作業系統 #HW2 Programming Projects

Team Member

- 資工三 110590011 劉承軒 - 整理程式、撰寫文件
- 資工三 110590018 劉承翰 - Chap.5、debug
- 資工三 110590056 林星主 - Chap.4

Chap.5 Scheduling Algorithms

SJF

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

typedef struct {
    int process_id;
    int burst_time;
} Process;

void SJF(Process processes[], int n) {
    int i, j, pos, temp;
    float total = 0, avg_wt, avg_tat;
    int *wait_time = malloc(sizeof(int) * n);
    int *turnaround_time = malloc(sizeof(int) * n);

    // 根據 burst_time 對進程進行排序
    for (i = 0; i < n; i++) {
        pos = i;
        for (j = i + 1; j < n; j++) {
            if (processes[j].burst_time < processes[pos].burst_time) {
                pos = j;
            }
        }

        // 交換位置
        temp = processes[i].burst_time;
        processes[i].burst_time = processes[pos].burst_time;
        processes[pos].burst_time = temp;

        temp = processes[i].process_id;
        processes[i].process_id = processes[pos].process_id;
        processes[pos].process_id = temp;
    }

    // 計算等待時間和周轉時間
    wait_time[0] = 0; // 第一個進程的等待時間為0

    for (i = 1; i < n; i++) {
        wait_time[i] = 0;
```

```
        for (j = 0; j < i; j++)
            wait_time[i] += processes[j].burst_time;

        total += wait_time[i];
    }

    avg_wt = total / n; // 平均等待時間
    total = 0;

    printf("\nProcess ID\tBurst Time\tWaiting Time\tTurnaround Time");
    for (i = 0; i < n; i++) {
        turnaround_time[i] = processes[i].burst_time + wait_time[i]; // 周轉時間
        total += turnaround_time[i];
        printf("\n%d\t\t%d\t\t%d\t\t%d", processes[i].process_id,
processes[i].burst_time, wait_time[i], turnaround_time[i]);
    }

    avg_tat = total / n; // 平均周轉時間
    printf("\n\nAverage Waiting Time = %.2f", avg_wt);
    printf("\nAverage Turnaround Time = %.2f\n", avg_tat);
}

int main() {
    Process processes[] = {{1, 6}, {2, 8}, {3, 7}, {4, 3}};
    int n = sizeof(processes)/sizeof(processes[0]);
    SJF(processes, n);
    return 0;
}
```

 image

FCFS

```
#include <stdio.h>

// 定義進程結構
typedef struct {
    int process_id;
    int arrival_time;
    int burst_time;
} Process;

// 函數原型聲明
void FCFS(Process processes[], int n);

// 主函數
int main(int argc, char **argv) {
    Process processes[] = {{1, 0, 10}, {2, 1, 5}, {3, 2, 8}};
    int n = sizeof(processes) / sizeof(processes[0]);
    FCFS(processes, n);
```

```
    return 0;
}

void FCFS(Process processes[], int n) {
    int wait_time = 0;
    int total_wait_time = 0;
    int total_turnaround_time = 0;

    printf("Process ID\tBurst Time\tWait Time\tTurnaround Time\n");

    for (int i = 0; i < n; i++) {
        int turnaround_time = wait_time + processes[i].burst_time;
        printf("%d\t\t%d\t\t%d\t\t%d\n", processes[i].process_id,
processes[i].burst_time, wait_time, turnaround_time);

        total_wait_time += wait_time;
        total_turnaround_time += turnaround_time;
        wait_time += processes[i].burst_time;
    }

    printf("\nAverage Waiting Time = %.2f\n", (float)total_wait_time / n);
    printf("Average Turnaround Time = %.2f\n", (float)total_turnaround_time / n);
}
```

 image

Priority-based

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

typedef struct {
    int process_id;
    int burst_time;
    int priority;
} Process;

void SJF(Process processes[], int n) {
    int i, j, pos, temp;
    float total = 0, avg_wt, avg_tat;
    int *wait_time = malloc(sizeof(int) * n);
    int *turnaround_time = malloc(sizeof(int) * n);

    // 根據 burst_time 對進程進行排序
    for (i = 0; i < n; i++) {
        pos = i;
        for (j = i + 1; j < n; j++) {
            if (processes[j].priority < processes[pos].priority) {
                pos = j;
            }
```

```
            }

            // 交換位置
            temp = processes[i].burst_time;
            processes[i].burst_time = processes[pos].burst_time;
            processes[pos].burst_time = temp;

            temp = processes[i].process_id;
            processes[i].process_id = processes[pos].process_id;
            processes[pos].process_id = temp;

            temp = processes[i].priority;
            processes[i].priority= processes[pos].priority;
            processes[pos].priority = temp;
        }

        // 計算等待時間和周轉時間
        wait_time[0] = 0; // 第一個進程的等待時間為0

        for (i = 1; i < n; i++) {
            wait_time[i] = 0;
            for (j = 0; j < i; j++)
                wait_time[i] += processes[j].burst_time;

            total += wait_time[i];
        }

        avg_wt = total / n; // 平均等待時間
        total = 0;

        printf("\nProcess ID\tBurst Time\tPriority\tWaiting Time\tTurnaround Time");
        for (i = 0; i < n; i++) {
            turnaround_time[i] = processes[i].burst_time + wait_time[i]; // 周轉時間
            total += turnaround_time[i];
            printf("\n%d\t\t%d\t\t%d\t\t%d\t\t%d", processes[i].process_id,
    processes[i].burst_time,processes[i].priority, wait_time[i], turnaround_time[i]);
        }

        avg_tat = total / n; // 平均周轉時間
        printf("\n\nAverage Waiting Time = %.2f", avg_wt);
        printf("\nAverage Turnaround Time = %.2f\n", avg_tat);
    }

    int main() {
        Process processes[] = {{1, 6,3}, {2, 8,1}, {3, 7,2}, {4, 3,4}};
        int n = sizeof(processes)/sizeof(processes[0]);
        SJF(processes, n);
        return 0;
    }
```

Round-Robin

```
#include <stdio.h>
#include <stdlib.h>
#define TIME_QUANTUM 2

typedef struct {
    int process_id;
    int burst_time;
} Process;

void RoundRobin(Process processes[], int n) {
    int i, total = 0, x, counter = 0, time_quantum;
    int wait_time = 0, turnaround_time = 0, remain;
    int *remaining_burst = malloc(sizeof(int)*n);
    remain = n;
    for (i = 0; i < n; i++)
        remaining_burst[i] = processes[i].burst_time;

    printf("Process ID\t\tRemaining Burst Time\n");
    while(remain>0)
    {
        for(int i = 0; i < n; i++)
        {
            if(remaining_burst[i]<TIME_QUANTUM)
            {
                remain--;
                remaining_burst[i] = 0;
                printf("Process[%d] is finish\n",processes[i].process_id);
            }
            else
            {
                remaining_burst[i]-=TIME_QUANTUM;
                if(remaining_burst[i] == 0)
                {
                    printf("Process[%d] is finish\n",processes[i].process_id);
                }
                else
                {
                    printf("Process[%d]\t\t\t%d\n", processes[i].process_id,
remaining_burst[i]);
                }
            }

        }
    }

}

int main() {
    Process processes[] = {{1, 6}, {2, 8}, {3, 7}, {4, 3}};
    int n = sizeof(processes)/sizeof(processes[0]);
```

```
    RoundRobin(processes, n);
    return 0;
}
```

Priority with round-robin

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

typedef struct {
    int process_id;
    int burst_time;
    int priority;
} Process;

int time_quantum = 4;

int compare_priority(const void *a, const void *b) {
    Process *p1 = (Process *)a;
    Process *p2 = (Process *)b;
    return p1->priority - p2->priority;
}

void priorityRoundRobin(Process processes[], const int n) {
    // Sort processes by priority
    qsort(processes, n, sizeof(Process), compare_priority);

    int total_time = 0, done = 0, current = 0;
    int remain_burst[n];
    for (int i = 0; i < n; i++) {
        remain_burst[i] = processes[i].burst_time;
    }

    printf("Process ID\tPriority\tBurst Time\tTime Quantum\n");

    while (!done) {
        done = 1; // Assume all processes are handled until proven otherwise
        for (int i = 0; i < n; i++) {
            if (remain_burst[i] > 0) {
                done = 0; // There is still a process that needs to be handled
                if (remain_burst[i] > time_quantum) {
                    total_time += time_quantum;
                    remain_burst[i] -= time_quantum;
                    printf("Process[%d]\t%d\t\t%d\t\t%d\n",
processes[i].process_id, processes[i].priority, remain_burst[i], time_quantum);
                } else {
                    total_time += remain_burst[i];
                    printf("Process[%d]\t%d\t\t%d\t\t%d\n",
processes[i].process_id, processes[i].priority, 0, remain_burst[i]);
```

```
                    remain_burst[i] = 0;
                }
            }
        }
    }

    printf("\nTotal Time Spent: %d\n", total_time);
}

int main() {
    Process processes[] = {{1, 10, 1}, {2, 15, 2}, {3, 20, 1}, {4, 20, 2}};
    int n = sizeof(processes)/sizeof(processes[0]);
    priorityRoundRobin(processes, n);
    return 0;
}
```

Chap.4 Sudoku solution validator

Description

```
#include <stdio.h>
#include <stdlib.h>
#include <stdint.h>
#include <pthread.h>

#define SIZE 9 // 數獨大小

int sudoku[SIZE][SIZE] = {
    {5, 8, 7, 1, 3, 6, 9, 2, 4},
    {9, 2, 4, 7, 5, 8, 1, 6, 3},
    {3, 2, 6, 9, 2, 4, 8, 5, 7},
    {2, 4, 3, 6, 1, 9, 7, 8, 5},
    {7, 9, 8, 4, 7, 5, 3, 1, 2},
    {6, 5, 1, 8, 2, 3, 4, 9, 6},
    {1, 6, 2, 3, 8, 7, 5, 4, 9},
    {8, 7, 9, 5, 4, 1, 6, 3, 8},
    {4, 3, 5, 2, 6, 9, 2, 7, 1}
};

void *checkRow(void *arg) {
    int row = *((int *)arg);
    free(arg);
    int check[SIZE] = {0};
    for (int i = 0; i < SIZE; i++) {
        int num = sudoku[row][i];
        if (num < 1 || num > SIZE || check[num - 1]) {
            pthread_exit((void *)0);
        }
```

```
            check[num - 1] = 1;
        }
        pthread_exit((void *)1);
    }

    void *checkColumn(void *arg) {
        int col = *((int *)arg);
        free(arg);
        int check[SIZE] = {0};
        for (int i = 0; i < SIZE; i++) {
            int num = sudoku[i][col];
            if (num < 1 || num > SIZE || check[num - 1]) {
                pthread_exit((void *)0);
            }
            check[num - 1] = 1;
        }
        pthread_exit((void *)1);
    }

    void *checkSubgrid(void *arg) {
        int startRow = *((int *)arg);
        int startCol = *((int *)(arg + sizeof(int)));
        free(arg);
        int check[SIZE] = {0};
        for (int i = startRow; i < startRow + 3; i++) {
            for (int j = startCol; j < startCol + 3; j++) {
                int num = sudoku[i][j];
                if (num < 1 || num > SIZE || check[num - 1]) {
                    pthread_exit((void *)0);
                }
                check[num - 1] = 1;
            }
        }
        pthread_exit((void *)1);
    }

    int main() {
        pthread_t threads[SIZE];
        int *arg;
        void *result_column, *result_row, *result_subgrid;
        // 检查行
        for (int i = 0; i < SIZE; i++) {
            arg = malloc(sizeof(int));
            *arg = i;
            pthread_create(&threads[i], NULL, checkRow, arg);
        }
        // 等待行检查完成
        for (int i = 0; i < SIZE; i++) {
            pthread_join(threads[i], &result_row);
            if ((intptr_t)result_row == 0) {
                printf("Invalid solution in row %d\n", i);
                break;
                // return 0; // 退出程序，不再继续检查
            }
```

```
    }

    // 检查列
    for (int i = 0; i < SIZE; i++) {
        arg = malloc(sizeof(int));
        *arg = i;
        pthread_create(&threads[i], NULL, checkColumn, arg);
    }
    // 等待列检查完成
    for (int i = 0; i < SIZE; i++) {
        pthread_join(threads[i], &result_column);
        if ((intptr_t)result_column == 0) {
            printf("Invalid solution in column %d\n", i);
            break;
            // return 0; // 退出程序
        }
    }
    int indices[SIZE][2] = {{0, 0}, {0, 3}, {0, 6}, {3, 0}, {3, 3}, {3, 6}, {6,
0}, {6, 3}, {6, 6}};

    // 检查子网格
    for (int i = 0; i < SIZE; i++) {
        arg = malloc(2 * sizeof(int));
        arg[0] = indices[i][0];
        arg[1] = indices[i][1];
        pthread_create(&threads[i], NULL, checkSubgrid, arg);
    }
    // 等待子网格检查完成
    for (int i = 0; i < SIZE; i++) {
        pthread_join(threads[i], &result_subgrid);
        if ((intptr_t)result_subgrid == 0) {
            printf("Invalid solution in subgrid %d starting at (%d, %d)\n", i,
indices[i][0], indices[i][1]);
            return 0; // 退出程序
        }
    }
    if((intptr_t)result_subgrid && (intptr_t)result_column &&
(intptr_t)result_row)
    {
        printf("Valid Sudoku solution\n");

    }
    return 0;
}
```

image