Leow Wee Kheng
CS3249 User Interface Development

Multithreading
My processor has 4 cores!
Can it run faster!?
Process

- A process is a running program managed by OS.
- A process consists of
  - executable code in memory
  - static and dynamic data
  - execution context
    - program counter
    - contents of registers
    - stack pointer
    - memory management information,
    - etc.
 Memory protection
  ○ A process has its own address space or virtual memory space.
  ○ Cannot access address spaces of other processes.
  ○ OS maps virtual memory to physical memory.

 Multiprogramming or multitasking
  ○ A process is given a fixed time to run.
  ○ When time is up, OS suspends the process, and switches context to another waiting process.

 Processes are heavy-weight
  ○ Context switching is costly.
Multithreading

- **Thread**
  - A light-weight sequence of a running program.
  - All threads of a process use the same address space.
  - Context switching is cheaper.

- **Multithreading**
  - Run multiple threads at the same time.
  - Different threads can run in different processors.
  - Some programming languages support multithreading
    - Ada, Java
  - In Unix/Linux, use pthread to implement multithreading.
Multithreading in Qt

- Qt application can run in multiple threads.
  - All window operations must run in the main thread.
  - Non-window operations can run in secondary threads.

- Qt has a QThread class for easy creation of threads.
// Thread.h

class Thread: public QThread {

    Q_OBJECT

public:
    Thread();
    void stop();

protected:
    void run();

private:
    volatile bool stopped;
};

inherits QThread

Redefines QThread::run()

volatile tells compiler not to optimise; variable is accessed from other threads.
// Thread.cpp

Thread::Thread()
{
    stopped = false;
}

void Thread::run()
{
    while (!stopped)
    {
        cout << ".";
        stopped = false;
        cout << "\n";
    }
}

void Thread::stop()
{
    stopped = true;
}
Use dialog to illustrate thread creation and stopping.

- click start to start thread
- click stop to stop thread
// ThreadDialog.h

class ThreadDialog : public QDialog
{
    Q_OBJECT

public:
    ThreadDialog(QWidget *parent = 0);

protected:
    void closeEvent(QCloseEvent *event);

private slots:
    void startOrStopThread();

private:
    Thread thread;
    QPushButton *threadButton;
    QPushButton *quitButton;
};
// ThreadDialog.cpp

ThreadDialog::ThreadDialog(QWidget *parent):
    QDialog(parent)
{
    // Create widgets.

    threadButton = new QPushButton(tr("Start"));
    connect(threadButton, SIGNAL(clicked()),
            this, SLOT(startOrStopThread()));

    quitButton = new QPushButton(tr("Quit"));
    quitButton->setDefault(true);
    connect(quitButton, SIGNAL(clicked()),
            this, SLOT(close()));
// Layout widgets.

QHBoxLayout *layout = new QHBoxLayout;
layout->addWidget(threadButton);
layout->addWidget(quitButton);
setLayout(layout);
setWindowTitle(tr("Thread test"));

void ThreadDialog::closeEvent(QCloseEvent *event) {
    thread.stop();  // Stop the thread.
    thread.wait();  // Wait for thread to finish.
    event->accept();
}
void ThreadDialog::startOrStopThread()
{
    if (thread.isRunning())
    {
        thread.stop();
        threadButton->setText(tr("Start"));
    }
    else
    {
        thread.start();
        threadButton->setText(tr("Stop"));
    }
}

- isRunning() and start() are inherited from QThread.
Staying Responsive

- One way is to dispatch events regularly.

```cpp
long-operation()
{
    for each iteration
    {
        do work of one iteration;
        qApp->processEvents(); // dispatch events
    }
}
```

- This method is impossible if
  - you don't have source code of `long-operation`, or
  - you don't want to change the source code.

- Alternative: run long operation in another thread.
Example: Volume Image Viewer

- 3D image is large, takes a while to load.
- Want to show progress while loading.
// program fragment

void VolumeViewer::loadImage(const QString &dirName)
{
  VTKDICOMReader *reader = VTKDICOMReader::New();
  char *dir = dirName.toAscii().data();
  reader->SetDirectoryName(dir);
  reader->Update();

  // display image ...
}

// Method 1: use wait cursor

void VolumeViewer::loadImage(const QString &dirName)
{
    VTKDICOMReader *reader = VTKDICOMReader::New();
    char *dir = dirName.toAscii().data();
    reader->SetDirectoryName(dir);

    QApplication::setOverrideCursor(Qt::WaitCursor);
    reader->Update();
    QApplication::restoreOverrideCursor();

    // display image ...
}
// Method 2: Multithreading

class ReadThread: public QThread
{
    Q_OBJECT

public:
    ReadThread(vtkDICOMImageReader *rd);
    bool isDone();

protected:
    void run();

private:
    volatile bool done;
    vtkDICOMImageReader *reader;
};
ReadThread::ReadThread(vtkDICOMImageReader *rd)
{
    reader = rd;  // Keep it for run function.
    done = false; // Init.
}

bool ReadThread::isDone()
{
    return done;
}

void ReadThread::run()
{
    reader->Update();  // Read image.
    done = true;
}
void VolumeViewer::loadImage(const QString &dirName)
{
    VTKDICOMReader *reader = VTKDICOMReader::New();
    char *dir = dirName.toAscii().data();
    reader->SetDirectoryName(dir);

    QProgressDialog progress(
        QString("Reading, please wait...") ,
        QString("Close"), 0, 0);
    progress.show();
    qApp->processEvents();  // Optional

    ReadThread thread(rd);
    thread.start();
    while(!thread.isDone())
    {
        qApp->processEvents();
        thread.quit();  // Stop thread’s event loop.
        progress.hide();
    // display image ...
    }
}
In general, can allow certain interactions in main thread instead of just displaying progress dialog.

**Caution:**
- Operations in different threads must not conflict.
  - Example: While doing long calculation on data, must disable GUI's read, edit functions.
    ```java
    openFileAction->setEnabled(false);
    ```
- If different threads refer to same data (shared data), must synchronise threads.
  - Otherwise, may get corrupted data or unstable execution.
Thread Synchronisation

- Producer-consumer problem:
  - Producer and consumer share common buffer.
  - Producer adds to **front** of buffer until it is full.
  - Consumer removes from **back** of buffer until it is empty.
  - Increment pointer before read / write data.
- Empty buffer

- Full buffer
Program Structure

Control

Producer: start / stop

Buffer: write

Consumer: read

Buffer test:
- start producer
- start consumer
// Buffer.h

class Buffer
{
    friend class Producer;
    friend class Consumer;

public:
    Buffer() {front = back = 0;}  // Init to empty buffer.

protected:
    enum {size = 10};
    int data[size];
    int front;  // Points to front of buffer.
    int back;   // Points to back of buffer.
};
// Producer.h

class Producer: public QThread
{
    Q_OBJECT

public:
    Producer(Buffer *b);
    void stop();

protected:
    void run();

private:
    Buffer *buffer;  // Points to shared buffer.
    volatile bool stopped;
};
// Producer.cpp

Producer::Producer(Buffer *b)
{
    buffer = b;  // Keeps pointer to shared buffer.
    stopped = false;
}

void Producer::stop()
{
    stopped = true;
}
void Producer::run()
{
    static int data = 0;

    while (!stopped)
    {
        if (buffer->front != buffer->back - 1 and
            !(buffer->front == buffer->size - 1 and
            buffer->back == 0)) // Not full
        {
            ++(buffer->front); // Increment front.
            if (buffer->front == buffer->size)
                buffer->front = 0; // Wrap around.

            usleep(500); // Simulate put to sleep by OS.
        }
    }
}
++data;
// Add data to buffer.
buffer->data[buffer->front] = data;

cout << "p." << data << " " << flush;
}
}

stopped = false;
}
// Consumer.h

class Consumer: public QThread
{
    Q_OBJECT

public:
    Consumer(Buffer *b);
    void stop();

protected:
    void run();

private:
    Buffer *buffer;  // Points to shared buffer.
    volatile bool stopped;
};
// Consumer.cpp

Consumer::Consumer(Buffer *b) 
{
    buffer = b;  // Keeps pointer to shared buffer.
    stopped = false;
}

void Consumer::stop()
{
    stopped = true;
}
void Consumer::run()
{
    while (!stopped)
    {
        if (buffer->back != buffer->front) // Not empty.
            { 
                ++$buffer->back;  // Increment back.
                if ($buffer->back == $buffer->size)
                    $buffer->back = 0;  // Wrap around.

                usleep(100);  // Simulate put to sleep by OS.

                // Get data.
                int data = buffer->data[$buffer->back];
                cout << "c." << data << " " << flush;
            }
    }

    stopped = false;
}
/main.cpp

int main(int argc, char **args)
{
    QApplication app(argc, args);

    Buffer *buffer = new Buffer;
    Producer *producer = new Producer(buffer);
    Consumer *consumer = new Consumer(buffer);
    Control *control = new Control(producer, consumer);
    control->show();

    return app.exec();
}
Buffer overrun can happen.

- Example: Empty buffer with previously read data (grey).

<p>| | | | | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td></td>
</tr>
</tbody>
</table>

Producer increments pointer, then it is put to sleep by OS before writing data to buffer.

<p>| | | | | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td></td>
</tr>
</tbody>
</table>

Consumer sees non-empty buffer.
- Consumer increments pointer and reads 6 (garbage).

| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |

- Producer wakes up and writes new data 11.

| 1 | 2 | 3 | 4 | 5 | 11 | 7 | 8 | 9 | 10 |

- Consumers sees empty buffer, never reads 11.
○ Producer increments pointer, writes 12.

```
  1  2  3  4  5  11  12  8  9  10
  back  front
```

○ Consumer increments pointer, reads 12, misses 11.

```
  1  2  3  4  5  11  12  8  9  10
  back  front
```
If producer always sleeps after incrementing pointer...

- Example: Empty buffer with previously read data (grey).

```
+---+---+---+---+---+---+---+---+---+---+
|1  |2  |3  |4  |5  |6  |7  |8  |9  |10 |
+---+---+---+---+---+---+---+---+---+---+
         back           front
```

- Producer increments pointer, then it is put to sleep by OS before writing data to buffer.

```
+---+---+---+---+---+---+---+---+---+---+
|1  |2  |3  |4  |5  |6  |7  |8  |9  |10 |
+---+---+---+---+---+---+---+---+---+---+
         back           front
```

- Consumer sees non-empty buffer.
- Consumer increments pointer and reads 6 (garbage).

- Producer wakes up and writes new data 11.
- Producer increments pointer, then goes to sleep.

- Consumer increments pointer, reads 7 (garbage).
Unsafe execution

- Buffer overrun
- Consumer sees empty buffer before producer writes new data
- Consumer reads garbage before producer writes new data

Consumer sleeps less...
Qt provides QMutex to protect critical region.

- mutex = mutual exclusion
- If a thread tries to lock a locked mutex, it is put to sleep until mutex is unlocked.
/ Buffer.h

class Buffer
{
    friend class Producer;
    friend class Consumer;

public:
    Buffer() {front = back = 0;}  // Init to empty buffer.

protected:
    enum {size = 10};
    int data[size];
    int front;  // Points to front of buffer.
    int back;   // Points to back of buffer.
    QMutex mutex;
};
```cpp
void Producer::run()
{
    static int count = 0;

    while (!stopped)
    {
        buffer->mutex.lock();
        if (buffer->front != buffer->back - 1 and
            !(buffer->front == buffer->BufferSize - 1 and
            buffer->back == 0)) // Not full
        {
            ...
        }
        buffer->mutex.unlock();
    }

    stopped = false;
}
```
void Consumer::run()
{
    while (!stopped)
    {
        buffer->mutex.lock();
        if (buffer->back != buffer->front) // Not empty.
        {
            ...
        }
        buffer->mutex.unlock();
    }
    stopped = false;
}
Safe execution

- Producer and consumer rotate to access whole buffer.
- Reason:
  - Producer keeps locking mutex immediately after unlocking.
  - Consumer can't lock mutex until buffer is full.
- No benefit with multithreading.
In practice, after unlocking mutex, producer does something before locking mutex again.

```cpp
void Producer::run()
{
    while (!stopped)
    {
        spend time to produce data;
        buffer->mutex.lock();
        if (buffer is not full)
        {
            write to buffer;
        }
        buffer->mutex.unlock();
    }
    stopped = false;
}
```
Safe execution

Producer and consumer run concurrently. More efficient.
Question:

- In the previous examples, producer and consumer increment pointers before writing and read data.
- What if they increment pointers after writing and reading? (exercise)
Caution

- Locking and unlocking mutex in complex functions or functions that use C++ exceptions can be error-prone: mutex is locked but not unlocked.
- QMutexLocker's destructor automatically unlocks mutex.
Instead Of

```cpp
void Consumer::run()
{
    while (!stopped)
    {
        buffer->mutex.lock();
        ... // read data
        buffer->mutex.unlock();
    }
}
```

do

```cpp
do
void Consumer::run()
{
    while (!stopped)
    {
        QMutexLocker locker(&(buffer->mutex));
        ... // read data
    }
}
```
Other Protection Mechanisms

- QReadWriteLock
  - Only one object can acquire write lock.
  - Many objects can acquire read lock.

- QSemaphore
  - Can acquire locks on multiple resources.
Communication with Main Thread

- Qt application starts with one **main thread**.
  - Only main thread can create QApplication object, call exec().
  - Main thread runs main event loop.
  - Main thread can start secondary threads.

- Secondary threads
  - Run their own event loops.
  - Can talk to each other via shared variables and mutex.
  - Cannot talk to main thread via shared variables and mutex.
    - Would lock main event loop and freeze GUI.
  - Qt's solution: signal-slot.
Implementation of signal-slot: same thread

- Synchronous: implemented as function calls via meta object.
  - connection $\Rightarrow$ metacall(id), id is signal or slot ID.
  - emit signal $\Rightarrow$ QMetaObject::activate(signalID).

![Diagram showing the implementation of signal-slot with source and target objects, connect(), valuedChange(), metacall(), QMetaObject::activate(), and setValue().]
Implementation of signal-slot: different threads

- Asynchronous: implemented by posting event.
  - Still have \texttt{metacall(id)}.
  - emit signal $\rightarrow$ \texttt{postEvent(receiver, event)}.
Thread Safety

- **Thread-safe function**
  - Can be called **safely** from different threads simultaneously.

- **Thread-safe class**
  - All class functions are thread-safe.

- **Reentrant function**
  - Can be called from different threads simultaneously.
  - May not be thread-safe.
  - Functions that don't access shared variables are reentrant.

- **Reentrant class**
  - Different class instances can be used simultaneously in different threads.
 Most of Qt's non-GUI classes are reentrant.

 C++ classes that don't reference global or shared data are reentrant.

 QWidget and its subclasses are not reentrant.
  ○ They have to run in the same main thread.
  ○ Cannot directly call GUI functions from secondary thread. Have to emit signals which are connected to slots.

 Qt's network classes are reentrant.
Caution

- Multithreading programs can be difficult to debug.
- Use only if necessary, and with caution.
Summary

- Multithreading runs multiple threads simultaneously.
- UI operations must run in main thread.
- Non-UI operations can run in secondary threads.
- Secondary threads communicate with main thread using signal-slot.
- With multiple threads, need to synchronise threads.
- Protect critical regions that access shared data.
- Avoid locking mutex immediately after unlocking.
Further Reading

[Blan2008] chap. 18:
- Other protection mechanisms:
  - read-write-lock, semaphore, etc.
- Communicating with main thread.

[Summ2011] chap 7:
- QtConcurrent: high-level API for mutlithreading.

[Lewi1995]:
- Introduction to multithreading.
Reference