UNIVERSIDADE DO VALE DO ITAJAÍ
CAMPUS ITAJAÍ
CIÊNCIA DE COMPUTAÇÃO
DISCIPLINA SISTEMAS OPERACIONAIS
PROFESSOR FELIPE VIEL

EDUARDO SAVIAN DE OLIVEIRA, MARCOS AUGUSTO FEHLAUER PEREIRA,
VINÍCIUS WENTZ RITZEL, YURI RODRIGUES

ESCALONADOR

ITAJAÍ
28/05/2023

Sumário

Introdução

Neste projeto, serão implementados três algoritmos de escalonamento de tarefas: Round-Robin (RR), Round-Robin com prioridade e First Come, First Served (FCFS). O escalonamento de tarefas desempenha um papel fundamental na distribuição da capacidade de processamento, e a implementação desses algoritmos permitirá compreender como diferentes estratégias de escalonamento podem afetar o desempenho do sistema operacional.

O algoritmo Round-Robin irá atribuir a cada tarefa uma fatia de tempo igual para execução, garantindo um compartilhamento do processador entre as tarefas. Já o Round-Robin com prioridade levará em consideração a importância das tarefas, garantindo que as mais prioritárias sejam atendidas adequadamente. Por fim, o algoritmo First Come, First Served executará as tarefas na ordem em que chegaram, evitando atrasos e proporcionando uma execução previsível.

Com a implementação desses algoritmos, será possível analisar os resultados obtidos por meio de simulações e avaliar a eficiência e a previsibilidade de cada um em diferentes cenários. Isso permitirá identificar suas vantagens e desvantagens, buscando melhorar a capacidade de gerenciamento do sistema operacional.

Desenvolvimento

First-Come, First-Served

A função sched_run implementa o algoritmo de escalonamento First-Come, First-Served (FCFS), que é um dos algoritmos mais simples de escalonamento. Ele opera com

uma abordagem de fila, em que as tarefas são executadas na ordem em que chegam, sem levar em consideração suas prioridades ou tempos de execução.

A função começa verificando se o ponteiro para o Scheduler (sc) é nulo, o que indica que o escalonador não foi inicializado corretamente. Caso seja nulo, uma mensagem de erro é exibida e a função é encerrada. Em seguida, é verificado se não há tarefas na lista de tarefas do escalonador (sc->tasks). Se a lista estiver vazia, uma mensagem indicando que não há tarefas no escalonador é exibida, e a função é encerrada.

A partir desse ponto, inicia-se a execução das tarefas na ordem em que estão na lista. A variável cur é inicializada com o ponteiro para o primeiro nó da lista de tarefas (sc->tasks.head). Em um loop, a função itera sobre os nós da lista, sendo que cur é atualizado para o próximo nó em cada iteração.

Para cada nó, a função obtém a tarefa correspondente (task) e chama a função run para executar a tarefa pelo tempo de execução total (task->burst). Em seguida, o tempo de execução da tarefa é definido como zero, indicando que a tarefa foi totalmente executada. Em seguida, o nó atual é removido da lista de tarefas (list_rm) e é atualizado o ponteiro para o próximo nó (cur = next).

É exibida uma mensagem informando que a tarefa foi concluída, contendo o nome da tarefa (task->name) e o identificador da tarefa (task->tid).

Após o processamento de uma tarefa, o ponteiro cur é atualizado para o próximo nó na lista. O loop continua até que todos os nós da lista de tarefas tenham sido percorridos, ou seja, até que cur se torne nulo.

O algoritmo FCFS continua executando as tarefas em ordem sequencial, uma após a outra, até que todas as tarefas sejam concluídas. Não há reordenamento ou interrupção das tarefas durante a execução, garantindo que cada tarefa tenha a oportunidade de ser concluída antes de iniciar a próxima.

Round-Robin

A função sched_run implementa o algoritmo de escalonamento Round-Robin (RR), que é um algoritmo de escalonamento por fatia de tempo. Esse algoritmo é utilizado para permitir que múltiplas tarefas sejam executadas de forma justa e equilibrada, dando a cada tarefa um período limitado de tempo de execução, chamado de "quantum".

A função começa realizando algumas verificações. Primeiro, é verificado se o ponteiro para o Scheduler (sc) é nulo, indicando que o escalonador não foi inicializado corretamente. Em caso afirmativo, uma mensagem de erro é exibida e a função é encerrada. Em seguida, é

verificado se não há tarefas na lista de tarefas do escalonador (sc->tasks). Se a lista estiver vazia, uma mensagem indicando que não há tarefas no escalonador é exibida, e a função é encerrada.

A partir desse ponto, inicia-se a execução das tarefas usando o algoritmo Round-Robin. A variável cur é inicializada com o ponteiro para o primeiro nó da lista de tarefas (sc->tasks.head). Em um loop, a função itera sobre os nós da lista, sendo que cur é atualizado para o próximo nó em cada iteração.

Para cada nó, a função obtém a tarefa correspondente (task) e verifica se o tempo de execução restante da tarefa (task->burst) é maior que zero. Se for, significa que a tarefa ainda precisa ser executada. A função então verifica se o tempo de execução restante da tarefa é maior do que o quantum definido (QUANTUM). Se for, a função run é chamada para executar a tarefa pelo tempo de quantum (QUANTUM). O tempo de execução da tarefa é diminuído pelo valor do quantum (task->burst -= QUANTUM).

Caso o tempo de execução restante da tarefa seja menor ou igual ao quantum, significa que a tarefa será concluída dentro dessa fatia de tempo. A função run é chamada para executar a tarefa pelo tempo de execução restante (task->burst). O tempo de execução da tarefa é definido como zero, indicando que a tarefa foi totalmente executada. É exibida uma mensagem informando que a tarefa foi concluída, contendo o nome da tarefa (task->name) e o identificador da tarefa (task->tid).

Em seguida, o ponteiro cur é atualizado para o próximo nó na lista. Se o ponteiro cur chegar ao final da lista (tornar-se nulo), ele é posicionado no início da lista (sc->tasks.head). Esse processo de ir para o próximo nó e, se necessário, voltar para o início da lista continua até que todas as tarefas tenham sido executadas ou até que a lista de tarefas esteja vazia.

Após cada iteração do loop, a função verifica se a lista de tarefas (sc->tasks) está vazia. Se estiver vazia, isso significa que todas as tarefas foram concluídas, e é exibida uma mensagem indicando que todas as tarefas foram concluídas. Em seguida, o loop é interrompido e a função é encerrada.

Round-Robin com Prioridade

A função sched_consume_list recebe um ponteiro para uma estrutura Scheduler (sc) e um ponteiro para uma lista de tarefas (l). Essa função itera sobre os elementos da lista e para cada elemento, recupera o nó correspondente (node). Se o nó existir, a função extrai a tarefa associada (t) e define o ponteiro da tarefa no nó como nulo. Em seguida, é feito um ajuste na prioridade da tarefa para garantir que ela esteja dentro de um intervalo específico. Por fim, a

tarefa é adicionada à lista de tarefas correspondente à sua prioridade na estrutura Scheduler (sc). Após iterar sobre todos os elementos da lista, a lista original (l) é removida.

A função rr_exec executa o algoritmo de escalonamento Round-Robin para a lista de tarefas associada a um determinado índice (index) na estrutura Scheduler (sc). Essa função utiliza um loop para iterar sobre os nós da lista de tarefas. Para cada nó, a função recupera a tarefa correspondente (task).

Se a tarefa ainda tiver um tempo de execução (burst) maior que zero, verifica-se se o tempo de execução restante é maior que o quantum definido (QUANTUM). Se for, a função run é chamada para executar a tarefa pelo tempo de quantum. Em seguida, o tempo de execução da tarefa é diminuído pelo valor do quantum.

Caso o tempo de execução restante seja menor ou igual ao quantum, a função run é chamada para executar a tarefa pelo tempo de execução restante. O tempo de execução da tarefa é definido como zero e é exibida uma mensagem indicando que a tarefa foi concluída. Em seguida, o nó atual é removido da lista de tarefas (list_rm) e é atualizado o ponteiro para o próximo nó (cur = next).

Após realizar as operações relacionadas a uma tarefa, o ponteiro cur é movido para o próximo nó na lista. Se o ponteiro cur chegar ao final da lista (tornar-se nulo), ele é posicionado no início da lista (sc.task_lists[index].head). Esse processo continua até que todas as tarefas tenham sido executadas ou até que a lista de tarefas correspondente a index na estrutura Scheduler esteja vazia. Quando a lista de tarefas estiver vazia, é exibida uma mensagem indicando que todas as tarefas foram concluídas.

Códigos

Github - Scheduler

First-Come, First-Served

```c
#include "schedule.h"
#include "list.h"
#include "task.h"
#include <stdio.h>
#include <stdlib.h>

struct Scheduler {
```

```
    List tasks;
    int last_id;
};

Scheduler* sched_new(){
    Scheduler* sc = malloc(sizeof(*sc));
    sc->tasks = list_new();
    sc->last_id = 0;
    return sc;
}

void sched_consume_list(Scheduler* sc, List* l){
    list_del(&sc->tasks);
    sc->tasks = *l;
    *l = (List){.len = 0, .head = NULL};
}

void sched_remove(Scheduler* sc, Task* task){
    list_rm(&sc->tasks, task);
}

void sched_run(Scheduler* sc){
    if (sc == NULL) {
        printf("Scheduler not initialized.\n");
        return;
    }

    if (sc->tasks.len == 0) {
        printf("No tasks in the scheduler.\n");
        return;
    }

    ListNode* cur = sc->tasks.head;
```

```
    while (cur != NULL) {
        Task* task = cur->task;

        run(task, task->burst);
        task->burst = 0;
         printf("Task [%s] [%d] finished.\n", task->name,
task->tid);
        ListNode* next = cur->next;
        list_rm(&sc->tasks, task);
        cur = next;
        continue;
    }
}


void sched_add(Scheduler* sc, char *name, int priority,
int burst){
    Task* new_task = malloc(sizeof(*new_task));
    sc->last_id += 1;
      *new_task  =  task_new(name,  sc->last_id,  priority,
burst);
    list_add(&sc->tasks, new_task);
}

void sched_del(Scheduler* sc){
    if(sc == NULL){ return; }
    list_del(&sc->tasks);
    free(sc);
}
```

Round-Robin

```
#include <stdio.h>
#include <stdlib.h>
#include "schedule.h"
```

```
struct Scheduler{
    List tasks;
    int last_id;
};

Scheduler* sched_new(){
                            Scheduler*          sc          =
(Scheduler*)malloc(sizeof(Scheduler));
    sc->tasks = list_new();
    sc->last_id = 0;

    return sc;
}

void sched_consume_list(Scheduler* sc, List* l){
    list_del(&sc->tasks);
    sc->tasks = *l;
    *l = (List){.len = 0, .head = NULL};
}

void sched_remove(Scheduler* sc, Task* task){
    list_rm(&sc->tasks, task);
}

void sched_run(Scheduler* sc) {
    if (sc == NULL) {
        printf("Scheduler not initialized.\n");
        return;
    }

    if (sc->tasks.len == 0) {
        printf("No tasks in the scheduler.\n");
        return;
```

```
    }

    ListNode* cur = sc->tasks.head;
    while (cur != NULL) {
        Task* task = cur->task;

        if (task->burst > 0) {
            if (task->burst > QUANTUM) {
                run(task, QUANTUM);
                task->burst -= QUANTUM;
            }
            else {
                run(task, task->burst);
                task->burst = 0;
                    printf("Task [%s] [%d] finished.\n",
task->name, task->tid);
                ListNode* next = cur->next;
                list_rm(&sc->tasks, task);
                cur = next;
                continue;
            }
        }

        cur = cur->next;

        if (cur == NULL) {
            cur = sc->tasks.head;
        }

        if(sc->tasks.len == 0){
            printf("All tasks finished.\n");
            break;
        }
    }
```

```
}

void sched_add(Scheduler* sc, char *name, int priority,
int burst){
    Task* new_task = malloc(sizeof(*new_task));
    sc->last_id += 1;
      *new_task  =  task_new(name,  sc->last_id,  priority,
burst);

    list_add(&sc->tasks, new_task);
}

void sched_del(Scheduler* sc){
    list_del(&sc->tasks);
    free(sc);
}
```

Round-Robin com Prioridade

```
#include "schedule.h"
#include <stdio.h>
#include <stdlib.h>

#define UNUSED(x) (void)(x)

struct Scheduler {
    List task_lists[MAX_PRIORITY - MIN_PRIORITY];
    int last_id;
};

static
int clamp(int lower, int n, int upper){
    if(n < lower){
        n = lower;
    } else if (n > upper){
```

```
        n = upper;
    }
    return n;
}

Scheduler* sched_new(){
    Scheduler* sc = malloc(sizeof(*sc));
    sc->last_id = 0;
      for(int  i  =  0;  i  <  MAX_PRIORITY  -  MIN_PRIORITY;
i++){
        sc->task_lists[i] = list_new();
    }
    return sc;
}

void sched_consume_list(Scheduler* sc, List* l){
    for(int i = 0; i < l->len; i += 1){
        ListNode* node = list_at(l, i);
        if(node){
            Task* t = node->task;
            node->task = NULL;
                int  p  =  clamp(MIN_PRIORITY,  t->priority,
MAX_PRIORITY);
            list_add(&sc->task_lists[p], t); }
        }
    list_del(l);
}

void rr_exec(Scheduler sc, int index) {
    ListNode* cur = sc.task_lists[index].head;
    while (cur != NULL) {
        Task* task = cur->task;

        if (task->burst > 0) {
```

```
            if (task->burst > QUANTUM) {
                run(task, QUANTUM);
                task->burst -= QUANTUM;
            }
            else {
                run(task, task->burst);
                task->burst = 0;
                    printf("Task [%s] [%d] finished.\n",
task->name, task->tid);
                ListNode* next = cur->next;
                list_rm(&sc.task_lists[index], task);
                cur = next;
                continue;
            }
        }

        cur = cur->next;

        if (cur == NULL) {
            cur = sc.task_lists[index].head;
        }

        if(sc.task_lists[index].len == 0){
            printf("All tasks finished.\n");
            break;
        }
    }
}

void sched_run(Scheduler* sc){
    for(int i = MAX_PRIORITY - MIN_PRIORITY -1; i >= 0;
i--){
        rr_exec(*sc, i);
    }
```

```
}

void sched_remove(Scheduler* sc, Task* task){
    UNUSED(sc); UNUSED(task);
     fprintf(stderr, "sched_remove() is not supported by
priority RR\n");
    abort();
}

void sched_add(Scheduler* sc, char *name, int priority,
int burst){
    int p = clamp(MIN_PRIORITY, priority, MAX_PRIORITY);

    Task* new_task = malloc(sizeof(*new_task));
    sc->last_id += 1;
    *new_task = task_new(name, sc->last_id, p, burst);

    list_add(&sc->task_lists[p], new_task);
}

void sched_del(Scheduler* sc){
    if(sc == NULL){ return; }
    free(sc);
}
```

Resultados

First-Come, First-Served

Running task = [T5] [1] [50] for 50 units.

Task [T5] [4] finished.

Running task = [T4] [1] [50] for 50 units.

Task [T4] [3] finished.

Running task = [T3] [1] [50] for 50 units.

Task [T3] [2] finished.

Running task = [T2] [1] [50] for 50 units.
Task [T2] [1] finished.
Running task = [T1] [1] [50] for 50 units.
Task [T1] [0] finished.

A execução começa pela tarefa T5, que é a primeira da lista. Ela é executada por completo, utilizando 50 unidades de tempo. Após a conclusão da execução da tarefa T5, é exibida uma mensagem indicando que a tarefa foi concluída.

Em seguida, a tarefa T4 é executada, novamente utilizando as 50 unidades de tempo. Após sua conclusão, é exibida uma mensagem correspondente.

Esse processo é repetido para as tarefas T3, T2 e T1, onde cada uma delas é executada por completo, uma após a outra, até a conclusão das 50 unidades de tempo de execução. Para cada tarefa concluída, é exibida uma mensagem correspondente.

Ao final, todas as tarefas foram executadas em ordem de chegada, utilizando seus tempos de execução completos.

Round Robin

Running task = [T5] [1] [50] for 10 units.
Running task = [T4] [1] [50] for 10 units.
Running task = [T3] [1] [50] for 10 units.
Running task = [T2] [1] [50] for 10 units.
Running task = [T1] [1] [50] for 10 units.
Running task = [T5] [1] [40] for 10 units.
Running task = [T4] [1] [40] for 10 units.
Running task = [T3] [1] [40] for 10 units.
Running task = [T2] [1] [40] for 10 units.
Running task = [T1] [1] [40] for 10 units.
Running task = [T5] [1] [30] for 10 units.
Running task = [T4] [1] [30] for 10 units.
Running task = [T3] [1] [30] for 10 units.
Running task = [T2] [1] [30] for 10 units.
Running task = [T1] [1] [30] for 10 units.
Running task = [T5] [1] [20] for 10 units.
Running task = [T4] [1] [20] for 10 units.

Running task = [T3] [1] [20] for 10 units.
Running task = [T2] [1] [20] for 10 units.
Running task = [T1] [1] [20] for 10 units.
Running task = [T5] [1] [10] for 10 units.
Task [T5] [4] finished.
Running task = [T4] [1] [10] for 10 units.
Task [T4] [3] finished.
Running task = [T3] [1] [10] for 10 units.
Task [T3] [2] finished.
Running task = [T2] [1] [10] for 10 units.
Task [T2] [1] finished.
Running task = [T1] [1] [10] for 10 units.
Task [T1] [0] finished.

O resultado apresenta a execução de cinco tarefas (T1, T2, T3, T4, T5) com um tempo de execução de 50 unidades cada. Cada tarefa é executada em ordem, seguindo o algoritmo Round-Robin, onde cada tarefa recebe uma fatia de tempo chamada "quantum" para ser executada.

Na primeira iteração, a tarefa T5 é executada por 50 unidades de tempo, pois o tempo de execução restante é igual ao quantum. Após a conclusão da execução da tarefa T5, é exibida uma mensagem indicando que a tarefa foi concluída.

Na segunda iteração, a tarefa T4 é executada por 50 unidades de tempo. Da mesma forma, após a conclusão da execução da tarefa T4, é exibida uma mensagem indicando que a tarefa foi concluída.

Esse processo é repetido para as tarefas T3, T2 e T1, onde cada uma delas é executada por 50 unidades de tempo. Após a conclusão da execução de cada tarefa, é exibida uma mensagem correspondente.

Ao final da execução das cinco tarefas, todas as tarefas foram concluídas e é exibida uma mensagem indicando que todas as tarefas foram finalizadas.

Round Robin com Prioridade

Running task = [T5] [3] [50] for 10 units.
Running task = [T5] [3] [40] for 10 units.
Running task = [T5] [3] [30] for 10 units.

```
Running task = [T5] [3] [20] for 10 units.
Running task = [T5] [3] [10] for 10 units.
Task [T5] [4] finished.
Running task = [T3] [2] [50] for 10 units.
Running task = [T4] [2] [50] for 10 units.
Running task = [T3] [2] [40] for 10 units.
Running task = [T4] [2] [40] for 10 units.
Running task = [T3] [2] [30] for 10 units.
Running task = [T4] [2] [30] for 10 units.
Running task = [T3] [2] [20] for 10 units.
Running task = [T4] [2] [20] for 10 units.
Running task = [T3] [2] [10] for 10 units.
Task [T3] [2] finished.
Running task = [T4] [2] [10] for 10 units.
Task [T4] [3] finished.
Running task = [T1] [1] [50] for 10 units.
Running task = [T2] [1] [50] for 10 units.
Running task = [T1] [1] [40] for 10 units.
Running task = [T2] [1] [40] for 10 units.
Running task = [T1] [1] [30] for 10 units.
Running task = [T2] [1] [30] for 10 units.
Running task = [T1] [1] [20] for 10 units.
Running task = [T2] [1] [20] for 10 units.
Running task = [T1] [1] [10] for 10 units.
Task [T1] [0] finished.
Running task = [T2] [1] [10] for 10 units.
Task [T2] [1] finished.
```

A execução começa pela tarefa T5, que possui prioridade 3. Ela é executada por 10 unidades de tempo, seguindo o esquema Round Robin. A cada fatia de 10 unidades, a tarefa é interrompida e volta para o final da fila de tarefas com a mesma prioridade. Esse processo se repete até que a tarefa T5 complete seu tempo de execução total de 50 unidades.

Após a conclusão da tarefa T5, a execução passa para as tarefas T3 e T4, que possuem prioridade 2. As tarefas T3 e T4 são executadas da mesma forma que a tarefa T5, em fatias de

10 unidades de tempo, seguindo o esquema Round Robin. Cada tarefa é executada até completar seu tempo total de execução, interrompendo-se a cada fatia de 10 unidades e retornando ao final da fila de tarefas com a mesma prioridade.

Após a conclusão das tarefas T3 e T4, a execução passa para as tarefas T1 e T2, que possuem prioridade 1. O mesmo esquema Round Robin é aplicado, com as tarefas sendo executadas em fatias de 10 unidades de tempo até completarem seus tempos totais de execução.

A cada vez que uma tarefa é concluída, é exibida uma mensagem indicando a finalização da tarefa. A execução continua até que todas as tarefas tenham sido concluídas.

Conclusão

Round Robin, Round Robin com Prioridade e FCFS (First-Come, First-Served) são algoritmos de escalonamento comuns utilizados em sistemas operacionais. Cada um desses algoritmos possui vantagens e desvantagens distintas que devem ser consideradas ao escolher o mais adequado para um determinado ambiente de execução.

O algoritmo Round Robin é caracterizado pela distribuição equitativa do tempo de CPU entre os processos. Ele garante uma justa alocação de recursos, uma vez que cada processo recebe uma fatia de tempo igual antes de ser interrompido e ceder a CPU para o próximo processo. Essa abordagem é especialmente útil em sistemas multitarefa, pois garante uma resposta rápida para todos os processos. No entanto, a desvantagem do Round Robin é que ele pode levar a uma sobrecarga de comutação de contexto, já que os processos são interrompidos em intervalos regulares, mesmo que ainda não tenham concluído sua execução.

O Round Robin com Prioridade é uma extensão do algoritmo Round Robin, no qual cada processo recebe uma prioridade associada. Isso permite que processos de alta prioridade sejam executados com mais frequência em comparação com os de baixa prioridade. Dessa forma, o sistema pode dar mais importância aos processos críticos ou interativos, garantindo um melhor desempenho em cenários específicos. No entanto, uma desvantagem deste algoritmo é que os processos de baixa prioridade podem sofrer de inanição, já que podem não receber tempo suficiente de CPU para concluir suas tarefas.

Por outro lado, o algoritmo FCFS é bastante simples e intuitivo. Ele executa os processos na ordem em que chegam, sem levar em consideração a duração ou prioridade. A principal vantagem do FCFS é a simplicidade de implementação, pois não requer nenhum mecanismo adicional de priorização ou divisão do tempo de CPU. No entanto, ele pode levar

a um problema conhecido como "efeito de latência" ou "starvation", onde processos de longa duração podem bloquear processos de curta duração que estão aguardando na fila. Além disso, o FCFS pode não ser eficiente em termos de tempo de resposta, especialmente em sistemas com variação significativa de tempo de execução dos processos.

Em resumo, o algoritmo Round Robin é adequado para ambientes multitarefa, onde a equidade na alocação de recursos é importante. O Round Robin com Prioridade é vantajoso em cenários onde diferentes processos têm níveis de prioridade distintos, permitindo uma execução mais eficiente dos processos críticos. Por fim, o FCFS é simples de implementar, mas pode apresentar problemas de latência e não ser eficiente em termos de tempo de resposta. A escolha do algoritmo de escalonamento mais adequado dependerá das características do sistema, das prioridades dos processos e dos requisitos de desempenho desejados.