Московский Авиационный Институт

(Национальный Исследовательский Университет)

Институт №8 “Компьютерные науки и прикладная математика”

Кафедра №806 “Вычислительная математика и программирование”

Лабораторная работа №3 по курсу

«Операционные системы»

Группа: М8О-210Б-23

Студент: Жданович Е.Т.

Преподаватель: Бахарев В.Д.

Оценка: _____

Дата: 09.01.24

Москва, 2024

Постановка задачи

Вариант 7.

Составить программу на языке Си, обрабатывающую данные в многопоточном режиме. При обработки использовать стандартные средства создания потоков операционной системы. Два человека играют в кости. Правила игры следующие: каждый игрок делает бросок 2-ух костей K раз; побеждает тот, кто выбросил суммарно большее количество очков. Задача программы экспериментально определить шансы на победу каждого из игроков. На вход программе подается K, какой сейчас тур, сколько очков суммарно у каждого из игроков и количество экспериментов, которые должна произвести программа.

Общий метод и алгоритм решения

Использованные системные вызовы:

- **`pthread_create`**:
  Создаёт новый поток для выполнения функции.
- **`pthread_join`**:
  Ожидает завершения потока и блокирует выполнение до его завершения.
- **`pthread_mutex_lock` и `pthread_mutex_unlock`**:
  Захватывает и освобождает мьютекс для синхронизации доступа к общим данным между потоками.
- **`rand` и `srand`**:
  Генерируют случайные числа. `srand` используется для инициализации генератора случайных чисел, а `rand` для получения случайных чисел.
- **`scanf` и `printf`**:
  Используются для ввода и вывода данных, взаимодействуя с пользователем.
- **`time`**:
  Получает текущее время, которое может быть использовано, например, для инициализации генератора случайных чисел.

- **`perror`**:
  Выводит сообщение об ошибке, связанной с последней системной ошибкой.

Child1.c

1. **Проверка аргументов командной строки:**

Программа ожидает два аргумента: имена семафоров `sem_empty` и `sem_full`.
Если аргументы не переданы, выводится сообщение об ошибке, и программа завершает выполнение.

2. **Открытие семафоров:**

С помощью `sem_open` открываются два семафора:

`sem_empty`: отвечает за блокировку чтения до тех пор, пока данные не будут готовы для обработки.

`sem_full`: отвечает за сигнализацию о завершении обработки данных.
Если открытие семафоров завершается неудачей, программа выводит сообщение об ошибке и завершает выполнение.

3. **Подключение к разделяемой памяти:**

С помощью `shm_open` открывается объект разделяемой памяти, имя которого задано в константе `SHM_NAME`.
Если операция завершается неудачей, выводится сообщение об ошибке, и программа завершает выполнение.
С помощью `mmap` разделяемая память отображается в адресное пространство программы, предоставляя доступ к структуре `SharedData`.

4. **Ожидание готовности данных:** Программа вызывает `sem_wait`, ожидая, пока данные станут доступны для обработки.

5. **Обработка данных:** В цикле строка из разделяемой памяти преобразуется: каждый символ переводится в верхний регистр с использованием функции `toupper`.

6. **Обновление флага:** После обработки строка помечается как обработанная, устанавливая флаг в значение `1`.

7. **Сигнал о завершении обработки.**

8. **Освобождение ресурсов.**

9. **Завершение программы:** Программа успешно завершает выполнение с кодом `0`.

Child2.c

1. **Проверка аргументов командной строки**:

Программа ожидает два аргумента: имена семафоров `sem_empty` и `sem_full`.
Если аргументы не переданы, выводится сообщение об ошибке, и программа завершает выполнение.

2. **Открытие семафоров**:

С помощью `sem_open` открываются два семафора:
`sem_empty`: для управления доступом к разделяемой памяти.
`sem_full`: для ожидания завершения обработки данных другим процессом.
Если открытие семафоров завершается неудачей, программа выводит сообщение об ошибке и завершает выполнение.

3. **Подключение к разделяемой памяти**:

С помощью `shm_open` открывается объект разделяемой памяти, имя которого задано в константе `SHM_NAME`.
Если операция завершается неудачей, выводится сообщение об ошибке, и программа завершает выполнение.
С помощью `mmap` разделяемая память отображается в адресное пространство программы, предоставляя доступ к структуре `SharedData`.

4. **Ожидание обработки данных**:

Программа вызывает `sem_wait(sem_full)`, ожидая сигнал от другого процесса о завершении обработки данных (например, преобразования строки в верхний регистр).

5. **Обработка данных**:

В цикле символы строки из разделяемой памяти проверяются:
Если символ — пробел `' '`, он заменяется на символ подчеркивания `'_'`.

6. **Вывод обработанного сообщения**:

Сообщение из разделяемой памяти выводится на стандартный вывод:
Предварительно печатается строка `"Child 2 processed message: "`.
Затем — само сообщение.
Вывод завершается символом новой строки.

7. **Освобождение ресурсов**:

Освобождаются ресурсы:

Операция `munmap` удаляет отображение разделяемой памяти.
Операция `close` закрывает дескриптор разделяемой памяти.

8. **Завершение программы**:

Программа успешно завершает выполнение с кодом `0`.

Parent.c

1. **Ввод сообщения**:

Программа запрашивает у пользователя сообщение для обработки.
Пользователь вводит строку, которая записывается в разделяемую память.

2. **Создание общей памяти и семафоров**:

Создается общая память с помощью `shm_open()` и выделяется необходимое пространство с помощью `ftruncate()`.
Память отображается в адресное пространство программы через `mmap`.
Создаются два семафора: `sem_empty` для синхронизации записи в память; `sem_full` для синхронизации обработки данных.

3. **Создание дочерних процессов**:

С помощью `fork()` создаются два дочерних процесса:

Первый процесс запускает программу `child1` через `execl()`.
Второй процесс запускает программу `child2` через `execl()`.

4. **Передача данных и ожидание обработки**:

Родительский процесс использует `sem_post(sem_empty)` для сигнализации о готовности данных.
Родитель ожидает завершения дочерних процессов с помощью `wait()`.

5. **Вывод результата**:

После завершения обработки сообщение из разделяемой памяти выводится на экран.

6. **Очистка ресурсов**:

Семафоры закрываются и удаляются с помощью `sem_close()` и `sem_unlink()`.
Разделяемая память удаляется с помощью `munmap()` и `shm_unlink()`.

Код программы

```
#include <stdio.h>

#include <stdlib.h>

#include <pthread.h>
```

```
#include <time.h>

#include <unistd.h>

#define MAX_THREADS 4

typedef struct {

 int  K;

 int round;

 int score1;

 int score2;

} GameData;


pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;

int roll_die() {

 return rand() % 6 + 1;

}

void* experiment(void* arg) {

 GameData* gameData = (GameData*)arg;

int player1_score = 0, player2_score = 0;

for (int i = 0; i < gameData->K; i++) {
    player1_score += roll_die() + roll_die();
}

for (int i = 0; i < gameData->K; i++) {
    player2_score += roll_die() + roll_die();
}

pthread_mutex_lock(&mutex);


gameData->score1 += player1_score;
gameData->score2 += player2_score;

if (player1_score > player2_score) {
    gameData->round++;
}
pthread_mutex_unlock(&mutex);
```

```
pthread_exit(NULL);


    }

    int main() {

     int K, round, score1, score2, max_threads;

   printf("Enter the number of rolls per player (K): ");
   scanf("%d", &K);

   printf("Enter the current round number: ");
   scanf("%d", &round);

   printf("Enter the initial score for player 1: ");
   scanf("%d", &score1);

   printf("Enter the initial score for player 2: ");
   scanf("%d", &score2);

   printf("Enter the maximum number of threads (experiments) to run
   simultaneously: ");
   scanf("%d", &max_threads);

   srand(time(NULL));

   pthread_t threads[max_threads];
   GameData gameData = {K, round, score1, score2};

   for (int i = 0; i < max_threads; i++) {
       if (pthread_create(&threads[i], NULL, experiment, (void*)&gameData)
   != 0) {
           perror("Error creating thread");
           return 1;
       }
   }


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

   printf("\nTotal score of player 1: %d\n", gameData.score1);
   printf("Total score of player 2: %d\n", gameData.score2);
   if (gameData.score1 > gameData.score2) {
       printf("Player 1 wins!\n");
   } else if (gameData.score1 < gameData.score2) {
       printf("Player 2 wins!\n");
   } else {
       printf("It's a draw!\n");
   }

   return 0;
```

```
}
```

Протокол работы программы

Тестирование:

```
lizka@LizaAlisa:~/ЛАБЫ_ОС/Лаба2$ ./main
Enter the number of rolls per player (K): 5
Enter the current round number: 34
Enter the initial score for player 1: 6
Enter the initial score for player 2: 8
Enter the maximum number of threads (experiments) to run simultaneously: 400

Total score of player 1: 14020
Total score of player 2: 14128
Player 2 wins!
lizka@LizaAlisa:~/ЛАБЫ_ОС/Лаба2$ ./main
Enter the number of rolls per player (K): 23
Enter the current round number: 56
Enter the initial score for player 1: 7
Enter the initial score for player 2: 8
Enter the maximum number of threads (experiments) to run simultaneously: 79

Total score of player 1: 12657
Total score of player 2: 12792
Player 2 wins!
```

Strace:

execve("./main", ["./main"], 0x7ffec8bd4898 /* 28 vars */) = 0 brk(NULL) = 0x5637d4ae7000

mmap(NULL, 8192, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x7f0e84d95000

access("/etc/ld.so.preload", R_OK) = -1 ENOENT (No such file or directory)

openat(AT_FDCWD, "/etc/ld.so.cache", O_RDONLY|O_CLOEXEC) = 3

fstat(3, {st_mode=S_IFREG|0644, st_size=19163, ...}) = 0

mmap(NULL, 19163, PROT_READ, MAP_PRIVATE, 3, 0) = 0x7f0e84d90000

close(3) = 0

openat(AT_FDCWD, "/lib/x86_64-linux-gnu/libc.so.6", O_RDONLY|O_CLOEXEC) = 3

read(3, "\177ELF\2\1\1\3\0\0\0\0\0\0\0\0\3\0>\0\1\0\0\0\220\243\2\0\0\0\0\0"..., 832) = 832

pread64(3, "\6\0\0\0\4\0\0\0@\0\0\0\0\0\0\0@\0\0\0\0\0\0\0@\0\0\0\0\0\0\0"..., 784, 64) = 784

fstat(3, {st_mode=S_IFREG|0755, st_size=2125328, ...}) = 0

pread64(3, "\6\0\0\0\4\0\0\0@\0\0\0\0\0\0\0@\0\0\0\0\0\0\0@\0\0\0\0\0\0\0"..., 784, 64) = 784

mmap(NULL, 2170256, PROT_READ, MAP_PRIVATE|MAP_DENYWRITE, 3, 0) = 0x7f0e84b7e000

mmap(0x7f0e84ba6000, 1605632, PROT_READ|PROT_EXEC, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x28000) = 0x7f0e84ba6000

mmap(0x7f0e84d2e000, 323584, PROT_READ, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x1b0000) = 0x7f0e84d2e000

mmap(0x7f0e84d7d000, 24576, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x1fe000) = 0x7f0e84d7d000

mmap(0x7f0e84d83000, 52624, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_ANONYMOUS, -1, 0) = 0x7f0e84d83000 close(3) = 0

mmap(NULL, 12288, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x7f0e84b7b000

arch_prctl(ARCH_SET_FS, 0x7f0e84b7b740) = 0

set_tid_address(0x7f0e84b7ba10) = 872

set_robust_list(0x7f0e84b7ba20, 24) = 0

rseq(0x7f0e84b7c060, 0x20, 0, 0x53053053) = 0

mprotect(0x7f0e84d7d000, 16384, PROT_READ) = 0

mprotect(0x5637a2f98000, 4096, PROT_READ) = 0

mprotect(0x7f0e84dcd000, 8192, PROT_READ) = 0

*prlimit64(0, RLIMIT_STACK, NULL, {rlim_cur=8192*1024,

rlim_max=RLIM64_INFINITY}) = 0

munmap(0x7f0e84d90000, 19163) = 0

fstat(1, {st_mode=S_IFCHR|0620,

st_rdev=makedev(0x88, 0), ...}) = 0

getrandom("\x69\xc8\xdd\x8b\xae\x0b\xef\x81", 8, GRND_NONBLOCK) = 8

brk(NULL) = 0x5637d4ae7000 brk(0x5637d4b08000) = 0x5637d4b08000

fstat(0, {st_mode=S_IFCHR|0620, st_rdev=makedev(0x88, 0), ...}) = 0

write(1, "Enter the number of rolls per

pl"..., 42 Enter the number of rolls per player

(K):) = 42 **read(0, "\n", 1024) = 1**

read(0, "\n", 1024) = 1

read(0, "\n", 1024) = 1

read(0, "\n", 1024) = 1

read(0, "", 1024) = 0

write(1, "Enter the current round number: "...,

209Enter the current round number:

Enter the initial score for player 1:

Enter the initial score for player 2:

Enter the maximum number of threads (experiments) to run simultaneously:

Total score of player 1: 0) = 209

write(1, "Total score of player 2: 0\n", 27Total

score of player 2: 0) = 27

write(1, "It's a draw!\n", 13It's a draw!) = 13

exit_group(0) = ? +++

exited with 0 +++

Вывод

Во время выполнения лабораторной работы я разработал программу, которая использует многопоточность для симуляции игры с подбрасыванием кубиков. Основная сложность возникла из-за работы с общими данными между потоками. Поскольку несколько потоков одновременно изменяли общие переменные, возникала угроза гонки данных, что могло привести к некорректным результатам. Я решил эту проблему с помощью мьютексов, которые обеспечили безопасный доступ к данным в критических секциях программы.

Кроме того, возникли вопросы, связанные с генерацией случайных чисел в многопоточном контексте. Я использовал стандартный генератор случайных чисел `rand()`, однако в будущем хотелось бы рассмотреть использование более устойчивых и потокобезопасных методов генерации случайных чисел, чтобы избежать неожиданных результатов.

В процессе работы я также столкнулся с необходимостью тщательно контролировать синхронизацию потоков и корректное завершение всех потоков, что потребовало дополнительного внимания к использованию функций `pthread_join`.

В целом, работа была полезной и помогла мне лучше понять основы многопоточности в C, синхронизацию потоков с использованием мьютексов, а также особенности работы с общей памятью в многозадачной среде.