ОПИСАНИЕ ИНТЕРФЕЙСА ЗАПУСКА ВЫЧИСЛИТЕЛЬНЫХ ПОТОКОВ

Версия № 54 от 25.10.2016

Оглавление

Общие сведения

Интерфейс для запуска расчетных потоков разработан с целью обеспечения реализации выполнения высокоинтенсивных расчетных задач, прозрачную с точки зрения программиста, выполняющего код этих расчетных задач.
Интерфейс предоставляет следующие возможности:
- указание максимального количества вычислительных функций (потоков);
- указание размера стека, выделяемого отдельному потоку при его создании;
- добавление вычислительной функции в список потоков;
- запуск расчета в одном из возможных режимов (см далее) и ожидание завершения расчета.

Расчетные функции могут быть запущены в одном из следующих режимов:
- параллельно, с ожиданием окончания расчета одним из потоков и принудительным завершением всех остальных;
- параллельно, с ожиданием окончания расчета всеми потоками;
- последовательно, от первой к последней позиции в списке, в этом случае возможности распараллеливания не используются.

Расчетной функции предоставляются средства:
- передачи необходимых исходных данных на вход функции;
- установки признака окончания расчета;
- проверки признака принудительного завершения расчета (с завершением потока).

К расчетным функциям предъявляются следующие требования:
- не использовать выделение/освобождение памяти (за исключением стека);
- не использовать функции чтения/записи;
- при использовании общих участков памяти организация блокировок возлагается на разработчика расчетных функций.

Интерфейс реализован на языке C++, с использованием стандартных функций языка C++11. Для запуска потоков используются функции POSIX, при сборке под Windows – функции WinAPI. Для взаимодействия между потоками и родительским процессом используются атомарные булевские переменные.
В целях портирования на платформы, не поддерживающие POSIX, реализована возможность принудительного отключения распараллеливания на этапе компиляции, в этом случае использование атомарных переменных также отключается.
Тестирование проведено на платформе x86 под управлением операционной системы QNX 6.5, сборка осуществлялась компилятором GCC 4.8.1. Кроме того, осуществлялось тестирование под управлением операционной системы Windows 10, сборка осуществлялась компилятором Visual Studio Exoress 2015 в родной среде и компилятором GCC 4.9 в среде MSYS2, эмулирующей POSIX.

Использование интерфейса

Интерфейс реализован в виде класса mkCalcPool. Объявления конструкторов, методов и переменных класса, а также сервисных функций располагаются в заголовочном файле mkcalcpool.hpp. Описания методов класса и сервисных функций располагаются в файле mkcalcpool.cpp. Таким образом, для использования класса mkCalcPool достаточно подключить заголовочный файл mkcalcpool.hpp в конкретный модуль, а также включить файл mkcalcpool.cpp в состав проекта (makefile).
Для использования возможностей распараллеливания расчета в проекте должна быть определена директива _MK_PARALLEL_ (при помощи #define или флага командной строки

компилятора). Если директива не определена, распараллеливание отключается.

Инициализация и закрытие интерфейса

Для инициализации интерфейса необходимо создать объект класса mkCalcPool. Конструктор класса принимает следующие параметры:

```
mkCalcPool(size_t cnt, size_t stk)
```

Здесь

cnt – максимальное количество вычислительных функций в списке;

stk – размер стека, выделяемый каждому потоку при параллельном запуске вычислений.

Параметр stk может быть равен 0, в этом случае размер стека остается на усмотрение операционной системы (так, в QNX 6.5 это 132 Кбайт).

Внимание! В программе может быть создан только один экземпляр класса mkCalcPool. При создании второго и более экземпляров функции объекта возвращают ошибку EBUSY (16).

При использовании конструктора по умолчанию объект может быть инициализирован функцией

```
int Init(size_t cnt, size_t stk)
```

с аналогичными параметрами. Данная функция возвращает:

EOK (0) в случае успешной инициализации;

ENOMEM (12) в случае ошибки распределения памяти;

EINVAL (22), если максимальное количество вычислительных функций указано равным 0.

По завершении работы с интерфейсом он может быть уничтожен вызовом функции

```
Free()
```

либо вызовом деструктора (автоматическим, при создании объекта на стеке, или ручным, при помощи оператора delete, при создании объекта в динамической памяти).

Правила оформления расчетных функций

Расчетная функция должна быть объявлена следующим образом:

```
void *func(calc_func_data_t *data)
```

Функция должна принимать на вход указатель на следующую структуру:

```
struct calc_func_data_t
{
      void *parameter;              ///< Указатель на некий параметр функции
      calc_run_mode_t mode;         ///< Режим запуска функции
      calc_atomic_flag_t *finished; ///< Указатель на флаг завершения расчета
};
```

Режим запуска функции может принимать следующие значения:

CRM_INIT (0) – значение, присваимое при инициализации структуры.

CRM_WAIT_ONE (1) – расчет будет запущен в параллельных потоках, с ожиданием завершения одного из потоков закончившего расчет первым. По окончании расчета функция должна установить флаг окончания расчета.

CRM_WAIT_ALL (2) – расчет будет запущен в параллельных потоках, с ожиданием завершения всех потоков. Установка флага завершения расчета желательна, но не обязательна.

CRM_SEQENCE (3) – расчетные функции будут запущены последовательно, без распараллеливания.

Флаг окончания расчета устанавливается при помощи функции (макроса)

```
SET_FINISHED(calc_atomic_flag_t *ptr)
```
Здесь
ptr – указатель на флаг завершения расчета, переданный в функцию в структуре calc_func_data_t.
Кроме того, во всех режимах функция должна контролировать признак принудительного прекращения расчета, с немедленным завершением потока. Контроль данного признака производится при помощи функции (макроса)
```
TST_POINT()
```
в произвольном месте кода функции.
Кроме того, для контроля признака прекращения расчета в условии цикла for или while может быть использована функция
```
bool NOT_BREAK(void)
```
которая возвращает true, если цикл может быть продолжен, или false, если цик расчета должен быть прерван. В последнем случае, при параллельном расчете, функция вызывает завершение потока (pthread_exit).

Добавление расчетных функций в список

Расчетная функция добавляется в конец или в определенное место списка при помощи функции объекта mkCalcPool
```
int Inject(void *(*f)(calc_func_data_t*), void *p, int ind)
```
Здесь
f – указатель на расчетную функцию;
p – указатель на параметр, передаваемый функции (см calc_func_data_t::parameter);
ind – порядковый номер ячейки в списке, или -1, если необходимо добавить функцию в первую свободную ячейку.
Функция возвращает:
EOK (0) в случае успеха;
ENOENT (2), если список потоков не инициализирован;
ESRCH (3), если порядковый номер указан неверно (превышает максимальное количество функций);
EAGAIN (11), если свободное место в списке исчерпано.
Если в качестве указателя на расчетную функцию передан NULL, данная позиция в списке будет занята, но не будет никаким образом участвовать в расчете.

Запуск расчета

Расчет запускается при помощи функции объекта mkCalcPool
```
int Start(calc_run_mode_t mode)
```
Здесь
mode – режим запуска. Может принимать следующие значения:
CRM_WAIT_ONE (1) – расчет будет запущен в параллельных потоках, с ожиданием завершения одного из потоков, закончившего расчет первым. Остальным потокам будет взведен признак принудительного завершения расчета.
CRM_WAIT_ALL (2) – расчет будет запущен в параллельных потоках, с ожиданием завершения всех потоков.
CRM_SEQENCE (3) – расчетные функции будут запущены последовательно, без распараллеливания.
Внимание! В случае, если директива _MK_PARALLEL_ не определена на этапе компиляции, режим запуска принудительно меняется на CRM_SEQENCE, и обращений

к функциям POSIX-потоков не происходит.
Функция Start дожидается окончания вычислений и возвращает:
EOK (0) в случае успеха;
ENOENT (2), если список потоков не инициализирован;
признак ошибки запуска потока, возвращаемый функциями pthread_create или pthread_attr_setstacksize, либо _beginthread (если определена директива _MK_PARALLEL_).
При включенном распараллеливании расчетные функции запускаются по возможности одновременно. В случае ошибки создания потока, запуск расчетных функций не производится, остальные потоки уничтожаются.
В режиме CRM_WAIT_ONE функция периодически, с интервалами 100 мс, проверяет состояние флага окончания расчета, и, если флаг взведен каким-либо из потоков, взводит признак принудительного завершения всех остальных потоков.

Доступ к списку потоков

При необходимости, указатель на список потоков может быть получен при помощи функции объекта mkCalcPool
`calc_item_t *getItems(void)`
Функция возвращает:
указатель на список в случае успеха;
NULL, если список потоков не инициализирован.

Доступ к количеству потоков и последней занятой позиции

При необходимости, текущее количество потоков может быть получено при помощи функции объекта mkCalcPool
`size_t getCount(void)`
Последняя занятая позиция в списке может быть получена при помощи функции
`size_t getLast(void)`

Пример использования класса mkCalcPool

Рассмотрим в качестве примера использования класса mkCalcPool исходный текст программы тестирования данного класса (файл mkcalctest.cpp):

```
/**
  @file Тест пула потоков
*/

#include <stdio.h>
#include "mkcalcpool.hpp"

////////////////////////////////////////////////////////////////////////////
/// Расчетная функция потока
/// \param calc_func_data_t *data - указатель на структуру данных для расчета
/// \return нет
void *test_func(calc_func_data_t *data)
{
       if(data)
       {
              const int work_cycles = 3000;
              int id = (int)data->parameter;

              printf("test func %d: mode = %d\n",
               id, data->mode);
```

```
		for( int j = 0; j < work_cycles && NOT_BREAK(); j++ )
		{  // ... якобы вычисления
			for( unsigned long i = 0; i < (work_cycles * id) && NOT_BREAK(); i++
)
			{
				i = ( i + 1 ) - 1;
				TST_POINT();
			}
			TST_POINT();
		}
		SET_FINISHED(data->finished);

		printf("test func %d finished\n", id);
	}
	return(NULL);
}
////////////////////////////////////////////////////////////////////////////////

////////////////////////////////////////////////////////////////////////////////
/// Стандартная точка входа приложения
int main(int argc, char *argv[])
{
	int result = EOK;
	int threads = 1;
	int stack = 0;
	bool thr_key = false, stk_key = false;
	const char *UseMsg = "Usage: mkcalctest [-t <threads>] [-s <stacksize>]\n"
		"-t <threads> - number of threads (1)\n"
		"-s <stacksize> - thread stack size, in bytes (0, system stack size)\n";

	mkCalcPool *testpool = new mkCalcPool;
	if(!testpool)
	{
		printf("Error %d, exiting...\n", errno);
		return(errno);
	}

	for(int i = 1; i < argc; i++)
	{
		char *arg = argv[i];
		if(arg)
		{
			if(arg[0] == '-')
			{
				switch(arg[1])
				{
					case 't':
						if(arg[2])
						{
							arg += 2;
							threads = atoi(arg);
						}
						else
							thr_key = true;
					break;
					case 's':
						if(arg[2])
						{
							arg += 2;
							stack = atoi(arg);
						}
						else
							stk_key = true;
					break;
					case 'h':
```

```
                                printf("%s", UseMsg);
                                return(result);
                        break;
                }
        }
        else if(thr_key)
        {
                threads = atoi(arg);
                thr_key = false;
        }
        else if(stk_key)
        {
                stack = atoi(arg);
                stk_key = false;
        }
    }
}

result = testpool->Init(threads, stack);
if(result == EOK)
{
    printf("Thread list ptr = %p\n", testpool->getItems());
    printf("Thread list stack size = %d\n", testpool->getStackSize());

    printf("Run %d threads\n", threads);
    for(int i = 0; i < threads; i++)
    {
        result = testpool->Inject(test_func, (void *)(i + 1));
        if(result != EOK)
        {
            printf("Inject error %d\n", result);
            break;
        }
    }
    if(result == EOK)
    {
        printf("Threads count = %d\n", testpool->getCount());
        printf("Last thread position = %d\n", testpool->getLast());

        printf("Test mode %d (wait for one)\n", CRM_WAIT_ONE);
        result = testpool->Start(CRM_WAIT_ONE);
        if(result != EOK)
        {
            printf("Pool start error %d\n", result);
            return(result);
        }
        printf("Test mode %d finished\n", CRM_WAIT_ONE);
        printf("Test mode %d (wait for all)\n", CRM_WAIT_ALL);
        result = testpool->Start(CRM_WAIT_ALL);
        if(result != EOK)
        {
            printf("Pool start error %d\n", result);
            return(result);
        }
        printf("Test mode %d finished\n", CRM_WAIT_ALL);
        printf("Test mode %d  (run sequentaly)\n", CRM_SEQENCE);
        result = testpool->Start(CRM_SEQENCE);
        if(result != EOK)
        {
            printf("Pool start error %d\n", result);
            return(result);
        }
        printf("Test mode %d finished\n", CRM_SEQENCE);
    }
}
else
```

```
            printf("Initialization error %d\n", result);
      delete testpool;
      return(result);
}
//////////////////////////////////////////////////////////////////////////
```

Файл проекта Makefile для сборки в QNX 6.5 компилятором GCC 4.8.1:

```
CC=g++
CFLAGS  =  -std=gnu++11 -fpack-struct -Wno-pragmas -Wpadded -D_MK_PARALLEL_
OFILES= mkcalcpool.o mkcalctest.o

BUILD_NUMBER_FILE=build.txt
BUILD_NUMBER_FLAGS  = -D__BUILD_DATE=$$(date +'%Y%m%d')
BUILD_NUMBER_FLAGS += -D__BUILD_NUMBER=$$(cat $(BUILD_NUMBER_FILE))

all: ../bin/mkcalctest $(BUILD_NUMBER_FILE)

../bin/mkcalctest : $(OFILES) $(LDLIBS)
      @echo ------------------------------------------------------------------------
      $(CC) $(LDFLAGS) $(LFLAGS) -o $@ $(OFILES) $(LDLIBS) 2>mkcalctest.link.err
      @if ! test -s mkcalctest.link.err; then rm mkcalctest.link.err; fi
      @echo -- build number $$(cat $(BUILD_NUMBER_FILE)) from $$(date +'%Y%m%d') --
      @echo ------------------------------------------------------------------------

clean:
      rm -f $(OFILES) ../bin/mkcalctest *.err

.cpp.o:
      $(CC) $(BUILD_NUMBER_FLAGS) $(CCFLAGS) $(CFLAGS) -c $< 2>$<.err
      @if ! test -s $<.err; then rm $<.err; fi

mkcalcpool.o: mkcalcpool.cpp mkcalcpool.hpp
mkcalctest.o: mkcalctest.cpp

$(BUILD_NUMBER_FILE): $(OFILES)
      @if ! test -f $(BUILD_NUMBER_FILE); then echo 0 > $(BUILD_NUMBER_FILE); fi
      @echo $$(($$(cat $(BUILD_NUMBER_FILE)) + 1)) > $(BUILD_NUMBER_FILE)
```