به نام خدا

تهیه کننده: ابراهیم صدیقی تهیه کننده: ۹۹۳۱۰۹۸

سوال ۱)

1. دلیل اصلی Deadlock در برنامه نویسی چند هسته ای: Deadlock در برنامه نویسی چند هسته ای زمانی رخ می دهد که چندین رشته در انتظار یکدیگر هستند تا منابعی را که برای ادامه نیاز دارند آزاد کنند. این وضعیت منجر به standstill می شود که در آن هیچ رشته ای نمی تواند پیشرفت کند و باعث می شود کل سیستم متوقف شود.
2. شانس Deadlock در تابع داده شده: در تابع ارائه شده، به دلیل نحوه به دست آوردن و رها شدن lock ها، احتمال Deadlock وجود دارد. این تابع از lock های مجزا برای هر عنصر در آرایه استفاده می‌کند و در طول فرآیند زدن، نخ‌ها به طور همزمان دو lock می‌گیرند. اگر دو رشته سعی کنند lock ها را با ترتیب متفاوتی بدست آورند، Deadlock ممکن است رخ دهد. به عنوان مثال، اگر Thread A عنصر X را lock کند و سپس سعی کند عنصر Y را lock کند، در حالی که Thread B عنصر Y را lock کند و سپس سعی کند عنصر X را lock کند، یک وضعیت Deadlock ایجاد می شود.
3. حداکثر تعداد Thread ها برای جلوگیری از Deadlock: برای اطمینان از احتمال کمتر از 10٪ Deadlock در تابع، تعداد نخ ها باید به تعداد عناصر موجود در آرایه محدود شود. این محدودیت تضمین می‌کند که هر رشته فقط تلاش می‌کند دو lock را در یک زمان به دست آورد و احتمال تضاد اکتساب lock که منجر به Deadlock می‌شود را کاهش می‌دهد. بنابراین، حداکثر تعداد رشته هایی که می توان بدون بیش از 10 درصد احتمال Deadlock استفاده کرد، برابر طول آرایه است.
4.
5. بازنویسی کد برای جلوگیری از Deadlock: برای بازنویسی کد و حذف پتانسیل Deadlock، می توان به جای lock های جداگانه برای هر عنصر آرایه از یک lock جهانی استفاده کرد. با حصول اطمینان از اینکه thread ها lock ها را در یک نظم ثابت به دست می آورند، می توان از Deadlock ها جلوگیری کرد. در ادامه یک نسخه تغییر یافته از عملکرد shuffle با یک قفل جهانی است:

```
omp_lock_t global_lock;
omp_init_lock(&global_lock);

void shuffle(int* array, int len) {
    #pragma omp parallel
    {
        unsigned int a, b;
        unsigned int seed = omp_get_thread_num();

        for (int i = 0; i < 1000; i++) {
            a = rand_r(&seed) % len;
            do {
                b = rand_r(&seed) % len;
            } while (a == b);

            omp_set_lock(&global_lock);
            swap(&array[a], &array[b]);
            omp_unset_lock(&global_lock);
        }
    }

    omp_destroy_lock(&global_lock);
}

```

سوال ۲)

برای موازی کردن الگوریتم LCS با استفاده از وظایف OpenMP، می‌توانیم ماتریس را به تکه‌های کوچک‌تر تقسیم کنیم و هر تکه را به یک کار جداگانه اختصاص دهیم. این رویکرد به چندین کار اجازه می دهد تا روی بخش های مختلف ماتریس به طور همزمان کار کنند و زمان اجرای کلی را کاهش می دهد.

در ادامه اجرای موازی الگوریتم LCS با استفاده از وظایف OpenMP آورده شده است:

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

#define max(x, y) (((x) > (y)) ? (x) : (y))

int lcs(char *a, char *b, size_t a_len, size_t b_len)
{
    // Set up dynamic programming memoization matrix
    int(*dp)[a_len + 1][b_len + 1] = malloc(sizeof(int[a_len + 1][b_len + 1]));
    size_t y_len = a_len + 1;
    size_t x_len = b_len + 1;

    // Zero out first row and first column
    for(int i = 0; i < y_len; i++)
    {
        (*dp)[i][0] = 0;
    }

    for(int i = 0; i < x_len; i++)
    {
        (*dp)[0][i] = 0;
    }

    // Parallelize the LCS computation using OpenMP tasks
    #pragma omp parallel
    {
        #pragma omp single
        {
            for (int i = 1; i < y_len; i += 128)
            {
                for (int j = 1; j < x_len; j += 128)
                {
                    #pragma omp task
                    {
                        int end_i = i + 128 < y_len ? i + 128 : y_len;
                        int end_j = j + 128 < x_len ? j + 128 : x_len;

                        for (int k = i; k < end_i; k++)
                        {
                            for (int l = j; l < end_j; l++)
                            {
                                if (a[k-1] == b[l-1])
                                {
                                    (*dp)[k][l] = (*dp)[k - 1][l - 1] + 1;
                                }
                                else
                                {
                                    (*dp)[k][l] = max((*dp)[k - 1][l], (*dp)[k][l - 1]);
                                }
                            }
                        }
                    }
                }
```

```
                    }
                }
            }
        }
    }

    int retval = (*dp)[a_len][b_len];
    free(dp);
    return retval;
}

int main()
{
    size_t string_size = 1 << 15;

    char *str1 = malloc(string_size + 1);
    char *str2 = malloc(string_size + 1);
    srand(500); // Fix rand seed to have the same string for every run

    for(int i = 0; i < string_size; i++)
    {
        str1[i] = (rand() % 26) + 97;
        str2[i] = (rand() % 26) + 97;
    }

    // Null terminate strings
    str1[string_size] = '\0';
    str2[string_size] = '\0';

    printf("%d\n", lcs(str1, str2, string_size, string_size));

    return 0;
}

```