Глава 17

Написание многопоточных приложений

©Перевод сделан Тюрюмовым А.Н.

Большую часть времени основанная на событиях архитектура GUI приложений создает хорошую иллюзию выполнения нескольких задач одновременно. Перерисовка окон обычно занимает малую часть секунды и пользовательский ввод может быть обработан почти мгновенно. Однако существуют ситуации когда задача не может быть выполнена в короткий промежуток времени в одном потоке. В этом случае на помощь приходит концепции многопоточного программирования. В данной главе рассказывается о том, как управлять потоками в wxWidgets, а в конце делается обзор некоторых альтернатив использованию потоков.

17.1 Когда стоит и не стоит использовать потоки

Потоком (thread) называется отдельный путь выполнения внутри программы. Потоки иногда называют легковесными процессами (process), однако фундаментальным отличием потоков и процессов является то, что память различных процессов отделена, а потоки работают в одном и том же адресном просранстве. Хотя последнее обстоятельство и упрощает использование совместных ресурсов потоками, но также и делает более вероятным обшибку, когда один поток меняет данные другого, поэтому при многопоточном программировании рекомендуется использовать объекты синхронизации, такие как мютексы (mutexes) и критические секции (critical sections).

При правильном использовании многопоточность позволяет программисту провести декомпозицию пользовательского интерфейса и «реальной работы». Заметим, что такое разделение не приведет к увеличению скорости работы программы (по крайней мере пока у вас не появится многопроцессорный компьютер), но сделает интерфейс более отзывчивым.

wxWidgets реализует как класс для потоков, так и необходимые объекты синхронизации (мютексы, критические секции и условия). API потоков в wxWidgets очень похож на API потоков в POSIX (pthreads), хотя несколько функций имеют другое имя, а также доступны некоторые возможности, имеющиеся в потоковом API в Win32.

Эти классы упрощают написание многопоточных приложений, а также осуществляют дополнительную проверку на ошибки по сравнению с родным API потоков. Однако использование пооков до сих пор является нетривиальной задачей, особенно для крупных проектов. Перед тем, как начать писать многопоточное приложение или до-

бавлять многопоточность в существующее рекомендуется рассмотреть возможность использования альтернатив многопоточности для реализации требуемой функциональности. Однако в некоторых ситуациях многопоточность является единственным выбором, например при написании FTP сервера, который должен запускать новый поток для каждого нового клиента. Однако использование потоков для показа прогреса при длительном вычислении является плохой идеей, так как в этом случае достаточно перенести вчисления в обработчик сообщений о простое (idle) и периодически вызывать `wxWindow::Update` для обновления экрана. За деталями обратитесь к разделу «Альтернативы многопоточности» в конце этой главы.

Если вы решили использовать потоки в вашем приложении, то лучше сразу добиваться того, чтобы только главный поток вызывал функции GUI. В примере использования потоков дистрибутива wxWidgets показывается, как множество потоков могут вызывать GUI функции одновременно, однако обычно это свидетельствует о непродуманном дизайне программы. Дизайн при котором в приложении существует только один GUI поток и несколько рабочих потоков, которые общаются с гавным потоком через механизм сообщений, является более мощным и убережет вас от многих проблем. Например, в Win32 потоки имеют доступ только к GDI объектам (кистям, обводкам и так далее), которые они создали сами, но не создаными другими потоками.

Для передачи сообщений между потоками вы можете использовать `wxEvtHandler::AddPendingEvent` или ее короткую версию — `wxPostEvent`. Эти функции имеют безопасную многопоточную (thread-safe) реализацию, поэтому они могут быть использованы для передачи сообщений между потоками.

17.2 Использование wxThread

Если вы хотите реализовать некоторую функциональность используя потоки, то вам необходимо сделать класс-наследник от `wxThread` и реализовать по крайней мере метод `Entry`, который отвечает за выполнение работы потока. Предположим, что вы хотите создать отдельный поток, чтобы посчитать число цветов в изображении. Объявляем соответствующий класс потока:

```
class MyThread : public wxThread
{
public:
    MyThread(wxImage* image, int* count):
        m_image(image), m_count(count) {}
    virtual void *Entry();
private:
    wxImage* m_image;
    int*     m_count;
};

// Идентификатор, чтобы уведомить приложение, когда
// работа будет закончена
#define ID_COUNTED_COLORS    100
```

Метод `Entry` делает все вычисления и возвращает код завершения, который возвращается функцией `Wait` (только для присоединенных потоков). Вот наша реализация для `Entry`:

```
void *MyThread::Entry()
{
    (*m_count) = m_image->CountColours();

    // Используем существующее сообщение, чтобы оповестить приложение
    // о том, что вычисления закончены
    wxCommandEvent event(wxEVT_COMMAND_MENU_SELECTED,
                         ID_COUNTED_COLORS);
    wxGetApp().AddPendingEvent(event);

    return NULL;
}
```

Для упрощения мы используем существующий класс сообщений для оповещения приложения о том, что подсчет цветов закончен.

17.2.1 Создание

Потоки создаются в два шага. Сначала объект инстанцируется, а далее для него вызывается метод `Create`:

```
MyThread *thread = new MyThread();
if ( thread->Create() != wxTHREAD_NO_ERROR )
{
    wxLogError(wxT("Can't create thread!"));
}
```

Существует два вида потоков: одни работают по принципу «запустил и забыл», а от других программа жде результатов работы, чтобы продолжить выполнение. Первые называются отсоединенными потоками (detached threads), а вторые — присоединенными (joinable threads). Тип потока обозначается с помощью передачи флага `wxTHREAD_DETACHED` (по умолчанию) или `wxTHREAD_JOINABLE` в конструктор `wxThread`. Результат присоединенного потока возвращается функцией `Wait`. Вы не можете вызывать эту функцию для неприсоединенных потоков.

Однако это не озанчает, что вы должны созадвать все потоки как присоединяемые, так как это тип имеет некорые недостатки. Например, вы должны ждать окончания выполнения фнукции потока, используя `wxThread::Wait` (иначе ресурсы, которые он использует никогда не будут освобождены), а также удалить соответствующий объект `wxThread` (однажды будучи использованым, экземпляр такого объекта нельзя использовать снова). В отличие от описаного неприсоединенные потоки отражают концепцию «запустил и забыл». Вам необходимо только его запустить, а остановится и освободит память этот объект уже сам. Присоедененные потоки могут быть созданы в стеке, хотя обычно также создаются в куче. Никогда не создавайте глобальный объект-поток, так как он выделяет память в своем конструкторе, что может вызвать ложное срабатывание у программ следящими за корректным освобожением памяти.

Определяем размер стека

Вы можете указать требуемый размер стека для потока как параметр в метода `Create`. Если в качестве размера указать ноль, то это будет означать, что будет использовано значение по умолчанию для данного платформы.

Определяем приоритет

Некоторые операционные системы позволяют приложению давать некоторые подсказки на счет того, сколько процессорного времени необходимо потоку. Для этого нужно вызвать метод `wxThread::SetPriority` с целым числом из диапазона от 0 до 100, где 0 — это минимум, а 100 — максимум. Существуют константы `WXTHREAD_MIN_PRIORITY`, `WXTHREAD_DEFAULT_PRIORITY` и `WXTHREAD_MAX_PRIORITY`, которые имеют значения 0, 50 и 100 соответственно. Вызов данного метода необходимо делать после вызова `Create`, но до вызова `Run`.

17.2.2 Запуск потока

После вызова Create поток создается, но не запускается. Для его запуска необходимо вызвать `wxThread::Run`, который запустит функцию `Entry` потока.

17.2.3 Как приостановить поток или ждать внешнего события

Если потоку необходимо дождаться, чтобы что-то произошло, то вы должны избегать опрашивания и ожидания в цикле, что всегда означает, что процессор будет занят выполнением бесполезной работы (busy waiting).

Если вам необходимо, чтобы поток подождал несколько секунд, то пошлите поток в спячку используя метод `wxThread::Sleep`.

Если вам необходимо дождаться какого-то события, то вы должны использовать вызов, который заблокирует выполнение потока, пока не прийдет уведомление об изменении. Например, если в вашем приложении используется сокет внутри потока, то вы должны использовать блокирующий вызов сокета, которы просто остановит поток, пока не появяться данные. Как вы видите в этом случае не нужно организовывать никаких циклов ожидания. Или если вы ждете данные из присоединенного потока, то вы должны заблокировать текущий поток с помощью метода `Wait`.

Используя методы `Pause` и `Resume` вы можете временно посылать поток в спячку. Однако существует несколько проблем о которых вы должны помнить. Во-первых, так как `Pause` может эмулироваться на некоторых платформах (по большей части это касается POSIX-систем), то поток должен периодически вызывать `TestDestroy` и как можно быстрее завершить свою работу, если этот метод возвратил `true`. Вторую проблему гораздо труднее обойти. Операционная система может отправить в спячку поток в любое время, которое может легко оказаться временем, когда поток блокирует мютекс, что может привести к неразрешимой взаимоблокировке приложения (deadlock).

Таким образом в большинстве случаев использование `Pause` и `Resume` говорит о плохом дизайне архитектуры программы. Вы должны пытаться преобразовать ваш код для использования синхронизирующих объектов (смотрите следующий раздел «Объекты синхронизации»).

17.2.4 Завершение работы потока

Как мы упоминали до этого отсоединенные потоки автоматически уничтожаются после своего завершения. Для присоединенных потоков вы можете просто вызвать `wxThread::Wait`, или в GUI-приложении вы можете проверять `wxThread::IsAlive` в обработчике для простоя главного потока и вызывать `Wait` только если `IsAlive` возвратит `false`. Вызов `Wait` гарантирует, что ресурсы потока будут освобождены. Конечно, более разумной альтернативой будет просто использовать отсоединенные потоки, которые бросают сообщение после своего завершения.

Вы можете использовать `wxThread::Delete`, чтобы послать запрос на завершение потоку. Чтобы этот метод работал поток должен периодически вызывать `TestDestroy`.

17.3 Объекты синхронизации

Почти при любом использовании потоков необходимо уметь разделять данные между потоками. Когда два потока пытаются получить доступ к одним данным (независимо от того являются ли эти данные объектом или ресурсами), необходимо синхронизировать доступ к ним, чтобы избежать доступа или изменения данных более чем одним потоком одновременно. Всегда существуют так называемые инварианты в предположения программы о соответствующих элементах данных. Например для списка таким предположением является то, что указатель указхывает на первый элемент, а каждый элемент содержит указатель на следующий или `NULL` в случае последнего элемента. Во время вставки нового элемента в сисок существует момент, когда этот инвариант нарушен. Если список используется из двух потоков, то вы должны защитить этот момент, чтобы в этот момент список больше никем не использовался.

Очень важно убедится, что разделяемые данные не только захватываются потоком, но и что другие потоки не могут в этот момент работать с ними. Сейчас мы опишем классы, которые позволят вам реализовать такие гарантии.

17.3.1 wxMutex

Имя данного объекта произошло от "mutual exclusion"[1]. Объект представляет собой простейший в использовании объект синхронизации. Он позволяет убедиться, что только один поток работает с заданным куском данных. Чтобы получить доступ к данным приложение вызывает метод `wxMutex::Lock`, который блокирует (останавливает выполнение) до тех пор пока ресурс не освободиться. `wxMutex::Unlock` освобождает занятый ресурс. Хотя возможно использовать `Lock` и `Unlock` мютекса непосредственно, но лучше использовать класс `wxMutexLocker`, чтобы быть уверенным, что мютекс всегда будет корректно освобожден даже если в вашем коде произойдет исключение.

В следующем примере предполагается, что класс `MyApp` содержит поле `m_mutex` типа `wxMutex`.

```
void MyApp::DoSomething()
```

[1] взаимное исключение (англ.)

```
{
    wxMutexLocker lock(m_mutex);
    if (lock.IsOk())
    {
        ... делаем что-нибудь
    }
    else
    {
        ... мы не можем захватить мютекс
        ... фатальная ошибка
    }
}
```

Существует три важных правила использования мютексов:

- Поток не может захватить мютекс, который уже захвачен (даже рекурсивно). Хотя есть системы, которые позволяют это, но это не верно для всех операционных систем.
- Поток не может освободить мютекс, захваченный другим потоком. Если вам необходимо нечто подобноо вы должны использовать семафоры (рассматриваются далее).
- Если вы в потоке который способен делать другую работу, если не сможет захватить мютекс, то вы должны вызвать `wxMutex::TryLock`. Метод возвратит результат немедленно и укажет смог ли поток захватить мютекс (`wxMUTEX_NO_ERROR`) или нет (`wxMUTEX_DEAD_LOCK` или `wxMUTEX_BUSY`). Особенно важно применять данный метод для главного (GUI) потока, который не должен никогда блокироваться, так как в этом случае приложение станет невосприимчивым к пользовательскому вводу.

17.3.2 Взаимная блокировка (deadlocks)

Взаимная блокировка возникает когда два потока ждут освобождения ресурса, которые другой поток уже занял. Предположим поток А уже захватил мютекс 1 и поток Б захватил мютекс 2. Если поток А ждет освобождения мютекса 2, а Б — освобождения 1, то они будут ждать вечно. Некоторые ОС способны распознать такие случаи и возвратить специальный код ошибки `wxMUTEX_DEAD_LOCK` при вызове методов `Lock`, `Unlock` или `TryLock`. В остальных системах программа просто зависает, пока пользователь вручную не убъет ее.

Существует два стандартных пути решения данной проблемы:

1. Использование фиксированного порядка. Строится постоянная иерархия захвата объектов. Для предыдущего примера каждый поток обязан сначала захватить мютекс 1, а только потом захватить мютекс 2. В этом случае взаимная блокировка произойти не сможет.
2. Использование тестового захвата. Захватить первый объект, а далее вызвать TryLock для захвата следующего объекта. Если вызов закончился неудачей

освободить все ресурсы и начать захват заново. Это более трудоемкий путь, но вы можете его использовать, если использование первого метода кажется вам недостаточно гибким.

17.3.3 wxCriticalSection

Критические секции существуют для защиты некоторого участка кода, но на практике они очень похожи на мютексы. Единственная разница в том, что мютексы обычно видны за пределами приложения и могут разделяться между процессами, а критические секции видны только внутри приложения. Это делает их более эффективными на платформах на которых они имеют родную реализацию. В соответствии с целью разлечается также и терминология: если мютекс может быть захвачен и освобожден, то программа может войти и выйти из критической секции.

Вы должны по возможности использовать класс `wxCriticalSectionLocker` вместо прямого использования `wxCriticalSection` по тем же причинам по которым желательно использовать `wxMutexLocker` вместо `wxMutex`.

17.3.4 wxCondition

Переменная-условие используется для ожидания некоторых изменений разделяемых данных. Например, вы можете иметь условие (condition), означающее, что очередь содержит некоторые данные. Разделяемые данные, находящиеся в очереди в данном примере защищаются с помощью мютекса.

Вы можете решить данную проблему с помощью цикла, который блокирует мютекс, тестирует доступность данных в очереди и освобождает мютекс, если данных не обнаружено. Однако этот путь очень неэффективный, так как все время работает цикл, который просто ждет момента, чтобы занять мютекс. В таких ситуациях гораздо эффективнее использовать условия, так как поток можно остановить до тех пор, пока другой поток не подаст сигнал об изменении состояния данных.

Множество потоков могут ждать выполенния одного и того же условия. В этом случае мы можем разбудить один или несколько потоков. Вызов `Signal` разбудит один ожидающий поток, а вызов `Broadcast` разбудит все потоки, ожидающие выполнения одного и того же условия. Если несколько предикатов реализованы через один и тот же `wxCondition`, то необходимо использовать `Broadcast`, так как разбуженый поток может не суметь стартовать, так как ждет выполнения другого предиката.

Пример использования wxCondition

Предположим у нас есть два потока:

- Производящий поток, который кладет десять элементов в очередь, после чего дает сигнал «очередь заполнена». Далее он ждет событие «очередь пуста» и заполняет ее снова.
- Обслуживающий поток, который ждет событие «очередь заполнена», после чего очищает ее.

Нам необходим один мютекс `m_mutex`, защищающий очередь и две переменных-условия `m_isFull` и `m_isEmpty`. Их можно создать передав в конструктор переменную

m_mutex в качестве параметра. Очень важно, чтобы вы всегда явно тестировали предикат, так как сигнал о выполнении условия может быть получен до того как вы начали его ждать.

В псевдокоде функция Entry производящего потока будет иметь вид:

```
while ( notDone )
{
   wxMutexLocker lock(m_mutex) ;
   while( m_queue.GetCount() > 0 )
   {
      m_isEmpty.Wait() ;
   }
   for ( int i = 0 ; i < 10 ; ++i )
   {
      m_queue.Append( wxString::Format(wxT("Element %d"),i) ) ;
   }
   m_isFull.Signal();
}
```

А вот код обслуживающего потока:

```
while ( notDone )
{
   wxMutexLocker lock(m_mutex) ;
   while( m_queue.GetCount() == 0 )
   {
      m_isFull.Wait() ;
   }
   for ( int i = queue.GetCount() ; i > 0 ; i )
   {
      m_queue.RemoveAt( i ) ;
   }
   m_isEmpty.Signal();
}
```

Метод Wait освобождает мютекс, а далее ждет когда выполнится условие. Когда метод сработает, он захватит мютекс снова, гарантируя правильную синхронизацию.

Очень важно проверять выполнение предиката не только перед входом, но также после выхода из Wait, так как что-нибудь может измениться между получением сигнала и просыпанием потока, в результате чего предикат может уже не быть истинным. Существуют системы, которые вызывают ложные пробуждения.

Заметим, что вызов Signal может произойти до того, как остальные потоки вызовут Wait и, как и в случае с условиями в pthread, сигнал будет потерян. Поэтому, если вы хотите быть уверенными, что не пропустили сигнал вы должны держать мютекс, ассоциированный с условием при старте в захваченном состоянии, а далее захватить его снова перед вызовом Signal. Это означает, что этот вызов заблокируется до того, как вызовется Wait из другого потока.

Следующий пример показывает как главный поток может запустить рабочий, который стартует а далее ждет пока главный поток не просигналит о продолжении:

```
class MySignallingThread : public wxThread
{
public:
    MySignallingThread(wxMutex *mutex, wxCondition *condition)
    {
        m_mutex = mutex;
        m_condition = condition;

        Create();
    }

    virtual ExitCode Entry()
    {
        ... делаем нашу работу ...

        // говорим другому(им) потоку(ам), что можно завершить работу:
        // сначала мы блокируем мютекс, в противном случае мы сможем
        // сигнализировать о событии до того, как ожидающие потоки
        // начнут его ожидать!
        wxMutexLocker lock(m_mutex);
        m_condition.Broadcast(); // в случае одного ожидающего то
                                 // же самое, что и Signal()

        return 0;
    }

private:
    wxCondition *m_condition;
    wxMutex *m_mutex;
};

void TestThread()
{
    wxMutex mutex;
    wxCondition condition(mutex);

    // мютекст должен быть вначале захвачен
    mutex.Lock();

    // создаем и запускаем поток, но оповещаем, что он не может
    // выйти (и сигнализировать о своем выходе) до того как мы
    // не разблокируем мютекс, расположенный ниже
    MySignallingThread *thread =
        new MySignallingThread(&mutex, &condition);

    thread->Run();
```

```
    // ждем завершения потоков: Wait() атомарно разблокирует
    // мютекс, который позволит потоку продолжить выполнение и
    // начать процедуру ожидания
    condition.Wait();

    // теперь мы можем выйти
}
```

Конечно, было бы намного лучше просто использовать присоединенный поток и вызывать для него `wxThread::Wait`, но этот пример просто иллюстрирует выжность правильной блокировки мютекса при использовании `wxCondition`.

17.3.5 wxSemaphore

Семафор — это некоторая смесь из счетчика и мютекса. Главное его отличие от мютекса в том, что семафор может активироваться из любого потока, а не только из владельца, поэтому можно представлять семафор как счетчик без владельца.

Поток, вызывающий `Wait` для семафора, ожидает когда этот счетчик станет потоложительным, далее метод уменьшает счетчик и делает возврат. Поток, вызывающий `Post` для семафора, увеличивает значение счетчика.

Для семафоров в wxWidgets существует дополнительная функциональность, заключающаяся в том, что в конструкторе можно указать его максимальное значение. Число 0, подставляемое по умолчанию, означает, что верхней границы не существует. Если вы укажете ненулевое максимальное значение и поток вызовет `Post` в неправильный момент (когда значение счетчика достигло максимального значения), то метод возвратит ошибку `wxSEMA_OVERFLOW`. Чтобы проиллюстрировать данную концепцию давайте рассмотрим «универсальный мютекс», описанный ранее:

- Мютекс может быть заблокирован и разблокирован из разных потоков, что можно реализовать с помощью меиафора с максимальным значением 1. Метод `mutex.Lock` можно реализовать с помощью `semaphore.Wait`, а метод `mutex.Unlock` с помощью `semaphore.Post`.
- Первый поток вызывает `Lock` (что в нашем случае `Wait`), находит положительное значение в семафоре, уменьшает его и немедленно продолжает работу.
- Другой поток вызывает `Lock`, видит нулевое значение и начинает ждать, пока кто-нибудь (не обязательно первый поток) вызовет `Unlock` (соответственно `Post` для семафора).

17.4 Пример использования потоков в wxWidgets

Вы можете найти работающий пример использования большинства рассмотренных здесь возможностей в каталоге samples/thread вашего дистрибутива wxWidgets (рис. 17.1). В этом примере вы можете запускать, останавливать, приостанавливать и продолжать работу потоков. Пример демонстрирует «рабочий поток», который периодически посылает сообщение главному потоку с помощью `wxPostEvent`, которое говорит о том, что поток закончил свое выполнение и полностью заполнил строку прогресса в диалоге.

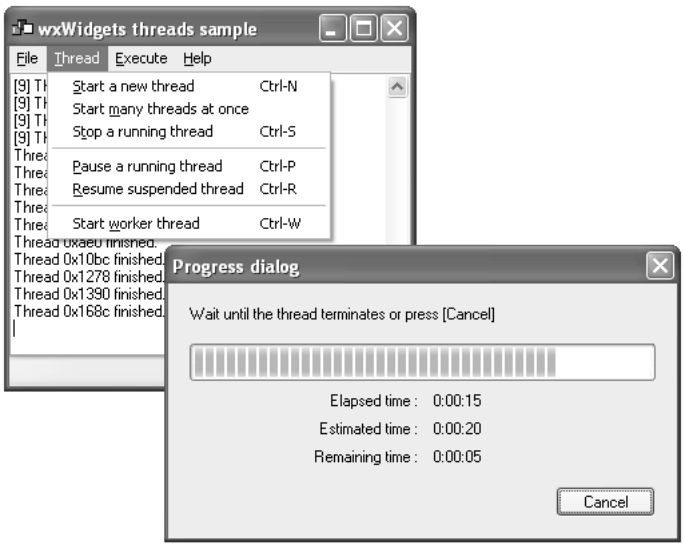

Рис. 17.1: Пример использования потоков в wxWidgets

17.5 Альтернативы многопоточности

Если вы находите, что использование потоков слишком сложно возможно вы сможете выполнить ту же задачу используя таймеры, обработку событий о простое или оба этих метода одновременно.

17.5.1 Использование wxTimer

Класс `wxTimer` позволяет вашему приложению получать периодические напоминания один раз или на постоянной основе. Вы можете использовать `wxTimer` как альтернативу потокам в сдучае, если сможете разбить вашу задачу на группу мелких подзадач, которые возможно выполнять раз в несколько миллисекунд, давая таким образом приложению достаточно времени для обработки событий от пользователя.

Вы можете выбирать каким именно образом уведомлять ваше приложение. Если вы предпочитаете виртуальные функции, то необходимо создать наследника от `wxTimer` и перегрузить метод `Notify`. Если вы предпочитаете получение события `wxTimerEvent`, то передайте указатель на `wxEvtHandler` в таймер (через конструктор или используя `SetOwner`) и используйте `EVT_TIMER(id, func)` для присоединения таймера к функции-обработчику.

Дополнительно вы можете указать идентификатор, который вы до этого передали таймеру в конструктор или через `SetOwner`, чтобы идентифицировать объект-таймер и далее передавать данный идентификатор в `EVT_TIMER`. Ота технология используется когда у вас в программе используется несколько различных таймеров.

Запускается таймер с помощью вызова метода `Start`, которому передается временной интервал в миллисекундах. Вызов `Stop` останавливает таймер, а `IsRunning` используется для определения запущен ли таймер в данный момент.

Следующий пример показывает как организовать обработку сообщений от таймера.

```
#define TIMER_ID 1000

class MyFrame : public wxFrame
{
public:
    ...
    void OnTimer(wxTimerEvent& event);

private:
    wxTimer m_timer;
};

BEGIN_EVENT_TABLE(MyFrame, wxFrame)
    EVT_TIMER(TIMER_ID, MyFrame::OnTimer)
END_EVENT_TABLE()

MyFrame::MyFrame()
       : m_timer(this, TIMER_ID)
{
    // Используем интервал в 1 секунду
    m_timer.Start(1000);
}

void MyFrame::OnTimer(wxTimerEvent& event)
{
    // Код в этой функции выполняется каждую секунду
    // ...
}
```

Заметим, что нет никакой гарантии, что ваш обработчик событий будет вызываться точно каждые n миллисекунд. Указанная величина зависит от того, что происходит в системе между в этот интервал.

Если уж мы коснулись темы измерения времени, то стоит упомянуть про `wxStopWatch`. Этот полезный класс служит для измерения временных интервалов. Конструктор запускает таймер, вы можете останавливать и продолжать его, а также получить прошедшее время в миллисекундах. Например:

```
wxStopWatch sw;

SlowBoringFunction();

// Останавливаем таймер
sw.Pause();

wxLogMessage("The slow boring function took %ldms to execute",
             sw.Time());
```

```
// Снова включаем таймер
sw.Resume();

SlowBoringFunction();

wxLogMessage("And calling it twice took %ldms in all", sw.Time());
```

17.6 Обработка при простое

Другим способом периодичным оповещением приложения является реализация обработчика для простоя (idle). Все объекты приложений и все окна генерируют событие о простое когда заканчивают обработку всех остальных событий. Если обработчик данного события вызывает `wxIdleEvent::RequestMore`, то данное событие генерируется снова. В противном случае е будет сгенерировано никаких событий о простое до тех пор пока не будет сгенерирована и обработана следующая группа событий от пользовательского интерфейса. Обычно стоит вызвать `wxIdleEvent::Skip`, чтобы это событие обрабатывалось базовым классом.

В следующем примере гипотетическая функция `FinishedIdleTask` делает некоторую часть работы, а когда заканчивает возвращает `true`.

```
class MyFrame : public wxFrame
{
public:
    ...
    void OnIdle(wxIdleEvent& event);

    // Делаем маленькую часть работы. Возвращаем true
    // если задача завершена
    bool FinishedIdleTask();
};

BEGIN_EVENT_TABLE(MyFrame, wxFrame)
    EVT_IDLE(MyFrame::OnIdle)
END_EVENT_TABLE()

void MyFrame::OnIdle(wxIdleEvent& event)
{
    // Обрабатываем простой. Если работа не закончена
    // запрашиваем дополнительное событие о простое

    if (!FinishedIdleTask())
        event.RequestMore();

    event.Skip();
}
```

Хотя в данном примере мы использоватли фрейм, обработка событий о простое не ограничивается окнами верхнего уровня. Любое окно может перехватывать такие

сообщения. Например, вы можете реализовать элемент управления для показа изображения, который масштабирует изображение по своему размеру только при простое, что позволяет избежать мерцаний при изменении размера окна. Чтобы быть уверенными, что обработка простоя приложения случайно не припятствует работе контрола мы можем перегрузить виртуальную функцию `OnInternalIdle` нашего элемента управления. Из ее реализации не забудьте вызвать `OnInternalIdle` базового класса. Часть кода нашего элемента управления может выглядеть следующим образом:

```
void wxImageCtrl::OnInternalIdle()
{
    wxControl::OnInternalIdle();

    if (m_needResize)
    {
        m_needResize = false;
        SizeContent();
    }
}

void wxImageCtrl::OnSize(wxSizeEvent& event)
{
    m_needResize = true;
}
```

Иногда вам может потребоваться, чтобы сгенерировалось событие о простое, в случае когда нет других ожидающих событий, которые могли бы сгенерировать событие о простое. Вы можете сделать это с помощью функции `wxWakeUpIdle`. Другой способ заключается в том, чтобы запустить `wxTimer`, который не выполняет никакой работы. Так как в последнем случае всегда генерируется событие таймера, а после него событие о простое. Чтобы немедленно обработать все сообщения о простое необходимо вызвать `wxApp::ProcessIdle`, но учтите, что для некоторых платформ это может повлиять на внутреннюю обработку событий о простое (в GTK+ перерисовка окон выполняется именно в этом обработчике).

Обработка событий обновления интерфейса, рассмотренная в Главе 9 «Создание дислогов», является одной из форм обработки событий о простое, которое позволяет элементам управления обновлять свое состояние, перехватывая `wxUpdateUIEvent`.

17.7 Yield

Когда приложение долгое время занято рассчетом продолжительной задачи и пользовательский интерфейс не обновляется, вы можете сделать его обновление с помощью периодического вызова `wxApp::Yield` (или его синонима `wxYield`) для обработчки ожидающих событий. Эту технику надо использовать очень осторожно, так как она может привести к нежелательным эффектам, таким как повторный вход (reentrancy). Например, `Yield` может обработать пользовательскую команду, которая приведет к повторному запуску вычисления алгоритма, даже если он до сих пор

вычисляется. Функция `wxSafeYield` блокирует все окна, обрабатывает события, а после включает все окна снова. Таким образом исключается возможность повторного входа. Если передать `true` в `wxApp::Yield`, то обработка произойдет только, если программа не находится в этом состоянии. Это является еще одной возможностью, чтобы избежать проблемы повторного входа.

Если вам необходимо периодически обновлять только некоторое окно, то рекомендуется использовать метод `wxWindow::Update`. Он обработает только ожидающие сообщения о рисовании для данного окна.

17.8 Итоги

Если дать продавцу в супермаркете на обслуживание две линии, то это не приведет к увеличению числа обслуженных покупателей. Таким же образом многопоточность не сделает ваше приложение быстрее (по крайней мере без специального оборудования). Однако приложение станет выглядеть быстрее для пользователя, так как интерфейс станет более отзывчивым. Пока на одной линии продавец ждет от покупателя кредитную карточку, то он может обслуживать покупателя на другой линии, так и мультипоточность может помогать более эффективно использовать доступные ресурсы. Некоорые задачи можно решить с помощью многопоточности более элегантно, чем при использовании только одного потока. В этой главе мы также коснулись темы эмуляции многопоточности с помощью таймеров и обработки событий о простое.

Тема многопоточного программирования гораздо сложнее, чем можно рассказать в одной главе. За дополнительной информацией мы рекомендуем книгу «Programming with POSIX Threads» Девида Батенхоффа.

В следующей главе мы расскажем о использовании сокетов для передачи данных между процессами.