ФИМиКН, каф. ПМИ

2 ноября 2022

Распределенные вычисления

Тема 2. Программирование систем с общей памятью на примере Pthreads

Федор Сергеевич Пеплин
fpeplin@hse.ru
https://www.hse.ru/org/persons/749540639

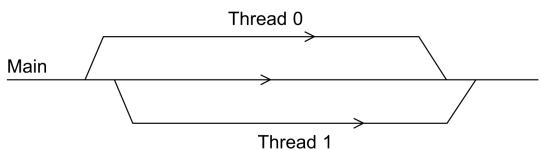
Создание и завершение потоков

Создание потока

```
int pthread_create(
   pthread_t* thread_p, //out
   pthread_attr_t* attr_p, //in
   void* (*routine)(void*), //in
   void* args_p) //in
```

Завершение потока

```
int pthread_join(
   pthread_t Thread //in
   void** return_value_p) //out
int pthread_detach(
    pthread_t Thread)
```



Hello, world

```
1  int main(int argc, char* argv[]) {
2      long       thread;
3      pthread_t* thread_handles;
4      thread_count = strtol(argv[1], NULL, 10);
5      thread_handles = malloc (thread_count*sizeof(pthread_t));
6      for (thread = 0; thread < thread_count; thread++)
7          pthread_create(&thread_handles[thread], NULL, Hello, (void*) thread);
8      printf("Hello from the main thread\n");
9      for (thread = 0; thread < thread_count; thread++)
10         pthread_join(thread_handles[thread], NULL);
11     free(thread_handles);
12     return 0;
13 }
14 void *Hello(void* rank) {
15     long my_rank = (long) rank;   printf("Hello from thread %ld of %d\n",
16         my_rank, thread_count);
17     return NULL;
18 }
```

Критическая секция

Определение

Участки кода, которые не должны выполняться одновременно более чем одним процессом. Выполнение критической секции несколькими процессами может привести к **состоянию гонки** (**race condition**).

Состояние гонки возникает при попытке разными процессами одновременно обновить общий ресурс (например, изменить глобальную переменную).

Способы изоляции критической секции

1. Холостой цикл (busy-wait)
2. Mutex

Необходимо уменьшать размер критической секции насколько возможно, так как код в ней выполняется как однопоточный.

Холостой цикл

Busy-wait

```
flag = 0;
void* Routine(void* rank)
{
    while(flag != my_rank);
    /*commands in the critical section*/
    flag = (flag + 1) % thread_count;

    return NULL;
}
```

Минусы холостого цикла

1. Может быть изменен компилятором с включенной опцией оптимизации.
2. Порядок, в котором процессы входят в критическую секцию, фиксирован. ОС об этом ничего не знает $\Rightarrow$ потеря производительности.

Mutex
Mutual Exclusion

```
pthread_mutex_t mutex;
int main()
{
    pthread_mutex_init(&mutex, NULL);
    /*...*/
    pthread_mutex_destroy(&mutex);
}
void* Routine(void* rank)
{
    pthread_mutex_lock(&mutex);
    /*commands in the critical section*/
    pthread_mutex_unlock(&mutex);

    return NULL;
}
```

Сравнение холостого цикла и мьютекса

Характерные значения времени выполнения программы

Количество ядер — 8

Количество потоков	Холостой цикл	Мьютекс
1	3	3
2	1.5	1.5
4	0.75	0.75
8	0.35	0.35
16	0.5	0.35
32	1	0.35
64	4	0.35

1. При количестве процессов $\leq$ количеству процессоров производительность программ с мьютексом и холостым циклом обычно сопоставима.
2. При количестве процессов $>$ количества процессоров время работы программы с холостым циклом резко растет: порядок выполнения процессами критической секции жестко фиксирован и планировщик ОС не знает о нем.

Семафоры

Семафор — это счетчик (unsigned int), допускающий выполнение двух операций — увеличение на единицу (sem_post) и уменьшение на единицу (sem_wait). Если при выполнении sem_wait семафор имеет нулевое значение, то операция является блокирующей.

```
int sem_init(
    sem_t *semaphore_p,    //out
    int shared             //in
    unsigned initial_val) //in
```

```
int sem_destroy(sem_t *s_p);
int sem_post(sem_t *s_p);
int sem_wait(sem_t *s_p);
```

В MacOS X вместо sem_init использовать sem_open, вместо sem_destroy — sem_close и sem_unlink.

Сферы применения семафоров:

1. Защита критической секции
2. Реализация модели «производитель — потребитель»
3. Контроль над ограниченными ресурсами

Барьерная синхронизация

Общие сведения

Решаемые задачи

- Разбиение программы на этапы, разделенные барьерами
- Отладка
- Замер времени выполнения

В стандарте pthreads нет готового механизма для обеспечения барьерной синхронизации

Возможные реализации

- Холостой цикл и мьютекс
- Семафоры
- Условные переменные

Барьерная синхронизация

Реализация с помощью холостого цикла и мьютекса

```
int counter = 0;
pthread_mutex_t mutex;
void *Routine(){
    pthread_mutex_lock(&mutex);
    counter++;
    pthread_mutex_unlock(&mutex);
    while(counter < thread_count);
}
```

Недостатки реализации

1. Расходование ресурсов CPU
2. Сниженная производительность при количестве процессов большем, чем ядер
3. Нужен отдельный счетчик для каждого барьера

Барьерная синхронизация

Реализация с помощью семафоров и мьютексов

```
int counter = 0;
sem_t count_sem, barrier_sem; //count_sem=1, barrier_sem=0
void* Routine(){
    sem_wait(&count_sem);
    if(counter == thread_count - 1){
        counter = 0;
        sem_post(&count_sem);
        for(int i = 0; i < thread_count - 1; i++)
            sem_post(&barrier_sem);
    }
    else{
        counter++;
        sem_post(&count_sem);
        sem_wait(&barrier_sem);
    }
```

Барьерная синхронизация

Реализация с помощью семафора

Недостаток реализации

Счетчик counter обнуляется, но sem_post может не вернуться в исходное состояние, если заблокированный поток был завершен планировщиком. Возникнет состояние гонки.

Барьерная синхронизация

Реализация с помощью условных переменных

Псевдокод

```
lock mutex;
if condition occured
    tell Thread/Threads;
else
    unlock mutex and block;
unlock mutex;
```

Разблокировать один поток

```
int pthread_cond_signal(
    pthread_cond_t *cond_p);
```

Разблокировать все потоки

```
int pthread_cond_broadcast(
    pthread_cond_t *cond_p);
```

Создать условную переменную

```
int pthread_cond_init(pthread_cond_t*, pthread_cond_attr*)
```

Удалить условную переменную

```
int pthread_cond_destroy(pthread_cond_t *cond_p)
```

Освободить mutex_p, блокировать поток, пока не сработает cond_signal или cond_broadcast, затем заблокировать mutex_p

```
int pthread_cond_wait(pthread_cond_t* cond_var_p, pthread_mutex_t* mutex_p)
```

Барьерная синхронизация

Условные переменные

```
1  int counter = 0;
2  pthread_mutex_t mutex;
3  pthread_cond_t cond_var;
4  void* Routine(){
5      pthread_mutex_lock(&mutex);
6      counter++;
7      if(counter == thread_count){
8          counter = 0;
9          pthread_cond_broadcast(&cond_var);
10     }
11     else
12         while(pthread_cond_wait(&cond_var,&mutex) != 0)
13     pthread_mutex_unlock(&mutex);
14 }
```

Блокировка чтения-записи

Задача о читателях-писателях

Читать могут сколько угодно, писать — никто

```
int pthread_rwlock_rdlock(pthread_rwlock_t* rwlock_p);
```

Никто не может писать и читать, кроме одного потока

```
int pthread_rwlock_rdlock(pthread_rwlock_t* rwlock_p);
```

Снимаем запрет

```
int pthread_rwlock_unlock(pthread_rwlock_t* rwlock_p);
```

Объявляем блокировку

```
pthread_rwlock_t rwlock = PTHREAD_RWLOCK_INITIALIZER;
```

Удаляем блокировку

```
int pthread_rwlock_destroy(pthread_rwlock_t* rwlock_p);
```

Умножение матрицы на вектор

Оценка производительности

Потоки	8000000x8	8000x8000	8x8000000
1	$t = 0.4, E = 1.0$	$t = 0.35, E = 1.0$	$t = 0.45, E = 1.0$
2	$t = 0.22, E = 0.9$	$t = 0.18, E = 0.97$	$t = 0.3, E = 0.75$
4	$t = 0.14, E = 0.71$	$t = 0.1, E = 0.87$	$t = 0.4, E = 0.28$

Вопросы

1. Почему время работы программы с «широкой и низкой» матрицей выше без распараллеливания? Количество кэш-промахов.
2. Почему эффективность программы с «широкой и низкой» матрицей падает с ростом количества процессов? False sharing.

Потокобезопасность
Thread-safety

Код потокобезопасен, если его можно использовать из нескольких потоков одновременно.

Причины потоконебезопасности

1. Доступ к глобальным переменным или динамической памяти
2. Неявный доступ через указатели
3. Побочные эффекты функций

Способы обеспечения потоковой безопасности

Реентерабелность, локализация хранения данных