تمرین سوم سیستم‌های عامل

اشکان شکیبا (۹۹۳۱۰۳۰)

سوال اول

در پردازه فرزند، مقدار متغیر pid که حاصل تابع fork است صفر و مقدار pid1 برابر آی‌دی خود پردازه می‌شود. در پردازه پدر نیز حاصل تابع fork برابر آی‌دی پردازه فرزند و مقدار pid1 برابر آی‌دی خود پردازه می‌شود.

بنابراین خروجی‌ها به صورت A: 0، B: 2603، C: 2603 و D: 2600 هستند.

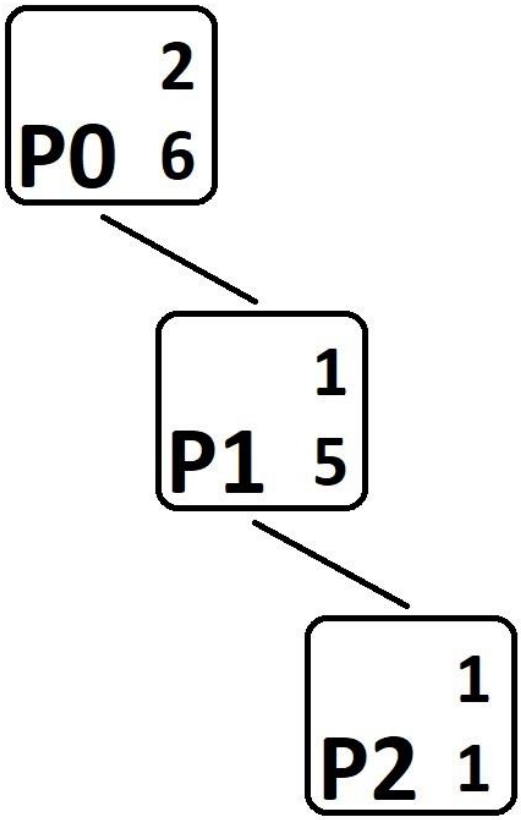
سوال دوم

ابتدا پردازه P0 پردازه P1 را ساخته و وارد if اول می‌شود و با افزایش count خود به ۲، آن را پرینت می‌کند. سپس از if دوم رد شده و با برقراری شرط pid وارد if سوم می‌شود و با افزایش count به ۶، آن را پرینت می‌کند.

در پردازه P1 اولین if به دلیل عدم برقراری شرط pid رد شده و در دومین if مقدار count به ۱ افزایش می‌یابد. سپس پردازه P2 ساخته شده و مقدار count پرینت می‌شود. در ادامه نیز با برقراری شرط pid2 و ورود به سومین if، مقدار count به ۵ افزایش یافته و پرینت می‌شود.

در پردازه P2 ابتدا در if دوم مشابه پردازه قبل مقدار count برابر با یک پرینت می‌شود. سپس به دلیل عدم برقراری هیچ یک از شرط‌های pid و pid2 از سومین if رد شده و مجددا همان مقدار قبلی count پرینت می‌شود.

درخت پردازه‌ها:



سوال سوم

زمانی که برنامه به خط B برسد حداقل ۲ ترد فعال وجود دارند، یکی ترد اصلی برنامه و دیگری ترد اول از آرایه تردها که مربوط به تابع print_message1 است. ترد سومی نیز خواهیم داشت که دومین ترد آرایه و مربوط به تابع print_message2 است و ممکن است تا آن لحظه فعال نشده باشد، یا فعال باشد، و یا حتی پیش‌تر فعالیت آن به اتمام رسیده باشد.

بنابراین در زمان اجرای خط B، ۲ یا ۳ ترد فعال وجود خواهند داشت.

سوال چهارم

الف) با توجه به همزمانی دو ترد برای افزایش مقدار shared_value، پدیده race condition رخ می‌دهد که بسته به میزان رخ دادن آن، مقدار نهایی می‌تواند عددی بین 1000000 تا 2000000 باشد.

ب) خیر، در صورت استفاده از فورک، هر یک از پردازه‌های فرزند و پدر، مقدار متفاوتی از متغیر را نگه‌داری می‌کنند. در واقع این یکی از اصلی‌ترین تفاوت‌های برنامه‌های مالتی پراسس و مالتی ترد است، که تردها فضای مشترک دارند اما پراسس‌ها ایزوله‌اند و هر یک PCB مختص به خود را دارند.

سوال پنجم

حالت اول: یکی از دستورها پس از پایان دیگری اجرا شود، خروجی ۴

حالت دوم: دستورها همزمان اجرا شده و ابتدا دستور اول پایان یابد، خروجی ۳

حالت سوم: دستورها همزمان اجرا شده و ابتدا دستور دوم پایان یابد، خروجی ۶

بخش عملی

جمع ماتریس به صورت تک ترد:

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

#define ROWS 100
#define COLS 100

int matrix[ROWS][COLS];
int thread[ROWS];
int m = 100;
int n = 100;

int main() {
    // initialize matrix
    for (int i = 0; i < m; i++) {
        for (int j = 0; j < n; j++) {
            matrix[i][j] = rand() % 10;
        }
    }

    // start time
    struct timespec start_time, end_time;
    clock_gettime(CLOCK_MONOTONIC, &start_time);

    // calculate sum of matrix
    int total_sum = 0;
    for (int row = 0; row < m; row++) {
        int sum = 0;
        for (int j = 0; j < n; j++) {
            sum += matrix[row][j];
        }
        thread[row] = sum;
    }
    for (int i = 0; i < m; i++) {
        total_sum += thread[i];
    }

    // end time
    double elapsed_time;
    clock_gettime(CLOCK_MONOTONIC, &end_time);
    elapsed_time = (end_time.tv_sec - start_time.tv_sec) +
(end_time.tv_nsec - start_time.tv_nsec) / 1000000000.0;
    printf("elapsed time: %f seconds\n", elapsed_time);

    return 0;
}
```

جمع ماتریس به صورت مالتی ترد:

```c
#include <time.h>
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>

#define ROWS 100
#define COLS 100

int matrix[ROWS][COLS];
int thread[ROWS];
int m = 100;
int n = 100;

void *row_sum(void *arg) {
    int j, sum = 0;
    int row = *((int *) arg);
    for (j = 0; j < n; j++) {
        sum += matrix[row][j];
    }
    thread[row] = sum;
    pthread_exit(NULL);
}

int main() {
    // initialize matrix
    for (int i = 0; i < m; i++) {
        for (int j = 0; j < n; j++) {
            matrix[i][j] = rand() % 10;
        }
    }

    // start time
    struct timespec start_time, end_time;
    clock_gettime(CLOCK_MONOTONIC, &start_time);

    // calculate sum of matrix
    pthread_t threads[m];
    int thread_args[m];
    int total_sum = 0;
    for (int i = 0; i < m; i++) {
        thread_args[i] = i;
        pthread_create(&threads[i], NULL, row_sum, (void *)
&thread_args[i]);
    }
    for (int i = 0; i < m; i++) {
        pthread_join(threads[i], NULL);
    }
    for (int i = 0; i < m; i++) {
        total_sum += thread[i];
    }
```

```
    // end time
    double elapsed_time;
    clock_gettime(CLOCK_MONOTONIC, &end_time);
    elapsed_time = (end_time.tv_sec - start_time.tv_sec) +
(end_time.tv_nsec - start_time.tv_nsec) / 1000000000.0;
    printf("elapsed time: %f seconds\n", elapsed_time);

    return 0;
}
```

زمانی که ماتریس کوچک و تعداد درایه‌های هر ردیف کم باشد، اجرای برنامه به شکل تک ترد بهینه‌تر است؛ چرا که هزینه ساخت تردها از میزان بهبود همزمان کردن محاسبات بیشتر می‌شود. البته که این هزینه با افزایش اندازه ماتریس کمرنگ‌تر شده و اجرای مالتی ترد عملیات راه حل بهینه‌تری خواهد بود. همچنین در ماتریس‌های بسیار بزرگ و با ردیف‌های زیاد، ممکن است تعداد تردها از حداکثر ظرفیت میزبانی سیستم بیشتر شده و منجر به اختلال شود.

ضرب ماتریس به صورت تک ترد:

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

#define M 1000

int matrix1[M][M];
int matrix2[M][M];
int matrix3[M][M];

int main() {
    // initialize matrices
    for (int i = 0; i < M; i++) {
        for (int j = 0; j < M; j++) {
            matrix1[i][j] = rand() % 10;
        }
    }
    for (int i = 0; i < M; i++) {
        for (int j = 0; j < M; j++) {
            matrix2[i][j] = rand() % 10;
        }
    }

    // start time
    struct timespec start_time, end_time;
    clock_gettime(CLOCK_MONOTONIC, &start_time);

    // calculate multiply of matrix
    for (int i = 0; i < M; i++) {
        for (int j = 0; j < M; j++) {
            for (int k = 0; k < M; k++) {
                matrix3[i][j] += matrix1[i][k] * matrix2[k][j];
            }
        }
    }

    // end time
    double elapsed_time;
    clock_gettime(CLOCK_MONOTONIC, &end_time);
    elapsed_time = (end_time.tv_sec - start_time.tv_sec) +
(end_time.tv_nsec - start_time.tv_nsec) / 1000000000.0;
    printf("elapsed time: %f seconds\n", elapsed_time);

    return 0;
}
```

ضرب ماتریس به صورت مالتی ترد:

```c
#include <time.h>
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>

#define M 1000

int matrix1[M][M];
int matrix2[M][M];
int matrix3[M][M];

void *multiply(void *arg) {
    int i = *((int *) arg), j, k;
    for (j = 0; j < M; j++) {
        for (k = 0; k < M; k++) {
            matrix3[i][j] += matrix1[i][k] * matrix2[k][j];
        }
    }
    pthread_exit(NULL);
}

int main() {
    // initialize matrices
    for (int i = 0; i < M; i++) {
        for (int j = 0; j < M; j++) {
            matrix1[i][j] = rand() % 10;
        }
    }
    for (int i = 0; i < M; i++) {
        for (int j = 0; j < M; j++) {
            matrix2[i][j] = rand() % 10;
        }
    }

    // start time
    struct timespec start_time, end_time;
    clock_gettime(CLOCK_MONOTONIC, &start_time);

    // calculate multiply of matrix
    pthread_t threads[M];
    srand(time(NULL));
    for (int i = 0; i < M; i++) {
        for (int j = 0; j < M; j++) {
            matrix1[i][j] = rand() % 10;
            matrix2[i][j] = rand() % 10;
        }
    }
    for (int i = 0; i < M; i++) {
        int *arg = malloc(sizeof(int));
        *arg = i;
        pthread_create(&threads[i], NULL, multiply, arg);
```

```
    }
    for (int i = 0; i < M; i++) {
        pthread_join(threads[i], NULL);
    }

    // end time
    double elapsed_time;
    clock_gettime(CLOCK_MONOTONIC, &end_time);
    elapsed_time = (end_time.tv_sec - start_time.tv_sec) +
(end_time.tv_nsec - start_time.tv_nsec) / 1000000000.0;
    printf("elapsed time: %f seconds\n", elapsed_time);

    return 0;
}
```

کد بخش دوم (search and sort):

```c
#include <time.h>
#include <stdio.h>
#include <pthread.h>

#define SIZE 1000

int a[SIZE] = {...};
int value = 2101;

void *sort(void *arg) {
    for (int i = 0; i < SIZE; i++) {
        for (int j = 0; j < SIZE - i - 1; j++) {
            if (a[j] > a[j + 1]) {
                int t = a[j];
                a[j] = a[j + 1];
                a[j + 1] = t;
            }
        }
    }
    printf("sorted\n");
    pthread_exit(NULL);
}

void *search(void *arg) {
    for (int i = 0; i < SIZE; i++) {
        if (a[i] == value) {
            printf("found at %d\n", i);
            pthread_exit(NULL);
        }
    }
    printf("not found in the array\n");
    pthread_exit(NULL);
}

int main() {
    // start time
    struct timespec start_time, end_time;
    clock_gettime(CLOCK_MONOTONIC, &start_time);

    // search and sort
    pthread_t thread1, thread2;
    pthread_create(&thread2, NULL, search, NULL);
    pthread_create(&thread1, NULL, sort, NULL);
    pthread_join(thread1, NULL);
    pthread_join(thread2, NULL);

    // end time
    double elapsed_time;
    clock_gettime(CLOCK_MONOTONIC, &end_time);
    elapsed_time = (end_time.tv_sec - start_time.tv_sec) +
(end_time.tv_nsec - start_time.tv_nsec) / 1000000000.0;
```

```
    printf("elapsed time: %f seconds\n", elapsed_time);

    return 0;
}
```

۱) اگر عدد مورد نظر برای جست‌وجو در آرایه مرتب‌شده زودتر از آرایه اولیه ظاهر شود، مرتب‌سازی آرایه پیش از جست‌وجو می‌تواند منجر به سریع‌تر یافتن عدد مورد نظر شود.

۲) به طور کلی بله، مگر اینکه تابع جست‌وجو به شکل دودویی و یا سایر الگوریتم‌های خاص مشابه پیاده‌سازی شده باشد.

۳) می‌توان ابتدا ترد مرتب‌سازی را join و سپس ترد جست‌وجو را اجرا کرد. این کار ممکن است مطابق بخش ۱، منجر به کاهش زمان جست‌وجو شود؛ اما به طور کلی به دلیل غیر همزمان کردن عملیات، احتمالا زمان اجرا را افزایش خواهد داد.

۴) نحوه بهینه‌سازی بیش از هر چیز به خواسته ما از برنامه بستگی دارد. مثلا اگر تنها هدف از مرتب‌سازی آرایه کمک به جست‌وجوی سریع‌تر در آن بوده باشد، می‌توان از مرتب‌سازی چشم‌پوشی کرده و از الگوریتم‌های جست‌وجوی سازگار با آرایه‌های نامرتب استفاده کرد.

۵) در چنین حالتی، بسیار محتمل است که مرتب‌سازی منجر به افزایش زمان جست‌وجو شود، به خصوص اگر داده مورد جست‌وجو پس از

مرتب‌سازی در انتهای آرایه قرار گیرد، زیرا تا پیش از آن در سراسر آرایه پراکنده است و شانس رسیدن به آن در ایندکس‌های کمتر وجود دارد.