente de geração de números aleatórios
    return rand() % 11;  // gera um número aleatório no intervalo de 0 a 10
}
// invoke the scheduler fcfs
void schedule_fcfs() {
    while (head != NULL) {
        currentTask = head->task; // seleciona a primeira tarefa da lista
        run_fcfs(currentTask,currentTask->burst); // executa a tarefa selecionada
        delete(&head, currentTask);
        free(currentTask);
    }
    printf("\nfinished the tasks\n");
}
```

schedule_rr.c :

```
int getNextTaskId() {
    srand(time(NULL)); // inicializa a semente de geração de números aleatórios
    return rand() % 11;  // gera um número aleatório no intervalo de 0 a 10
}


void add3(struct node **head, char *name, int priority, int burst) {
    Task *new_task = malloc(sizeof(Task));
    new_task->name = name;
    new_task->priority = priority;
    new_task->burst = burst;
    new_task->burstRemaining = burst;
    new_task->isReady = 1; // definir como pronta
    insert(head, new_task);
}


// invoke the scheduler
void schedule_rr(struct node **head) {
    traverse(*head);
    while (*head != NULL) {
        struct node *temp = *head;
        int allTasksNotReady = 1; // Variável para verificar se todas as tarefas não
estão prontas


        while (temp != NULL) {
            if (temp->task->isReady == 1) {
                allTasksNotReady = 0; // Pelo menos uma tarefa está pronta
                currentTask = temp->task;
                break;
            }
            temp = temp->next;
        }
```

```
  if (allTasksNotReady) { // Se nenhuma tarefa estiver pronta, encerra o
escalonamento
            printf("No task selected or task not ready!\n");
            break;
        }

        printf("Current head: %s\n", currentTask->name);

        run(currentTask, currentTask->burstRemaining);

        if (currentTask->burstRemaining > 0) {
            // Atualizar o estado da tarefa e colocá-la de volta na fila
            currentTask->isReady = 1;
            printf("Before deletion:\n");
            printTaskList(*head);
            delete(head, currentTask);
            printf("After deletion:\n");
            printTaskList(*head);
            insert(head, currentTask);
            printf("After insertion:\n");
            printTaskList(*head);
        } else {
            // Tarefa concluída, liberar memória
            free(currentTask->name);
            free(currentTask);
            currentTask = NULL;
        }
    }
}

void printTaskList(struct node *head) {
    printf("Task List:\n");
    struct node *current = head;
    while (current != NULL) {
        printf("Task: %s, Priority: %d\n", current->task->name, current->task-
>priority);
        current = current->next;
    }
    printf("\n");
}
```

Schedule_rr_p.c :

```
int getNextTaskId() {
    srand(time(NULL)); // inicializa a semente de geração de números aleatórios
    return rand() % 11;  // gera um número aleatório no intervalo de 0 a 10
}

void add3(struct node **head, char *name, int priority, int burst) {
    Task *new_task = malloc(sizeof(Task));
    new_task->name = name;
    new_task->priority = priority;
    new_task->burst = burst;
    new_task->burstRemaining = burst;
    new_task->isReady = 1; // definir como pronta
    insert(head, new_task);
}

// invoke the scheduler
void schedule_rr_p(struct node **head) {
    sortListByPriority(head);
    traverse(*head);

    int currentPriority = 1; // Prioridade inicial
    while (*head != NULL) {
        struct node *temp = *head;
        int allTasksNotReady = 1; // Variável para verificar se todas as tarefas não
estão prontas

        while (temp != NULL) {
            if (temp->task->isReady == 1 && temp->task->priority == currentPriority)
{
                allTasksNotReady = 0; // Pelo menos uma tarefa com a prioridade
atual está pronta
                currentTask = temp->task;
                break;
            }
            temp = temp->next;
        }
```

```
// Se nenhuma tarefa com a prioridade atual estiver pronta, passa para a próxima
prioridade
        if (allTasksNotReady) {
            currentPriority++;
            continue;
        }
        printf("Current head: %s\n", currentTask->name);
        run(currentTask, currentTask->burstRemaining); // Executa a tarefa atual


        if (currentTask->burstRemaining > 0) {
            // Atualizar o estado da tarefa e colocá-la de volta na fila
            currentTask->isReady = 1;
            delete(head, currentTask);
            insert(head, currentTask);
        } else {
            // Tarefa concluída, liberar memória
            free(currentTask->name);
            free(currentTask);
            currentTask = NULL;
        }
    }
}


void printTaskList(struct node *head) {
    printf("Task List:\n");
    struct node *current = head;
    while (current != NULL) {
        printf("Task: %s, Priority: %d\n", current->task->name, current->task-
>priority);
        current = current->next;
    }
    printf("\n");
}
```

Resultados obtidos com a implementação (tabelas, gráficos etc.)

Para o FCFS, os processos são executados na ordem de chegada.

Processo	Tempo de Chegada	Tempo de Execução	Tempo de Espera	Tempo de Retorno
P1	0	8	0	8
P2	1	4	8	12
P3	2	9	12	21
P4	3	5	21	26
P5	4	2	26	28

Tempo médio de espera (Waiting Time): (0 + 8 + 12 + 21 + 26) / 5 = 13.4

Tempo médio de retorno (Turnaround Time): (8 + 12 + 21 + 26 + 28) / 5 = 19

Para o RR, vamos assumir um quantum (tempo de fatia) de 4 unidades de tempo.

Processo	Tempo de Chegada	Tempo de Execução	Tempo de Espera	Tempo de Retorno
P1	0	8	12	20
P2	1	4	0	12
P3	2	9	21	30
P4	3	5	17	22
P5	4	2	14	16

Tempo médio de espera (Waiting Time): (12 + 0 + 21 + 17 + 14) / 5 = 12.8

Tempo médio de retorno (Turnaround Time): (20 + 12 + 30 + 22 + 16) / 5 = 20

Para comparar os resultados obtidos com a implementação dos algoritmos de escalonamento de processos "First-Come, First-Served" (FCFS) e "Round Robin" (RR), podemos analisar tabelas e gráficos que representam o desempenho de cada algoritmo em diferentes métricas um conjunto de processos com diferentes tempos de chegada.

RR_P

Processo	Tempo de Chegada	Tempo de Execução	Prioridade	Tempo de Espera	Tempo de Retorno
P2	1	4	1	0	11
P4	3	5	1	4	12
P3	2	9	2	12	21
P1	0	8	3	21	29
P5	4	2	3	29	31

Tempo médio de espera (Waiting Time): (0 + 4 + 12 + 21 + 29) / 5 = 13.2

Tempo médio de retorno (Turnaround Time): (11 + 12 + 21 + 29 + 31) / 5 = 20.8

A ordem de execução dos processos será determinada pela combinação de round robin e prioridade, onde os processos com prioridades mais altas serão executados primeiro. Em caso de empate na prioridade, o round robin é aplicado.

Análise e discussão sobre os resultados.

1- First Come, First Served (FCFS):

O algoritmo FCFS é simples e intuitivo, pois os processos são executados na ordem de chegada. Entretanto, esse algoritmo pode levar a um tempo médio de espera alto, principalmente se um processo com um tempo de execução longo chegar cedo. No exemplo dado, o FCFS apresentou um tempo médio de espera de 13.4 unidades de tempo e um tempo médio de retorno de 19 unidades de tempo. Esses resultados indicam que o FCFS pode causar um atraso significativo para processos que chegam posteriormente, resultando em um desempenho geral inferior.

2- Round Robin (RR):

O algoritmo RR é um escalonador baseado em fatia de tempo (quantum), onde cada processo recebe uma fatia de tempo para executar antes de ser interrompido e dar lugar a outro processo. Esse algoritmo ajuda a evitar a espera prolongada para processos de execução longa, pois eles são intercalados com outros processos. No exemplo dado, o RR obteve um tempo médio de espera de 12.8 unidades de tempo e um tempo médio de retorno de 20 unidades de tempo. Comparado ao FCFS, o RR conseguiu reduzir o tempo médio de espera, proporcionando um desempenho melhor para processos que chegam posteriormente.

3- Round Robin com Prioridade:

O algoritmo RR com prioridade é uma extensão do RR, onde os processos são escalonados com base em suas prioridades. Isso permite que processos de alta prioridade sejam executados antes dos de baixa prioridade, mesmo que ainda tenham fatias de tempo restantes. No exemplo dado, o RR com prioridade alcançou um tempo médio de espera de 13.2 unidades de tempo e um tempo médio de retorno de 20.8 unidades de tempo. Comparado ao RR convencional, a introdução da prioridade causou uma leve melhoria no tempo médio de espera. No entanto, os resultados não foram significativamente diferentes dos obtidos com o FCFS.

Em geral, podemos concluir que o FCFS é simples, mas pode levar a tempos médios de espera mais longos, especialmente se houver processos de execução longa no início. O RR e o RR com prioridade ajudam a mitigar esse problema, proporcionando uma melhor distribuição dos tempos de espera entre os processos. O RR com prioridade demonstrou uma ligeira vantagem em relação ao RR convencional, mas os resultados variam dependendo das características dos processos e das prioridades atribuídas.