Разработка многопотоковых программ

Судаков А.А.

“Параллельные и распределенные вычисления” Лекция 20

План

- Многопотоковые библиотеки
- Стандарт POSIX
- Создание потоков
- Синхронизация
- Стандарт OpenMP

Литература

- Учебное пособие по многопоточному программированию http://www.yolinux.com/TUTORIALS/LinuxTutorialPosixThreads.html
- Учебное пособие по OpenMP

http://www.llnl.gov/computing/tutorials/openMP/

Многопотоковость

- Поток – последовательность команд, которые выполняются параллельно с другими потоками в одном адресном пространстве
- Все (почти все) ресурсы потоков – общие
- Преимущества
  - Простота взаимодействия между потоками
  - Возможность использования нескольких процессоров одним процессом
  - Большая производительность
- Недостатки
  - Сложность синхронизации
  - Большая вероятность появления ошибок

Поддержка многопотоковости

- Существует несколько стандартов
- SUN threads – первая библиотека многопоточной работы
- Windows thread - M$ библиотека
- pthreads - стандарт POSIX на создание многопоточных программ
- Большое количество пользовательских библиотек по созданию многопотоковых программ
  - Java green threads

Стандарт POSIX

- Функции
  - Создания потоков
  - Завершения потоков
  - Синхронизации между потоками
  - Данные потоков

Создание потоков

```
int  pthread_create(
pthread_t  *  thread,  //идентификатор потока
pthread_attr_t * attr,     // атрибуты потока
void * (*start_routine)(void *),    // функция потока
void * arg              //агрумент функции потока
);
```

- Идентификатор потока – обязательный структура
- Функция потока – функция, которая будет выполняться параллельно с другими
- Атрибуты – специальные свойства
- Аргумент – аргумент, который передается потоку

Завершение потоков

- Выход из функции потока
- Принудительное завершение из другого потока
  - Не рекомендуется из-за сложности обработки асинхронных сообщений

Пример

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

void* thread_function(void* arg){
        int num = (int) arg;
        int i;
        for (i=0; i<100000/num; i++);
        printf("I am thread number %d\n",num);
        return 0;
}

int main(){
pthread_t threads[10];
int i;
for (i=0; i<10; i++)
        pthread_create(threads+i,NULL,thread_function,(void*)i+1);
return 0;
}
```

Пример выполнения

```
[saa@cluster threads]$ gcc -pthread create.c
[saa@cluster threads]$ ./a.out
I am thread number 2
I am thread number 3
I am thread number 1
I am thread number 4
I am thread number 5
I am thread number 7
I am thread number 6
I am thread number 8
I am thread number 9
```

Функции потоков

- Функции должны правильно работать с общими ресурсами
- Должны корректно выполняться параллельно одна другой
  - Быть реентерабельными
- Реентерабельные
  - Нет работы с общими данными
  - Работа с общими данными корректно синхронизирована

Пример нереентерабельной функции

```
char* mem ; // общая переменная

void* thread_function(void* arg){
        int num = (int) arg;
        int i;

        mem = malloc(1000);
        ....
        free(mem);
        return 0;
}
```

Можно удалить память дважды или присвоить используемому указателю новое значение

Реентерабельные версии библиотечных функций

- Функция форматирования даты в виде текстовой строки
  - `char *ctime(const time_t *timep);`
  - Использует общий статически выделенный буфер
  - Не может выполняться параллельно две функции
- Реентерабельная функция
  - `char *ctime_r(const time_t *timep, char *buf);`
  - Принимает аргумент - уникальный буфер пользователя
  - Несколько функций может выполняться параллельно

Пример использования

```
#include <pthread.h>
#include <stdio.h>
#include <time.h>
#include <sys/time.h>

void* thread_function(void* arg){
        time_t t = time(0);
        char buf[30];
        printf("time is %s",ctime_r(&t,buf));
        return 0;
}

int main(){
pthread_t threads[10];
int i;
for (i=0; i<10; i++)
        pthread_create(threads+i,NULL,thread_function,(void*)i+1);
return 0;
}
```

Синхронизация

- Защита данных
  - Обращение к общим переменным
  - Гарантия, что при асинхронном завершении общие данные будут в непротиворечивом состоянии
- Синхронизация действий
  - Привязка запуска/завершения одного потока к запуску/завершению другого потока

Защита данных

- Мьютексы
  - Взаимоисключающие блокировки
- Типы
  - Быстрый – обычный тип блокировки
  - Рекурсивный – поддерживается счетчик захватов
  - С проверкой ошибок
- Создание
  - Статически
  - В динамически созданной структуре
- Операции
  - Блокировка
  - Освобождение
  - Проверка

Создание мьютексов

- Статическое создание
  - `pthread_mutex_t fastmutex = PTHREAD_MUTEX_INITIALIZER;`
  - `pthread_mutex_t recmutex = PTHREAD_RECURSIVE_MUTEX_INITIALIZER_NP;`
  - `pthread_mutex_t errchkmutex = PTHREAD_ERRORCHECK_MUTEX_INITIALIZER_NP;`
- Динамическое создание в любой (даже динамически выделенной) памяти
  - `int pthread_mutex_init(pthread_mutex_t *mutex, const pthread_mutex_attr_t *mutexattr);`
- Удаление (для динамически создаваемых)
  - int pthread_mutex_destroy(pthread_mutex_t *mutex);

Блокировка - освобождение

- Блокировка
  - `int pthread_mutex_lock(pthread_mutex_t *mutex);`
- Освобождение
  - `int pthread_mutex_unlock(pthread_mutex_t *mutex);`
- Проверка
  - `int pthread_mutex_trylock(pthread_mutex_t *mutex);`
  - Аналогично захвату, но не захватывает уже захваченную блокировку

Пример программы без блокировки

```
#include <pthread.h>
#include <stdio.h>
#include <time.h>
#include <sys/time.h>

long counter = 0; // счетчик

void* thread_function(void* arg){
        int num = (int) arg;
        int i;
        for (i=0; i<1000000;i++)counter++;
        printf("thread # %d, counter=%ld\n", num, counter);
        return 0;
}

int main(){
pthread_t threads[10];
int i;
for (i=0; i<10; i++)
        pthread_create(threads+i,NULL,thread_function,(void*)i+1);
         sleep(10);
return 0;
}
```

Выполнение программы без блокировок

- [saa@cluster threads]$ gcc -pthread mutex.c
- [saa@cluster threads]$ ./a.out
- thread # 1, counter=1320529
- thread # 2, counter=2004893
- thread # 3, counter=2062666
- thread # 5, counter=3396949
- thread # 6, counter=3400423
- thread # 4, counter=4741143
- thread # 7, counter=4751892
- thread # 9, counter=6096112
- thread # 8, counter=6102053

Пример той же программы с блокировками

```
#include <pthread.h>
#include <stdio.h>
#include <time.h>
#include <sys/time.h>

long counter = 0;

pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
void* thread_function(void* arg){
        int num = (int) arg;
        int i;
        for (i=0; i<100000;i++) {
                pthread_mutex_lock(&mutex);
                counter++;
                pthread_mutex_unlock(&mutex);
        }
        printf("thread # %d, counter=%ld\n", num, counter);
        fflush(stdout);
        return 0;
}

int main(){
pthread_t threads[10];
int i;
for (i=0; i<10; i++)
        pthread_create(threads+i,NULL,thread_function,(void*)i+1);
sleep(10);
return 0;
}
```

Семафоры

- Семафор – целочисленный атомарный счетчик с блокировкой
- Поддерживаются семафоры POSIX
- Отличия от семафоров UNIX
  - Другие прототипы функций
  - Нет третьего состояния (требование нуля)
- Создание
  - `int sem_init(sem_t *sem, int pshared, unsigned int value);`
- Удаление
  - `int sem_destroy(sem_t * sem);`
- Уменьшение
  - `int sem_wait(sem_t * sem);`
- Увеличение
  - `int sem_post(sem_t * sem);`

Условные переменные

- Ожидание наступления некоторого условия
- Поток проверки
  - Проверка условия
  - Захват блокировки
  - Установка на ожидание
  - Повторить
- Поток, который установил условие
  - Сигнализирует ожидающим потокам

Инициализация и удаление

- Статическая
  - `pthread_cond_t cond = PTHREAD_COND_INITIALIZER;`
- Динамическая
  - `int pthread_cond_init(pthread_cond_t *cond, pthread_condattr_t *cond_attr);`
- Удаление (только для динамических)
  - `int pthread_cond_destroy(pthread_cond_t *cond);`

Условия

- Условие – некоторая переменная стала иметь некоторое значение
  - Установлен флаг
  - Счетчик стал достаточно большим
- Данные, которые соответствуют условию должны защищаться с помощью мьютекса

Проверка условия

- Захватить мьютекс связанный с условием
- Проверить условие, если не выполнено
- Вызвать функцию проверки
  - int pthread_cond_wait(pthread_cond_t *cond, pthread_mutex_t *mutex);
  - Функция переводит поток в состояние ожидания
  - Функция автоматически освобождает указанную блокировку
- Если выполнено, освободить блокировку

Сигнал о выполнении условия

- Вызвать функцию для указанной условной переменной
  - `int pthread_cond_broadcast(pthread_cond_t *cond);`
- Функция переводит в состояние выполнения все потоки, которые ожидают выполнения условия

Пример

```
#include <pthread.h>
#include <stdio.h>
#include <time.h>
#include <sys/time.h>
long counter = 0;
int thr_count = 0;
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t cond = PTHREAD_COND_INITIALIZER;
void* thread_function(void* arg){
        int num = (int) arg; int i;
        for (i=0; i<100000;i++) {
                pthread_mutex_lock(&mutex);
                counter++;
                pthread_mutex_unlock(&mutex);
        }
        printf("thread # %d, counter=%ld\n", num, counter); fflush(stdout);
        pthread_mutex_lock(&mutex);
        thr_count++;
        pthread_cond_broadcast(&cond);
        pthread_mutex_unlock(&mutex);
        return 0;
}

int main(){
pthread_t threads[10];
int i;
for (i=0; i<10; i++)
        pthread_create(threads+i,NULL,thread_function,(void*)i+1);
pthread_mutex_lock(&mutex);
while(thr_count < 10) pthread_cond_wait(&cond,&mutex);
pthread_mutex_unlock(&mutex);
return 0;
```

Синхронизация действий

- Ожидание окончания потока
  - `int pthread_join(pthread_t th, void **thread_return);`
- Вызывающий поток ждет завершения потока th
- Поток th не должен иметь атрибут `PTHREAD_CREATE_DETACHED`

Пример join

```
#include <pthread.h>
#include <stdio.h>
#include <time.h>
#include <sys/time.h>

long counter = 0;

pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
void* thread_function(void* arg){
        int num = (int) arg;
        int i;
        for (i=0; i<100000;i++) {
                pthread_mutex_lock(&mutex);
                counter++;
                pthread_mutex_unlock(&mutex);
        }
        printf("thread # %d, counter=%ld\n", num, counter);
        fflush(stdout);
        return 0;
}

int main(){
pthread_t threads[10];
int i;
for (i=0; i<10; i++)
        pthread_create(threads+i,NULL,thread_function,(void*)i+1);
for(i=0; i<10; i++)
        pthread_join(threads[i],NULL);
return 0;
}
```

Пример выполнения

```
[saa@cluster threads]$ gcc -pthread join.c -g
[saa@cluster threads]$ ./a.out
thread # 2, counter=974887
thread # 7, counter=977525
thread # 5, counter=980255
thread # 10, counter=985973
thread # 4, counter=998066
thread # 8, counter=998216
thread # 1, counter=998277
thread # 9, counter=999596
thread # 3, counter=999784
thread # 6, counter=1000000
```

Данные связанные с потоками

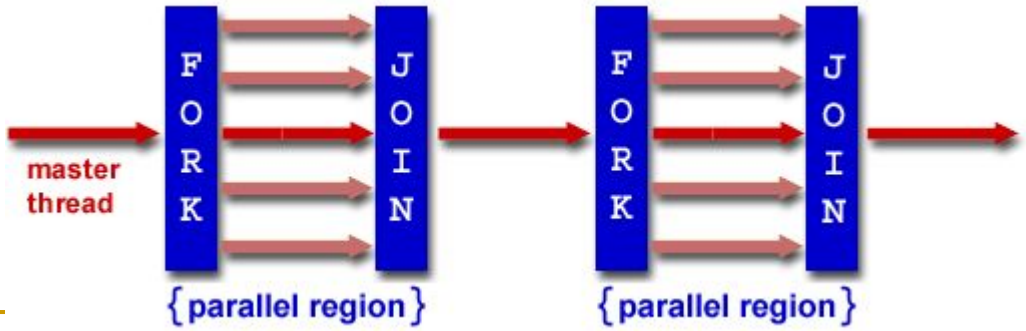
- Все глобальные переменные общие для всех потоков
- Можно создать ключ – переменную, к которой имеют доступ все потоки, но значение переменной для каждого потока - свое
- Пример – переменная `errno`

Стандарт OpenMP

- Разработка параллельных программ с использованием многопотоковости требует стандартных действий и стандартных правил
- Некоторые структуры (циклы) очень легко распараллеливаются
- Синхронизация доступа к данным выполняется стандартным образом
- Ручное использование многопоточности
  - Больше ручной работы
  - Большая вероятность ошибок

Как распараллеливается

- Программа разбивается на параллельные участки, которые выполняются последовательно
  - Fork-Join модель
- Каждый параллельный участок выполняется с помощью некоторого количества потоков
  - По умолчанию – равно количеству процессоров
  - Можно установить с помощью системной переменной OMP_NUM_THREAD



Изменение программного кода

- Изменение кода выполняется путем указания компилятору какие участки и как распараллеливать
- Указания вводятся с помощью директив препроцессора или специальных комментариев, чтобы не «портить» код

Как вводятся директивы

- Фортран
  - **!$OMP PARALLEL *[clause ...]***
    IF *(scalar_logical_expression)*
    PRIVATE *(list)*
    SHARED *(list)*
    DEFAULT (PRIVATE | SHARED | NONE)
    FIRSTPRIVATE *(list)*
    REDUCTION *(operator: list)*
    COPYIN *(list)*
    block
  - **!$OMP END PARALLEL**
- C/C++
  - **#pragma omp parallel *[clause ...] newline***
    if *(scalar_expression)*
    private *(list)*
    shared *(list)*
    default (shared | none)
    firstprivate *(list)*
    reduction *(operator: list)*
    copyin *(list)*
    structured_block

Типы директив

- Какие участки распараллеливать
  - #pragma omp parallel
- Какие участки выполнять в разных потоках
  - #pragma omp sections –начало набора участков
  - #pragma omp section – начало участка
- Какие участки выполнять одним потоком
  - single
- Как планировать выполнение
  - Shadule(тип, размер порции)
- Синхронизация
  - Критический раздел, выполнение только мастер-потоком, барьер
- Какие данные являются общими, а какие - нет

Распараллеливание циклов for

- #pragma omp parallel for

```
#include <iostream>
#include <cmath>
using namespace std;

int main (void){
#pragma omp parallel for
        for (int i =0; i<10; i++)
                cout << i<<endl<<flush;
return 0;
}
```

Пример выполнения

```
[saa@cluster omp]$ icc -openmp for.cpp
for.cpp(7) : (col. 1) remark: OpenMP DEFINED LOOP WAS PARALLELIZED.
[saa@cluster omp]$ ./a.out
05

1
2
3
4
6
7
8
9
[saa@cluster omp]$
```

Участи параллельного выполнения

- #pragma omp parallel sections
- #pragma omp section
  - Каждая секция будет выполняться в своем потоке

```
#include <iostream>
#include <cmath>
using namespace std;


int main (void){
#pragma omp parallel sections
{
#pragma omp section
        for (int i =0; i<5; i++) cout << i<<endl<<flush;
#pragma omp section
        for (int i =5; i<10; i++) cout << i<<endl<<flush;
}
return 0;
}
```

Пример выполнения

```
[saa@cluster omp]$ icc -openmp section.cpp
[saa@cluster omp]$ OMP_NUM_THREADS=4 ./a.out
05
1
2
3
4

6
7
8
9
```

Типы планирования

- Применяется совместно с for
- Shedule(тип, порция)
- Порция – количество итераций
- Типы
  - Static – работа статически разбивается на порции одинакового размера
  - Dynamic -работа разбивается на порции заданного размера. После выполнения одной порции поток динамически выполняет другую
  - GUIDED размер порции уменьшается экспоненциально по мере выполнения. Размер соответсвует минимальному размеру порции
  - RUNTIME Решение принимается при запуске программы с помощью установки системной переменной OMP_SCHEDULE

Синхронизация

- Указывается для блока команд
- Critical – указание критического раздела
- Master – выполняется только master потоком
- Barrier – указание барьера
- ORDERED -выполнение итераций цикла в той же последовательности, что и в последовательной программе

Пример critical

```
#include <iostream>
#include <cmath>
using namespace std;


int main (void){
#pragma omp parallel for
        for (int i =0; i<10; i++){
#pragma omp critical
                cout << i<<endl<<flush;
        }
return 0;
}
```

Пример выполнения критического раздела

- Без critical

```
[saa@cluster omp]$ OMP_NUM_THREADS=10 ./a.out
0756893241
```

- С указанием critical

```
[saa@cluster omp]$ icc -openmp ./single.cpp
./single.cpp(7) : (col. 1) remark: OpenMP DEFINED LOOP WAS PARALLELIZED.
[saa@cluster omp]$ OMP_NUM_THREADS=10 ./a.out
0
5
2
7
9
3
6
1
4
8
```

Видимость данный

- Используется совместно с for, section или после определения данных
- SHARED (данные) – данные совместного использования – все сложности работы ложатся на программиста
- PRIVATE (данные) – данные являются частными данными потока, после выполнения потока не сохраняются
- THREADPRIVATE (данные) – глобальные данные являются частными данными потока, но должны быть консистентны для всех потоков и сохранятся после выполнения

Пример частных и общих данных

```
#include <stdio.h>

int alpha[10], beta[10], i;
#pragma omp threadprivate(alpha)

main ()  {

/* First parallel region */
#pragma omp parallel private(i,beta)
  for (i=0; i < 10; i++)
     alpha[i] = beta[i] = i;

/* Second parallel region */
#pragma omp parallel
  printf("alpha[3]= %d and beta[3]= %d\n",alpha[3],beta[3]);

}
```

Пример выполнения

```
[saa@cluster omp]$ icc -openmp ./threadprivate.c
./threadprivate.c(9) : (col. 1) remark: OpenMP DEFINED REGION WAS PARALLELIZED.
./threadprivate.c(14) : (col. 1) remark: OpenMP DEFINED REGION WAS PARALLELIZED.
[saa@cluster omp]$ OMP_NUM_THREADS=2 ./a.out
alpha[3]= 3 and beta[3]= 0
alpha[3]= 3 and beta[3]= 0
```

Beta[] – данные потерялись

Alpha[] – данные не потерялись

Операции редукции

- Reduce(оператор:данные)
- Используется для указания параллельных блоков в котором выполняется операция редукции
- Опепраторы могут быть +,-,*,+=,-=,*=

Пример редукции

```
#include <iostream>
#include <cmath>
using namespace std;

int k=0,l=0;
int main (void){
#pragma omp parallel for  shared(l) reduction(+:k)
        for (int i =0; i<100000; i++){
                k++;
                l++;
        }
        cout << "k="<<k<<endl<<flush;
        cout << "l="<<l<<endl<<flush;
return 0;
}
```

Результат выполнения

```
[saa@cluster omp]$ icc -openmp ./reduce.cpp
./reduce.cpp(7) : (col. 1) remark: OpenMP
   DEFINED LOOP WAS PARALLELIZED.
[saa@cluster omp]$
   OMP_NUM_THREADS=10 ./a.out
k=100000
l=60000
L потеряно
```

Вопросы?