

Лабораторные работы по курсу
Операционные системы

Лабораторная работа 4
Тупики и синхронизация потоков

Москва, 2023

Теоретическая часть

Многопоточность при разработке приложений является очень мощным инструментом для повышения эффективности программ. Однако, применение данного подхода влечет за собой множество «подводных камней», которые могут повлиять на работоспособность всей системы.

Первым подобным «подводным камнем» является проблема тупиков (deadlock). Представим ситуацию, когда несколько потоков конкурируют за один и тот же ресурс. Если данный ресурс недоступен, то потоки переводятся в режим ожидания. В случае, если ресурс не будет освобожден, ожидание становится бесконечным и программа «зависает». Подобная ситуация называется тупиком. Поток находится в состоянии тупика, если он ожидает события, которое никогда не произойдет. [1] [2]

Примером тупика служит транспортная пробка, представленная на Рисунок 1.

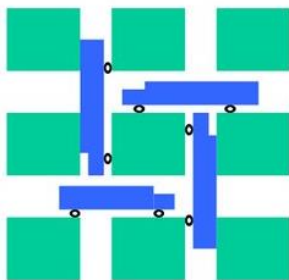


Рисунок 1 Транспортная пробка

В данной ситуации машины могут двигаться только вперед и разъехаться будет невозможно.

Вторым подводным камнем является проблема синхронизации потоков. В предыдущей лабораторной работе мы убедились, что без средств синхронизации невозможна корректная работа многопоточной программы, использующей глобальные переменные. Наиболее популярным средством синхронизации являются мьютексы. В ОС Linux в своей основе мьютексы содержат аппаратную поддержку, представленную в виде *spinlock* [1], основанного на алгоритме MCS [1]. В данном случае используется специальная ассемблерная инструкция, позволяющая производить атомарную операцию. В старых версиях Linux подобной инструкцией была *tsl* – test-and-set instruction. [3]

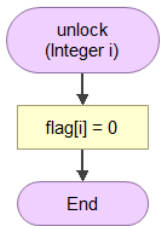
Однако существуют и программные алгоритмы синхронизации потоков. Рассмотрим наиболее популярные из них.

1. Алгоритм Деккера

lock	<pre> graph TD Start([lock (Integer i)]) --> J["j = 1-i"] J --> Flag1{"flag[i] = 1"} Flag1 -- True --> TurnJ1{"turn = j"} TurnJ1 -- True --> Flag0["flag[i] = 0"] Flag0 --> TurnJ2{"turn = j"} TurnJ2 -- True --> Flag1_2["flag[i] = 1"] Flag1_2 --> TurnJ2 TurnJ2 -- False --> Flag1 Flag1 -- False --> End([End]) </pre>	<p>Алгоритм реализуется при помощи двух флагов и переменной <i>turn</i>.</p> <p>При захвате мьютекса потоком, переменной <i>j</i> присваивается значение, равное 1 – номер текущего потока, переданного в качестве параметра – номера конкурирующего потока. Далее до тех пор, пока значение флага конкурирующего потока равно 1 мы проводим проверку условия очереди. Если значение очереди соответствует номеру конкурирующего потока, то мы выставляем флаг текущего потока в 0 и заходим в бесконечный цикл, до тех пор, пока значение очереди не изменится. Далее значение флага текущего потока возвращается в 1.</p>
unlock	<pre> graph TD Start([unlock (Integer i)]) --> Turn["turn = 1-i"] Turn --> Flag["flag[i] = 0"] Flag --> End([End]) </pre>	<p>Значению переменной <i>turn</i> присваивается номер конкурирующего потока. Значение флага готовности опускается в 0.</p>

2. Алгоритм Петерсона

lock	<pre> graph TD Start([lock (Integer i)]) --> J["j = 1-i"] J --> FlagI["flag[i] = 1"] FlagI --> TurnJ["turn = j"] TurnJ --> Cond{"flag[j] == 1 && turn == j"} Cond -- True --> TurnJ Cond -- False --> End([End]) </pre>	<p>Алгоритм реализуется при помощи двух флагов и переменной <i>turn</i>.</p> <p>При захвате мьютекса потоком, переменной <i>j</i> присваивается значение, равное 1 – номер текущего потока, переданного в качестве параметра – номера конкурирующего потока. Значение флага текущего потока выставляется в 1, а значению очереди присваивается номер конкурирующего потока.</p> <p>Далее до тех пор, пока значение флага конкурирующего потока равно 1 и значение очереди равно номеру конкурирующего потока мы находимся в ожидании в бесконечном цикле.</p>
------	---	---

unlock	 <pre> graph TD A([unlock (Integer i)]) --> B[flag[i] = 0] B --> C([End]) </pre>	Значению переменной флага готовности текущего потока опускается в 0.
--------	---	---

3. Алгоритм пекаря (булочной)

lock	 <pre> graph TD A([lock (Integer i)]) --> B[choosing[i] = 1] B --> C[ticket[i] = max(ticket[0], ticket[i]) + 1] C --> D[choosing[i] = 0] D --> E[j = 0 to Nthreads] E -- Done --> F([End]) E -- Next --> G{choosing[j]=1} G -- True --> G G -- False --> H{ticket[j] && ((ticket[j] < ticket[i]) ((ticket[j] = ticket[i]) && j < i))} H -- True --> G H -- False --> E </pre>	<p>При входе в критическую секцию каждому потоку назначаются номера. Номер пришедшего потока будет на 1 больше, чем номер предыдущего.</p> <p>Общий счётчик показывает номер исполняемого в данный момент потока. Все остальные потоки ждут, пока не закончится обслуживание текущего клиента и счетчик покажет следующий номер.</p> <p>Первым шагом флаг <i>choosing</i> текущего потока выставляется в 1. После происходит получение билета – на один больше уже получивших и опускание флага <i>choosing</i>. Далее в цикле для всех потоков проходит проверка того, что ни один поток не находится в состоянии получения билета – значение <i>choosing</i> у всех потоков равно 0. Следующим шагом текущий поток ожидает, пока конкурирующие потоки с меньшим номером или с таким же номером, но с более высоким приоритетом (номером потока), закончат свою работу.</p>
unlock	 <pre> graph TD A([unlock (Integer i)]) --> B[ticket[i] = 0] B --> C([End]) </pre>	Происходит возврат билета – значение номера текущего потока приравнивается 0

Практическая часть

1. Анализ многопоточных программ.

1.1. Создайте файл thread.c и скопируйте в него данную программу. Для компиляции воспользуйтесь следующим ключом:

```
gcc thread.c -o thread -lpthread
```

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

void* func(void *args) {
    printf("Hello from thread!\n");
}

int main() {
    pthread_t thread;
    printf("Hello from main!\n");
    pthread_create(&thread, NULL, func, NULL);
    pthread_join(thread, NULL);
    return 0;
}
```

*Запустите полученную программу? Какой результат был получен?
Модернизируйте программу так, чтобы было создано 10 потоков с функцией func.*

1.2. Создайте файл data_race.c и скопируйте в него данную программу. Для компиляции воспользуйтесь следующим ключом:

```
gcc data_race.c -o data_race -lpthread
```

```
#include <pthread.h>
#include <stdio.h>
int Global = 0;
void *Thread1() {
    Global++;
}
void *Thread2() {
    Global--;
}
int main() {
    pthread_t t1,t2;
    pthread_create(&t1, NULL, Thread1, NULL);
    pthread_create(&t2, NULL, Thread2, NULL);
    pthread_join(t1, NULL);
    pthread_join(t2, NULL);
    printf("%d\n",Global);
    return Global;
}
```

Запустите полученную программу. Какой результат был получен?

1.3. Проверим программу на наличие гонок данных. Для этого скомпилируем её с ключом -fsanitize=thread.

```
gcc -fsanitize=thread data_race.c -o data_race -lpthread
```

Запустите программу. Поясните полученный результат.

- 1.4. С помощью приведенного ниже скрипта происходит запуск программы до тех пор, пока она не выведет значение, отличное от 1 или число итераций не превысит 1000. Для этого создайте файл script.sh, сделайте его исполняемым и запустите его, в качестве параметра указав название исполняемого файла:

```
touch script.sh
nano script.sh
#скопируйте текст скрипта, сохраните его
chmod +x script.sh
./ script.sh data_race
```

Листинг скрипта:

```
#!/bin/bash
var=2
N=0
while [[ $var -eq 2 ]] && [[ $N -ne 2000 ]]
do
var=$(./$1)
N=$((N+1))
done
echo $N
```

Какой результат вернул скрипт? Запустите его несколько раз. Прокомментируйте результат работы. Проверьте программу на гонки данных с помощью ThreadSanitizer. Модернизируйте программу таким образом, чтобы создавалось 100 потоков Thread1 и 100 потоков Thread2. Запустите программу и проанализируйте результат.

Добавьте в программу мьютексы. Проверьте программу на гонки данных с помощью ThreadSanitizer.

2. Тупики и методы их решения

- 2.1. Скомпилируйте и запустите следующий код.

```
gcc deadlock.c -o deadlock -lpthread
```

```

#include <stdio.h>
#include <pthread.h>
#include <unistd.h>
void *function1();
void *function2();
pthread_mutex_t first_mutex; // mutex lock
pthread_mutex_t second_mutex;

int main()
{
    pthread_mutex_init(&first_mutex, NULL);
    pthread_mutex_init(&second_mutex, NULL);
    pthread_t one, two;
    pthread_create(&one, NULL, function1, NULL);
    //sleep(1);
    pthread_create(&two, NULL, function2, NULL);
    pthread_join(one, NULL);
    pthread_join(two, NULL);
    printf("Thread joined\n");
}

void *function1()
{
    pthread_mutex_lock(&first_mutex);
    printf("Thread ONE acquired first_mutex\n");
    //sleep(3);
    pthread_mutex_lock(&second_mutex);
    printf("Thread ONE acquired second_mutex\n");
    pthread_mutex_unlock(&second_mutex);
    printf("Thread ONE released second_mutex\n");
    pthread_mutex_unlock(&first_mutex);
    printf("Thread ONE released first_mutex\n");
}

void *function2()
{
    pthread_mutex_lock(&second_mutex);
    printf("Thread TWO acquired second_mutex\n");
    pthread_mutex_lock(&first_mutex);
    printf("Thread TWO acquired first_mutex\n");
    pthread_mutex_unlock(&first_mutex);
    printf("Thread TWO released first_mutex\n");
    pthread_mutex_unlock(&second_mutex);
    printf("Thread TWO released second_mutex\n");
}

```

Какой результат выполнения кода?

Скомпилируйте программу с ключом *Thread Sanitizer*. Запустите программу, используя анализатор поиска тупиков.

```

gcc deadlock.c -o deadlock -lpthread -fsanitize=thread
TSAN_OPTIONS=detect_deadlocks=1:second_deadlock_stack=1 ./deadlock

```

Какой результат выполнения кода?

Раскомментируйте строки в коде, содержащие вызов `sleep` и заново скомпилируйте и запустите программу.

Объясните полученный результат. Измените программу таким образом, чтобы избежать тупика. (Thread Sanitizer не должен обнаруживать ошибки)

2.2. Скомпилируйте и запустите следующий код.

```

#include <stdio.h>
#include <pthread.h>
#include <unistd.h>
static int global_var = 0;
void *function1();
pthread_mutex_t mutex;

int main()
{
    pthread_mutex_init(&mutex, NULL); // initialize the lock
    pthread_t one, two;
    pthread_create(&one, NULL, function1, NULL); // create thread
    pthread_join(one, NULL);
    printf("Global var = %d\n", global_var);
    return 0;
}

void summ_f()
{
    pthread_mutex_lock(&mutex);
    if(global_var!=5)
    {
        global_var++;
        summ_f();
    }
    pthread_mutex_unlock(&mutex);
}

void *function1()
{
    summ_f();
}

```

Объясните полученный результат. Измените программу таким образом, чтобы избежать тупика.

3. Алгоритмы синхронизации потоков.

В приложении к лабораторной работе содержатся листинги программ *mutex.c*, *Deccer.c*, *Bakery.c*, *Peterson.c*, *Deccer.h*, *Bakery.h*, *Peterson.h*.

В программе происходит вызов функции *Thread* из двух потоков. В случае запуска программы без механизмов синхронизации произойдет гонка данных и результат будет непредсказуем. Ваша задача – реализовать методы *lock* и *unlock* самостоятельно на основе одного из приведенных алгоритмов – Петерсона, Деккера и Булочника.

Выбор варианта: (Номер в списке % 3) + 1.

Вариант	Алгоритм
1	Петерсона
2	Деккера
3	Булочника

Для проверки результата запустите программу через скрипт, приведенный выше.

Запустите программу с мьютексом и без него. Объясните полученный результат. Скомпилируйте и запустите программу с утилитой *Thread Sanitizer*. Какой результат был получен? Объясните почему.

Контрольные вопросы

1. Что такое тупики в ОС?
2. Приведите примеры тупиков?
3. Чем аппаратные алгоритмы синхронизации отличаются от программных?
4. Опишите принцип работы алгоритма Петерсона
5. Опишите принцип работы алгоритма Деккера
6. Опишите принцип работы алгоритма булочника

Список рекомендованной литературы

1. Карпов В.Е., Коньков К.А. Основы операционных систем. Москва: Физматкнига, 2019. 326 pp.
2. Таненбаум Э., Бос Х. Современные операционные системы. Санкт-Петербург: Питер, 2021. 1119 pp.
3. mutex.c [Электронный ресурс] // Linux kernel code: [сайт]. URL: <https://elixir.bootlin.com/linux/latest/source/kernel/locking/mutex.c> (дата обращения: 11.10.2022).
4. MCS locks and qspinlocks [Электронный ресурс] // LWN: [сайт]. URL: <https://lwn.net/Articles/590243/> (дата обращения: 11.10.2022).
5. Test and set instruction [Электронный ресурс] // ibm: [сайт]. URL: <https://www.ibm.com/docs/en/ztpf/1.1.0.15?topic=configuration-test-set-instruction> (дата обращения: 11.10.2022).